VHDL

Modular Design and Synthesis of Cores and Systems

ZAINALABEDIN NAVABI, PH.D.

VHDL: Modular Design and Synthesis of Cores and Systems

About the Author

Dr. Zainalabedin Navabi is a professor of electrical and computer engineering at Northeastern University. Dr. Navabi is the author of several textbooks and computer-based trainings on VHDL, Verilog, and related tools and environments. Dr. Navabi's involvement with hardware description languages began in 1976 when he started the development of a register-transfer level simulator for one of the very first Hardware Description Languages (HDLs). In 1981 he completed the development of a synthesis tool that generated MOS layout from an RTL description. Since 1981, Dr. Navabi has been involved in the design, definition, and implementation of HDLs. He has written numerous papers on the application of HDLs in simulation, synthesis, and test of digital systems. He started one of the first full HDL courses at Northeastern University in 1990. Since then he has conducted many short courses and tutorials on this subject in the United States and abroad. In addition to being a professor, he is also a consultant to Electronic Design Automation (EDA) companies. Dr. Navabi received his M.S. and Ph.D. from the University of Arizona in 1978 and 1981, and his B.S. from the University of Texas at Austin in 1975. He is a senior member of IEEE, and a member of IEEE Computer Society, ASEE, and ACM. Dr. Navabi is the author of eight books on various aspects of digital system design automation.

VHDL: Modular Design and Synthesis of Cores and Systems

Zainalabedin Navabi, Ph.D.

Professor of Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts

Third Edition

McGraw Hill **Professional**

Want to learn more?

We hope you enjoy this McGraw-Hill eBook! If you'd like more information about this book, its author, or related books and websites, please click here.

In the memory of my father, Mohammad-Hossein Navabi.

This page intentionally left blank

CONTENTS

APPENDIXES

PREFACE

This book is on the IEEE Standard 1076 VHDL hardware description language and the utilization of this language for the design of modern digital systems. The intended audiences are engineers involved in various aspects of digital systems design and manufacturing and students with the basic knowledge of digital system design. The emphasis of the book is on using VHDL for the design, test, and synthesis of digital systems. We will discuss Register Transfer (RT) level digital system design, and discuss how VHDL can be used in this design flow. The book covers development, test, and utilization of configurable cores for RT level design. In addition, we discuss design and utilization of a processor core for embedded system designs.

In the last few years RT level design of digital systems has gone through significant changes. Beyond simulation and synthesis that are now parts of any RTL design process, we are looking at testbench generation and design validation tools and schemes. As with any book on VHDL, this book covers digital design and VHDL for simulation and synthesis. However, to ready design engineers for designing, testing, and verifying large digital system designs, the book contains material for testbench development and testable design.

Design strategy, digital design tools, and the role of VHDL in a modern digital design environment are discussed in Chapter 1 of this book. In the design strategy part we discuss where and how RT level cores are used, what parts of a design are handled by a program running on a processor core, and what parts of a large design are coded in an HDL like VHDL. With this multi domain design methodology, the role of design tools and a common design representation that suits all these domains become very important. We will show that the VHDL language is a suitable language for representation of a design, the components of which are described at different domains.

The introductory part of this book outlines the tools needed in a digital design environment and discusses how these tools benefit digital designers. We will show that at the present time the majority of designers do take advantage of simulation and synthesis tools. Because of this, the book focuses on VHDL simulation semantics and synthesizability of designs. Design error detection by simulation is only possible if the design is simulated with a proper testbench and good design coverage is obtained. For this reason, the book devotes a major chapter to the issue of testbench development. We will also discuss testable designs and develop testbenches for these designs as well.

One of the key subjects treated in this book is the linguistics of the VHDL, including its type oriented semantics, simulation model, and concurrency handling. We believe, to be able to utilize VHDL beyond just an RTL design front-end, and keep up with the changing technology, and to be able to find new ways of using this language for tomorrow's complex designs, a thorough understanding of the language semantics and linguistics aspects of VHDL is required. This belief is why we have devoted a major part of this book to the linguistics and simulation model of VHDL.

This book covers basic VHDL for simulation and synthesis, details of language simulation semantics and syntax, configurable core design, core testing, and processor cores and their applications. These topics are kept separate to allow a designer interested in basic simulation and synthesis to use the book for just that. On the other hand, those who are interested in the linguistics of VHDL can use the parts of the book that are intended for this purpose and use the book as a complete reference. Furthermore advanced designers and test engineers have their own chapters to gather their required information.

In an academic setting, in an undergraduate course the book can be used as a design book with less emphasis on the linguistics and language semantics, and for a graduate course the complete book provides enough material for one semester.

Zainalabedin Navabi
navabi@ece.neu.edu
Boston, Massachusetts
March, 2007

INTRODUCTION

Over the years, design of digital systems has evolved from transistor level logic to RTL. In today's technology, RT level design has reached its level of maturity and new, more abstract design methods are being searched for. With the present level of maturity of RTL and RT level tools, we should go one step beyond just simulation and synthesis and look into new ways a hardware description language like VHDL can help designers.

Toward this goal, and in order to prepare designers for future design challenges, we have concentrated on two key issues. One is that the linguistics and semantics of the VHDL are thoroughly covered. The second contribution of this book is its coverage of test-benches and testability techniques.

In addition to looking at the future of RTL, this book gets deep into today's RT level design by presenting synthesis techniques and developing configurable cores for a core-based design environment. The design of a general purpose processor core in the last chapter of this book is another way to enhance RT level designs by using embedded processors.

VHDL: Modular Design and Synthesis of Cores and Systems covers RTL, system level design methodology, VHDL language syntax and semantics, HDL simulation model, design of configurable cores, testbench and testable design description in VHDL, and processor cores. This book can be used in an academic or industrial setting by students or engineers. In either case it assumes a general knowledge of logic design. The early part of this book provides enough information for simulation and synthesis of basic RT level components. This information is useful for continuing to design and practice the language while more advanced topics are learned in the later chapters of the book.

The first two chapters cover design strategy and simulation and synthesis of basic components. With these two chapters, readers will be able to start writing VHDL codes and perform simulation and synthesis of basic components.

The next three chapters discuss VHDL language syntax and semantics at the structural, functional, and behavioral levels, respectively. The chapters that follow discuss VHDL types and simulation model. These five chapters cover VHDL from a linguistic point of view.

After the discussion of the language, a chapter discusses configurable core design and another chapter talks about core testing and testbench development. The final chapter shows design, test and utilization of a processor core. Advanced modern designers will greatly benefit from the last three chapters of the book.

Chapter 1 discusses the general flow of a system level design and the role of compilers and synthesis tools. We discuss where VHDL fits in a modern digital design environment. A brief history of VHDL is given in this chapter.

Chapter 2 presents VHDL for those who want to get immediately started with design with VHDL. A section in this chapter covers VHDL definitions and terminologies.

Chapters 3 to 5 cover VHDL structural, concurrent, and behavioral constructs for describing RT level components.

Chapter 6 focuses on the linguistics issues of VHDL. Issues such as type definitions and overloading are covered here.

Chapter 7 discusses the signal and simulation model of VHDL. Multiple concurrent assignments to signals and scheduling values for signals are discussed here.

Chapter 8 presents VHDL descriptions for several configurable RT level cores. Several arithmetic, memory, queue, and processor cores are presented here.

Chapter 9 shows test techniques for RT level cores. It presents several testbench generation techniques and several testable designs coded in VHDL.

Chapter 10 is the last chapter of this book. This chapter shows the RT level design of a processor core. The processor core is then tested using methods discussed in Chapter 9. The last part of this chapter shows design of a filter using this processor core.

ACKNOWLEDGMENTS

Several people helped me with preparation of this manuscript. My students Ms. Elnaz Koopahi and Ms. Mahshid Sedghi wrote models for many of the examples of Chapters 8 and 9. In addition they helped with the review of the book and made many useful comments for improving the book. Ms. Sedghi was very helpful in reviewing Chapter 10, debugging the processor model and developing a testbench for it.

Students in my VHDL class helped review the material and provided feedback on the flow of the material.

As with all my other publishing works, Ms. Fatemeh Asgari helped me with the preparation of the manuscript. She worked with me on the initial planning of this work, distribution of tasks during the project, and final assembly of this book. Her planning and organization has always been a key to successful completion of such projects.

I also thank my wife, Irma Navabi, for help, encouragement, and understanding my working habits. Such an intensive work could not be done without the support of my wife and two sons, Arash and Arvand. I thank them for this and my other scientific achievements.

This page intentionally left blank

1 Digital System Design Automation with VHDL

The early schematic capture and design entry programs used for design of digital systems have given way to complex design entry programs utilizing software programs, hardware cores, and components described in hardware description languages. Today's hardware designers designing complex hardware/software systems must be familiar with software programming, hardware description languages, and utilization of embedded cores for implementation of hardware and software parts of a system. In addition, hardware designers must be familiar with design environments utilizing software programs, HDL hardware descriptions, and embedded processor and intellectual property (IP) cores.

This chapter gives an overview of the hardware design process and the role of VHDL in a modern digital design methodology. We begin by an overview of levels of abstraction from systems to transistors. This discussion becomes useful in presenting design strategies that are based on recursive partitioning of a design into lower abstraction levels. Following this, we talk about hardware/software codesign, and discuss partitioning a design into a part that is to be implemented using various hardware design abstractions, and a part that is to be implemented with a software program. The section that comes after this discussion of partitioning focuses on RT level design that is the major abstraction level that the VHDL language is used for. We will talk about RT level simulation, synthesis and device programming tools, and will discuss the existing CAD tools used in this hardware design process. The last part of this chapter gives a general overview of the properties of the VHDL language.

1

1.1 Abstraction Levels

Digital design started with putting transistors together to implement a given hardware function. Obviously this handcrafted method of design and flexibilities offered in choice of transistor size and routing of wires, achieves an optimum design for a given function.

On the other hand, as designs become more complex, this level of design had to change to allow design of large circuits. In an evolutionary process, gate level designs replaced transistor level designs. With this move to an upper abstraction level, compromise for timing, silicon utilization, and power consumptions had to be made. In addition, design tools were developed to help designers with utilization of gates verification of designs, and translation to the transistor level.

As designs became more complex, another higher abstraction level evolved that included even less detail than the gate level. The main focus of this level of abstraction is how transfer of data happens between registers, logic units, and busses; and because of this, it is referred to as register transfer level, or RTL. As in the move from transistor level to gate level, moving from gates to RT level carries with it compromises and tradeoffs. Furthermore, this higher level of abstraction requires use of tools and various software and hardware packages to aid the designer in the design process. As in the gate level, RT level tools include those for design capture, verification, and translation from RT level to the lower abstraction level, i.e., gate level synthesis.

For the same reasons that design had to go up from gate to RT level, the time of sole RT level design had to expire, and this level of abstraction had to give way to an upper level of abstraction, which for now, we refer to as Electronic System Level (ESL) or just system level. At the system level, a designer is only concerned with the functionality of the system being designed, and describes the algorithm that is going to be implemented. The algorithm is described using a procedural language like the C language. The description at this level does not contain clock or gate level timing.

System level tools include design entry tools, simulators, and, of course, hardware generation programs. Hardware generation from a system level description can be done in one of two possible ways. As in other abstraction levels, one way of generating hardware is to translate a system level description to a lower level of abstraction, i.e., RTL. Alternatively, a system level procedural description can be compiled to run on a given processor. This alternative is possible at the system level because the description is procedural and a software language like C can be used for it.

The above mentioned method of hardware generation (using a software program) from a system level description is what has be-

come embedded system design. The former method, i.e., translation from system to RTL, is often referred to as C synthesis, or system level synthesis. C synthesis refers to generation of hardware from a C program, or a procedural description. Figure 1.1 shows abstraction levels discussed here.
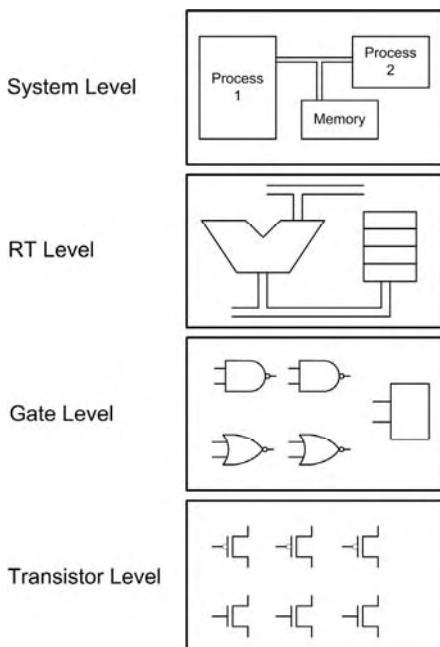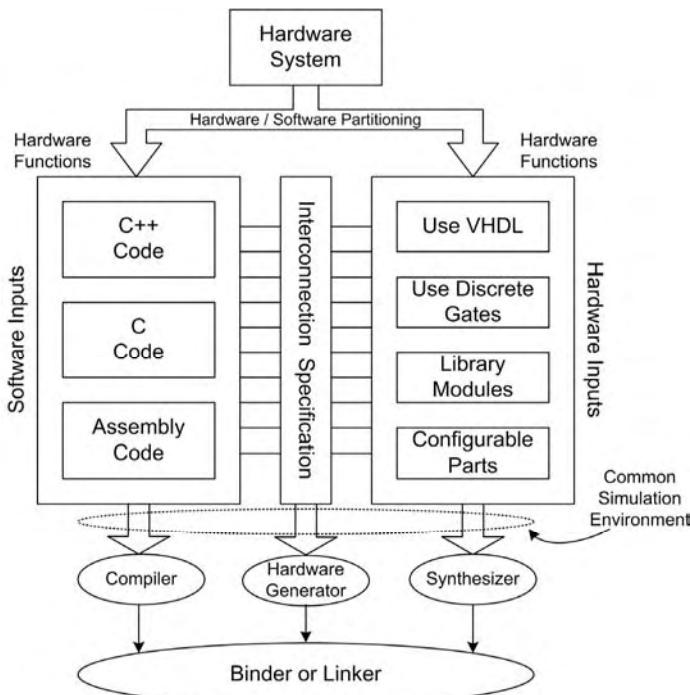
Figure 1.1 Abstraction Levels

1.2 System Level Design Flow

Figure 1.2 shows a design flow that consists of hardware and software parts. In this flow some hardware functions we implemented with RTL coding and same other functions are implemented by programs running on processor cores. The subsections below describe the details of this block diagram.

1.2.1 Hardware/Software Partitioning

The first step in design at the system level is to decide what parts are to be implemented using predefined cores, RTL VHDL modules, or discrete components, and which parts are to be implemented with a program running on a processor core. This decision is referred to as hardware/software partitioning. This is a manual (or semi-manual) process, and is perhaps the most difficult system design phase.

Figure 1.2 Hardware/Software Design Flow

The hardware part becomes a description of various hardware modules that are described in an HDL or are available as predefined hardware modules. The software part is a high level C/C++ program that after being compiled becomes the memory contents of processor that runs the program.

1.2.2 Hardware Part

The hardware part (right flow in Figure 1.2) of a complete hardware/software system may be composed of components that are described in VHDL, IP cores or discrete parts. Using tools and design environments, a hardware designer chooses to code parts of his or her design in VHDL, or use parts from a library of predefined modules.

Often a design environment provides intellectual property (IP) cores that designers can use and integrate in their designs. Hardware design environments also include configurable parts for commonly used components such as arithmetic functions, register banks, and counters.

1.2.3 Software Part

The left flow in Figure 1.2 shows the implementation of the software part of a system. The part of a design that is to be implemented in software must become a machine language program in a given processor. The designer may choose to code this part in a high level language and compile it, or directly code it in assembly or machine language.

All the necessary software tools and compilers are available to a designer who uses a supported processor core. In this case, use of C/C++ for describing the software part of a system is the most logical choice, since compilation and debugging tools are provided for the designer. On the other hand, if a designer uses his or her own processor or a processor core that does not have a strong support, the designer is responsible for generating the machine language of the program he or she is implementing.

Regardless of how the programming task is done, after the completion of the design of the software part, this part looks like any hardware block with inputs and outputs. The inputs and outputs are either external to the system being designed, or they are to interconnect the hardware and software parts.

The sections that follow elaborate on the hardware and software sides of the design processes depicted in Figure 1.2.

1.3 RTL Design Flow

For the design of a digital system using an automated design environment, the design flow begins with specification of the design at various levels of abstraction and ends with generating netlist for an ASIC (application specific integrated circuits), layout for a custom IC, or a program for a PLD (programmable logic devices). Figure 1.3 shows steps involved in this design flow.

In the design entry phase, a design is specified as a mixture of behavioral VHDL code, instantiation of VHDL modules, and bus and wire assignments. A design engineer is also responsible for generating testbenches for his or her design for verification of the design and later for verifying the synthesis output. Design verification can be done by simulation, assertion verification, formal verification, or a mix of all three. After performing this design validation phase (this is called the pre-synthesis verification), this design is taken through the synthesis process to translate it into actual hardware of a target device. Here, target device refers to the specific field programmable logic device (FPLD) that is being programmed, the ASIC that is being manufactured by an outside source, or the custom IC that is being fabricated. After the synthesis process and before the actual hard-

ware is generated, another simulation, that is referred to as post-synthesis simulation, is done. This simulation can take advantage of the same testbench generated for the VHDL model of the system before it is synthesized. This way, the behavioral model of the design and its hardware model are tested with the same data. The difference between pre- and post-synthesis simulations is in the level of details obtained from each simulation.

The sections that follow elaborate on each of the blocks shown in Figure 1.3. Most VHDL based EDA environments provide blocks shown in this figure.

1.3.1 Design Entry
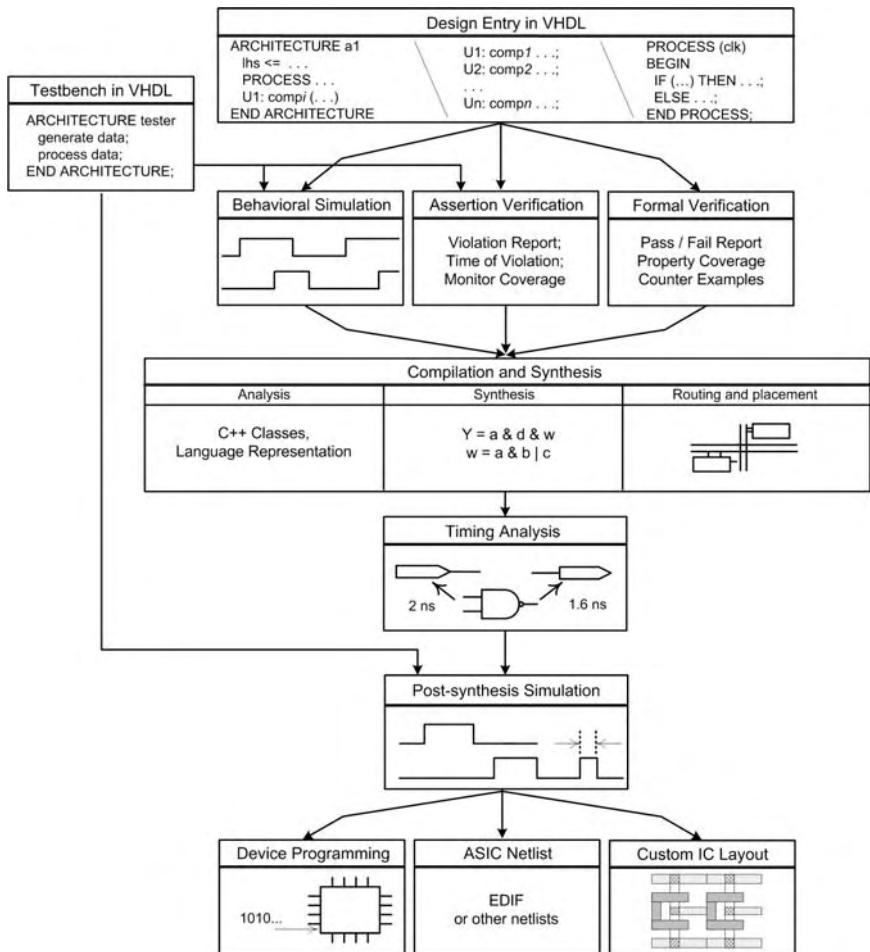
The first step in the design of a digital system is the design entry phase. In this phase, the design is described in VHDL in a top-down hierarchical fashion. A complete design may consist of components at the gate or transistor level, behavioral parts describing high level functionality of a hardware module, or components described by their bussing structure.

Because high-level VHDL designs are usually described at the level that specifies system registers and transfer of data between registers through busses, this level of system description is referred to as register transfer level. A complete design described as such has a clear hardware correspondence. VHDL constructs used in an RT level design are sequential statements, signal assignments, and instantiation statements.

VHDL *sequential statements* are used for high level behavioral descriptions. A system or a component is described in a sequential fashion similar to the way processes are described in a software language. For example, we can describe a component by checking its input conditions, setting flags, waiting for events to occur, monitoring handshaking signals, and issuing outputs. Describing a system sequentially, VHDL if-then, case and other software-language-like constructs can be used.

VHDL *signal assignments* are statements for representing logic blocks, bus assignments and bus and input/output interconnect specifications. Combined with Boolean and conditional assignments, these language constructs can be used for describing components and systems in terms of their register and bus assignments.

VHDL *instantiation statements* are for using lower level components in an upper level design. Instead of describing behavior, functionality, or bussing of a system, we can describe a system in VHDL in terms of its lower level components. These sub-components can be as small as a gate or a transistor, or as large as a complete processor.

Figure 1.3 HDL Based Design Flow

1.3.2   Testbench in VHDL

A system designed in VHDL must be simulated and tested for functionality before it is turned into hardware. In this simulation pass, design errors and incompatibility of components used in the design can be detected. Simulating a design requires generation of test data and observation of simulation results. This process can be done by use of a VHDL module that is referred to as a testbench. A VHDL testbench uses high-level constructs of this language for data generation, response monitoring, and even handshaking with the design. Inside the testbench, the design that is being simulated is instantiated. The

testbench together with the design form a simulation model used by a VHDL simulation engine.

1.3.3 Design Validation

An important task in any digital design is design validation. Design validation is the process that a designer checks his or her design for any design flaws that may have occurred in the design process. A design flaw can happen due to ambiguous problem specifications, designer errors, or incorrect use of parts in the design. Design validation can be done by simulation, assertion verification, or formal verification.

1.3.3.1 Simulation. Simulation for design validation is done before a design is synthesized. This simulation pass is also referred to as behavioral, RT-level, or pre-synthesis simulation. At the RT level a design includes clock level timing but no gate and wire delays are included. Simulation at this level is accurate to the clock level. Timing of RT level simulation is at the clock level and does not usually consider hazards, glitches, race conditions, setup and hold violations and other detailed timing issues. The advantage of this simulation is its speed compared with simulations at the gate or transistor levels.
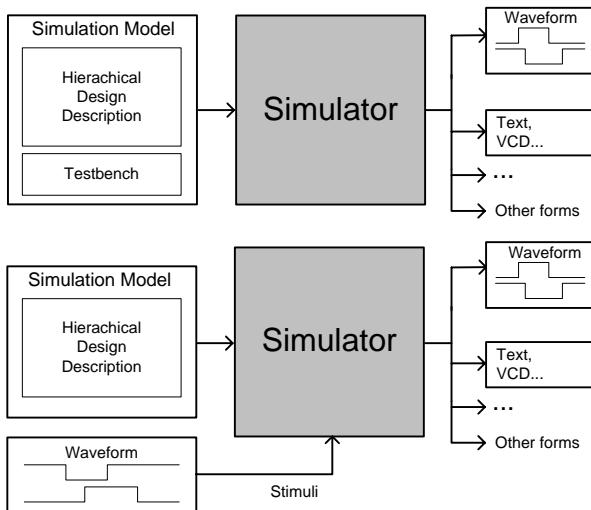
Figure 1.4 Using a Testbench or a Waveform Editor for Simulation

Simulation of a design requires test data, and usually VHDL simulation environments provide various methods for application of

this data to the design being tested. Test data can be generated graphically using waveform editors, or through a testbench. Figure 1.4 shows two alternatives for defining test input data for a simulation engine. Outputs of simulators are in the form of waveforms (for visual inspection) and text for large designs for machine processing.


```
ENTITY counter4_tester IS END ENTITY;
ARCHITECTURE timed OF counter4_tester IS
   SIGNAL r : std_logic;
   SIGNAL c : std_logic := '0';
   SIGNAL cnt : std_logic_vector (3 DOWNTO 0);
BEGIN
   UUT1: ENTITY WORK.counter4 (procedural) PORT MAP (r, c, cnt);
   r <= '0', '1' AFTER 09 NS, '0' AFTER 17 NS,
        '1' AFTER 59 NS, '0' AFTER 67 NS;
   c <= NOT c AFTER 3.5 NS WHEN NOW <= 75 NS ELSE '0';
END ARCHITECTURE timed;
```

Testbench

```
ENTITY counter4 IS
   PORT (Reset, Clk : IN std_logic;
            Count : OUT std_logic_vector (3 DOWNTO 0));
END ENTITY;
ARCHITECTURE procedural OF counter4 IS
   SIGNAL cnt_reg : std_logic_vector (3 DOWNTO 0);
BEGIN
   PROCESS (Clk) BEGIN
      IF (Clk = '0' AND Clk'EVENT) THEN
         IF (Reset = '1') THEN cnt_reg <= "0000";
         ELSE cnt_reg <= cnt_reg + 1; END IF;
      END IF;
   END PROCESS;
   Count <= cnt_reg;
END ARCHITECTURE procedural;
```

Simulator

Design to Simulate

Figure 1.5 VHDL Simulation with a Testbench

For simulating with a VHDL testbench, the testbench instantiates the design under test, and as part of the code of the testbench it applies test data to the instantiated circuit. Figure 1.5 shows VHDL code of a counter circuit, its testbench, and its simulation results in form of a waveform. As shown here, simulation validates the functionality of the counter circuit being tested. With every clock pulse the counter is incremented by 1. Note in the timing diagram it is shown that the counter output changes with the rising edge of the clock and no gate delays and propagation delays are shown here. Simulation results show a correct functionality of the counter regardless of the clock frequency.

Obviously, an actual hardware component behaves differently. Based on the timing and delays of the parts used, there will be a non-zero delay between the active edge of the clock and the counter output. Furthermore, if the clock frequency applied to an actual part is too fast for propagation of values within the gates and transistors of a design, the output of the design becomes unpredictable.

The simulation shown here is not provided with the details of the timing of the hardware being simulated. Therefore, potential timing

problems of the hardware that are due to gate delays cannot be detected. This is typical of a pre-synthesis or high-level behavioral simulation. What is being verified in Figure 1.5 is that our counter counts binary numbers. How fast the circuit works and what clock frequency it requires can only be verified after the design is synthesized.
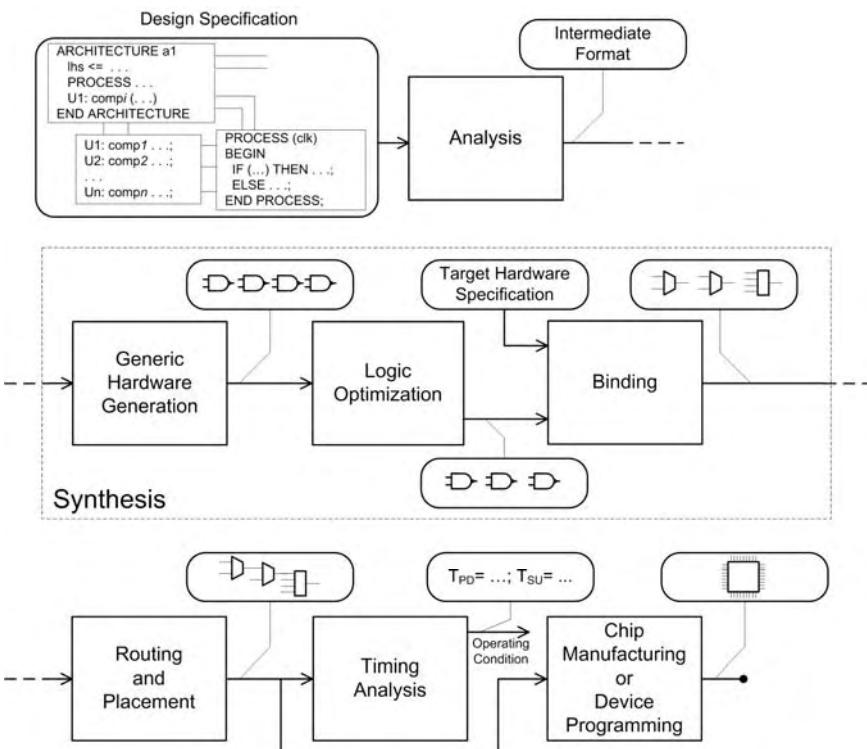
1.3.3.2 Assertion Verification. Instead of having to inspect simulation results manually or by developing sophisticated testbenches, assertion monitors can be used to continuously check for design properties while the design is being simulated. Assertion monitors are put in the design being simulated by the designer. The designer decides that if the design functions correctly, certain conditions have to be met. These conditions are regarded as design properties, and assertion monitors are developed by designer to assert that these properties are not violated. An assertion monitor fires if a design property put in by the designer is violated. This alerts the designer that the design is not functioning according to the designer's expectation. OVL (open verification library) provides a set of assertion monitors for monitoring common design properties. Designers can use their own assertions and use them in conjunction with their testbenches.

1.3.3.3 Formal Verification. Formal verification is the process of checking a design against certain properties. When a design is completed, the designer develops a set of properties reflecting correct behavior of his or her design. Without running simulation, and without requiring any simulation data, a formal verification tool examines the design to make sure that the described properties hold under all conditions. If a situation is found that the property will not hold, the property is said to have been violated. Input conditions that make a property to fail are regarded as the property's counter examples. Property coverage indicates how much of the complete design is exercised by the property.

1.3.4 Compilation and Synthesis

Synthesis is the process of automatic hardware generation from a design description that has an unambiguous hardware correspondence. A VHDL description for synthesis cannot include signal and gate level timing specifications, file handling, and other language constructs that do not translate to sequential or combinational logic equations. Furthermore, VHDL descriptions for synthesis must follow certain styles of coding for combinational and sequential circuits. These styles and their corresponding VHDL constructs are defined under VHDL for RTL synthesis.

In the design process, after a design is successfully entered and its pre-synthesis simulation results have been verified by the designer, it must be compiled to make it one step closer to an actual hardware on silicon. This design phase requires specification of the hardware that the design is to be realized in. For example, we have to specify a specific ASIC, or an FPGA part as our "target hardware". When the target hardware is specified, technology files of that hardware (ASIC, FPGA, or Custom IC) with detailed timing and functional specification become available to the compilation process. The compilation process, translates various parts of the design to an intermediate format (analysis phase), links all parts together, generates the corresponding logic (synthesis phase), places and routes components of the target hardware and generates timing details.

Figure 1.6 Compilation and Synthesis Process

Figure 1.6 shows the compilation process and a graphical representation for each of the compilation phase outputs. As shown, the input of this phase is a hardware description that consists of various levels of VHDL, and its output is a detailed hardware for program-

ming an FPLD (Field Programmable Logic Devices) or manufacturing an ASIC.

1.3.4.1 Analysis. A complete design that is described in VHDL may consist of behavioral VHDL, bus and interconnection specifications, and wiring of other VHDL components. Before the complete design is turned into hardware, the design must be analyzed and a uniform format must be generated for all parts of the design. This phase also checks the syntax and semantics of the input VHDL code.

1.3.4.2 Generic Hardware Generation. After obtaining a uniform presentation for all components of a design, the synthesis pass begins its operation by turning the design into a generic hardware format, such as a set of Boolean expressions or a netlist of basic gates.

1.3.4.3 Logic Optimization. The next phase of synthesis, after a design has been converted to a set of Boolean expressions, is the logic optimization phase. This phase is responsible for reducing expressions with constant input, removing redundant logic expressions, two level minimization, and multi-level minimization that includes logic sharing.

   This is a very computationally intensive process, and some tools allow users to decide on the level of optimization. Output of this phase is in form of Boolean expressions, tabular logic representations, or primitive gate netlists.

1.3.4.4 Binding. After logic optimization, the synthesis process uses information from target hardware to decide exactly what logic elements and cells are needed for the realization of the circuit that is being designed. This process is called binding and its output is specific to the FPLD, ASIC, or Custom IC being used.

1.3.4.5 Routing and Placement. The routing and placement phase decides on the placement of cells of the target hardware. Wiring inputs and outputs of these cells through wiring channels and switching areas of the target hardware are determined by the routing and placement phase. The output of this phase is specific to the hardware being used and can be used for programming an FPLD or manufacturing an ASIC.

   An example of a synthesis run is shown in Figure 1.7. In this figure, the counter circuit used in the simulation run of Figure 1.5 is being synthesized. In addition to the VHDL description of the design, the synthesis tool shown requires specification of the target hardware to synthesize to. The output of the synthesis tool is a list of gates and flip-flops available in the target hardware, and their interconnections.

A graphical representation of this output that is automatically gener-ated by the synthesis tool of Altera's Quartus II is shown in Figure 1.7. What is shown here is referred to as the technology map view that shows FPGA cells that are utilized for the implementation of the complete circuit.
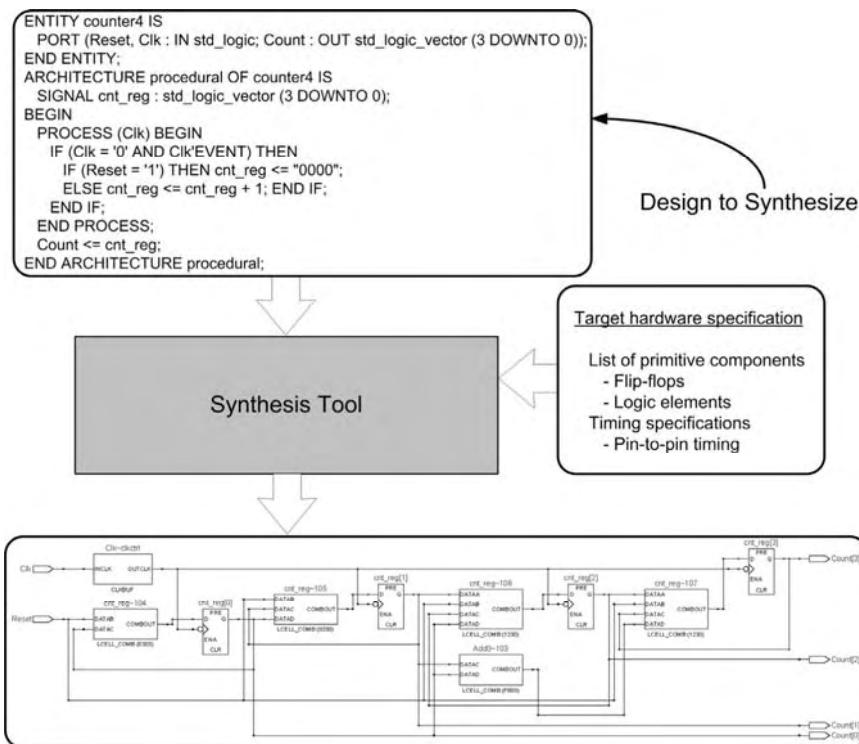
Figure 1.7 An Example Synthesis Run

1.3.5 Timing Analysis

As shown in Figure 1.3, as part of the compilation process, or in some tools after the compilation process, there is a timing analysis phase. This phase generates worst-case delays, clocking speed, delays from one gate to another, as well as required setup and hold times. Results of timing analysis appear in tables and / or graphs. Designers use this information to decide on their clocking speed and, in general, speed of their circuits.

1.3.6 Post-Synthesis Simulation

After synthesis is done, the synthesis tool generates a complete net-list of target hardware components and their timings. The details of gates used for the implementation of the design are described in this netlist. The netlist also includes wiring delays and load effects on gates used in the post-synthesis design. The netlist output is made available in various netlist formats including VHDL. Such a descrip-tion can be simulated and its simulation is referred to post-synthesis simulation. Timing issues, determination of a proper clock frequency and race and hazard considerations can only be checked by a post-synthesis simulation run after a design is synthesized. As shown in Figure 1.3 the same testbench testing the original VHDL design be-fore synthesis can be used for post-synthesis simulation.

Due to delays of wires and gates, it is possible that the behavior of a design as intended by the designer and its behavior after post-synthesis simulation are different. In this case, the designer must modify his or her design and try to avoid close timings and race situa-tions.

1.3.7 Hardware Generation

The last stage in an automated VHDL based design is hardware gen-eration. This stage generates a netlist for ASIC manufacturing, a program for programming FPLDs, or layout of custom IC cells.

1.4 VHDL

The previous section showed steps involved in taking an RT level de-sign from a VHDL description to hardware implementation. This de-sign process is only possible because VHDL is a language that can be understood by system designers, RT level designers, test engineers, simulators, synthesis tools, and machines. Because of this important role in design, VHDL has become an IEEE standard. The standard is used by users as well as tool developers.

1.4.1 VHDL Initiation

In the search for a standard design and documentation tool for the VHSIC (Very High Speed Integrated Circuits) program, the United States Department of Defense (DoD) in the summer of 1981 spon-sored a workshop on hardware description languages at Woods Hole, Massachusetts. This workshop was arranged by the Institute for De-fense Analysis (IDA) to study various hardware description methods, the need for a standard language, and the features that might be re-

quired by such a standard. Because the VHSIC program was under the restrictions of the United States International Traffic and Arms Regulations (ITAR), the VHDL component of this program was also initially subject to such restrictions.

In 1983, DoD established requirements for a standard VHSIC Hardware Description Language (VHDL), based on the recommendations of the "Woods Hole" workshop. A contract for the development of the VHDL language, its environment, and its software was awarded to IBM, Texas Instruments, and Intermetrics corporations. Work on VHDL started in the Summer of 1983. At that time language specifications were no longer under ITAR restrictions, but these restrictions still applied to government developed software.

VHDL 2.0 was released only 6 months after the project began. This version, however, allowed only concurrent statements, and lacked the capability to describe hardware in a sequential software-like fashion, a shortcoming that would seriously jeopardize the applicability of the language for high level behavioral descriptions. The language was significantly improved, as this and other shortcomings were corrected when VHDL 6.0 was released in December of 1984. Development of VHDL-based tools also began in 1984.

In 1985, ITAR restrictions were lifted from VHDL and its related software, and the VHDL 7.2 Language Reference Manual (LRM) copyright was transferred to IEEE for further development and standardization. This led to the development of the IEEE 1076/A VHDL Language Reference Manual (LRM), which was released in May of 1987. Later that year version B of the LRM was developed and approved by REVCOM (a committee of the IEEE Standards Board). VHDL 1076-1987 formally became the IEEE standard hardware description language in December of 1987.

Efforts for defining a new version of VHDL started in 1990 by a team of volunteers working under the IEEE DASC (Design Automation Standards Committee). In October of 1992 a new VHDL referred to as VHDL'93 was completed and was released for review. After minor modifications, this new version was approved by the VHDL balloting group members and became the new VHDL language standard. The present VHDL standard is formally referred to as VHDL 1076-1993. Standardization work for this version did not complete until the middle of 1994.

1.4.2 Existing Languages

Early in the VHSIC program in 1981 it was found that none of the existing hardware description languages could be used as a standard tool for the design, manufacturing, and documentation of digital circuits ranging from integrated circuits to complete systems. Part of the

study for the development of the requirements of a VHSIC language, however, concentrated on the capabilities, shortcomings, and other characteristics of eight hardware description languages that were available at that time. These languages were AHPL, CDL, CONLAN, IDL, ISPS, TEGAS, TI-HDL, and ZEUS. We briefly describe the important features of these languages in order to provide a framework for understanding the VHDL requirements that are discussed in the next section.

1.4.2.1   AHPL.   AHPL (a hardware programming language) is an HDL for describing hardware at the dataflow level of abstraction. This language uses an implicit clock for synchronizing assignments of data to registers and flip-flops, but does not provide support for describing asynchronous circuits. The language descriptions consist of interacting concurrent modules, and hierarchy of modules is not supported. Data types in AHPL are fixed and restricted to bits, vectors of bits, and arrays of bits. Procedures or functions are only allowed in the context of combinational logic units. Delay and constraint specifications are not allowed in AHPL and assignment of values to buses and registers all occur at the same time without delay, since they are synchronized with an implicit clock.

1.4.2.2   CDL.   CDL (computer design language) is a hardware description language developed in an academic environment mainly for instruction in digital systems. This language is strictly a dataflow language, and does not support design hierarchy. In CDL, micro-statements are used for transfer of data into registers. Conditional micro-statements use if-then-else constructs and can be nested.

1.4.2.3   CONLAN.   The CONLAN (CONsensus LANguage) project began as an attempt to establish a standard hardware description language. This platform consists of a family of languages for describing hardware at various levels of abstraction. Base CONLAN (bcl), for example, is the base language for all member languages. All operations in CONLAN are executed concurrently. CONLAN allows hierarchical description of hardware but has limited external use.

1.4.2.4   IDL.   IDL (interactive design language) is an internal IBM language with limited outside use. IDL was originally designed for automatic generation of PLA structures, but it was later extended to cover more general circuit descriptions. Hardware in IDL can be described in a hierarchy of structures. This language is primarily a concurrent hardware description language.

1.4.2.5   ISPS. ISPS (instruction set processor specification) is a very high level behavioral language and was mainly designed to create an environment for designing software based on a given hardware. Although the language is primarily targeted for CPU-like architectures, other digital systems can easily be described in it. Timing control in ISPS is limited. The "NEXT" construct allows timing control between statements of behavioral descriptions, but it is not possible to specify gate level timing and structural details.

1.4.2.6   TEGAS. TEGAS (TEst Generation And Simulation) is a system for test generation and simulation of digital circuits. Although several extended versions of this language have behavioral features, the main language (TEGAS Description Language or TDL) is only structural. Digital hardware can be described hierarchically in this language. Detailed timing specification can be specified in TDL.

1.4.2.7   TI-HDL. TI-HDL (Texas Instruments Hardware Description Language) is a multi-level language for the design and description of hardware. It allows hierarchical specification of hardware and supports description of synchronous, asynchronous, and combinatorial logic circuits. Behavioral descriptions in TI-HDL are sequential and software-like, and use if-then-else, case, for, and while constructs for program flow control. This language has fixed data types with no provision for adding user defined types.

1.4.2.8   ZEUS. The ZEUS hardware description language is a non-procedural language that was created at General Electric Corporation. This language supports design hierarchy and allows definition of systems by their functionality or their structural arrangements. Timing in ZEUS is at the clock level and there are no provisions for gate delay specification or detailed timing constraints. Because of this timing arrangement, asynchronous circuits cannot be described in ZEUS. This language provides a close link to physical layout.

1.4.3 VHDL Requirements

A DoD document entitled "Department of Defense Requirements for Hardware Description Languages", released in January of 1983, clearly stated the requirements for the VHSIC hardware description language. The present VHDL satisfies the requirements set forth in this detailed document. This section briefly describes the main features of VHDL requirements.

1.4.3.1   General Features. The DoD requirement document specifies that the VHSIC hardware description language should be a language

for design and description of hardware. It indicates that VHDL should be usable for design documentation, high-level design, simulation, synthesis, and testing of hardware, as well as a driver for a physical design tool.

It emphasizes that VHDL is for the description of hardware from system to gate, and it clearly specifies that system software is not an issue and that physical design does not need to be addressed. Since in an actual digital system, all small or large elements of the system are active simultaneously and perform their tasks concurrently, the concurrency aspect of VHDL is heavily emphasized. In a hardware description language, concurrency means that transfer statements, descriptions of components, and instantiations of gates or logical units are all executed such that in the end they appear to have been executed simultaneously.

1.4.3.2 Support for Design Hierarchy. The DoD requirement document specified the need for hierarchical specification of hardware in VHDL. This feature is essential for a multi-level hardware language. A design consists of an interface description and a separate part for describing its operation. Several descriptions may exist for describing the operation of a design, all corresponding to the same interface description. The operation of a system can be specified based on its functionality, or it can be specified structurally in terms of its smaller sub-components. Structural description of a component can be accomplished at all design levels. At the lowest levels, components are described by their functionality and use no sub-components.

1.4.3.3 Library Support. For design management, the need for libraries is specified for VHDL. User defined and system defined primitives and descriptions reside in the library system. The language should provide mechanism for accessing various libraries. A library can contain an interface description of a design. At the same time, several specifications of the operation of this design can simultaneously reside in this library.

Descriptions and models that are correct should be placed in the library after the language compiler has compiled them. In addition, libraries should be accessible to different designers.

1.4.3.4 Sequential Statement. Although the strong features of a hardware description language should be its support for concurrent execution of processes and statements, the VHDL language requirements also specified the need for software-like sequential control. When a hardware designer partitions a system into concurrent components or subsections, the designer should then be able to describe

the internal operational details by sequential programming language constructs such as case, if-then-else, and loop statements.

Sequential statements provide an easy method for modeling hardware components based on their functionality. Sequential or procedural capability is only for convenience and the overall structure of the VHDL language remains highly concurrent.

1.4.3.5 Generic Design. In addition to inputs and outputs of a hardware component, other conditions may influence the way it operates. These include the environment where the hardware component is used, and the physical characteristics of the hardware component itself. It should not be necessary to generate a new hardware description for every specific condition. Furthermore, many hardware components in various logic families or FPGAs are functionally equivalent, and differ only in their timing and loading characteristics. For example, logic elements in Altera's Cyclone, Cyclone II, and Stratix are very similar in cell structure but have different timing properties.

A good hardware description language should allow the designer to configure the generic description of a component when it is used in a design. Generic descriptions should be configurable for size, physical characteristics, timing, loading, and environmental conditions. The ability to describe generic models of hardware was a DoD requirement for the VHDL language.

1.4.3.6 Type Declaration and Usage. A language for the description of hardware at various levels of abstractions should not be limited to Bit or Boolean types. VHDL requirements specified that the language ought to allow integer, floating point, and enumerate types, as well as user defined types. Types defined by the system or by the user should be placed in the library of the language environment and their use should be transparent to the user.

The language should provide the capability to redefine language operators for types that are defined by the user. For example, the language provides Boolean operators such as AND, OR, and NOT for the predefined logic values. A user needing a multi-level logic should be able to redefine these operators for the newly defined multi-level logic type.

In addition, a hardware description language should allow array type declarations and composite type definitions, such as structures or records in programming languages. The DoD document also specified a strongly typed language and strong type checking.

1.4.3.7 Use of Subprograms. The ability to define and use functions and procedures was another VHDL requirement. Subprograms can be used for explicit type conversions, logic unit definitions, operator re-

definitions, new operation definitions, and other applications commonly used in programming languages.

1.4.3.8   Timing Control.   The ability to specify timing at all levels is another requirement for the VHDL language. VHDL should allow the designer to schedule values to signals and delay the actual assignment of values until a later time. For handshaking and gate or line delay modeling in the sequential descriptions, it should be possible to wait for the occurrence of an event or for a specific time duration.

The language should be general and should allow any number of explicitly defined clock signals. The clocking scheme should be completely up to the user, since the language does not have an implicit clocking scheme or signal.

Constructs for edge detection, delay specification, setup and hold time specification, pulse width checking, and setting various time constraints should be provided.

1.4.3.9   Structural Specification.   The DoD requirements for a standard hardware description language specified that the language should have constructs for specifying structural decomposition of hardware at all levels. It also should be possible to describe a generic one-bit design and use it when describing multi-bit regular structures in one or more dimensions. This requires constructs for iteration in the description of structures.

1.4.4   The VHDL Language

In its present form, VHDL satisfies all requirements of the 1983 DoD requirements document. The experience, researchers, software developers, and other users with VHDL since it became the IEEE standard in 1987 indicates this language is sufficiently rich for designing and describing today's digital systems.

As originally required, VHDL is a hardware description language with strong emphasis on concurrency. The language supports hierarchical description of hardware from system to gate or even switch level. VHDL has strong support at all levels for timing specification and violation detection. As expected, VHDL provides constructs for generic design specification and configuration.

A VHDL design entity is defined as an entity declaration and its associated architecture body. The entity declaration specifies its interface and is used by architecture bodies of design entities at upper levels of hierarchy. The architecture body describes the operation of a design entity by specifying its interconnection with other design entities, by its behavior, or by a mixture of both. The VHDL language groups subprograms or design entities by use of packages. For cus-

tomizing generic descriptions of design entities, configurations are used. VHDL also supports libraries and contains constructs for accessing packages, design entities, or configurations from various libraries.

Vendor specific VHDL libraries are used for time specification of various FPGA and ASIC libraries. Other libraries have packages that are specific for a certain design style promoted by an EDA manufacturers. The standard IEEE library is a library of packages for type definitions, and logical operations.

A typical VHDL design environment has an analyzer program that translates a VHDL description into an intermediate form and places it in a design library. The analyzer is responsible for lexical analysis and syntax check. Operations on the design library, such as creating new libraries, deleting the old, or deletion of packages or design entities from a library are done through the design environment. VHDL-based tools use the intermediate format from the design library. One well developed tool is the VHDL simulator, which simulates a design entity from the design library and produces a simulation report. A hardware synthesizer, test vector generator, and a physical design tool are examples of other VHDL-based tools.

1.5 Summary

This chapter gave an overview of mechanisms, tools, and processes used for taking a system level design description from the design stage to a hardware implementation. This overview contained information that will become clearer in the chapters that follow. This chapter also provided the reader with the history of VHDL evolution. With this standard HDL, the efforts of tool developers, researchers, and software vendors have become more focused, resulting in better tools and more uniform environments. The next chapter presents an overview of VHDL.

Problems

1.1   Study Altera's FPGA design environment and see their simulation and synthesis environments. How do you compare Altera's environment with the simulation and synthesis environments discussed in this chapter?

1.2   Search for several commercial Formal Verification and generate a report of their input formats, capabilities, and their verification utilities.

1.3   Study Accellera's OVL library and discuss how this library helps the design automation process.

1.4   Study SystemC and discuss tools available for this language.

1.5   Look up TLM and discuss this design abstraction level.

1.6   Study the Verilog hardware description language and discuss tools available for this language.

Suggested Reading

Accellera, Open Verification Library: Assertion Monitor Reference Manual, www.accellera.org, v1.0, 2005.

Bening, Lionel and Harry D. Foster, *Principles of Verifiable RTL Design Second Edition - A Functional Coding Style Supporting Verification Processes in Verilog*, Springer, 2nd edition, 2001, ISBN: 0792373685.

Brown, Stephen, and Zvonko Vranesic, *Fundamentals of Digital Logic with VHDL Design, 1st edition*, 1999, McGraw-Hill Science/Engineering/Math, ISBN: 978-0072355963.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Navabi, Zainalabedin, *Digital Design and Implementation with Field Programmable Devices*, Kluwer Academic Publishers, 2005, ISBN: 1-4020-8011-5.

Navabi, Zainalabedin, *Embedded Core Design with FPGAs*, 2006, McGraw Hill-Professional, ISBN: 0071474811.

Perry, Douglas L., and Harry Foster, *Applied Formal Verification for Digital Circuit Design*, 2005, McGraw-Hill Professional, ISBN: 978-0071443722.

2 RTL Design with VHDL

The level of hardware description that hardware description languages are most used for is the register transfer level (RTL). Between gate level on the low abstraction side, and system level on the high abstraction side, the RT level of abstraction is a good balance between correspondence to actual hardware and ease of description for hardware designers. At this level of abstraction, designs can be simulated with HDL simulators, they are synthesizable, and automatic generation of hardware is provided by most hardware design EDA tools.
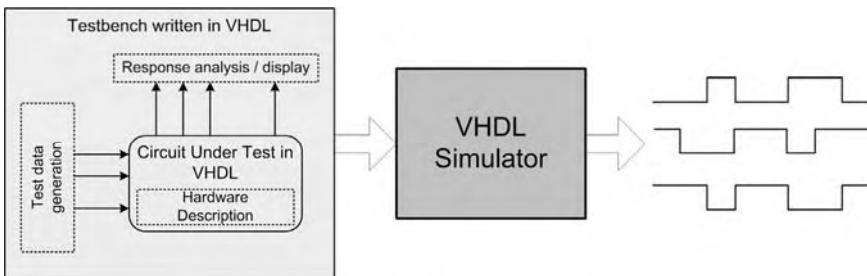
This chapter presents VHDL at the RT level. We discuss how a design is described in VHDL for simulation and synthesis. For this purpose, only a subset of VHDL is needed and many complex language structures that are used in cell modeling and higher level non-synthesizable designs are not covered here. In order to utilize this language in a design and test environment, certain language structures that do not necessarily correspond to specific hardware structures, but are used for testing RT level designs, are also described.

The chapter begins with a discussion of the main structures of VHDL. After this introductory presentation, we will start covering various constructs of the language using simple examples. The examples progressively become more complex and more constructs of the language are covered. After we present a sufficient set of constructs for design of hardware, we will turn our attention to developing testbenches for testing designs in VHDL. Several typical testbenches for the designs presented in the earlier parts of this chapter will be discussed in this part. The last part of this chapter covers several VHDL

linguistic issues and terminologies. This basic understanding of the linguistics of VHDL is necessary for a better understanding of the language that will be presented in the chapters that follow.

2.1 Basic Structures of VHDL

An entity-architecture pair forms the basic structure of VHDL with which all hardware components and testbenches are described. Language constructs, according to VHDL syntax and semantics form the inside of these language structures. Language constructs are designed to facilitate description of hardware components for simulation, synthesis, and specification of testbenches to specify test data and monitor circuit responses. Figure 2.1 shows a simulation model that consists of a design and its testbench in VHDL. VHDL constructs (shown by dotted lines) of the VHDL model being tested are responsible for description of its hardware, while language constructs used in a testbench are responsible for providing input data to the module being tested and analysis or display of its outputs. Simulation output is generated in form of a waveform for visual inspection or data files for machine readability.
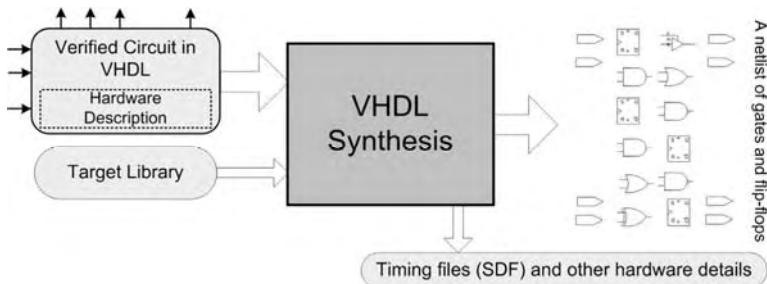
Figure 2.1 Simulation in VHDL
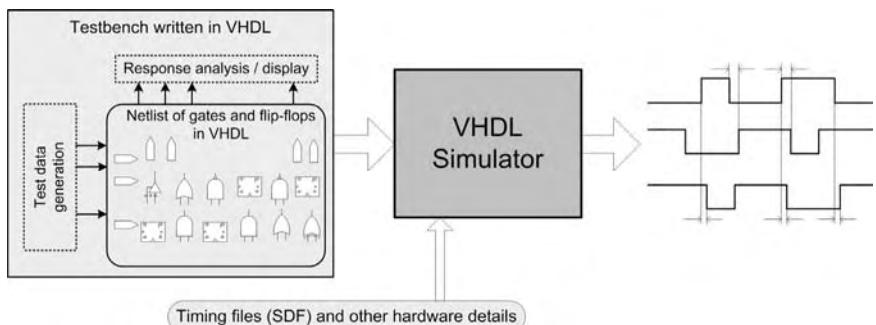
After a design passes basic functional validations, it must be synthesized into a netlist of components of a target library. The target library is the specification of the hardware that the design is being synthesized to. VHDL constructs used in the VHDL description of a design for its verification, or those for timing checks and timing specifications are not synthesizable. A VHDL design that is to be synthesized must use language constructs that have a clear hardware correspondence.

Figure 2.2 shows a block diagram specifying the synthesis process. Circuit being synthesized and specification of the target library are the inputs of a synthesis tool. The outputs of synthesis are a netlist of components of the target library, and timing specification and other physical details of the synthesized design.

Often synthesis tools have an option to generate this netlist in VHDL. In this case (Figure 2.3), the same testbench prepared for the pre-synthesis simulation can be used with the netlist generated by the synthesis tool. This simulation, which is often regarded as post-synthesis simulation, uses timing information generated by the synthesis tool and yields simulation results with detailed timing.

Figure 2.2 Synthesis of a VHDL Design

Figure 2.3 Post-synthesis Simulation in VHDL

2.1.1   Entities and Architectures

Entities used in VHDL for description of hardware components are defined as a pair of *entity* and *architecture* declaration. The interface of the circuit is specified by its entity, while its operation is described by architecture bodies associated with that entity. Allowing multiple architectures associated with an entity facilitates having configurable designs for a given interface. We use the term module to refer to an entity-architecture pair.

As shown in Figure 2.4 the interface specification of a circuit begins with the ENTITY keyword and is followed by the name of the entity together with IS. Entity declaration contains a list of the component's input-output ports and their types. On the other hand, the architecture specification begins with the ARCHITECTURE keyword

and describes the functionality of a defined entity. The functionality of the component is described using gate instantiations, signal assignments and processes in the architecture body. The architecture body consists of two parts; the declarative part and the statement part. The declarative part is the section before the BEGIN keyword, while the statement part is enclosed between BEGIN and END of the architecture. The repeat of architecture or entity names after the END keyword is optional in both entity and architecture specifications.
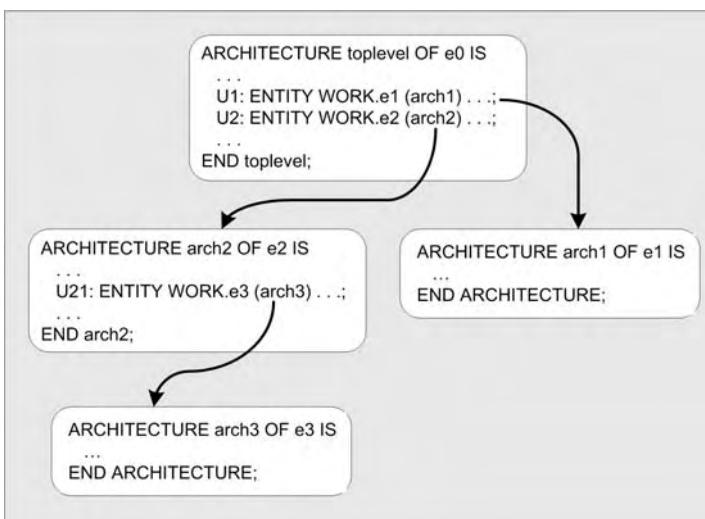
```
ENTITY entity_name IS
   input and output ports
END ENTITY entity_name;

ARCHITECTURE identifier OF entity_name IS
   declarative part
BEGIN
   statement part
END ARCHITECTURE identifier;
```

Figure 2.4 Entity Architecture Pair

A design may be described in a hierarchy of other modules. The top-level module is the complete design, and modules lower in the hierarchy are the design's components. Component instantiation is the construct used for bringing a lower level entity-architecture pair (module) into a higher level one. Figure 2.5 shows a hierarchy of several nested modules.

Figure 2.5 Hierarchy of Design Components

2.1.2 Entity-Architecture Outline

As discussed, ports of a module are specified in its entity declaration, and its operation is described in its architecture (Figure 2.6). Port declarations specify the mode of a port (i.e., input, output, etc.) and its size. Ports of an entity are visible to all architectures that are associated with it. The architecture description has a declarative part and a statement part. Signals local to a specific architecture are declared in the declarative part of an architecture. The statement part of the architecture consists of statements that are considered concurrent. These interacting statements form the description of the behavior of the module.

```
ENTITY entity1 IS PORT (i1, i2 : IN BIT; w1 : OUT BIT);
END ENTITY entity1;

ARCHITECTURE simple1 OF entity1 IS
    SIGNAL s1 : BIT;
BEGIN
    statement1;
    statement2;
    statement3;
END ARCHITECTURE simple1;
```

Figure 2.6 Entity Outline

The operation of an entity can be described in several ways. Architecture *simple_1a* in Figure 2.7 describes a circuit at the gate level and in terms of lower level components that have been defined before. This is done by component instantiation statements. Architecture *simple_1b* shows signal assignments, while architecture *simple_1c* uses a process statement to describe the functionality of the design. A process is used for behavioral descriptions of the design. A process is recognized with the PROCESS keyword and includes a sequence of statements. The execution of a process is triggered by events. These events are either listed in a sensitivity list enclosed in a set of parenthesis; alternatively wait statements are used to control the process execution. However we will be using the former throughout this chapter. The *simple_1c* architecture uses conditional if-statements to generate the proper functions of the circuit outputs.

The subsections that follow describe details of entity ports and description styles. In the examples in this chapter, and in the book, we use uppercase letters for VHDL keywords and reserved words. VHDL is not case sensitive. It allows letters, numbers and special character "_" to be used for names. Names are used for VHDL structures such as entities, architectures, ports, generic parameters, variables, signals, and instance of components.

```
ENTITY simple IS PORT (i1, i2, i3 : IN BIT; w1, w2 : OUT BIT); END ENTITY simple;
```

```
ARCHITECTURE simple_1a OF simple IS
   SIGNAL c1 : BIT;
BEGIN
   U1: ENTITY WORK.nor2 PORT MAP (i1, i2, c1);
   U2: ENTITY WORK.and2 PORT MAP (c1, i3, w1);
   U3: ENTITY WORK.xor2 PORT MAP (c1, i3, w2);
END ARCHITECTURE simple_1a;
```

```
ARCHITECTURE simple_1b OF simple IS
BEGIN
   w1 <= (i1 NOR i2) AND i3;
   w2 <= (i1 NOR i2) XOR i3;
END ARCHITECTURE simple_1b;
```

```
ARCHITECTURE simple_1c OF simple IS
BEGIN
   PROCESS (i1, i2, i3)
      VARIABLE c1 : BIT;
   BEGIN
      c1 := i1 NOR i2;
      IF (c1 = '1') THEN w1 <= i3;  ELSE w1 <= '0';
      IF (c1 = i3 ) THEN w2 <= '0'; ELSE w2 <= '1';
   END PROCESS;
END ARCHITECTURE simple_1c;
```

Figure 2.7 Architecture Definition Alternatives

2.1.3 Entity Ports

In the PORT keyword in the entity declaration is a set of parenthesis with a list of entity ports. This list includes inputs, outputs and bi-directional input/output lines. Ports may be listed in any order. This ordering can only become significant when the entity is instantiated, and does not affect the way its operation is described. Top-level entities used for testbenches have no ports.

Along with input and output names, in the set of parenthesis, sizes and types of ports may also be specified. A port may be IN for input, OUT for output, INOUT for bidirectional input-output ports, or BUFFER for buffered outputs. A BUFFER can be read in an architecture that is associated with the entity that declares it; an OUT cannot.

Range specification of a multi-bit port comes in a set of parenthesis as a pair of ascending or descending numbers. The keyword TO is used for ascending range and DOWNTO for descending. Figure 2.8 shows an example circuit with scalar, vectored, IN, OUT and INOUT ports. Ports named a, and b are one-bit inputs, and port c is a one-bit input/output. Ports av and bv are 8-bit inputs of $aCircuit$. Vectors are declared by the vector type (i.e., BIT_VECTOR) and a set of parenthesis that contain the range and direction of the indices. Another input/output is port cv that is an 8-bit vector. Port w of $aCircuit$ is declared as a one-bit output, and wv is an 8-bit output of this module.

```
ENTITY aCircuit IS
   PORT (a, b : IN BIT;
         c : INOUT BIT;
         av, bv : IN BIT_VECTOR (7 DOWNTO 0);
         cv : INOUT BIT_VECTOR (7 DOWNTO 0);
         w : OUT BIT;
         wv : OUT BIT_VECTOR (7 DOWNTO 0));
END ENTITY aCircuit;
```

Figure 2.8 Entity Declaration and Ports

As shown in Figure 2.8, each port also requires a type specifica-tion. Type BIT is a predefined VHDL type which has been used in this example. We will be using this type for now but keep in mind that other standard and user-defined types are also allowed. BIT_VECTOR is the vector version of the predefined BIT type.

2.1.4 Signals and Variables

In addition to ports, signals can be declared as data carriers in an architecture (Figure 2.9). Signal declarations take place in the archi-tecture declarative part between the header and the BEGIN keyword. Similar to signals, variables are used as intermediate carriers with the difference of only being accessible in the process bodies. Therefore in addition to ports, signals are the only data objects that can carry data between processes and other concurrent components.

```
ARCHITECTURE two_processes OF aCircuit IS
   SIGNAL d : BIT;
   SIGNAL dv : BIT_VECTOR (7 DOWNTO 0);
BEGIN
   p1: PROCESS (a, b, cv)
      VARIABLE e : BIT;
      VARIABLE ev : BIT_VECTOR (7 DOWNTO 0);
   BEGIN
      -- Can see all of aCircuit, plus d, dv, e, and ev.
      . . .
   END PROCESS;
   p2: PROCESS (av, bv, c)
      VARIABLE f : BIT;
      VARIABLE fv : BIT_VECTOR (7 DOWNTO 0);
   BEGIN
      -- Can see all of aCircuit, plus d, dv, f, and fv.
      . . .
   END PROCESS;
END ARCHITECTURE two_processes;
```

Figure 2.9 Signal and Variable Declaration

Signals are declared using the SIGNAL keyword. They are used for interconnections and have properties of actual signals in a hardware component. Variable declarations use the VARIABLE keyword. They are used for behavioral descriptions and are similar to variables in software languages. Figure 2.9 shows several signal and variable declarations. This architecture is associated with the entity declaration of Figure 2.8 and all ports of this entity are visible here.

Signals represent simple interconnection wires, busses, and simple gate or complex logical expression outputs. Signals can be used in scalar or vector form. Multiple concurrent assignments to a signal that is resolved are allowed and the value that the signal receives is the resolution of all concurrent assignments to the signal. Resolutions will be discussed in the next section. Assignments to signals use the <= symbol on the right hand side of the target signal. Figure 2.10 shows several examples of signals used on the right and left hand sides of concurrent signal assignments. The architecture shown here is associated with the entity of Figure 2.8.

```
ARCHITECTURE four_assignments OF aCircuit IS
   SIGNAL d : BIT;
   SIGNAL iv, jv, kv : BIT_VECTOR (7 DOWNTO 0);
BEGIN
   iv <= av AND cv;
   jv <= bv AND cv;
   kv <= av NOR bv;
   wv <= iv XOR jv WHEN c = '1' ELSE iv NAND kv;
END ARCHITECTURE four_assignments;
```

Figure 2.10 Using Signals

In contrast to signals, variables do not represent actual wires and are primarily used as variables are used in software languages. In VHDL, we use variables for temporary variables, intermediate values, and storage of data. A variable can only be used in a sequential body of VHDL. Since variables are only visible in the processes that they are declared in, multiple concurrent assignments to a variable is not possible. Assignments to variables use the := symbol on the right hand side of the target variable. Figure 2.11 shows several examples of variables used in a VHDL architecture. VHDL also supports shared variables that are very different from variables discussed here. We will not cover shared variables in this chapter.

In the vector form, inputs, outputs, signals and variables may be used as a complete vector, part of a vector, or a bit of the vector. The latter two are referred to as slicing and indexing. Examples of slicing and indexing on right and left hand sides of a concurrent assignment statement are shown in Figure 2.12.

```
ARCHITECTURE mixed_processes_assignments OF aCircuit IS
   SIGNAL d : BIT;
   SIGNAL dv : BIT_VECTOR (7 DOWNTO 0);
BEGIN

   p1: PROCESS (a, b, cv)
      VARIABLE e : BIT;
      VARIABLE ev : BIT_VECTOR (7 DOWNTO 0);
   BEGIN
      IF (a = b) THEN ev := av; ELSE ev := bv;
      IF (a = '1') THEN wv <= av; ELSE wv <= "1000111";
      d <= e;
   END PROCESS;

   dv <= av XOR bv;
   w <= d AND a;

END ARCHITECTURE mixed_processes_assignments;
```

Figure 2.11 Using Signals and Variables

```
ARCHITECTURE indexing_slicing OF aCircuit IS
   SIGNAL d : BIT;
   SIGNAL dv : BIT_VECTOR (7 DOWNTO 0);
BEGIN
   wv (3 DOWNTO 0) <= av (7 DOWNTO 4) AND cv (7 DOWNTO 4);
   w <= cv (4);
   cv (7) <= av (0);
END ARCHITECTURE indexing_slicing;
```

Figure 2.12 Using Indexing and Slicing

2.1.5 Logic Value System

The standard VHDL defines BIT for the basic logic value type. This type has '0' and '1' logic values. The *std_logic* standard package defines a logic value system consisting of nine logic values. However, *std_logic* is not a part of the VHDL language and is only an IEEE standard utility package for the language. The VHDL standard does include this package. The nine values of *std_logic*, which are shown in Table 2.1, are used to represent high impedance, unknown, un-initialized, capacitive and resistive 1 or 0, and driven 1 or 0. In most cases only four or five of these nine values are sufficient to express the logic-level behavior of a circuit. The 'U' logic value is considered as the default value for objects that do not specify an initial value.

Since the *std_logic* type includes all values of the BIT type, we will be using it instead of the BIT type from now on in the chapter. In

the examples presented thus far replacing BIT with *std_logic* and BIT_VECTOR with *std_logic_vector* is all that is needed to make them work with this logic value system. The complete name of this package is *std_logic_1164*, where 1164 is the IEEE reference number to this standard. The *std_logic_unsigned* package, which will be used for unsigned types, contains a set of unsigned arithmetic, conversion, and comparison functions. This package is based on the *std_logic_1164* package. Finally, keep in mind that special care needs to be taken as VHDL has very strict type checking rules.

Table 2.1 Logic Value System

Value	Representing
'U'	Uninitialized
'X'	Forcing Unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High Impedance
'W'	Weak Unknown
'L'	Weak 0
'H'	Weak 1
'-'	Don't care

2.1.6 Resolutions

As discussed above, VHDL allows multiple concurrent assignments to resolved signals. A resolved signal has a resolution function associated with it. Resolution functions are part of the type mark used for a signal declaration. The standard BIT type does not have a resolution function embedded in it, and thus, it is not a resolved type. However, the *std_logic* IEEE standard type is a resolved type and uses the *resolved* resolution function.

When multiple concurrent assignments are made to a resolved signal, we say that the signal has multiple drivers. In this case, the resolution function decides the final value of the signal. As an example consider the description of *selector* in Figure 2.13. Signal *yv* is declared as *std_logic_vector* and it is a resolved type. In the body of *multiple_drivers* architecture, two concurrent assignments are made to this signal. Depending on values of *as* and *bs*, *yv* simultaneously receives *av* and *bv*, *av* and "ZZZZZZZZ", *bv* and "ZZZZZZZZ", or "ZZZZZZZZ" and "ZZZZZZZZ". The final value assigned to *yv* will be determined by the *resolved* resolution function. The example in Figure 2.13 also shows how the *std_logic_1164* package can be made visible to a design.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTIRY selector IS
   PORT (av, bv: IN std_logic_vector (7 DOWNTO 0),
         as, bs: IN std_logic;
         yv: OUT std_logic_vector (7 DOWNTO 0));
END ENTITY selector;

ARCHITECTURE multiple_drivers OF selector IS
BEGIN
   yv <= av WHEN as = '1' ELSE "ZZZZZZZZ";
   yv <= bv WHEN bs = '1' ELSE "ZZZZZZZZ";
END ARCHITECTURE multiple_drivers;
```

Figure 2.13 Multiple Assignments to a Resolved Signal

Table 2.2 shows how the *resolved* function of the IEEE Std. 1164 package (the *std_logic* package for short) works for five or the more commonly used values. Based on this table if in the above example, *av* is "00001111", *bv* is "00111100", and *as* and *bs* are both '1', then *yv* becomes "00XX11XX". But, if for example *bv* is '0' instead of '1', then *yv* becomes "00001111".

Table 2.2 Partial *std_logic resolved* Function

U	U	X	0	1	Z
U	U	U	U	U	U
X	U	X	X	X	X
0	U	X	0	X	0
1	U	X	X	1	1
Z	U	X	0	1	Z

2.2 Combinational Circuits

A combinational circuit can be represented by its gate level structure, its Boolean functionality, or description of its behavior. At the gate level, interconnection of its gates are shown; at the functional level, Boolean expressions representing its outputs are written; and at the behavioral level a software-like procedural description represents its functionality. This section shows these three levels of abstraction for describing combinational circuits. Examples for combining various forms of descriptions and instantiation of already described components will also be described here.

2.2.1 Gate Level Combinational Circuits

A combinational circuit can be represented by its subcomponents at its gate level. VHDL does not provide primitive structures and thus any primitive gate (NAND, NOR, etc.) must be separately described using an entity-architecture pair. Figure 2.14 shows the VHDL code for several common gate level components.

```
ENTITY AND2 IS PORT (i1, i2 : IN std_logic; o1: OUT std_logic);
END ENTITY AND2;

ARCHITECTURE example OF AND2 IS
BEGIN
    o1 <= i1 AND i2 AFTER 3 NS;
END example;
        ENTITY OR3 IS PORT (i1, i2, i3 : IN std_logic; o1: OUT std_logic);
        END ENTITY OR3;

        ARCHITECTURE example OF OR3 IS
        BEGIN
            o1 <= i1 OR i2 OR i3 AFTER 6 NS;
        END example;
                ENTITY BUFIF1 IS PORT (i1, en : IN std_logic; o1: OUT std_logic);
                END ENTITY BUFIF1;

                ARCHITECTURE example OF BUFIF1 IS
                BEGIN
                    o1 <= i1 AFTER 4 NS WHEN en = '1' ELSE 'Z' AFTER 3 NS;
                END example;
                        ENTITY BUFIF0 IS PORT (i1, en : IN std_logic; o1: OUT std_logic);
                        END ENTITY BUFIF0;

                        ARCHITECTURE example OF BUFIF0 IS
                        BEGIN
                            o1 <= i1 AFTER 4 NS WHEN en = '0' ELSE 'Z' AFTER 3 NS;
                        END example;
```
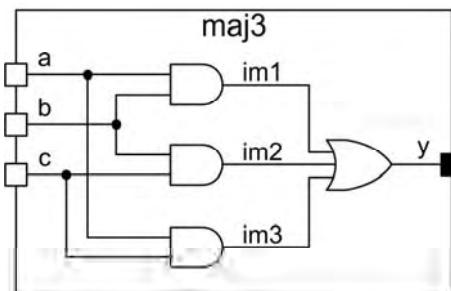
Figure 2.14 Basic Primitives Described in VHDL

The 2-input AND gate shown in Figure 2.14 has an entity declaration with *i1* and *i2* as inputs of type *std_logic* and a single output named *o1*. For defining the component's functionality a signal assignment is used. A signal assignment statement assigns values to its left-hand side signal or output, buffer or inout ports. An event observed on *i1* or *i2*, right-hand side signals of the signal assignment of the AND2 gate causes the evaluation of the right hand side expression. The VHDL code shows a 3 ns delay for the 2-input AND component. This results in the assignment of a new value to *o1* taking place 3 ns after the evaluation of the right-hand-side expression.

The BUFIF1 and BUFIF0 gates are implemented using conditional signal assignments which will be described later. Other primitive gates can be easily defined as shown above. These primitive gates provide a sufficient set of components for the description of larger structures. With the help of component instantiation, which will be shortly discussed, larger components can be described.

2.2.1.1 Majority Example. We use the majority circuit of Figure 2.15 to illustrate how primitive gates are used in a design. The description

shown in Figure 2.16 corresponds to this circuit. The module descrip-
tion has inputs and outputs according to the schematic of Figure 2.15.

Figure 2.15 A Majority Circuit

Figure 2.16 shows a structural description of a majority circuit
consisting of four subcomponent instantiations. Components that are
to be instantiated in this architecture need to be compiled and exist
in the default WORK library. Each component instantiation starts
with an arbitrary label followed by the ENTITY keyword, the name of
the library and the component that is being instantiated. If the archi-
tecture name is eliminated, the most recently compiled architecture
of the library mentioned will be used. Alternatively, the architecture
name of the component being instantiated can follow its name in a set
of parenthesis. Following the component name, a mapping between
the ports of the instantiated component and the actual signals is
made. The interconnection between these subcomponents is done
with signals declared in the declarative part of the architecture.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY maj3 IS
   PORT (a, b, c : IN std_logic;
         y       : OUT std_logic);
END maj3;

ARCHITECTURE gate_level OF maj3 IS
   SIGNAL im1, im2, im3 : std_logic;
BEGIN
   ANDa: ENTITY WORK.AND2 PORT MAP (a, b, im1);
   ANDb: ENTITY WORK.AND2 PORT MAP (b, c, im2);
   ANDc: ENTITY WORK.AND2 PORT MAP (a, c, im3);
   ORa : ENTITY WORK.OR3  PORT MAP (im1, im2, im3, y);
END ARCHITECTURE gate_level;
```

Figure 2.16 VHDL Code for the Majority Circuit

The use of the LIBRARY and USE clauses at the beginning of the code of Figure 2.16 and all other codes that use the *std_logic* type is necessary, and we will not repeat them in the examples that follow.

2.2.1.2 Multiplexer Example. Figure 2.17 shows a 2-to-1 multiplexer using three-state gates. This circuit is a good example to show how a resolution function works for a signal. As shown in Figure 2.18, the 2-to-1 multiplexer is described using BUFIF1 and BUFIF0, which we defined in Figure 2.14.

The VHDL code of this multiplexer instantiates BUFIF1 and BUFIF0 of Figure 2.14. In both cases the names of the architectures are also included. The output is *y*, and since it is driven by both gates, a resolution formed. When *s* is '1' BUFIF1 conducts and the value of *b* propagates to its output. At the same time, because *s* is '1', BUFIF0 does not conduct and its output becomes 'Z'. Resolution of these values driving *y* is determined by the resolution shown in Table 2.2.
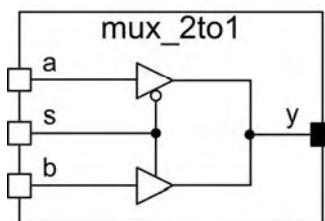
Figure 2.17 Multiplexer Using Three-state Gates

```
ENTITY mux_2to1 IS
   PORT (a, b, s: IN std_logic; y: OUT std_logic);
END ENTITY mux_2to1;

ARCHITECTURE gate_level OF mux_2to1 IS BEGIN
   BUFIF1a: ENTITY WORK.BUFIF1(example) PORT MAP (b, s, y);
   BUFIF1b: ENTITY WORK.BUFIF0(example) PORT MAP (a, s, y);
END ARCHITECTURE gate_level;
```
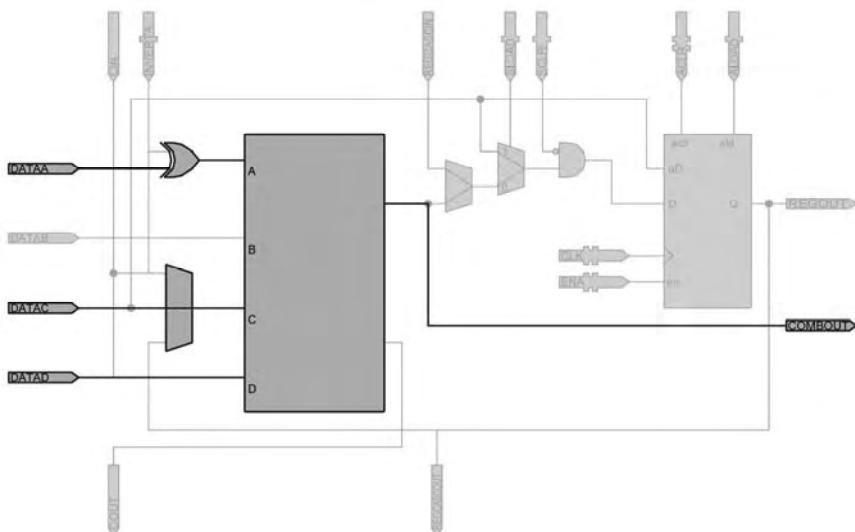
Figure 2.18 Multiplexer VHDL Code

2.2.2 Gate Level Synthesis

Gate level descriptions in VHDL are synthesizable. However, it must be noted that a designer using a gate level description cannot expect the same exact gates and interconnections to be used in the synthesized output. The hardware generated by a synthesis tool, and how
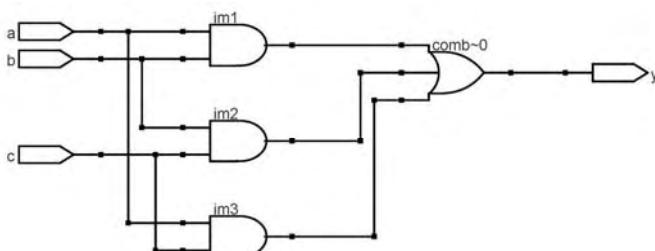
gates are implemented merely depends on the target technology. Furthermore, delays used in a gate level description are always ignored by the synthesis tools.

The synthesis of *maj3* module of Figure 2.16 using Altera's Quartus II and specifying Cyclone as the target FPGA results in using a single look-up table of a logic element, as shown in Figure 2.19. As shown, the three inputs of the circuit are used in a look-up table to produce the necessary combinational output.

Figure 2.19 Logic Element Used for *maj3*

The RTL view of this implementation that is also produced by our synthesis tool gives a better view of the functionality of this generated hardware. Figure 2.20 shows this view of our gate level *maj3* module. As expected, the AND-OR functionality of the Majority function is obtained from the synthesis tool.

Figure 2.20 RTL (logical) View of Synthesized *maj3*

Gate level descriptions using tri-state primitive structures are also synthesizable. If the synthesis target hardware does not have tri-state structures inside the chip (such as Altera's Cyclone) regular AND-OR gates will be used for the implementation of a description that uses tri-states. For example, when synthesized, the description of the multiplexer of Figure 2.18 uses a single look-up table.

2.2.3 Descriptions by Use of Equations

At a higher level than gates and transistors, a combinational circuit may also be described by the use of Boolean, logical and arithmetic expressions. The VHDL language has a set of standard operators that can be used for Boolean, logical and arithmetic equations. Table 2.3 shows this set of operators. These operators can be used in assign statements in order to create the desired functionality for the circuit.

Table 2.3 VHDL Operators

Boolean Operators	NOT	AND	OR	NAND	NOR	XOR	XNOR	
Comparison Operators	=	/=	<	<=	>	>=		
Arithmetic Operators	+	-	ABS	MOD	REM	*	/	**
Concat. Operators	&							

2.2.3.1 XOR Example. Consider the description of an XOR gate using Boolean equations in Figure 2.21. As discussed before, a signal assignment can be used to assign values to ports or other signals. Here the XOR operator has been used in a simple signal assignment to assign the result of *i1* XORed with *i2* to the output after a 3 ns delay. As shown, instead of having to write our own gates in the way described in Section 2.2.1, we can simply use the operators listed in Table 2.3 to produce any desired expression.

```
ENTITY xor2 IS
   PORT (i1, i2: IN std_logic; o1: OUT std_logic);
END ENTITY xor2;
--
ARCHITECTURE expression OF xor2 IS
BEGIN
   o1 <= i1 XOR i2 AFTER 3 NS;
END ARCHITECTURE expression;
```

Figure 2.21 XOR VHDL Code

2.2.3.2 Full-Adder Example. Figure 2.22 shows another example of using concurrent signal assignments. This code corresponds to a single-bit full adder. A full adder can be described by two signal assign statements; one for sum and one for carry out. The *sum* output can be generated with a Boolean expression using two XOR operators. The expression will be the same as the XOR example described above with the difference of having three operands this time. As for the carry-out (*cout*) output, again another signal assignment with AND and OR operators is used. Note that the use of parenthesis is necessary in the carry signal assignment for clarity of operator precedence.

```
ENTITY full_adder IS
   PORT (a, b, cin : IN std_logic;
         sum, cout : OUT std_logic);
END ENTITY full_adder;
--
ARCHITECTURE expression OF full_adder IS
BEGIN
   sum <= a XOR b XOR cin AFTER 0.3 NS;
   cout <= (a AND b) OR (a AND cin) OR (b AND cin)
           AFTER 0.2 NS;
END ARCHITECTURE expression;
```

Figure 2.22 Full Adder VHDL Code

The statements in the VHDL description of Figure 2.22 are concurrent. This means that the order in which they appear in this module is not important. These statements are sensitive to events on their right hand sides. When a change of value occurs on any of the right hand side signals, the statement is evaluated and the resulting value is scheduled for the left hand side signal.

2.2.3.3 Comparator Example. Consider the code shown in Figure 2.23, which corresponds to a 4-bit comparator. To define 4-bit arrays, the *std_logic_vector* predefined array type in the *std_logic* package is used. This array type represents is a collection of signals of the *std_logic* type. The (3 DOWNTO 0) is the range of indices for the vector (3 is the index of the leftmost element and 0 is for the right most element in the vector). Note that the range can also be declared as (0 TO 3), but the former declaration is recommended since the indices correspond to bit position weights in an array.

First, the result of XORing bits of *in1* and *in2* are assigned to *im*, which is defined as a 4-bit signal in the declaration part of the architecture. In the next line, a function call is used to produce the *eq* output. A function is a subprogram consisting of sequential statements that can be called anywhere in a VHDL code. Functions can be de-

clared in several places such as the declarative part of an architecture or a process. A function declaration starts with the FUNCTION keyword and a function name followed by its parameters and finally a single return value type. In this case, our function has a 4-bit input of type *std_logic_vector* and a *std_logic* return type. This function performs bit-by-next-bit operation, giving a 1-bit result. The function is called in the architecture statement body with *im* as its input.

```
ENTITY comp_4bit IS PORT (
      in1, in2 : IN std_logic_vector (3 DOWNTO 0);
      eq       : OUT std_logic );
END comp_4bit;

ARCHITECTURE functional OF comp_4bit IS
   SIGNAL im : std_logic_vector (3 DOWNTO 0);
   FUNCTION nor_reduce
      (in1: IN std_logic_vector (3 DOWNTO 0))
      RETURN std_logic
   IS
      VARIABLE result : std_logic ;
   BEGIN
      result:= NOT (in1(3) OR in1(2) OR in1(1) OR in1(0)) ;
      RETURN result;
   END;
BEGIN
   im <= in1 XOR in2;
   eq <= nor_reduce(im);
END functional;
```

Figure 2.23 Four-Bit Comparator

As shown in Figure 2.23, the function result is defined as a variable. Notice the use of := instead of <= in the assignment. There will be more on variables later in the chapter.

Another way to describe the comparator circuit is to use a conditional signal assignment. A conditional signal assignment is a signal assignment that takes place only when the condition stated after the WHEN keyword is met. In the case of our comparator example, the below conditional signal assignment produces the correct *eq* value. This statement allows using another conditional part after the ELSE keyword.

```
   eq <= '1' WHEN in1 = in2 ELSE '0';
```

2.2.3.4 Multiplexer Example. Figure 2.24 shows a 2-to-1 multiplexer using a conditional operator. Note here that the sizes of the *a* and *b* inputs of the multiplexer are not specified here. These will be determined when the multiplexer is instantiated in an upper level

structure. In that case, the sizes of the signals in the upper level structure that are associated with *a* and *b* will be passed to these signals.

```
ENTITY multiplexer IS
   PORT (a, b : IN std_logic_vector; s : IN std_logic;
         w : OUT std_logic_vector);
END ENTITY;

ARCHITECTURE expression OF multiplexer IS
BEGIN
   w <= a WHEN s = '0' ELSE b;
END ARCHITECTURE expression;
```

Figure 2.24 An Unconstrained 2-to-1 Mux using Condition Operator

2.2.3.5 Decoder Example. Figure 2.25 shows a 2-to-4 decoder, coded using a selected signal assignment. The behavior of a selected signal assignment is similar to the case statement. A case statement can only appear in VHDL sequential bodies where, selected signal assignments are for the concurrent bodies of VHDL, e.g., an architecture body. A selected signal assignment statement is similar to a conditional signal assignment in the way that it chooses from a number of expressions based on a condition, but they differ in that only conditions relating to one expression are used in a selected signal assignment.

```
ENTITY dcd2to4 IS
   PORT (sel: IN std_logic_vector (1 DOWNTO 0);
         y: OUT std_logic_vector (3 DOWNTO 0) );
END dcd2to4;

ARCHITECTURE structural OF dcd2to4 IS
BEGIN
   WITH sel SELECT
      y <= "0001" WHEN "00",
           "0010" WHEN "01",
           "0100" WHEN "10",
           "1000" WHEN "11",
           "0000" WHEN OTHERS;
END ARCHITECTURE structural;
```

Figure 2.25 Decoder Using Selected Signal Assignment

2.2.3.6 Adder Example. For another example, consider the 8-bit adder circuit with a carry-in and a carry-out output shown in Figure 2.26. For performing unsigned addition, this description uses the *std_logic_unsigned* package that is another standard VHDL package.

In the body of the *equation* architecture, an assignment statement is used to set the add result of the two operands added with the carry to the intermediate *mid* signal. This signal is declared as a 9-bit signal to be able to capture the carry out of the nine bit addition on its right hand side. For size matching the right and the left of the assignment to *mid*, '0's are concatenated with *a* and *b* to make them nine bit vectors.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY adder8 IS PORT (
   a    : in  std_logic_vector (7 DOWNTO 0);
   b    : in  std_logic_vector (7 DOWNTO 0);
   ci   : in  std_logic;
   s    : out std_logic_vector (7 DOWNTO 0);
   co   : out std_logic );
END ENTITY adder8;
--
ARCHITECTURE equation OF adder8 IS
   SIGNAL mid : std_logic_vector (8 DOWNTO 0);
BEGIN
    mid <= ('0'&a) + ('0'&b) + ci;
    co <= mid (8);
    s  <= mid (7 DOWNTO 0);
END equation;
```

Figure 2.26 Adder with Carry-in and Carry-out

3.2.3.7 ALU Example. As our final example of concurrent assignment statements, consider an ALU that performs add and subtract operations and has two flag outputs *gt* and *zero*. The *gt* output becomes '1' when input *a* is greater than input *b*, and the *zero* output becomes '1' when the result of the operation performed by the ALU is '0'.

Figure 2.27 shows the VHDL code of this ALU. Using a conditional signal assignment, the *addsub* input decides whether ALU inputs should be added or subtracted. Other VHDL constructs used in this description are arithmetic, concatenation, conditional, compare and relational operations.

```
ENTITY alu8 IS PORT (
   a, b   : in  std_logic_vector (7 DOWNTO 0);
   addsub : in  std_logic;
   gt, zero, co : out std_logic;
   r        : out std_logic_vector (7 DOWNTO 0));
END ENTITY alu8;

ARCHITECTURE assigns OF alu8 IS
   SIGNAL mid : std_logic_vector (8 DOWNTO 0);
BEGIN
   mid <= ('0'& a) + ('0'& b) WHEN addsub = '1' ELSE
           ('0'& a) - ('0'& b);
   co <= mid (8);
   r  <= mid (7 DOWNTO 0);
   gt <= '1' WHEN a > b ELSE '0';
   zero <='1' WHEN mid (7 DOWNTO 0) = "00000000" ELSE '0';
END assigns;
```

Figure 2.27 ALU VHDL Code Using a Mix of Operations

2.2.4 Instantiating Other Modules

Existing designs (entity-architecture pairs) can be instantiated in a upper-level structures and wired with other components. We show a simple example in this section.

2.2.4.1 ALU Example Using Adder. The ALU of Figure 2.27 starts from scratch and implements every function it needs inside its architecture. If we have a situation that we need to use a specific design from a given library, or we have a function that is too complex to be repeated everywhere it is used, we can describe it separately, compile it, and instantiate it when we need to use it.

Figure 2.28 shows another ALU that is different from the above ALU in that the new ALU performs adding only. In this new ALU, addition is handled by the adder circuit of Figure 2.26. The *adder8* circuit is instantiated using VHDL direct instantiation. In this instantiation, association by position is used for associating the ALUs local signals to the ports of the adder. Because we are not using the carry-out of the adder, we are associating it with OPEN.

Alternatively, association by name could be used that uses the actual port names of the component being instantiated. In this case, the actual port name comes first, followed by a right arrow and then followed by the local signal name. This alternative is shown in Figure 2.28 below the instantiation statement that uses association by position, and it is commented out. Note that the carry output from the adder circuit that we are not using here, is not listed in the associa-

tion list. This is equivalent to using OPEN in association by position. Order of association elements in association by name is not important.

```
ENTITY alu8add IS PORT (
   a, b   : in  std_logic_vector (7 DOWNTO 0);
   gt, zero, co : out std_logic;
   r      : out std_logic_vector (7 DOWNTO 0));
END ENTITY alu8add;

ARCHITECTURE assigns OF alu8add IS
   SIGNAL mid8 : std_logic_vector (7 DOWNTO 0);
   SIGNAL mid1 : std_logic;
BEGIN

   AD: ENTITY WORK.adder8 PORT MAP (a, b, '0', mid8, OPEN);

-- AD: ENTITY WORK.adder8 PORT MAP
--    (a => a, b => b, ci => '0', s => mid8);

   r   <= mid8;
   gt <= '1' WHEN a > b ELSE '0';
   zero <= '1' WHEN mid8 = "00000000" ELSE '0';
END assigns;
```

Figure 2.28 ALU VHDL Code Using Instantiating an Adder

2.2.5 Synthesis of Assignment Statements

Descriptions of the previous section concentrated on using signal assignment statements. We also showed components that instantiated other components consisting of signal statements. In general, concurrent signal assignments, regardless of the hierarchy they are used in and their complexity, are synthesizable.

As an example of this synthesis, consider the *alu8add* of Figure 2.28. The RTL view of this circuit after being synthesized by Quartus II is shown in Figure 2.29. As shown, the generated hardware uses multiplexers for conditional assignments and adders for the addition operations. The gray box in this diagram corresponds to the adder8 of Figure 2.26. Details of this adder are shown in the lower part of the diagram. The complete hardware uses 19 logic-elements of an Altera Cyclone II FPGA.
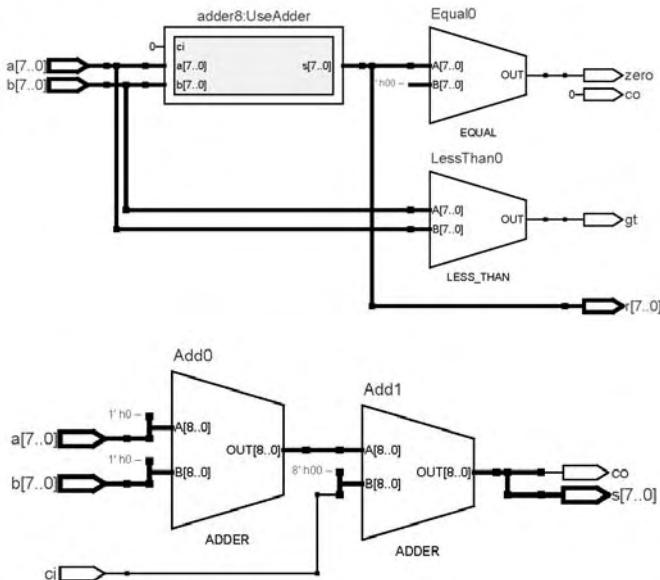
Figure 2.29 *ALU_Adder* **RTL View after Synthesis**

2.2.6 Descriptions with Sequential Flow

As described earlier in the chapter, VHDL provides constructs such as different kinds of signal assignments, component instantiation and processes to model concurrency. In addition to these constructs, VHDL also provides constructs for sequential description of hardware. This is referred to as the behavioral approach to describe hardware components. In this approach, the designer expresses the functionality or behavior of hardware instead of its structural details. Behavioral descriptions are supported with the process statements in VHDL.

A process statement runs concurrently with other concurrent components but its body consists of sequential statements. The process statement can appear in the architecture body just as signal assignments and, encloses in a begin-end pair. The sequential statements in a process statement run repeatedly during a simulation run. The PROCESS keyword starts the definition of a process statement. A list of signals enclosed in a pair of parenthesis can appear after the process keyword as its sensitivity list. Any event on any of these signals will cause the process to be executed again. If no sensitivity list

is specified for a process statement, that process will run continuously until the end of the simulation time.

A process statement consists of a declarative part and a statement part. The statement part is where the sequential statements are listed while the declarative part consists of functions, variables and other declarations (signal declarations are not allowed in the declarative part of a process). Variables are used to store intermediate values within a process. Variable declaration is similar to that of signals and it takes place in the declarative part of a process. Assigning values to variables is done with :=. Initial values can also be assigned to variables in their declarations.

Although variables can be data carriers like signals, they differ from them in several ways. Signals do not change in a process execution and therefore cannot hold intermediate values within a process. This is where variables become handy. They can store intermediate values by a variable assignment. Declarations including variable declarations take place once at the beginning of a simulation run and a variable's value is retained between process iterations. Another difference between signals and variables is that only signal assignments are timed and the after clause cannot be used with variable statements.

Another issue regarding signal assignments in concurrent bodies is the delta delay. A simulation cycle is referred to as delta delay in VHDL. Consider two signal assignments in an architecture body, one of which its output causes the evaluation of the right hand side of the other signal assignment. Since these are concurrent statements one expects both of their results to be ready at the same time. On the other hand, the first signal assignment causes an event on the right-hand side of the second signal assignments. So it is obvious that there is a time delay between these two statements. This delay is what is referred to as delay time or delta delay, and is shown by the δ symbol. The δ delay is what makes us think that the computer is performing our simulation concurrently.

2.2.6.1 Majority Example. Figure 2.30 shows a majority circuit described by the use of a process statement. This process statement is sensitive to a, b and c signals. Any event on these signals will cause the process block to run again and a new value will be assigned to the y output. No delay has been considered for this signal assignment. Therefore, the value of y will be updated δ time after any event on a, b or c.

```
ENTITY maj3 IS
   PORT (a, b, c : IN std_logic;
         y       : OUT std_logic);
END maj3;

ARCHITECTURE sequential OF maj3 IS
BEGIN
   PROCESS (a, b, c)
   BEGIN
      y <= (a AND b) OR (b AND c) OR (a AND c);
   END PROCESS;
END ARCHITECTURE sequential;
```

Figure 2.30 Procedural Block Describing a Majority Circuit

2.2.6.2 Majority Example with Delay. The VHDL code corresponding
to a majority circuit with delay is shown in Figure 2.31. The code is
very similar to the code described in Figure 2.30 with an after clause
with a 5 ns delay value. In this case the result of the right-hand side
will be placed on y's driver with the 5 ns delay value. After an event
on one of the inputs, the scheduling of value into the driver of y takes
place, but the value on y does not appear until 5 ns after the process
has been executed.

```
ARCHITECTURE sequential_delay OF maj3 IS
BEGIN
   PROCESS (a, b, c)
   BEGIN
      y <= (a AND b) OR (b AND c) OR (a AND c) AFTER 5 NS;
   END PROCESS;
END ARCHITECTURE sequential_delay;
```

Figure 2.31 Majority Gate with Delay

2.2.6.3 Procedural Multiplexer Example. Figure 2.32 illustrates an-
other example of a sequential process. This example uses an if state-
ment. VHDL allows the use of if statements in process statements.
An if statement is similar to the conditional signal assignment con-
struct, with the difference that it is only allowed in sequential bodies.
Depending on the condition specified for the if statement, a corre-
sponding branch is taken. Note that each if statement contains a cor-
responding then part. An if statement ends with the END IF key-
word. There can be any number of else-if branches to an if statement.
For the else-if parts, the ELSIF keyword must be used. On the other
hand a maximum of one else statement is allowed per each if-
statement. This optional else statement should be the last branch in
an if statement.

```
ENTITY multiplexer IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE procedural OF multiplexer IS BEGIN
   PROCESS (a, b, s) BEGIN
      IF (s = '0') THEN w <= a;
      ELSE w <= b;
      END IF;
   END PROCESS;
END ARCHITECTURE procedural;
```

Figure 2.32 Sequential Flow Multiplexer

In this example, if the selector (*s*) is equal to zero, the block after the THEN keyword is executed and *w* takes *a*, otherwise the block of statements after the ELSE are taken and therefore *w* takes *b*.

2.2.6.4 Procedural ALU Example. A case statement is similar to if statements, described in the previous example, in branching on a condition. However a case statement chooses a branch based on the value of the case expression, which does not need to be a Boolean. Case statements are preferred over if statements when many choices exist. This is why we have used the case statement in Figure 2.33. A case statement is somehow similar to the selected signal assignment described earlier. As discussed before, selected signal assignments and conditional signal assignments cannot be used in process statement bodies.

```
ENTITY alu8 IS
   PORT (left_i, right_i: IN std_logic_vector (7 DOWNTO 0);
         mode : IN std_logic_vector (1 DOWNTO 0);
         aluout : OUT std_logic_vector (7 DOWNTO 0));
END ENTITY;
--
ARCHITECTURE procedural OF alu8 IS BEGIN
   PROCESS (left_i, right_i, mode) BEGIN
      CASE mode IS
         WHEN "00" => aluout <= left_i + right_i;
         WHEN "01" => aluout <= left_i - right_i;
         WHEN "10" => aluout <= left_i AND right_i;
         WHEN "11" => aluout <= left_i OR right_i;
         WHEN OTHERS => aluout <= "XXXXXXXX";
      END CASE;
   END PROCESS;
END ARCHITECTURE procedural;
```

Figure 2.33 Sequential Flow ALU

This ALU receives *left_i*, *right_i* and *mode* as its inputs. The process statement, shown in the VHDL code, is sensitive to all its inputs. The case statement used here selects the correct operation corresponding to *mode* from the case alternatives. the case statement has a case expression and several case alternatives. Each case alternative begins with the WHEN keyword. The last case alternative is the OTHERS or the default alternative. This branch is taken when the condition does not match any of the other alternatives. In our example, inputs containing Z or X will take this branch.

2.2.7 Combinational Rules

Now that if and case statements are covered, consider a case where there are conditions under which the output of a combinational circuit is not assigned a value. Obviously the output retains its previous value, which implies the latching behavior. This latching behavior is unwanted in describing combinational circuits. Therefore two rules can be considered in describing combinational circuits with process statements.

1. List all inputs of the combinational circuit in the sensitivity list of the process statement describing it.
2. Make sure all combinational circuit outputs receive some value regardless of how the program flows in the conditions of if or case statements. If there are too many conditions to check, set all outputs to their inactive values at the beginning of the process statement.

2.2.8 Bussing

Bus structures can be implemented by the use of multiplexers or three-state logic structures. Various methods of describing combinational logic can be used for the description of a bus, in VHDL.

Figure 2.34 shows the VHDL code for a three-state bus with three sources, *busin1*, *busin2* and *busin3*. These sources are put on *busout* by active high enabling control signals: *en1*, *en2* and *en3*.

Three conditional signal assignments are used for assigning values to *busout*. Each conditional signal assignment either selects a bus driver or a 4-bit 'Z' value for *busout*. As said before, multiple assignments to a signal produce multiple drivers for that circuit. Therefore, multiple conditional signal assignments are appropriate for representing busses in VHDL. Note that only one enable should be active at a time. Multiple driving values for *busout* are resolved using the *resolved* resolution function, which is part of the *std_logic* logic value

system used in this example. Enabled inputs of *busout* produce logic values on their drivers, while inactive ones put all 'Z' values on their drivers. Resolving 'Z' with logic values results in a valid logic value, but resolving multiple logic values results in unknown. This is why we use high impedance for inactive enables.

```
ENTITY bussing IS
   PORT (
      busin1: IN std_logic_vector (3 DOWNTO 0);
      busin2: IN std_logic_vector (3 DOWNTO 0);
      busin3: IN std_logic_vector (3 DOWNTO 0);
      en1: IN  std_logic;
      en2: IN  std_logic;
      en3: IN  std_logic;
      busout: OUT std_logic_vector(3 DOWNTO 0) );
END bussing;
--
ARCHITECTURE structural OF bussing IS
BEGIN

   busout <= busin1 WHEN en1 = '1' ELSE (OTHERS => 'Z');

   busout <= busin2 WHEN en2 = '1' ELSE (OTHERS => 'Z');

   busout <= busin3 WHEN en3 = '1' ELSE (OTHERS => 'Z');

END structural;
```
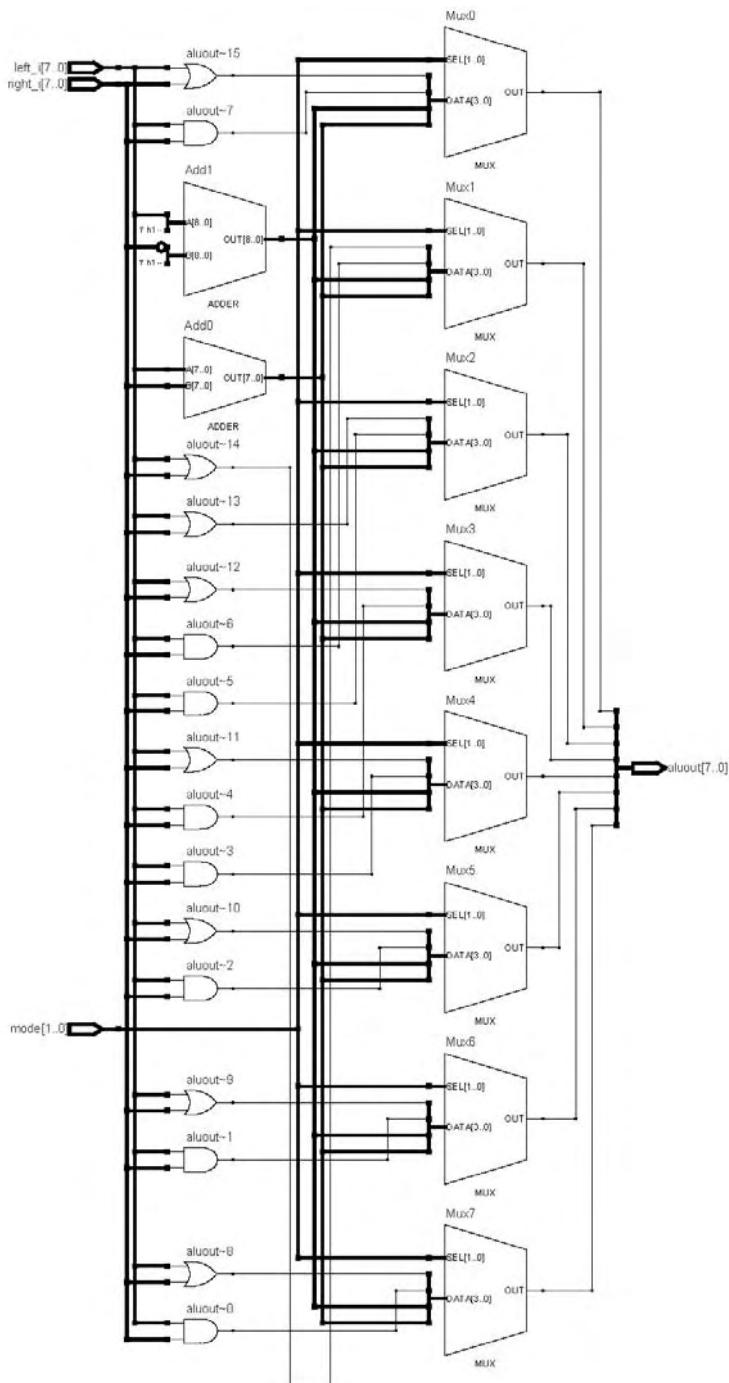
Figure 2.34 Three-state Bussing

2.2.9 Synthesizing Procedural Blocks

Following combinational synthesis rules discussed above, we can easily develop VHDL designs for synthesis. We use the *alu8* example of Figure 2.33 for demonstrating this synthesis. Figure 2.35 shows the RTL view of the synthesis result of this circuit using Altera's Quartus II. As shown, the circuit has eight one-bit 4-to-1 multiplexers. Each multiplexer selects one of the four ALU functions depending on the value of *mode*. The logic for the eight functions of *left_i* and *right_i* appear to the left of the multiplexers.

The complete implementation of this circuit uses 33 of the 32,216 logic-elements of an EP2C35F672C6 Cyclone II FPGA. The Quartus II FPGA design tool provides other post synthesis diagrams and reports, including several timing reports and exact FPGA cell interconnections.

Figure 2.35 Synthesis of Sequential Flow ALU

2.3 Sequential Circuits

As with any digital circuit, a sequential circuit can be described in VHDL by use of gates, Boolean expressions, or behavioral constructs (e.g., the process statement). While gate level descriptions enable a more detailed description of timing and delays, because of complexity of clocking and register and flip-flop controls, these circuits are usually described by use of sequential process statements. This section shows various ways sequential circuits are described in VHDL. The following discusses primitive structures like latch and flip-flops, and then generalizes coding styles used for representing these structures to more complex sequential circuits including counters and state machines.

2.3.1 Basic Memory Elements at the Gate Level

A clocked D-latch latches its input data during an active clock cycle. The latch structure retains the latched value until the next active clock cycle. This element is the basis of all static memory elements.

A simple implementation of the D-latch using cross-coupled NOR gates is shown in Figure 2.36. The VHDL code of Figure 2.37 corresponds to this D-latch circuit. Notice the use of buffer for *q* and *q_b*. This is due to the use of these signals on the right-hand-side of signal assignments. These signals could have been declared as OUT and two extra local signals could have done the job, but in this case two additional signal assignments would have been added to the code.
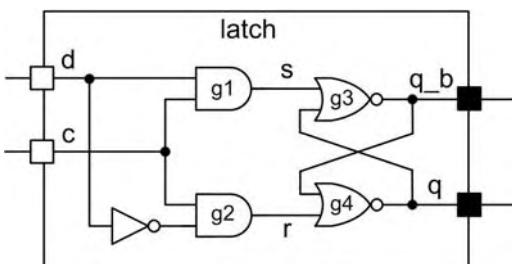
Figure 2.36 Clocked D-latch

```
ENTITY latch IS
   PORT (d, c: IN std_logic;
         q, q_b : BUFFER std_logic);
END latch;

ARCHITECTURE structural OF latch IS
   SIGNAL s, r : std_logic;
BEGIN
   s   <= c AND d       AFTER 6 ns;
   r   <= c AND (NOT d) AFTER 6 ns;
   q_b <= s NOR q       AFTER 4 ns;
   q   <= r NOR q_b     AFTER 4 ns;
END structural;
```

Figure 2.37 VHDL Code for a Clocked D-latch

Alternatively, the same latch can be described with a concurrent signal assignment. Three equivalent assignments are shown below. In all three cases, *q* receives a new value when *c* is '1' and retains its value otherwise.

```
q <= d WHEN c = '1' ELSE q;
q <= d WHEN c = '1';
q <= d WHEN c = '1' ELSE UNAFFECTED;
```

These statements describe what happens in a latch and describe latch transparency when *c* is '1'. Using two such statements with complementary clock values describe a master-slave flip-flop. As shown in Figure 2.38, the *qm* signal is the master output and *q* is the flip-flop output.

```
ENTITY master_slave IS
   PORT (d, c: IN std_logic;
         q : OUT std_logic);
END master_slave;

ARCHITECTURE dual OF master_slave IS
   SIGNAL qm : std_logic;
BEGIN
   qm <= d WHEN c = '1';
   q  <= qm WHEN c = '0';
END dual;
```

Figure 2.38 Master-Slave Flip-Flop

2.3.2 Memory Elements Using Procedural Statements

Although latches and flip-flops can be described in gate level by component instantiation and signal assignments, describing more complex register structures cannot be done this way. This section presents coding styles for describing latches and flip-flops using process statements. Later on, it will be shown that the same coding styles presented here can be used to describe memories with more complex control units as well as functional register structures such as counters and shift-registers.

2.3.2.1 Latches. Figure 2.39 shows a D-latch described with a process statement. This process statement is sensitive to the latch clock and data input represented by *c* and *d*, respectively. The if statement used in the process statement puts the data input into *q* when the latch clock is active. This implies that any change on *d* while *c* is 1, can be transparently seen on the output. This behavior is referred to as transparency and it is how latches work.

```
ENTITY latch1 IS
   PORT (d, c: IN std_logic; q: OUT std_logic);
END latch1;

ARCHITECTURE behavioral OF latch1 IS
BEGIN
   PROCESS (d, c)
   BEGIN
      IF c = '1' THEN
         q <= d;
      END IF;
   END PROCESS;
END ARCHITECTURE behavioral;
```

Figure 2.39 Procedural Latch

2.3.2.2 D Flip-Flop. While a latch is transparent, a change on the D-input of a D flip-flops does not directly pass on to its output. The D input will only have an effect on the specified edge of the clock. The VHDL code of Figure 2.40 describes a positive-edge trigger D-type flip-flop. The process statement used is sensitive to changes on the clock input. Since the code describes a positive-edge flip-flop, it only transfers the data on the D-input to its output when *clk* has made a '0' to '1' transition. A process statement sensitive to the *clk* and an if statement that determines a '0' to '1' transition on *clk* specifies the positive edge of *clk*. Similarly, a negative-edge trigger D flip-flop is created by changing the if statement to " IF *clk* = '0' ".

```
ENTITY DFF1 IS
   PORT (d, clk: IN std_logic; q : OUT std_logic);
END DFF1;
--
ARCHITECTURE behavioral OF DFF1 IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF clk = '1' AND clk'EVENT THEN
         q <= d;
      END IF;
   END PROCESS;
END ARCHITECTURE behavioral;
```

Figure 2.40 A Positive-Edge D Flip-Flop

2.3.2.3 Synchronous Control. The coding style presented for the above simple D flip-flop is a general one and can be expanded to cover many features found in flip-flops and even memory structures. The description shown in Figure 2.41 is a D-type flip-flop with synchronous set and reset (*s* and *r*) inputs.

```
ENTITY DFF1sr IS
   PORT (d, clk, s, r: IN std_logic; q : OUT std_logic);
END DFF1sr;
--
ARCHITECTURE behavioral OF DFF1sr IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF s = '1' THEN
            q <= '1';
         ELSIF r = '1' THEN
            q <= '0';
         ELSE
            q <= d;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE behavioral;
```

Figure 2.41 D Flip-Flop with Synchronous Control

The difference between this type of flip-flop and a simple D flip-flop is that on every positive edge of the clock, it first checks for the set and reset inputs and puts a '1' into the output if *s* is active and a '0' if *r* is active. Only when *s* and *r* are inactive, the flip-flop places the D-input on its output. Note that the *s* input has been given a higher priority to

r by first checking for *s* in the sequential process statement. The flip-flop structure corresponding to this description is shown in Figure 2.42.
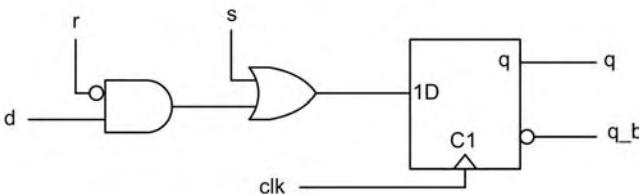
Figure 2.42 D Flip-Flop with Synchronous Control

Other synchronous control inputs can be added to this flip-flop in a similar fashion. A clock enable (*en*) input would only require inclusion of an if-statement in the last else part of the if-statement in the code of Figure 2.41.

2.3.2.4 Asynchronous Control. Consider the *asynchronous* architecture of *DFF1sr* of Figure 2.43 that describes a D flip-flop with asynchronous control. In order to have asynchronous control inputs we only need to add asynchronous control signals to the process sensitivity list and change the ordering of if statement conditions. In the previous case when the process statement was only sensitive to *clk*, the flow of the process statement would only start if any change happened on the *clk* input. In other words, changes on control inputs did not have any effect on starting the flow of the process statement. In this case however, by adding control signals to the sensitivity list, the process block flow can start by any event seen on these signals or the *clk*.

Figure 2.44 shows a graphical notation that corresponds to the asynchronous behavior described here.

```
ARCHITECTURE asynchronous OF DFF1sr IS
BEGIN
   PROCESS (clk, s, r) BEGIN
      IF s = '1' THEN
         q <= '1';
      ELSIF r = '1' THEN
         q <= '0';
      ELSIF clk = '1' AND clk'EVENT THEN
         q <= d;
      END IF;
   END PROCESS;
END ARCHITECTURE asynchronous;
```

Figure 2.43 D Flip-Flop with Asynchronous Control
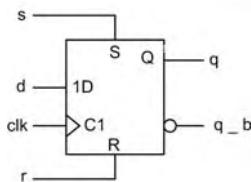
Figure 2.44 Flip-Flop with Asynchronous Control Inputs

2.3.3 Flip-flop Synthesis

In the above discussion, flip-flop and latches described by use of pro-
cedural statements are synthesizable. The flip-flop of the logic-
element used for the implementation of code of Figure 2.41 uses its
lookup-table to generate a '1' or '0' for setting and resetting the flip-
flop. As shown in Figure 2.45, all data (D, Set or Reset) of the flip-flop
go through the D input and its asynchronous inputs are unused. Data
on the D input is always controlled by the clock.
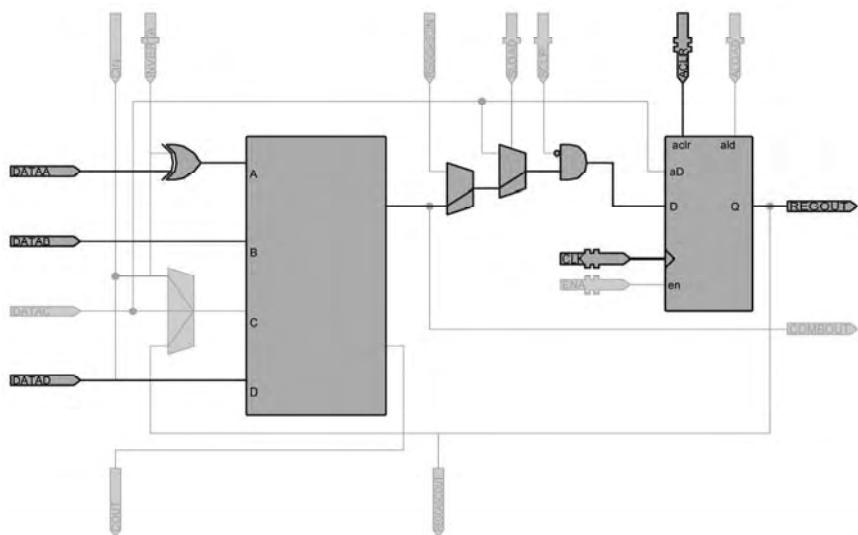
Figure 2.45 Synchronous Flip-Flop Synthesis

On the other hand, the flip-flop of the logic-element used for the
implementation of code of Figure 2.43 uses its asynchronous inputs
for setting and resetting. As shown in Figure 2.46 the lookup-table
used for the flip-flop with asynchronous control is only used for pass-
ing the D input of the circuit to the flip-flop D input. Logic-element
flip-flop inputs *aD* and *ald* are responsible for asynchronously loading
a '1' into the flip-flop and the *aclr* handles asynchronous resetting.
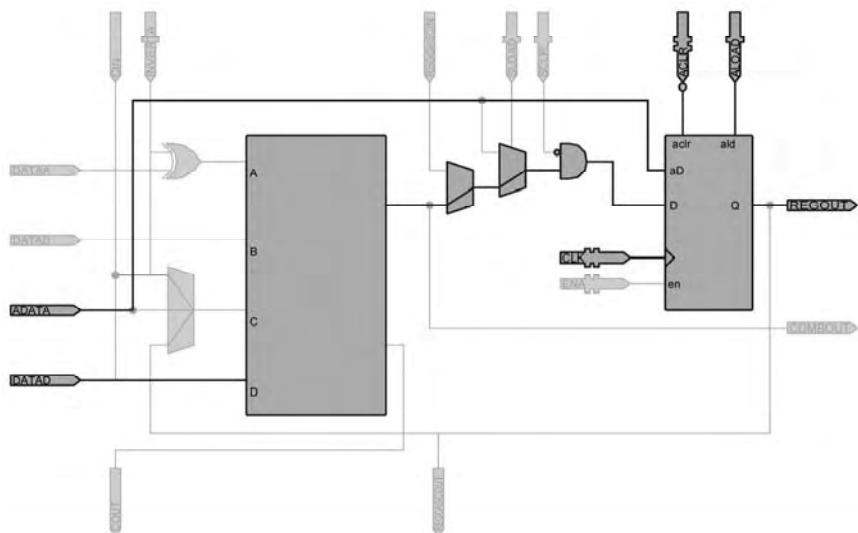
Figure 2.46 Asynchronous Flip-Flop Synthesis

RTL views of hardware generated for the two flip-flops are shown in Figure 2.47. Comparing these circuits, more clearly shows the use of the D-input of the flip-flop input for synchronous reset. The D-input of the flip-flop resulted from synthesizing Figure 2.41 has a logic block that involves d, r, and s (upper part of Figure 2.47). On the other hand, the circuit d input directly connects to the D-input of the flip-flop resulted from the synthesis of Figure 2.43 (lower part of Figure 2.47).
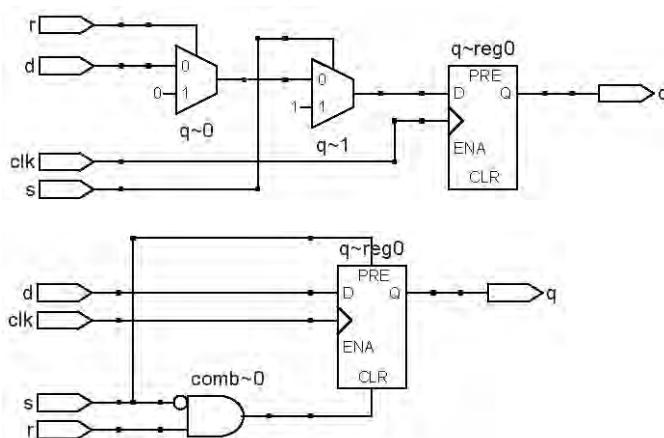
Figure 2.47 Synchronous versus Asynchronous Synthesis

2.3.4 Registers, Shifters and Counters

Registers, shifter-registers, counters and even sequential circuits with more complex functionalities can be described by simple extensions of the coding styles presented for the flip-flops. In most cases, the functionality of the circuit only affects the last else-part of the if-statement in the procedural statement of codes shown for the flip-flops.

2.3.4.1 Registers. Figure 2.48 shows the VHDL code for an 8-bit register with synchronous set and reset control. An 8-bit register is defined similar to a D-flip-flop but for a vector of eight elements. Assigning '1's ('0's) to bits of q is done by an aggregate operation on the right-hand side of the assignment. This operation selects bits of the right-hand side vector it is forming and associates values to these bits. The OTHERS keyword can be used in array aggregates to select all the indexes that have not been selected up to the appearance of OTHERS. A special case of this form, which is used in our example, is when no other indexes have been selected. In this case OTHERS selects all the indexes of the array.

```
ENTITY register8 IS
   PORT (
      d        : IN std_logic_vector (7 DOWNTO 0);
      clk, s, r  : IN std_logic;
      q        : OUT std_logic_vector ( 7 DOWNTO 0));
END register8;
--
ARCHITECTURE behavioral OF register8 IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF clk = '1' AND clk'event THEN
         IF s= '1' THEN
            q <= (OTHERS => '1');
         ELSIF r = '1' THEN
            q <= (OTHERS => '0');
         ELSE
            q <= d;
         END IF;
      END IF;
   END PROCESS;
END behavioral;
```

Figure 2.48 An 8-bit Register

2.3.4.2 Shift-Registers. A 4-bit shift-register with right- and left-shift capabilities, a serial-input, synchronous reset input, and parallel loading capability is shown in Figure 2.49. We have used the *ris-*

ing_edge function instead of the event expression used in other examples of this chapter. This function is part of the *std_logic_1164* package. All activities of the shift-register are synchronized with the clock input. If *rst* is '1', the register is reset, if *ld* is '1', parallel *d* inputs are loaded into the register, and if none are '1' shifting left or right takes place depending on the value of the *l_r* input ('1' for left, '0' for right). Shifting in this code is done by use of the concatenation operator &. For left-shift, *s_in* is concatenated to the right of *q(2 DOWNTO 0)* to form a 4-bit vector that is put into *q*. For right-shift, *s_in* is concatenated to the left of *q(3 DOWNTO 1)* to form a 4-bit vector that is clocked into *q*.

The style used for coding this register is the same as that used for flip-flops and registers presented earlier. In these examples, a single process statement handles function selection (e.g., zeroing, shifting, or parallel loading) as well as clocking data *d* into the register.

```
ENTITY shift_reg4 IS
    PORT (
        d   : IN std_logic_vector (3 DOWNTO 0);
        clk, ld, rst, l_r, s_in : IN std_logic;
        q   : OUT std_logic_vector (3 DOWNTO 0));
END shift_reg4;

ARCHITECTURE behavioral OF shift_reg4 IS
BEGIN
    PROCESS (clk)
        VARIABLE q_t: std_logic_vector (3 DOWNTO 0);
    BEGIN
        IF rising_edge (clk) THEN
            IF rst= '1' THEN
                q_t := (OTHERS => '0');
            ELSIF ld = '1' THEN
                q_t := d;
            ELSIF l_r = '1' THEN
                q_t :=  q_t (2 DOWNTO 0) & s_in ;
            ELSE
                q_t := s_in & q_t (3 DOWNTO 1);
            END IF;
        END IF;
        q <= q_t;
    END PROCESS;
END behavioral;
```

Figure 2.49 A 4-bit Shift Register

2.3.4.3 Counters. The style used for describing the shift-register in the previous discussion can be used for describing counters. A counter counts up or down, while a shift-register shifts right or left. We use

arithmetic operations in counting as opposed to shift or concatenation operators in shift-registers.

Figure 2.50 describes a 4-bit up-counter with a reset control input. The code for this counter is similar to the shift-register described above. The only difference is that the counter performs arithmetic addition instead of shifting. The *cnt_reg* is used for saving the status of the counter. The use of this signal eliminates the need for declaring *count* as a BUFFER.

```
ENTITY counter4 IS
   PORT (reset, clk : IN std_logic;
         count : OUT std_logic_vector (3 DOWNTO 0));
END ENTITY;
--
ARCHITECTURE procedural OF counter4 IS
   SIGNAL cnt_reg : std_logic_vector (3 DOWNTO 0);
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk = '0' AND clk'EVENT) THEN
         IF (reset='1') THEN
            cnt_reg <="0000" AFTER 1.2 NS;
         ELSE
            cnt_reg <= cnt_reg + 1 AFTER 1.2 NS;
         END IF;
      END IF;
   END PROCESS;
   count <= cnt_reg;
END ARCHITECTURE procedural;
```

Figure 2.50 An Up Counter

2.3.5 Synthesis of Shifters and Counters

Except for the operations that are performed in the process statements of the descriptions of the registers, counters, and shift registers, the above descriptions followed the same basic rules, and the same styles of coding could be used for them. The styles we presented are synthesizable, and for demonstration purposes we show the synthesis results obtained by synthesizing the shift register of Figure 2.49.

Synthesis of this shift register uses 4 logic-elements of an Altera Cyclone II chip. As shown in the RTL view of Figure 2.51, the generated hardware has a register part that feeds back to itself through a cluster of combinational logic parts.
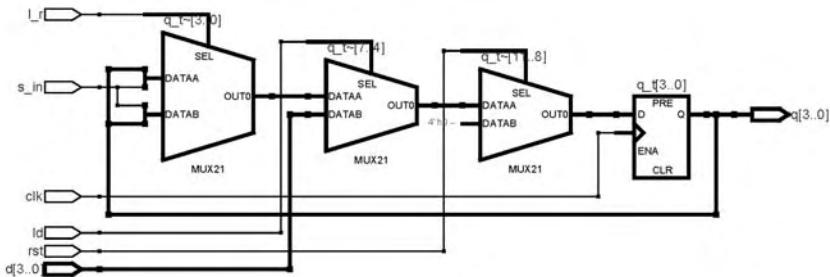
Figure 2.51 Shift Register Synthesis RTL View

2.3.6 State Machine Coding

Coding styles presented so far can be further generalized to cover finite state machines of any type. This section shows coding for Moore and Mealy state machines. The examples we will use are simple sequence detectors. These circuits represent the controller part of a digital system that has been partitioned into a data path and a controller. The coding styles used here apply to such controllers, and will be used in later chapters of this book to describe CPU and multiplier controllers.

2.3.6.1 Moore Detector. State diagram for a Moore sequence detector detecting **110** on its a input is shown in Figure 2.52. The machine has four states that are labeled, *S0*, *S1*, *S2*, and *S3*. Starting in *S0*, if the **110** sequence is detected, the machine goes into the *S3* state in which the output becomes '1'. In addition to the a input, the machine has a *reset* input that forces the machine into its *S0* state. The resetting of the machine is synchronized with the clock.
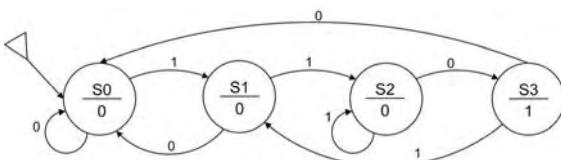
Figure 2.52 A Moore Sequence Detector

The VHDL code of the Moore machine of Figure 2.52 is shown in Figure 2.53. The current state is represented by an enumeration type. This type has a value for each state; therefore for detecting **110** this type should have four values. As shown, the *state* type is declared as an enumeration type that has four states, *S0*, *S1*, *S2*, and *S3*. Type

declarations take place in the declarative part of the architecture. Another example of an enumeration type is the BIT type of the standard package with enumeration values of '0' and '1'.

The process statement of Figure 2.53, which is sensitive to *clk*, is in charge of state transitions and register clocking. Upon the negative edge of the clock, checked by " *clk = '0' AND clk'EVENT* " the process statement checks for the *reset* input. If this input is active, the next state becomes *S0* by assigning *S0* to *current*. Note that the value put into *current* in this pass gets checked in the next run of the process statement. On the other hand if the *reset* input is not active, the program flow reaches the case statement and takes one of the case alternatives according to *current*. Each case alternative schedules a new value into *current* that will be observed the next time the process statement in entered.

```vhdl
ENTITY detector110 IS
   PORT (a, clk, reset : IN std_logic; w : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF detector110 IS
   TYPE state IS (S0, S1, S2, S3);
   SIGNAL current : state := S0;
BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '0' AND clk'EVENT) THEN
         IF reset = '1' THEN current <= S0;
         ELSE
            CASE current IS
               WHEN S0 =>
                  IF a='1' THEN current <= S1;
                  ELSE current <= S0; END IF;
               WHEN S1 =>
                  IF a='1' THEN current <= S2;
                  ELSE current <= S0; END IF;
               WHEN S2 =>
                  IF a='1' THEN current <= S2;
                  ELSE current <= S3; END IF;
               WHEN S3 =>
                  IF a='1' THEN current <= S1;
                  ELSE current <= S0; END IF;
               WHEN OTHERS => current <= S0;
            END CASE;
         END IF;
      END IF;
   END PROCESS;
   w <= '1' WHEN current = S3 ELSE '0';
END ARCHITECTURE procedural;
```

Figure 2.53 Moore Machine VHDL Code

The *w* output of the circuit is assigned values using a single conditional signal assignment. This statement is outside of the process statement and becomes active when an event occurs on *current*.

2.3.6.2 Pulse Synchronizer. Figure 2.54 shows a pulse synchronizer circuit. The circuit has an asynchronous *adata* input and a *synched* output. It generates synchronous pulses on its *synched* output while its *adata* input is high. This is a two-state state machine. The state of the machine is the same as the value of *synched*.

```
ENTITY synchronizer IS
   PORT (clk, adata : IN std_logic;
         synched : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF synchronizer IS
   TYPE state IS (S0, S1);
   SIGNAL current : state;
BEGIN
   PROCESS (clk) BEGIN
      IF (rising_edge(clk)) THEN
         IF current = S0 THEN
            IF adata = '0' THEN
               current <= S0;
            ELSE
               current <= S1;
            END IF;
         ELSE -- current = S1
            current <= S0;
         END IF;
      END IF;
   END PROCESS;
   synched <= '1' WHEN current = S1 ELSE '0';
END ARCHITECTURE procedural;
```

Figure 2.54 VHDL Code of a Synchronizer

2.3.7 State Machine Synthesis

All state machine descriptions discussed above are synthesizable. For demonstration purposes we discuss synthesis results of the *synchronizer* circuit of Figure 2.54. We synthesized this circuit using Altera Quartus II synthesis tool targeting the Cyclone II FPGA. The synthesized circuit uses one logic element of this FPGA. The RTL view and the single logic-element implementing this circuit are shown in Figure 2.55.
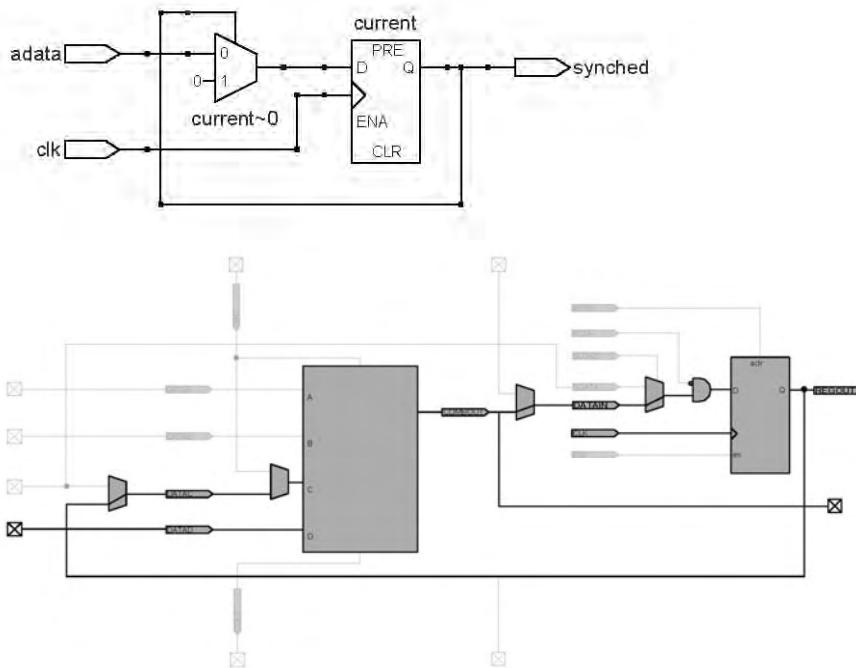
Figure 2.55 Synchronizer synthesis results, RTL View, Cell View

2.3.8   Memories

VHDL allows declaration and usage of multidimensional arrays for signals and variables. Memories are represented with user-defined types. Figure 2.56 shows a memory declaration and several valid memory operations. The memory type declaration has taken place in the declarative part of the architecture. Here, type *memory* is defined as an unconstrained array of 8-bit *std_logic* vectors. Then the *mem* signal is defined as a 1023-element array of memory elements. In addition, the process statement of Figure 2.56 declares *memv* as a variable of type *memory*. This variable is a memory array with 16 8-bit words.

2.4   Writing Testbenches

VHDL coding styles discussed so far were for coding hardware structures, and in all cases synthesizability and direct correspondence to hardware were our main concerns. On the other hand, testbenches do not have to have hardware correspondence and they usually do not

follow any synthesizability rules. The VHDL code of Figure 2.53 which is a **110** Moore detector will be used as the design under test (DUT) in this section.

Figure 2.57 shows a testbench developed for the Moore detector of Figure 2.53. The testbench entity which instantiates design under test (DUT), *detector110* in this example, has no ports. Initial values of DUT inputs are assigned in the signal declarations stated in the declarative part of this architecture. This testbench applies test data to DUT.

The testbench uses four processes (signal assignments and process statements are considered processes) that make assignments to the signals that are associated with the ports of DUT. The first process is a signal assignment for the *rst* signal. This assignment places a single resetting pulse on the reset input of DUT. The second process is a conditional signal assignment that generates a periodic signal on the circuit clock input. The clock toggles until the simulation time reaches 165 ns. The next process assigns values to the *aa* signal at given time instances. This is achieved by the sequential wait statements and sequential assignments to *aa*. The last process waits for an event on *ww* and then it reports the time and value of the circuit output.

```
ARCHITECTURE . . .
   TYPE memory IS
      ARRAY (INTEGER RANGE <>) OF
      std_logic_vector (7 DOWNTO 0);
   SIGNAL mem: memory(0 to 1023);
BEGIN
   PROCESS (mem)
      VARIABLE memv: memory(0 to 15);
      VARIABLE data: std_logic_vector(7 DOWNTO 0);
      VARIABLE short_data: std_logic_vector(3 DOWNTO 0);
   BEGIN
      . . .
      data := mem(956);
      short_data := mem(931)(6 downto 3);

      memv (12) := mem(189);

      mem (932) <= data ;
      mem (321)(5 DOWNTO 2) <= short_data;
      mem (940) <= "0000" & short_data ;

   END PROCESS;
   . . .
END ARCHITECTURE;
```

Figure 2.56 Memory Array Examples

```
ENTITY detector110_tester IS END ENTITY;
--
ARCHITECTURE timed OF detector110_tester IS
   SIGNAL aa, clock, rst, ww : std_logic := '0';
BEGIN
   UUT1: ENTITY WORK.detector110 (procedural)
         PORT MAP (aa, clock, rst, ww);

   rst <= '1' AFTER 31 NS, '0' AFTER 54 NS;
   clock <= NOT clock AFTER 7 NS WHEN NOW<=165 NS ELSE '0';

   PROCESS BEGIN
      WAIT FOR 23 NS; aa <= '1';
      WAIT FOR 21 NS; aa <= '0';
      WAIT FOR 19 NS; aa <= '1';
      WAIT FOR 31 NS; aa <= '0';
      WAIT;
   END PROCESS;

   PROCESS (ww) BEGIN
      REPORT "Signal w changed to:"& std_logic'IMAGE(ww)&
             " at " & TIME'IMAGE(NOW)
      SEVERITY NOTE;
   END PROCESS;

END ARCHITECTURE timed;
```
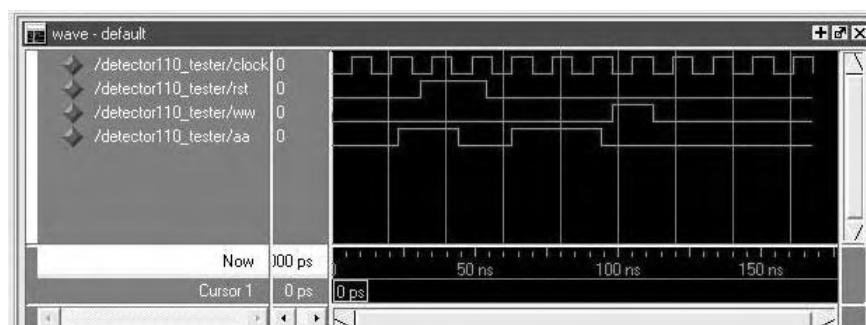
Figure 2.57 Moore Detector Testbench

The waveform resulting from the simulation of the testbench of Figure 2.57 is shown in Figure 2.58. Note from the waveform shown that the data we applied were able to cause the output of DUT to trigger.

Figure 2.58   Testbench Waveform Results

2.5 Synthesis Issues

VHDL constructs described in this chapter included those for cell modeling as well as those for designs to be synthesized. In describing an existing cell, timing issues are important and must be included in the VHDL code of the cell. At the same time, description of an existing cell may require parts of this cell to be described by interconnection of gates and transistors. On the other hand, a design to be synthesized does not include any timing information because this information is not available until the design is synthesized, and designers usually do not use gates and transistors for high level descriptions for synthesis.

Considering the above, knowing that the timings are ignored by synthesis tools, and only using gates when we really have to, the codes presented in this chapter all have one-to-one hardware correspondence and are synthesizable. For synthesis, a designer must consider his or her target library to see what and how certain parts can be synthesized. For example, most FPGAs do not have internal three-state structures and three-state bussings are converted to AND-OR busses.

2.6 VHDL Essential Terminologies

Several VHDL terminologies and linguistic issues were mentioned in this chapter in the context of examples that we presented. Most of these topics will be discussed and elaborated in the chapters that follow. However, a big-picture view of these topics and their role in the definition of the language is needed before we proceed into the presentation of the details. This section provides such a view of the linguistic aspects of the VHDL language.

2.6.1 Design

A design consists of an entity and an architecture. An entity declaration defines the interface of the design, while the architecture provides the operation of the design.

2.6.2 Analysis

Before a design is simulated, it must be analyzed. The analysis phase performs language syntax and semantic checks and translates the VHDL source file into an intermediate format. The source file is said to compile into a specific design library.

2.6.3 Library

A library is where designs and utilities reside. When a design is compiled, it will be placed in a library that the user specifies. In addition, there are libraries of standard definitions and packages that a design may use. These utilities are pre-compiled and are supplied by tool provides as part of their simulation or synthesis environment.

2.6.3.1 WORK Library. The default working library is called the WORK library. By default, all designs are compiled into this library. Designs in this library are referred to by preceding the keyword WORK with the design entity name.

2.6.3.2 STD Library. The STD library is part of the VHDL language and contains all basic definitions and types of the language.

2.6.3.3 IEEE Library. The IEEE library has packages of standard types and utilities for arithmetic, text, and complex mathematical processings.

2.6.4 Standard Packages

Standard packages are placed in standard libraries. They contain utility types, functions, and procedures. These packages are provided in compiled form by the tool vendors.

2.6.4.1 STANDARD Package. The STANDARD package defines BIT, INTEGER, TIME and other standard types of VHDL.

2.6.4.2 TEXTIO Package. The TEXTIO package resides in the STD library and contains procedures and functions for formatted ASCII input and output from and to external files.

2.6.4.3 *std_logic.1164* Package. The *std_logic_1164* package resides in the IEEE library. This package defines the industry standard *std_logic* type, and functions and procedures for using this type in designs. Basic operators of VHDL for the *std_logic* type are defined in this package.

2.6.4.4 *std_logic_unsigned* Package. The *std_logic_unsigned* resides in the IEEE library. This package contains functions and procedures for unsigned arithmetic. VHDL arithmetic operators are defined in this package.

2.6.5 Elaboration

After a design is successfully analyzed, the elaboration phase begins. In this phase, design libraries and components are put together and a simulatable model is created. Static parameters are defined in this phase.

2.6.6 Event Driven Simulation

VHDL simulation is event driven. This means that all components of a design are concurrently alive, and events on ports of components cause their activation. An event is regarded as a change in value of a signal.

2.6.7 Concurrency

VHDL is a concurrent language. This means that in its concurrent bodies, the order in which concurrent statements appear is not important. Concurrent statements in a concurrent body of VHDL are executed when an event occurs on one of their input ports.

Concurrency is the way the simulation of components or constructs appears to the user. Obviously, VHDL is a language for which simulators have been developed on single processor platforms, and true concurrency in execution of thousands of component cannot exist. Through the use of concurrent constructs, timing of interconnecting signals, and order of simulation of constructs or components, a VHDL simulators makes us (the users) <u>think</u> that such execution is being done concurrently.

2.6.8 Concurrent Bodies

The main concurrent body of VHDL is an architecture. Other concurrent blocks may exist within an architecture. Components instantiated within an architecture and all sub-blocks and subcomponents within these components are considered concurrent. Execution of these components is event based.

Concurrent bodies can only contain concurrent statements. Sequential statements contained in a sequential body become a concurrent statement that can be contained in a concurrent body.

2.6.9 Sequentiality

Sequentiality refers to the statements that are executed one after another. The order in which sequential statements appear in a sequential body is important. Sequentiality is like programs are executed in

a software language. Most sequential statements are similar to those of software languages, i.e., if-then, case and loop statements.

Although hardware components are concurrent, describing the behavioral of a component is more easily done sequentially. Therefore VHDL provides sequential bodies within which sequential statements can be used. A component that is described sequentially runs in parallel and concurrent with other components of a design.

2.6.10 Sequential Bodies

A process statement is the main sequential body of the VHDL language. A process statement is a concurrent statement, but its body is sequential, and only sequential statements can be used in a process statement. Other sequential bodies of VHDL are functions and procedures.

Under certain conditions, and for some sequential bodies, we can use wait statements. A wait statement is a sequential statement. If no wait statement appears in a sequential body, the sequential body runs in zero time.

2.6.11 VHDL Objects and Classes

An entity that has a value of a given type is a VHDL object. Objects may be explicitly declared, on they may be implicitly created in the language. Port signals, loop index variables, signal for interconnecting components, temporary variables, or files are some of the objects in VHDL. An object may be of one of the four classes of *signals*, *variables*, *constants*, and *files*. A subclass of the variable class is the *shared variable* class.

2.6.11.1 Signals. Objects of the signal class represent hardware wires and have timing associated with them. Values assigned to a signal are placed on the signal *driver* and will appear on the signal after a specified delay value. Explicit declaration of a signal may only appear in concurrent VHDL bodies. However, assignments to signals, or using them, can be done in sequential or concurrent bodies. The assignment symbol for signals is <= which has a nonzero time component. The scheduling for assignment of the right hand side to the signal can be specified by use of an AFTER clause.

In addition to explicitly declared signals, there are other signals such as input and output ports and signals carrying history of other signals that do not require explicit declarations. Issues related to timing of signals will be discussed later in this chapter.

2.6.11.2 Variables. Objects of the variable class are for storage of temporary values and have no hardware significance. Variables can be assigned values and their values can be used much the same way as variables in most software programming languages. Variables can only be declared, or assigned values to, in sequential bodies of VHDL. The standard := assignment is used to assign values to variables.

   The shared variable subclass of the variable class can be declared in concurrent or sequential bodies. Loop parameters are of the variable class and need not be explicitly declared.

2.6.11.3 Constants. Objects of the constant class represent constant values of a given type. They can be declared and they are available for use within concurrent and sequential bodies. An explicit constant declared in a concurrent or sequential body of VHDL, consists of the constant name, its type, and its value. The value given to the constant cannot be altered. VHDL provides a mechanism for passing non-hardware parameters for timing or specification of physical characteristics of a component to component descriptions. Such can be done using implicit constants that are referred to as *formal generics*.

2.6.11.4 Files. Another class of objects in VHDL is the file class. As with the constants, files can be declared and used in concurrent and sequential bodies.

2.6.12 Real Time

The smallest time increment in VHDL is femtoseconds. Activities are scheduled in terms of this real time, and value reports are provided at specific real time instances.

2.6.13 Delta Delay

The delta delay is an internal simulation cycle time in VHDL. The delta delay is used by VHDL simulators to hide from us the fact that processings and assignments are done sequentially, and make it appear as if concurrent assignments are really done concurrently.

   With a new real time increment, a VHDL simulator performs concurrent assignments that depend on each other in δ times. When all assignments are complete, it advances the real time. This makes it appear as if all assignments have taken place in a single real time instance, and thus concurrency.

2.6.14 Scheduling

A value assigned to a signal is said to be scheduled for the signal. A scheduled value has a time associated with it, which is the time that the value appears on the signal. A scheduled value is not guaranteed, and can be overwritten by another scheduling.

2.6.15 Resolution

VHDL allows multiple concurrent assignments to signals. However a signal that is to receive multiple assignments must be a resolved signal. A resolved signal has a resolution function associated with its type. A resolution function decides on a single value from multiple concurrent values assigned to a resolved signal.

2.6.16 Code Formal

VHDL is a free-format language. White space is used as separator. VHDL is not case sensitive, however, for clarity, we use upper case for VHDL keywords and reserved words.

Comments in VHDL are marked by two dashes (--). Two dashes anywhere in a line make the rest of the line comment.

2.7 Summary
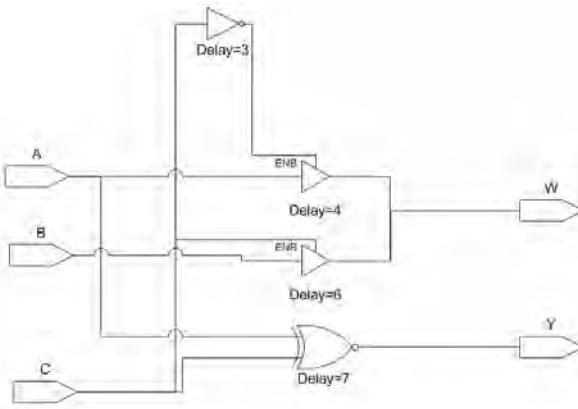
The focus of this chapter was on RT level description in the VHDL HDL language. The chapter used complete design examples at various levels of abstraction for showing ways in which VHDL could be used in a design. The examples that were presented had one-to-one hardware correspondence and were synthesizable. We have shown how combinational and sequential components can be described for synthesis. This introduction is enough for the reader to start coding in VHDL and use this language for simple designs. In addition, this introduction is needed for understanding the conceptual issues that are discussed in the next five chapters. The last part of this chapter introduces some VHDL terminologies that are needed for understanding the linguistics of VHDL in chapters that follow.

Problems

2.1 For the figure shown below, write the necessary code fragments.

      a. Use gate-level modeling (Write architectures for the gates you are using).
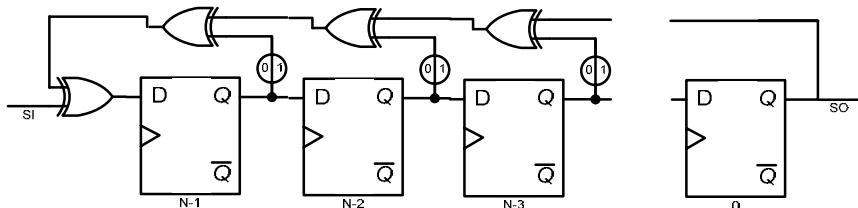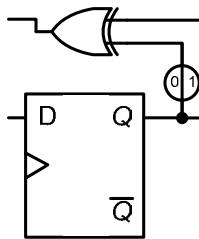
      b. Use concurrent assignment statements.

      c. Use a sequential PROCESS statement.

2.2 Show synthesizable VHDL code for a register unit that performs operations shown below. The unit has a 3-bit mode (*md*) input, an asynchronous reset (*rs*) input, a 1-bit output control (*oc*) input, and an 8-bit bi-directional *io* bus. The internal register drives the *io* bus when *oc* is '1' and *md* is not "111". Use *std_logic*.
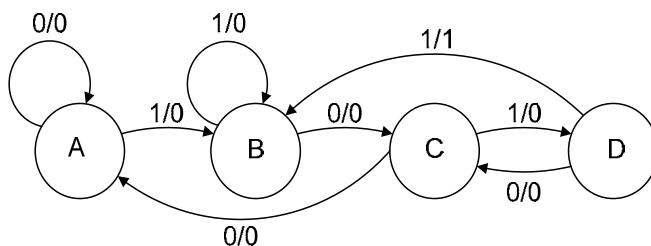
    md=000: does nothing
    md=001: right shift the register
    md=010: left shift the register
    md=011: up count, binary
    md=100: down count, binary
    md=101: complement register contents
    md=110: swap right and left 4 bits
    md=111: parallel load

2.3 Using D-flip-flops, generate an 8-bit LFSR (Linear Feedback Shift-Register). For every bit, include a Binary Control (*BC*) value that can turn the contribution of the flip-flop output to the XOR input on or off (1 for ON, 0 for OFF). For the 8-bit LFSR include a 7-bit generic BIT_VECTOR that can configure contribution of LFSR flip-flops to the LFSR feedback. The right-most flip-flop output has no XOR, and the left-most flip-flop input is fed by the feedback line through the input XOR.

2.4   Show VHDL code for a 4-bit counter that counts the following sequence: 0100, 0001, 1011, 1010, 0110, 1111, 0111, 0000, 1000. After a clock pulse, if the present count is 0001, the next count becomes 1011, and if the present count is 1000 the next count becomes 0100 (roll over).

2.5   Describe the state machine shown below in VHDL.

2.6   Show VHDL model for a memory chip with 8-bit bi-directional data input / output lines, a 16-bit address bus, a *cs* (chip select) and a *we* (write-enable) input. When selected (*cs* is 1) and *we* is '1', the memory drives its output with the memory contents at the specified address. When being written into, *cs* must be '1' while *we* is '0'. When not selected, the outputs remain at high impedance.

a. Write a behavioral VHDL code. You can assume all over-loaded arithmetic functions are available.
b. Generate a testbench to instantiate the memory, apply test data to each memory location read the data back, verify its correctness and continue this for all memory locations. Your test data must be walking '1' test that starts with "10000000" at location 0, changes to "01000000" for location 1, and continues to "00000001" for location 7 and "10000000" for location 8, etc.

2.7   You are to design a processing element that receives its instructions serially through its synchronous *serial_instruction* input. The system has a clock input with which all control and data activities are synchronized. The header for the start of an instruction is 111. An instruction begins after this sequence appears on the *serial_instruction* input. Instructions are two bits. The system has a 16-bit accumulator on which all activities happen, a 16-bit *DataInput* and a 16-bit *DataOutput*. After the start sequence, 000 is for *Reset_AC*, 001 for *Shiftright_AC*, 010 for *Addinput_AC*, 001 is for *IncrementAC*, 100 is for *swaprightleft_AC*, and 101 is for *Complement_AC*. The *AC* output is always available on the *DataOutput*.

a. Show a block diagram for your design.
b. Write synthesizable VHDL description of all your data components (two such components).
c. Write synthesizable VHDL description of your controller.
d. Show the complete wiring and testbench template.

Suggested Reading

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

Brown, Stephen, and Zvonko Vranesic, *Fundamentals of Digital Logic with VHDL Design, 1st edition*, 1999, McGraw-Hill Science/Engineering/Math, ISBN: 978-0072355963.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Navabi, Zainalabedin, *Embedded Core Design with FPGAs*, 2006, McGraw Hill-Professional, ISBN: 0071474811.

3 VHDL Constructs for Structure and Hierarchy Descriptions

This chapter describes methods used in VHDL for describing basic components and using components in upper level structures. The chapter shows various binding methods and use and configuration of generic components in upper level designs.

The chapter begins with basics of architectures and moves into structural descriptions. It then discusses issues related to binding components, generic structures and discussion of specification of timing parameters are discussed next. We limit this discussion to only these constructs that are essential for description of structures and hierarchies of VHDL. Using simple language constructs, we build several simple components and use them to demonstrate hierarchies and upper level design units. The next chapter presents more behavioral language constructs for description of more complex hardware components. Our discussion of hierarchies in this chapter applies to hardware components regardless of their complexity.

3.1 Basic Components

This section presents VHDL description of an inverter and 2-input NAND gate that we will use in subsequent designs in the sections that follow. Examples that follow will be used for demonstration of wiring, binding, configurations, and timing parameter specifications.

3.1.1 Basic Model

Figure 3.1 shows the logical symbol for an inverter, and its VHDL description. The entity declaration of the inverter of Figure 3.1 is further detailed in Figure 3.2.


```
ENTITY inv IS
   PORT (i1 : IN BIT; y : OUT BIT);
END ENTITY;
--
ARCHITECTURE delay1 OF inv IS
BEGIN
   y <= NOT i1 AFTER 3 NS;
END ARCHITECTURE delay1;
```

Figure 3.1 Inverter Symbol and VHDL Description

Figure 3.2 indicates that a port clause is bracketed between the beginning and end indications of an entity declaration. The port clause gives the declaration of all input and output ports of the entity. Two interface signal declarations are used to declare *i1* input and the *o1* output ports of the inverter. The type of *i1* port is BIT and its mode is IN. The *o1* port is also of type BIT and its mode is OUT. The IN and OUT modes for *i1* and *o1* specify that these signals are the input and the output of the inverter. Ports can also have an INOUT mode or BUFFER. INOUT is mainly used for bidirectional lines and defines an input and an output signal. INOUT port may have a multiple number of sources. On the other hand, BUFFER is primarily used for output ports with a single driver which needs to be read in the same body as it is being driven. Type BIT is a predefined VHDL type. In addition to standard types, user-defined types are also allowed. Type definition and usage are discussed in the next chapter.

Figure 3.2 Details of the Entity Declaration of the Inverter
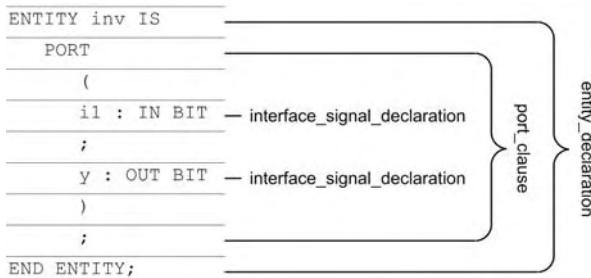
Figure 3.1 also shows the architecture body of *inv*, identified as *delay1*. This architecture describes the internal operation of the inverter. A single signal assignment bracketed between the BEGIN and END keywords constitutes what is referred to as the statement part of the *delay1* architecture of *inv*. This statement sets the complement of *i1* input to *o1* output, with a 4-ns delay. The <= symbol specifies the direction of assignment. When an event occurs on *i1* (*i1* changes value), the complement of the new value of *i1* is scheduled for the *o1* signal 4 ns later. TIME is a predefined type in VHDL, and its units range from femtoseconds (10^{-15} s) to hours. Other time units can also be defined.

Also shown in Figure 3.1 is a graphical representation of the interface description of *inv*. As shown, hollow square boxes are used for input ports, and filled boxes signify the outputs. Other port modes in VHDL are BUFFER and INOUT. Figure 3.3 shows various port modes and the way they are driven. An input (left-most diagram) can only be driven with any number of sources from outside of a component. Outputs can only be driven from inside of a component with zero or more sources. Bidirectional ports can be driven by any number of sources or read from inside and outside. Buffer ports (right-most diagram of Figure 3.3) may be updated by at most one source from inside of a component, and they may be read from inside and outside.
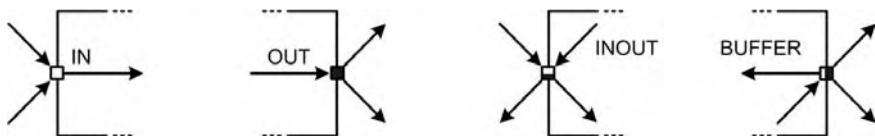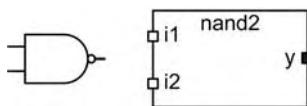
Figure 3.3 Entity Parts, (a) Inputs, (b) Outputs, (c) Bidirectional, (d) Buffers


```
ENTITY nand2 IS
   PORT (i1, i2 : IN BIT; y : OUT BIT);
END ENTITY;
--
ARCHITECTURE delay1 OF nand2 IS
BEGIN
   y <= i1 NAND i2 AFTER 5 NS;
END ARCHITECTURE delay1;
```

Figure 3.4 Two-input NAND: Symbol, Entity Declaration, Architecture Body

Another example of an entity-architecture pair is shown in Figure 3.4. The interface declaration of this *nand2* example has two signals, namely *i1* and *i2*. Figure 3.5 shows the details of the *nand2* port clause.
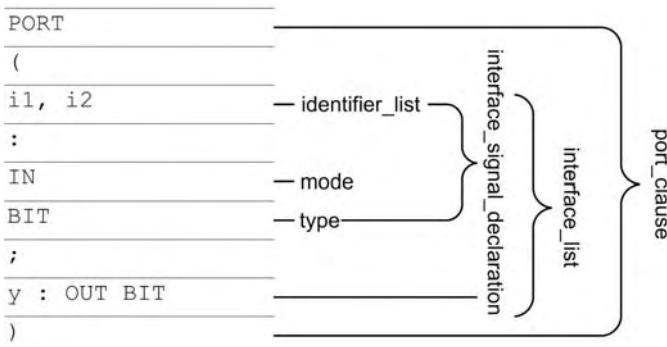
Figure 3.5 Port Clause Details for *nand2*

3.2 Component Instantiations

Wiring primitive structure for generation of larger designs is demonstrated in this section. In VHDL, the operation of a design entity can be described in terms of its subcomponents. To completely specify this operation, we must indicate the component interconnections and link them to a set of available library cells. The main language constructs that support this style of hardware description are signal declarations, component declarations, configuration specifications, and component instantiations. These constructs are discussed here.

For demonstrating the use of these constructs we use a 2-to-1 multiplexer example. The circuit shown in Figure 3.6 uses three *nand2* gates and an *inv*. The *a* input of the circuit is selected when s is **0** and *b* is selected when s is **1**.
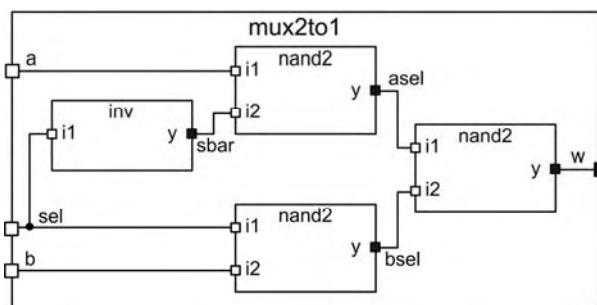
Figure 3.6 *mux2To1* Using Four *nand2*

3.2.1 Direct Instantiation

The simplest form of instantiating lower level components for building upper level structure is direct instantiation. Figure 3.7 shows instantiation of three *nand2* and an *inv* entities for building *multiplexer*. As shown, intermediate signals, *sbar*, *asel*, and *bsel* signals that are used for interconnecting *nand2* and *inv* entities (Figure 3.6), are declared as signals of type BIT in Figure 3.7.

```
ENTITY multiplexer IS
   PORT (a, b, s : IN BIT;
         w : OUT BIT);
END ENTITY;
--
ARCHITECTURE direct OF multiplexer IS
   SIGNAL sbar, asel, bsel : BIT;
BEGIN

   U1: ENTITY WORK.inv (delay1) PORT MAP (s, sbar);
   U2: ENTITY WORK.nand2 (delay1) PORT MAP (a, sbar, asel);
   U3: ENTITY WORK.nand2 (delay1) PORT MAP (b, s, bsel);
   U4: ENTITY WORK.nand2 (delay1) PORT MAP (asel, bsel, w);

END ARCHITECTURE direct;
```

Figure 3.7 Using Direct Instantiations

As shown in Figure 3.7, a *direct component instantiation* begins with an instantiation labeled and after a colon it is followed by the ENTITY keyword, which is then followed by the exact library (i.e., WORK), entity name, and architecture identifier. For the instantiation labeled *u2*, *nand2* is the entity name of the component being instantiated. As shown, the entity name is followed by the architecture identifier in a set of parenthesis. If this architecture identifier is not included, the most recently compiled architecture of *nand2* will be used as default.

3.2.2 Component Instantiation

Instead of direct instantiations shown in Figure 3.7, component instantiation statements may be used for more flexibility in component binding. Figure 3.8 shows this alternative architecture for our multiplexer of Figure 3.6. This figure only shows the *gates* architecture of multiplexer and its entity declaration remains as shown in Figure 3.7.

```
ARCHITECTURE gates OF multiplexer IS
   COMPONENT n1
      PORT (i1: IN BIT; y: OUT BIT);
   END COMPONENT;
   COMPONENT n2
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   FOR ALL : n1 USE ENTITY WORK.inv (delay1);
   FOR ALL : n2 USE ENTITY WORK.nand2 (delay1);
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1: n1 PORT MAP (s, sbar);
   U2: n2 PORT MAP (a, sbar, asel);
   U3: n2 PORT MAP (b, s, bsel);
   U4: n2 PORT MAP (asel, bsel, w);
END ARCHITECTURE gates;
```
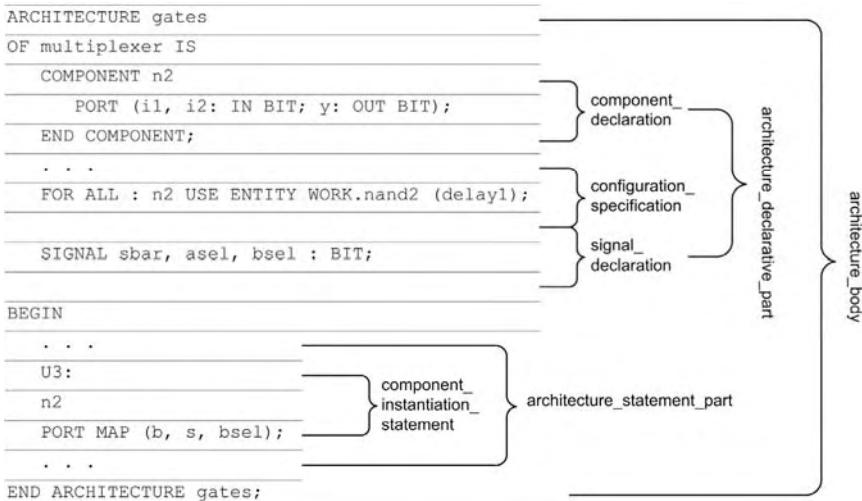
Figure 3.8 Gates Architecture of Multiplexer

Figure 3.9 Syntax Details of the Architecture Body of *multiplexer(gates)*
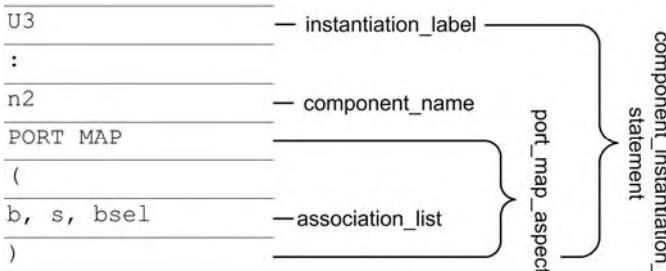
Figure 3.9 shows the syntax details of the architecture body in Figure 3.8. Subcomponents in the gates description of the multiplexer are *inv*, and *nand2*. The declarative part includes a component declaration and a configuration specification for each of these components. Component declarations define the interface to the component in an instantiation statement in the statement part of an architecture, and

configuration specifications associate such an instance with an existing entity. For example, in the case of *n2* component, a component declaration defines the ports of instantiations of *n2* (referred to as local ports) to be the same as those of the *nand2* entity, that is, (*i1*, *i2* : IN BIT; *y* : OUT BIT). The configuration of *n2* specifies that for all instantiations of this component (ALL : *n2*) the *delay1* architecture of the *nand2* entity which exists in the WORK library should be used. In the configuration specification following keyword FOR, the keyword ALL specifies that the association with the specified existing entity applies to *all* instances of that component. If different bindings are to be used for different instances of a component, the list of labels that binding applies to must be used instead of this keyword. Alternatives in the use of component declarations and configuration specifications are discussed later in this chapter. The word WORK in the configuration specification specifies the library in which the *delay1* architecture of the *nand2* entity resides. This is the default library and it refers to the current *work*ing library. Definition of new libraries is discussed later in this chapter.

In addition to the above constructs for defining components, the description in Figure 3.8 and the partial description in Figure 3.9 also include a signal declaration declaring several signals of type BIT. The keyword SIGNAL begins the declaration and is followed by a list of identifiers. BIT, the type indication for the signals, ends the declaration. Signals declared here are used as intermediate signals in the statement part of the *gates* architecture of *multiplexer* and are the same as those used in the diagram of Figure 3.6. Figure 3.9 indicates that the signal declarations are part of the architecture declarative part.

The architecture statement part describes the operation of *multiplexer* in terms of its components. Component instantiation statements include a label, component name, and association between the actual signals that are visible in the architecture body of the *multiplexer* and the ports of the component being instantiated. Syntax details of the instantiation statement shown in Figure 3.9 are given in Figure 3.10. For this statement, *u3* is a label for instantiation of *n2* which is bound to the *nand2* entity.

The mapping of ports specifies that the first two ports of this component are connected to the *b*, and *s* inputs of the *nand2*. These signal names are the primary ports of the *multiplexer* entity and therefore are visible within its *gates* architecture body. The last port of *nand2*, which is its output, is connected to the *bsel* intermediate signal. The next instantiation statement in the statement part of the architecture body of the *multiplexer* (Figure 3.8) uses *bsel* for the second input of a *nand2* component.

Figure 3.10 Component Instantiation Statement Syntax Details

The last statement in Figure 3.8 ends the *gates* structural description of the multiplexer. After a successful analysis of the *gates* architecture of the *multiplexer* and all its subcomponents by a VHDL simulation system, this design entity becomes available in a design library and can be used in other designs. Designs that can use this unit include a testbench and a multibit multiplexer. The next section illustrates the latter.
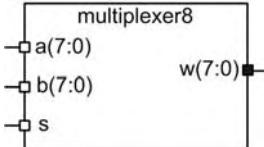
Compared to most hardware netlists, the description in Figure 3.8 is verbose. However, the language constructs in this description provide flexibilities that do not exist in most netlists. A configuration specification associates an instantiated component with an actual entity and architecture from a specified library. A component declaration declares a virtual design interface that defines the way a component can be instantiated locally. A configuration declaration maps a declared component to an actual component. Through the use of these constructs, a description may be bound to various libraries and packages without changing instantiation specifications. If a configuration declaration does not exist for a declared component, it is expected that the name of the component and port names of the component declaration match exactly those of the actual component. Furthermore, since an architecture identifier of an actual component is only specified with a configuration specification, e.g., (*delay1*) in Figure 3.8, in the absence of a configuration specification, architecture for the entity that has most recently been compiled will be used.

3.3 Iterative Networks

In addition to language constructs for declaration, configuration, and instantiation of components, VHDL includes higher-level constructs that can be used for definition of repetitive hardware at the structural level. Such constructs are discussed in this section. The example used is an 8-bit multiplexer and is referred to as *multiplexer8*. This circuit uses the *multiplexer* circuit.

3.3.1 Multi-bit Vectors

Figure 3.11 shows a graphical symbol for the multiplexer8 and its corresponding entity declaration.


```
ENTITY multiplexer8 IS
   PORT (a, b : IN BIT_VECTOR (7 DOWNTO 0); s : IN BIT;
         w : OUT BIT_VECTOR (7 DOWNTO 0) );
END ENTITY;
```
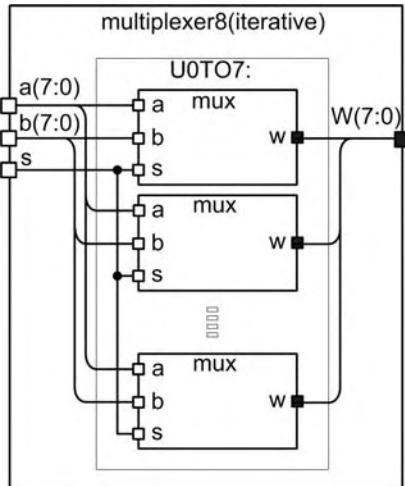
Figure 3.11 Interface Description *multiplexer8*

The entity declaration uses *multiplexer8* as the entity name. The port clause of this declaration has three interface declarations for declaring data inputs, select input, and the output. The *a* and *b* data inputs are 8-bit arrays of bits, and for their declaration the BIT_VECTOR type has been used. As with type BIT, BIT_VECTOR is a predefined type in VHDL and is available in the default standard package. Definition and usage of new packages and types are discussed at the end of this chapter.

3.3.2 Multi-instance Generations

Figure 3.12 shows schematic diagram of our *multiplexer8* circuit. The VHDL description corresponding to this circuit is shown in Figure 3.13. Instead of flatly instantiating all eight instances of *multiplexer* we have used VHDL's *generate statement* for generating multiple instantiations. For each iteration through the generate statement, an instance of the declared component, *mux*, is generated. The configuration specification statement in the declarative part of the generate statement binds each of the generated instances of *mux* to the *multiplexer* circuit of the WORK library (*gates* architecture of *multiplexer*, Figure 3.8).

Figure 3.12 *multiplexer8* Hierarchical Structure

```
ARCHITECTURE iterative OF multiplexer8 IS
   COMPONENT mux PORT (a, b, s : IN BIT; w : OUT BIT);
   END COMPONENT;
BEGIN
   U0TO7: FOR i IN 0 TO 7 GENERATE
      FOR ALL : mux USE ENTITY WORK.multiplexer (gates);
   BEGIN
      Ui: mux PORT MAP (a(i), b(i), s, w(i));
   END GENERATE;
END ARCHITECTURE iterative;
```

Figure 3.13 Iterative Architecture of *multiplexer8*

The syntax details of the generate statement in Figure 3.13 are shown in Figure 3.14. The statement begins with a label (*U0TO7*) and, using a FOR loop for the generation scheme, it generates eight instances of *mux*, that are all labeled *ui*. The region immediately after the GENERATE keyword is the generate statement declarative part. This region follows the syntax of the architecture declarative part and is referred to as *block_declarative_item*. With each iteration, the block-declarative-item shown here provides the *ui* instance of *mux*.
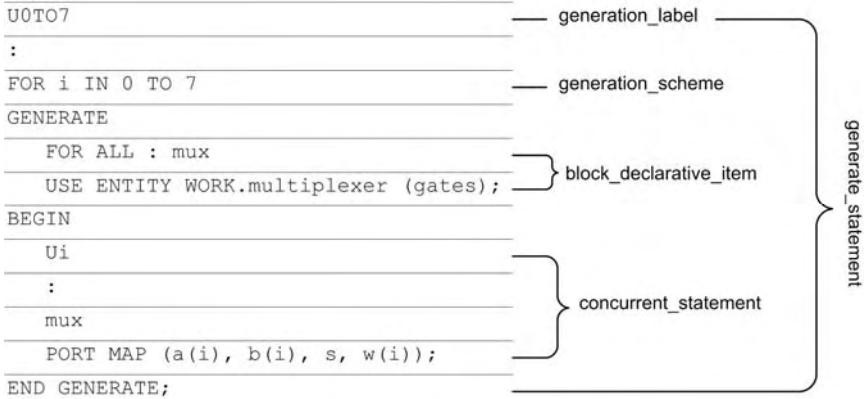
Figure 3.14 Generate Statement Syntax Details

There is also an *if generation* scheme in VHDL. This construct is used to separate out some instances from others. This becomes especially useful for iterative structures that have different interconnection schemes for first and last instantiation. An example is a series of full-adder where the first one uses carry-in, the last one uses carry-out, and all full-adders in between use intermediate carry signals.

3.3.3 Simplified Generations

Figure 3.15 shows a simplified version of the 8-bit multiplexer. In this description, a generate statement generates eight direct instances of *multiplexer*. Because direct instantiations do not require binding, the declarative part of the generate statement is not necessary.

```
ARCHITECTURE direct OF multiplexer8 IS
BEGIN
   U0TO7: FOR i IN 0 TO 7 GENERATE
      Ui: ENTITY WORK.multiplexer
          PORT MAP (a(i), b(i), s, w(i));
   END GENERATE;
END ARCHITECTURE direct;
```

Figure 3.15 Generating Direct Instantiation

3.4 Binding Alternatives

Configuration specifications in the architecture declarative part of a VHDL description associate an instance of a component with a design entity. So far in this chapter, we have used the combinations of *component declarations*, *configuration specifications*, and *component in-*

stantiation in their simplest forms. VHDL, however, allows other forms of configuration specifications that make component instantiations significantly more flexible than presented thus far.

For the purpose of illustration, consider the multiplexer circuit of Figure 3.6. The VHDL description corresponding to this diagram is shown in Figure 3.8. This description uses an inverter and four NAND gates. Alternatively, a NAND gate with two inputs tied together may be used instead of the *u1* inverter. This can be done either by changing *u1* instance to *n2* and using *s* for both gate inputs, or by keeping the *n1* component declaration and instantiation as is, and binding *u1* instance of *n1* to a NAND gate instead of an inverter. Figure 3.16 shows the latter alternative.

```
ARCHITECTURE alter OF multiplexer IS
   COMPONENT n1
      PORT (i1: IN BIT; y: OUT BIT);
   END COMPONENT;
   COMPONENT n2
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   FOR U1 : n1                               -- Line 8
      USE ENTITY WORK.nand2 (delay1) PORT MAP (i1, i1, y);
   FOR ALL : n2 USE ENTITY WORK.nand2 (delay1);
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1: n1 PORT MAP (s, sbar);
   U2: n2 PORT MAP (a, sbar, asel);
   U3: n2 PORT MAP (b, s, bsel);
   U4: n2 PORT MAP (asel, bsel, w);
END ARCHITECTURE alter;
```

Figure 3.16 Configuration Specification Port Map

The *alter* architecture of *multiplexer* of Figure 3.16 has *n1* and *n2* component declarations. As discussed before, these declarations specify how the declared components are used (instantiated) locally in this architecture. As required by declaration on *n1*, *u1* instance of *n1* uses one input and one output. Then, configuration specification that is used to bind *n1* to an actual component outside of this architecture, binds the *u1* instance of *n1* to the *delay1* architecture of *nand2*.

As stated before local port declaration (those of component declarations) are used if a binding indication does not include a port map. However, since the *nand2* entity that we are binding *n1* to, has different ports than those of *n1* declaration, a port map is used in the configuration specification of *u1* instance of *n1*. As shown on line 9 of Figure 3.16, the port map aspect shown, maps *i1* that is the declared port of *n1* to the first two ports of *nand2*. Note that the first two ports

of this entity are its two inputs. This same construct also binds the declared *y* port of *n1* to the third port of *nand2*, which becomes the output of this gate.

We refer to this method of port association as a two-step association. As shown in Figure 3.17, the first step associates *s* to *i1* and *sbar* to *y*, where *i1* and *y* are local declaration ports. The second step associates this *i1* to *i1* and *i2* of *nand2* and this local *y* to *y* of *nand2*.
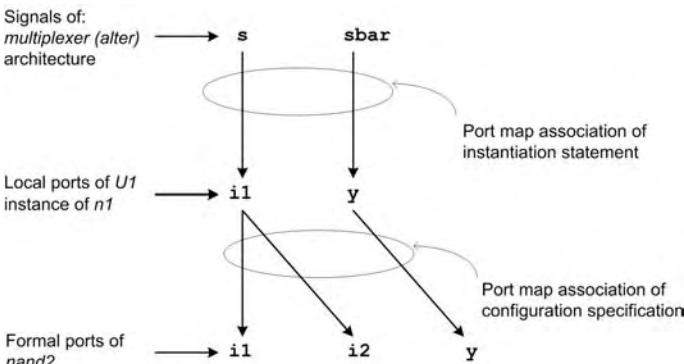
Figure 3.17 Two-step Associations

3.5 Association Methods

Examples discussed thus far use what is referred to as *association by position* for port associations. This means that ports are connected by the order that they appear in the port map aspect. See, for example, to *i1* ports connecting to the first two ports of *nand2* in the example of Figure 3.16. In this association, *i1* connects to the ports of *nand2* regardless of what port names *nand2* uses. However, since names are not used, the ordering must be correct.

An alternative port association is *association by name*. This association scheme uses actual names of signals to which local signals are being connected to. For example, the port map aspect on line 9 of Figure 3.16 can be rewritten as shown below.

```
PORT MAP (i1 => i1, i2 => i1, y => y);
```

In this statement, *i1*, *i2*, and *y* of *nand2* are in bold, and *i1* and *y* are the local signals. As shown, *i1* and *i2* of *nand2* connect to the local *i1*, and *y* of *nand2* connects to the local *y*.

VHDL reads "*w => y*" as *y* is associated with *w*.

3.6 Generic Parameters

Component models can be parameterized to better utilize gate or component models, and to make general models usable in different design environments. The specific behavior of these models is dependent on the parameters that are determined by the design entities that use them. VHDL generic parameters can be used for this purpose. For example, a generic parameter can be used for timing and delay of a generic gate model. When this gate is used in a specific design environment, its generic parameters are determined. Generic parameters are used and values are passed to these parameters in much the same way as with ports. The syntax of constructs related to ports and generics are similar, except that the generic clause and generic map aspect constructs use the keyword GENERIC instead of PORT. In general, generics are a means of communicating non-hardware and non-signal information between designs.

A NAND gate and an inverter with generic parameters are shown in Figure 3.18 to illustrate specification and usage of the generic language construct. The entity declarations for these gates include a generic clause and a port clause, as shown in Figure 3.19. The generic clause in each of the gates in Figure 3.18 consists of a generic interface list which contains interface constant declarations for *tplh* and *tphl*. For the inverter, these generic parameters have default values of 5 and 3 ns, respectively. The default values are useful for simplifying upper level structures.

```
ENTITY inv_t IS
   GENERIC (tplh : TIME := 5 NS; tphl : TIME := 3 NS);
   PORT (i1 : IN BIT; y : OUT BIT);
END ENTITY;
--
ARCHITECTURE delay2 OF inv_t IS
BEGIN
   y <= '1' AFTER tplh WHEN (NOT i1) = '1' ELSE
        '0' AFTER tphl;
END ARCHITECTURE delay2;
-- --
ENTITY nand2_t IS
   GENERIC (tplh : TIME := 6 NS; tphl : TIME := 4 NS);
   PORT (i1, i2 : IN BIT; y : OUT BIT);
END ENTITY;
--
ARCHITECTURE delay2 OF nand2_t IS
BEGIN
   y <= '1' AFTER tplh WHEN (i1 NAND i2) = '1' ELSE
        '0' AFTER tphl;
END ARCHITECTURE delay2;
```

Figure 3.18 Entity Declarations with Generic Parameters
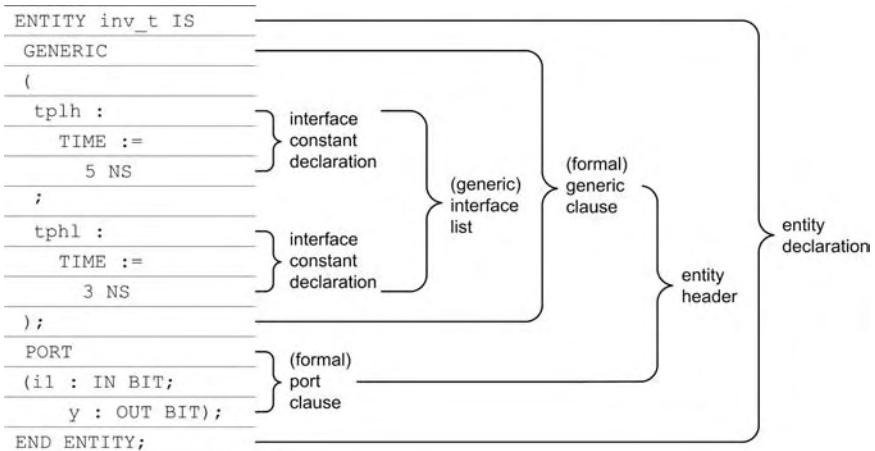
Figure 3.19 Details of the Entity Declaration of the Inverter with Generics

3.6.1 Using Generic Default Values

Generic parameters for which default values exist, e.g., those of Figure 3.18, can be left open. In which case the default values will be used for the generic parameters. An example of this case is shown in Figure 3.20.

```
ARCHITECTURE default OF multiplexer IS
   COMPONENT n2
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   FOR ALL : n2 USE ENTITY WORK.nand2_t (delay2);
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1: n2 PORT MAP (s, s, sbar);
   U2: n2 PORT MAP (a, sbar, asel);
   U3: n2 PORT MAP (b, s, bsel);
   U4: n2 PORT MAP (asel, bsel, w);
END ARCHITECTURE default;
```

Figure 3.20 Default Values for Generic Parameters

The *n2* component declaration in Figure 3.20, that is a local declaration, and the format of which all instantiations must follow, makes no mention of any generic parameters. Instantiations *U1* through *U4* use only PORT MAP which is how *n2* is declared. The configuration specification in this architecture of *multiplexer* binds all *n2* instances to *delay2* architecture of *nand2_t* (Figure 3.18). Since *nand2_t(delay2)* has default values for *tplh* and *tphl* generics, and

multiplexer(default) does not overwrite them, 6 ns and 4 ns will be used for *tplh* and *tphl*, respectively.

3.6.2 Generic Map Aspect

Constants or other generics may be associated with the formal generic parameters of a component. This is done in much the same way that signals are associated with the formal ports of components. Figure 3.21 shows another architecture for the *multiplexer*. In the declarative part of this architecture, the declaration of *n2* includes local generic clauses as well as local port clauses. An instantiation of this component in the statement part of the *fixed* architecture of the *multiplexer* contains a generic map aspect and a port map aspect (Figure 3.22).

```
ARCHITECTURE fixed OF multiplexer IS
   COMPONENT n2
      GENERIC (tplh, tphl : TIME);
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   FOR ALL : n2 USE ENTITY WORK.nand2_t (delay2);
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1:n2 GENERIC MAP (5 NS, 3 NS) PORT MAP (s, s, sbar);
   U2:n2 GENERIC MAP (5 NS, 3 NS) PORT MAP (a, sbar, asel);
   U3:n2 GENERIC MAP (5 NS, 3 NS) PORT MAP (b, s, bsel);
   U4:n2 GENERIC MAP (5 NS, 3 NS) PORT MAP (asel, bsel, w);
END ARCHITECTURE fixed;
```

Figure 3.21 Using Generic Map Aspect
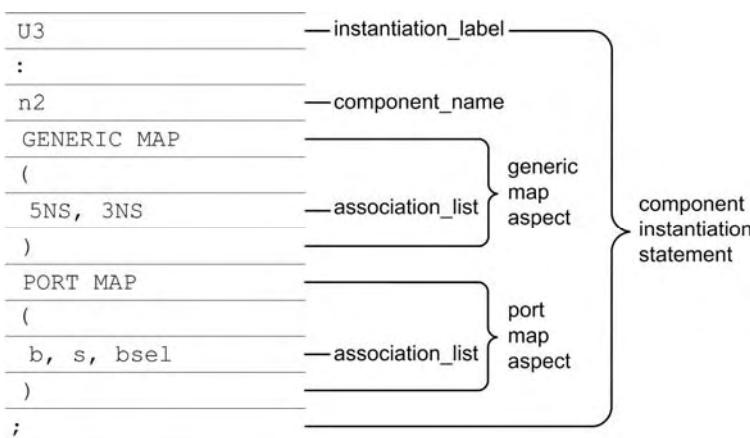
Figure 3.22 Component Instantiation Statement with a Generic Map Aspect

The values used in the association lists of the generic map aspects shown in Figure 3.22 are associated with the formal generics *nand2_t*. These values are used in place of the default values in the generic interface list in the entity declaration for these entities. Since the component declaration shown (*n2*) does not include default values to be associated with the formal generics of these components, inclusion of generic map aspects with instantiations of these components is required.

3.6.3 Generic Association List

The syntax used for the association list of generics and ports are similar, and for both association by position and association by name are possible. Association by position is easier and less verbose, while association by name is less error-prone. Furthermore, for situations that some generic parameters are to be specified and for others default values are to be used, association by name offers a more convenient format.

As an example of generic mapping by association by name, consider component instantiation shown below. In this example, *tplh* of *n2* is given 4.5 ns and *tphl* that is not specified uses the default value of 7 ns (*nand2_t* in Figure 3.18).

```
U3: n2 GENERIC MAP (tplh => 4.5 ns)
        PORT MAP (y => bsel, i1 => b, i2 =>s);
```

The above example also maps ports of *n2* by association by name. This association has the same result as the *u3* instance of *n2* shown in Figure 3.21.

Not specifying a generic or a port in association by name is interpreted as leaving it *open*. Using the keyword OPEN in association by position in place of a generic or a port has the same effect. Mappings of *u3* instance of *n2* as shown below are the same as those of the previous example of this component instantiation.

```
U3: n2 GENERIC MAP (OPEN, 4.5 ns)
        PORT MAP (y => bsel, i1 => b, i2 =>s);
```

We have presented examples for specifying gate delay values in order to demonstrate the use of generics. Other uses of these parameters include passing fan-ins and fan-outs, load resistance or capacitance, and properties such as count sequences and signatures. For example seed and polynomial of an LFSR may be passed to a general purpose signature register through its generic parameters.

3.7   Design Configuration

Binding a component instantiation to an actual component, as described earlier in this chapter, does not have to be done in the architecture that uses component. This binding can be deferred until later and accomplished by a configuration declaration. Therefore, it is possible to generate a generic design and specify the details of timing or a specific component library at a later stage. This way, a generic design can be tested for various logic families, or a single test bench can be used to test various versions of the same component. By use of configurations, trying different descriptions of a component in an upper-level design can easily be done even at the deepest level of nesting.

3.7.1   Basic Configuration Declaration

Instead of using configuration specifications to bind an instance of a component to an entity-architecture pair, a configuration declaration can pair an entity and an architecture into a *configuration*. In addition, as with configuration specification, a configuration declaration can be used for specifying port maps, generic maps, and libraries. As a simple example of this feature of the VHDL language, consider the *configurable* architecture of *multiplexer* shown in Figure 3.23.

```
ARCHITECTURE configurable OF multiplexer IS
   COMPONENT n2
      GENERIC (tplh, tphl : TIME := 4 NS);
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1: n2 PORT MAP (s, s, sbar);
   U2: n2 PORT MAP (a, sbar, asel);
   U3: n2 PORT MAP (b, s, bsel);
   U4: n2 PORT MAP (asel, bsel, w);
END ARCHITECTURE configurable;
```

Figure 3.23 c*onfigurable* **Architecture of** *multiplexer*

The architecture shown in this figure declares *n2* as a component with generic parameters and ports as shown. It then instantiates four *n2* instances. Since there is no configuration specification statement in this architectures, and because *n2* (which is a local declaration) is not an entity in our WORK library, this architecture has unbound component instances and cannot be simulated.

Configuration declaration of Figure 3.24 can be used to correct this problem by binding all instances of *n2* to *delay2* architecture of

nand2_t. In addition, this VHDL construct is used here to specify generic parameters of *nand2_t* as part of its component configuration.

```
     CONFIGURATION configured OF multiplexer IS
       FOR configurable
         FOR ALL : n2
            USE ENTITY WORK.nand2_t (delay2)
            GENERIC MAP (5 NS, 3 NS);
         END FOR;
       END FOR;
     END CONFIGURATION configured;
```

Figure 3.24 Basic Configuration Declaration

The configuration declaration of Figure 3.24 is identified as *configured*. There are three levels of nestings in this description, and they are numbered accordingly. Note that these numbers are for illustration only and are not part of the code. Level 3 is the bracketing of the configuration declaration. The second level is for obtaining visibility into the configurable architecture of *multiplexer*, and Level 1 constitutes the binding of ALL n2 instances to actual components.

Configuration declarations that configure the same architecture-entity pair and use different identifiers can coexist in the same library. As an example, consider the *another_configured* configuration declaration of Figure 3.25. As shown, this body associates *n2* instances of *multiplexer* with the *delay2* architecture of *nand2* and does not use a generic map aspect. An upper-level structure using this configuration must use the following binding indication.

```
CONFIGURATION another_configured OF multiplexer IS
   FOR configurable
      FOR ALL : n2
         USE ENTITY WORK.nand2_t (delay2);
      END FOR;
   END FOR;
END CONFIGURATION another_configured;
```

Figure 3.25 Another Configuration Declaration for Multiplexer

The syntax of a configuration declaration construct is shown in Figure 3.26.

```
CONFIGURATION
configured                    — indentifier
OF
multiplexer                   — entity_name
IS
    FOR configurable
        FOR ALL : n2
            USE
            ENTITY
            WORK.nand2_t (delay2)    binding_
            GENERIC                  indication
            MAP (5 NS, 3 NS)
            ;
        END FOR;
    END FOR;
END CONFIGURATION configured;
```

configuration_declaration

component_configuration

blobk_configuration
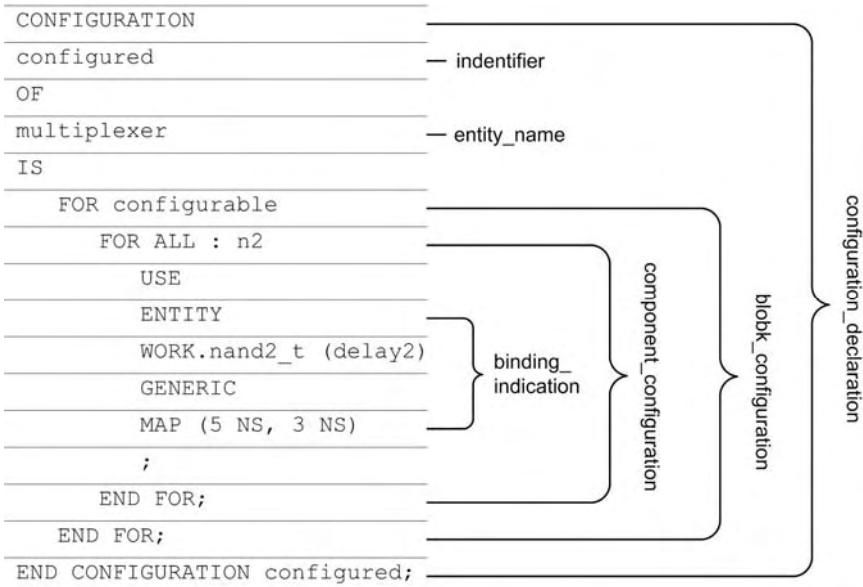
Figure 3.26 Details of Configuration Declaration

The heading in this statement specifies an identifier for the configuration declaration and the entity to be configured. This is followed by a block configuration within which the scope is limited to the statement part of the *configurable* architecture of the *multiplexer*. In this block configuration, binding for all visible components can be specified by individual component configurations. We have used a single component configuration for binding all *n2* instances. Component configurations begin with the FOR keyword followed by component specifications. Following the USE keyword, a binding indication is used to associate all instances of *n2* to the *delay2* architecture of the *multiplexer*. As shown in Figure 3.26, there is only a minor difference between the syntax of component configurations and the configuration specifications. The former requires the END FOR keywords, while the latter, which is used in the declarative part of the architectures, does not. Component configurations, however, offer more flexibility, and many levels of component nestings can be specified within these constructs.

3.7.2 Incremental Configuration

Configuration declarations can be used in conjunction and on top of configuration specifications. The former incrementally adds or overwrites some of the bindings of the latter. Figure 3.27 shows an archi-

tecture for *multiplexer* and its corresponding configuration declaration.

```
ARCHITECTURE reconfigurable OF multiplexer IS
   COMPONENT n2
      GENERIC (tplh, tphl : TIME := 4 NS);
      PORT (i1, i2: IN BIT; y: OUT BIT);
   END COMPONENT;
   --Primary Binding:
   FOR ALL : n2 USE ENTITY WORK.nand2_t (delay2);
   SIGNAL sbar, asel, bsel : BIT;
BEGIN
   U1: n2 PORT MAP (s, s, sbar);
   U2: n2 PORT MAP (a, sbar, asel);
   U3: n2 PORT MAP (b, s, bsel);
   U4: n2 PORT MAP (asel, bsel, w);
END ARCHITECTURE reconfigurable;
--
CONFIGURATION reconfigured OF multiplexer IS
   FOR reconfigurable
      FOR ALL : n2
         GENERIC MAP (5 NS, 3 NS);
      END FOR;
   END FOR;
END CONFIGURATION reconfigured;
```

Figure 3.27 Incremental Binding

The *reconfigurable* architecture of *multiplexer* of this example uses a configuration specification as its primary binding to associate all instances of *n2* with *nand2_t(delay2)*. Further down in Figure 3.27, the *reconfigured* configuration declaration of *multiplexer* adds a generic map aspect to the primary binding. This generic map aspect specifies 5 ns and 3 ns for the *tplh* and *tphl* of *nand2_t*.

3.7.3 Configuring Nested Components

Initially, a designer may use components based on their functionality rather than timings or specific technology. These designs may be used in upper-level designs, which may cause them to be buried under several levels of design hierarchy. Configuration declarations can be extended beyond configuring components of the immediate architectures and can be used to configure several levels of component nestings.

Consider for example, *default*, *fixed*, *configurable*, and *reconfigurable* architectures of *multiplexer*. These architectures may use *nand2* or *nand2_t* with various delay values. Any of these multiplexers can be used in an 8-bit multiplexer. VHDL allows the formation of

the 8-bit multiplexer without specifying the exact multiplexer architecture to use, and without specifying NAND gates to use for the multiplexers. Figure 3.28 shows such an 8-bit multiplexer.

```
ARCHITECTURE multiconfig OF multiplexer8 IS
   COMPONENT mux
      PORT (a, b, s : IN BIT; w : OUT BIT);
   END COMPONENT;
BEGIN
   U0TO7: FOR i IN 0 TO 7 GENERATE
      Ui: mux PORT MAP (a(i), b(i), s, w(i));
   END GENERATE;
END ARCHITECTURE multiconfig;
```

Figure 3.28 Unspecified Components and Sub-components

Note in Figure 3.28 that there is no binding indication for eight generated instances of *mux*. Furthermore, if we decide that *mux* is actually one of the multiplexers of this chapter, the bindings for the gates of the multiplexers are yet to be decided.

Figure 3.29 shows a multi level configuration. The block configuration that begins with *FOR multiconfig* gives visibility to the *multiconfig* architecture of *multiplexer8* of Figure 3.28. A generate statement, that itself is considered a block, is in this architecture. The block configuration of Figure 3.29 that begins with *FOR U0TO7* gives visibility to the inside of the generate statement of Figure 3.28. The *u1* instance of *mux* appears in this block (the generate statement).

```
01: CONFIGURATION multiconfiged OF multiplexer8 IS
02:    FOR multiconfig
03:       FOR U0TO7
04:          FOR ALL : mux
 :           USE ENTITY WORK.multiplexer (configurable);
05:             FOR configurable
06:                FOR ALL : n2
07:                   USE ENTITY WORK.nand2_t (delay2)
08:                   GENERIC MAP (5 NS, 3 NS);
09:                END FOR;
10:             END FOR;
11:          END FOR;
12:       END FOR;
13:    END FOR;
14: END CONFIGURATION multiconfiged;
```

Figure 3.29 Multi-level Configuration Declaration

The component configuration on line 4 of Figure 3.29 specifies that the *configurable multiplexer* of Figure 3.23 will be used for eight generated *mux* instantiations.

Line 5 of Figure 3.29 is a block configuration that makes the architecture body of *configurable* architecture of *multiplexer* visible. As shown in Figure 3.23, this architecture has four instances of *n2*. Line 6 in Figure 3.29 starts a component configuration that binds the instances of *n2* in Figure 3.23 to *nand2_t (delay2)* and specifies its generic parameters.

Figure 3.30 shows syntax constructs contained in configuration declaration of Figure 3.29. The purpose of each construct is specified here. For example it shows that the construct that begins on line 3 and end on line 12 is a block configuration that gives visibility to the *U0TO7* generate block of Figure 3.28.

Starts and Ends in Line	Language Structure	Visibility to:	Binding to:
1-14	Configuration Declaration	-	-
2-13	Block Configuration	*multiplexer8 (multiconfig)* Architecture	-
3-12	Block Configuration	*U0TO7:* Generate	-
4-11	Component Configuration	-	*multiplexer (configurable)*
5-10	Block Configuration	*multiplexer8 (multiconfig)* Architecture	
6-9	Component Configuration	-	*nand2_t (delay2)*

Figure 3.30 *multiconfiged* **Configuration Declaration Language Structures**

3.7.4 Indexing Block Configurations

A block configuration that applies to a generate statement can be indexed to specify specific iterations of the generate statement. For example, in Figure 3.31 *FOR U0TO7(0)* block configuration makes visible the body of the generate statement with iteration 0 and all bindings in this block apply to this iteration. Further down in the code of Figure 3.31, the *FOR U0TO7(1 to 7)* block configuration is used for bindings of 1 to 7 iterations of *ui* instance of *mux*.

```
CONFIGURATION differentlyconfiged OF multiplexer8 IS
   FOR multiconfig
      FOR U0TO7 (0)
         FOR Ui : mux
         USE ENTITY WORK.multiplexer (configurable);
            FOR configurable
               FOR U3 : n2
                  USE ENTITY WORK.nand2_t (delay2)
                  GENERIC MAP (3 NS, 2 NS);
               END FOR;
               FOR OTHERS : n2
                  USE ENTITY WORK.nand2_t (delay2)
                  GENERIC MAP (4 NS, 6 NS);
               END FOR;
            END FOR;
         END FOR;
      END FOR;
      FOR U0TO7 (1 TO 7)
         FOR Ui : mux
         USE ENTITY WORK.multiplexer (configurable);
            FOR configurable
               FOR ALL : n2
                  USE ENTITY WORK.nand2_t (delay2)
                  GENERIC MAP (8 NS, 5 NS);
               END FOR;
            END FOR;
         END FOR;
      END FOR;
   END FOR;
END CONFIGURATION differentlyconfiged;
```

Figure 3.31 Indexing Configurations for Generate Statements

3.7.5 Instantiating a Design Unit

Instantiation statements are used for using a design unit in an upper level design. As previously discussed, instantiation of a component can be done by direct instantiation or by instantiating a declared component and providing binding for it. Regardless of the method used, an actual component to which binding is done can be an entity-architecture pair, or a configuration declaration. Figure 3.32 shows a testbench that tests various multiplexers that we developed in this chapter. Direct instantiations are used for all designs in this test-bench. As shown, *UUT1* through *UUT5* instantiate multiplexer entity architecture pairs, while *UUT6* and *UUT7* instantiate configuration declarations that are written on top of the multiplexer entity.

```
ENTITY multiplexer_tester IS
END ENTITY;
--
ARCHITECTURE timed OF multiplexer_tester IS
   SIGNAL a, b, s, w1, w2, w3, w4, w5, w6, w7 : BIT;
BEGIN
   UUT1:ENTITY WORK.multiplexer(direct) PORT MAP(a,b,s,w1);
   UUT2:ENTITY WORK.multiplexer(gates) PORT MAP(a,b,s,w2);
   UUT3:ENTITY WORK.multiplexer(alter) PORT MAP(a,b,s,w3);
   UUT4:ENTITY WORK.multiplexer(default)PORT MAP(a,b,s,w4);
   UUT5:ENTITY WORK.multiplexer(fixed) PORT MAP(a,b,s,w5);
   UUT6:CONFIGURATION WORK.configured PORT MAP(a,b,s,w6);
   UUT7:CONFIGURATION WORK.reconfigured PORT MAP(a,b,s,w7);
   a <= '0', '1' AFTER 020 NS,'0' AFTER 065 NS,
            '1' AFTER 179 NS;
   b <= '1', '0' AFTER 045 NS, '1' AFTER 105 NS,
            '0' AFTER 215 NS;
   s <= '0', '1' AFTER 129 NS, '0' AFTER 211 NS,
            '1' AFTER 245 NS;
END ARCHITECTURE timed;
```
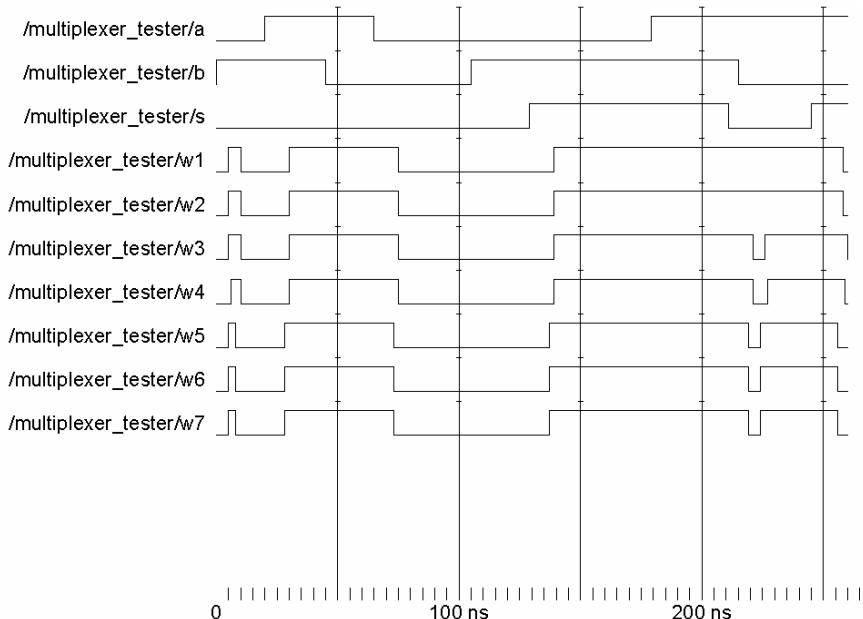
Figure 3.32 Instantiating Design Units

3.8 Design Simulation

A design that is compiled and fully configured without unbound components, can be simulated. The design unit for simulation can be a testbench architecture-entity pair such as that of Figure 3.32, or a configuration declaration that configures a testbench. In either case, the simulation model becomes a design entity in VHDL's library that a simulation tool can run. A top-level design entity for simulation cannot have ports. We end this chapter by discussing simulation of the multiplexer-tester and showing its simulation results.

After successful compilation of *timed* architecture of *multiplexer_tester* of Figure 3.32, our VHDL simulation tool will show this entity-architecture pair as a design unit in its WORK library. Simulation begins by selecting this design unit and running the simulation until it stops. Simulation continues until there are no more events occurring in the design being simulated. Since we have limited data on *a*, *b*, and *s*, this simulation ends a few nanoseconds after the last change on an input occurs.

Figure 3.33 shows simulation run result of testbench of Figure 3.32. In all eight designs *s* selects *a* or *b*, and with a small delay *w1* through *w7* outputs follow *a* or *b* depending on *s*. As shown, various output delays are different and in some cases glitches of varying durations appear on some outputs and not all outputs.

Figure 3.33 Multiplexer Simulation Report

Note here that we are simulating the various gate level architectures of the multiplexer, and delays and glitches depend on specific timing values that are given to the gates. As expected, a NAND design of a multiplexer that uses complementary reconvergent paths to the output has potential hazards. In the designs being tested in Figure 3.32, some outputs have hazards and some do not. It is recommended that an interested reader would follow the gate level structures and delay values used for *UUT7* and *UUT1* to understand why glitch appears on *w7* and not on *w1*.

3.9 Summary

This chapter focused on formation and configuration of upper level structures based on lower level design units. The lower level design example we used was a NAND gate, and we formed a multiplexer based on that. Various forms of component specifications, configurations, and delay parameter specifications were illustrated. We used simple examples to avoid discussion of more complex language constructs and focus only on the structural descriptions. However, all topics covered here apply to specification and configuration of designs that are made of simple gate-level components or complex system-level cores.

Problems

3.1   Problems 1, 2 and 3 are related; read all three before you start coding. In this problem you are to write a model for a T flip-flop using *std_logic*. The flip-flop toggles on the falling edge of its *clk* input when its *T* input is '1'. When *T* is '0', the flip-flop state does not change. It has t_{CQ1} and t_{CQ0} timing parameters that are for the output changing to '1' and '0' respectively. The default values for these parameters are 3 ns and 5 ns and are passed to the flip-flop via its generic parameters. Another generic parameter (t_{Cmin} , with a default of 30 ns) passes the minimum clock period to the flip-flop. Write the complete VHDL model of this flip-flop considering the output delay values. Generate a timing check so that if the clock period falls below the minimum allowed, t_{Cmin}, the flip-flop output becomes 'X'.

3.2   Using the flip-flop of Problem 3.1 in the structure shown below generate an n-bit unconstrained binary counter. Use generate statements; use the AND operation for the AND gate. The size of your counter must be adjustable when it is instantiated within an upper level architecture. The right-most inputs of this structure must be tied to '1' for it to work. You can also use the right-most inputs as active-high master-enable inputs for the entire counter.

3.3   Write a configuration declaration on top of your counter of Problem 3.2 to overwrite the timing parameters of the individual flip-flops (t_{CQ1}, t_{CQ0} and t_{Cmin}) to 3.5 ns, 5.5 ns and 33 ns.

3.4   Using only XOR gates, write a VHDL description for an 8-bit even/odd parity checker. The circuit has an 8-bit input vector and two outputs. The *odd* output is to become 1 when the number of 1s on the input in odd. The *even* output is the opposite of the *odd* output. Use generate statement (s).

3.5   A JK flip-flop with the following interface declaration is given:

```
ENTITY jkff IS
  GENERIC (tplh, tphl : TIME);
  PORT (j, k, c : IN BIT; q : OUT BIT);
END jkff;
```

   a. Wire 8 of these flip-flops using a generate statement to form an eight-bit D register. For each flip-flop, the D input connects to the J input and its complement connects to the K input. Use NOT operation for complementing the D inputs.
   b. On top of the D register, write a configuration declaration to bind the internal JK flip-flops of the 8-bit D register to *behav-*

ioral architecture of a flip-flop with the same name, generics, and ports in the *work* library.

3.6   In the following description, the configuration specification is missing. Assume that the only component that you have available is the *nand3 (single_delay)* model. Write appropriate configurations such that this description implements function *f(a,b,c,d,e)*.

$$f(a, b, c, d, e) = a \cdot b + c \cdot d' + e$$

```
ENTITY function_f IS
   PORT (a, b, c, d, e : IN BIT; f : OUT BIT);
END function_f;

ARCHITECTURE configurable OF function_f IS
   COMPONENT n1 PORT (w: IN BIT; z: OUT BIT);
   END COMPONENT;
   COMPONENT n2 PORT (w, x: IN BIT; z: OUT BIT);
   END COMPONENT;
   COMPONENT n3 PORT (w, x, y: IN BIT; z: OUT BIT);
   END COMPONENT;
   ...
   SIGNAL i1, i2, i3, i4 : BIT;
BEGIN
   g0 : n1 PORT MAP (d, i1);
   g1 : n1 PORT MAP (e, i2);
   g2 : n2 PORT MAP (i1, c, i3);
   g3 : n2 PORT MAP (a, b, i4);
   g4 : n3 PORT MAP (i2, i3, i4, f);
END configurable;
```

Suggested Reading

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL Hardware Description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

4 Concurrent Constructs for RT Level Descriptions

The previous chapter discussed structural descriptions in VHDL. Although complex structural constructs were discussed, constructs used for description of individual components were simple and limited to signal assignments. This chapter shows VHDL concurrent constructs for description of more complex components. We discuss those constructs that can be useful for description of combinational and sequential register transfer level components.

Although discussions of the previous chapter centered on basic gate level components, they apply equally well to hardware parts of higher complexities. This means that we can still apply structural language constructs discussed in Chapter 3 to more complex components that we will discuss in this chapter. We begin this chapter with a presentation of various forms of signal assignments. Language issues related to simple assignments used in the previous chapter are first discussed, and then signal assignments with condition and selection options will be discussed. We will then talk about block statements and guarded signal assignments. These constructs are useful for modeling connection and disconnection of busses at various abstraction levels.

4.1 Concurrent Signal Assignments

A concurrent signal assignment is the most basic form of describing hardware. We can use this structure for describing simple Boolean expressions or complex sequential components. However, the limited

syntax of this structure makes it more appropriate for describing simple logical functions.

A concurrent signal assignment has a left hand side signal and an expression on the right hand side. Being a concurrent statement, the execution of this construct is sensitive to its right hand side signals. When a right hand side signal changes value, the right hand side expression is evaluated and a value is scheduled for the left hand side signal.

4.1.1 Simple Assignments

The simplest form of a concurrent signal assignment is one without any condition, which makes it a simple signal assignment. This form of a concurrent signal assignment has a left hand side target signal and an expression on the right.

An example signal assignment is used in description of the *expression* architecture of *multiplexer* of Figure 4.1. As shown, an event on *a*, *b* or *s* causes the AND-OR expression to be evaluated and a value scheduled for *w* after 7 ns.
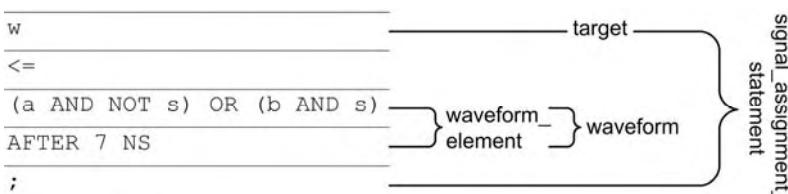
```
ENTITY multiplexer IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE expression OF multiplexer IS
BEGIN
   w <= (a AND NOT s) OR (b AND s) AFTER 7 NS;
END ARCHITECTURE expression;
```

Figure 4.1 Concurrent Signal Assignment

Figure 4.2 shows syntax details of this statement. The left hand side is called a *target* and the right hand side is a *waveform*. The *waveform* construct can consist of several *waveform_element*s. A *waveform_element*, only one of which is used in our example, consists of an *expression* this is optionally, followed by an AFTER keyword and a *time_expression*.

Figure 4.2 Signal Assignment Syntax Structure

4.1.2 Conditional Signal Assignment

VHDL allows the use of a condition on the right hand side of a signal assignment. The construct for this purpose is called a *conditional signal assignment*. This statement is shown in the *conditional* architecture of *multiplexer* shown in Figure 4.3.
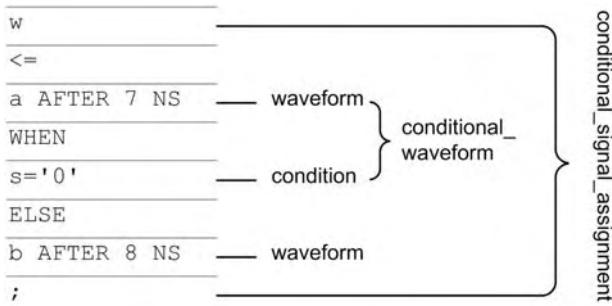
```
ARCHITECTURE conditional OF multiplexer IS
BEGIN
   w <= a AFTER 7 NS WHEN s='0' ELSE
        b AFTER 8 NS;
END ARCHITECTURE conditional;
```

Figure 4.3 Using Conditional Signal Assignment

As shown in Figure 4.4, a conditional signal assignment has a left hand side target and a series of right hand side *conditional waveform*s. A *conditional_waveform* has a waveform followed by the WHEN keyword, the condition and the ELSE keyword. The last conditional waveform must be a waveform construct that is optionally followed by a condition. In the example of Figure 4.4, the last waveform does not have a condition.

Figure 4.4 Syntax Structure of Conditional Signal Assignments

As another example of a conditional signal assignment consider the *async_reset* architecture of *flipflop* shown in Figure 4.5. This circuit is a D-type flip-flop with asynchronous reset input.

As shown, two conditional waveforms are used on the right hand side of *qout*. The first condition, i.e., *reset*='1', is first checked and if it is true a '0' is put into *qout*. If *reset* is not '1', then the second condition is checked, as the result of which *din* is put into *qout*. If none of the conditions are true, then *qout* retains its old value, and thus a memory behavior.

```
ENTITY flipflop IS
   PORT (reset, din, clk : IN BIT; qout : OUT BIT);
END ENTITY;
--
ARCHITECTURE async_reset OF flipflop IS
BEGIN
   qout <= '0' WHEN reset = '1' ELSE
           din WHEN clk'EVENT AND clk = '1';
END ARCHITECTURE async_reset;
```

Figure 4.5 Flip-Flop Using Conditional Signal Assignment

```
ENTITY latch IS
   PORT (din, clk : IN BIT; qout : OUT BIT);
END ENTITY;
--
ARCHITECTURE assign OF latch IS
BEGIN
   qout <= din WHEN clk = '1';
END ARCHITECTURE assign;
--
ARCHITECTURE unaffect OF latch IS
BEGIN
   qout <= din WHEN clk='1' ELSE UNAFFECTED;
END ARCHITECTURE unaffect;
```

Figure 4.6 Conditional Signal Assignment Used in Latch Description

Another use of a conditional signal assignment is shown in the *assign* architecture of *latch* of Figure 4.6. In this architecture when an event occurs on *din* or *clk*, the right hand side is evaluated and if at this time *clk* is '1', then *din* is put into *qout*. Transparency of the latch is implemented by the fact that while *clk* is '1', changes on *din* propagate to *qout*.

The statement shown in the *assign* architecture leaves *qout* unaffected when *clk* is not '1'. Alternatively, keeping the left hand side unaffected can be explicitly specified by use of the UNAFFECTED keyword. This is shown in the *unaffect* architecture of Figure 4.6. In fact, this keyword may be used in place of any waveform construct anywhere in a conditional signal assignment. For example, the conditional signal assignment in the body of the *unaffect* architecture of Figure 4.6 can be written as follows:

```
qout <= UNAFFECTED WHEN clk='0'
        ELSE din;
```

Note that the above statement is only equivalent to that of the *unaffect* architecture because *clk* is of type BIT and can only take '0' or '1' value.

4.1.3 Selected Signal Assignment

Another form of a concurrent signal assignment is a selected signal assignment. In this structure, an expression selects one of several waveforms. Figure 4.7 shows a binary decoder that is described with a selected signal assignment.

```
ENTITY binary3to8decoder IS
   PORT (bin_in : IN BIT_VECTOR (2 DOWNTO 0);
         en : IN BIT;
         dcd_ou : OUT BIT_VECTOR (0 TO 7));
END ENTITY binary3to8decoder;
--
ARCHITECTURE selected OF binary3to8decoder IS
   SIGNAL tmp : BIT_VECTOR(dcd_ou'RANGE);
BEGIN
   WITH bin_in SELECT
      tmp <= "10000000" WHEN "000",
             "01000000" WHEN "001",
             "00100000" WHEN "010",
             "00010000" WHEN "011",
             "00001000" WHEN "100",
             "00000100" WHEN "101",
             "00000010" WHEN "110",
             "00000001" WHEN "111";
   dcd_ou <= tmp WHEN en = '1' ELSE (OTHERS => '0');
END ARCHITECTURE selected;
```

Figure 4.7 Using a Selected Signal Assignment

In this example, when an event occurs on *bin_in*, it is checked against "000" to "111" in this order. These values ("000" to "111") are called choices. When a match is found between *bin_in* and one of the choices, the waveform that corresponds to that choice is selected and put into the left hand side signal, i.e., *tmp*.

   In addition to this statement, the code of Figure 4.7 has a conditional signal assignment that conditionally assigns *tmp* to the decoder output, *dcd_ou*. With the condition being en='1', this decoder becomes a 3-to-8 decoder with an active high enable input.

   The right hand side of the conditional signal assignment uses (OTHERS => '0'), which expands to as many zeros as the left hand side, i.e., *dcd_ou*, requires.

   The syntax of the selected signal assignment of Figure 4.7 is shown in Figure 4.8. This construct begins with the WITH keyword

that is followed by an *expression* and the SELECT keyword. Following this keyword is the left hand side *target*. On the right hand side of the arrow *selected waveform*s separated by commas appear.
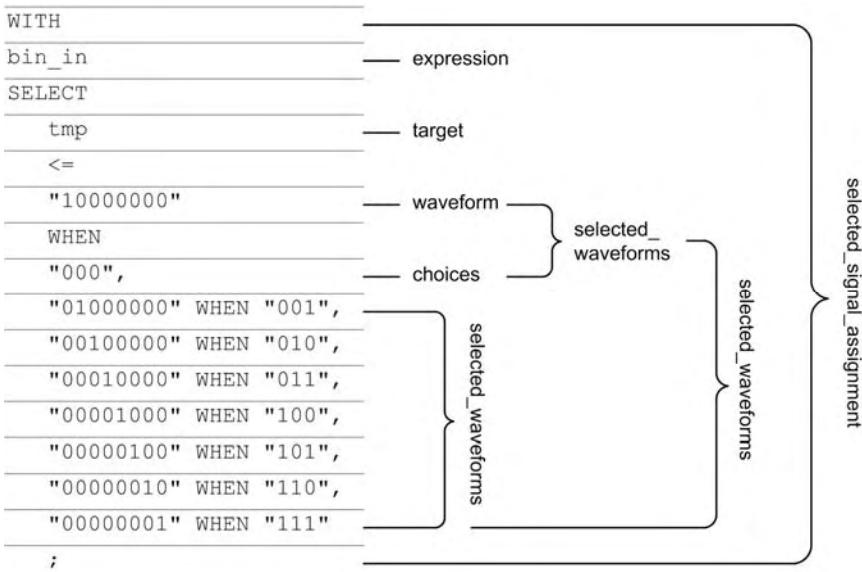
Figure 4.8 Syntax of Selected Signal Assignment

A selected waveform has a waveform and a choices part. The waveform is as described before, and the choices consists of several *choice* constructs separated by vertical bars (|). In our example we only have one choice for every waveform.

The processing of this construct begins after the arrow and individual choices are examined in the order that they appear (from left to right). If by the time the last choice is examined, the value of expression is not matched, the target signal remains unaffected. To avoid this situation the last choice can be specified to assign a default value if none of the other choices match the expression. For doing this, the last selected waveform should be as shown here:

```
"default_value" WHEN OTHERS;
```

In our example of the decoder, since all eight combinations of the 3-bit *bin_in* are covered, the use of the default selected waveform is not necessary. Using this default becomes especially useful when a multi-value logic system (instead of BIT) is used.

As an example of a selected signal assignment with default, choices consider the assignment to display shown in Figure 4.9. This

is a seven segment display decoder that displays "0" through "9" and an "E" for error if the input value is larger than 9.

```
ENTITY ssd IS
   PORT (bcd : IN BIT_VECTOR (3 DOWNTO 0);
         display : OUT BIT_VECTOR (1 TO 7));
END ENTITY;
--
ARCHITECTURE selected OF ssd IS
BEGIN
   WITH bcd SELECT
      display <= "1111110" WHEN "0000",
                 "0110000" WHEN "0001",
                 "1101101" WHEN "0010",
                 "1111001" WHEN "0011",
                 "0110011" WHEN "0100",
                 "1101101" WHEN "0101",
                 "1011111" WHEN "0110",
                 "1110000" WHEN "0111",
                 "1111111" WHEN "1000",
                 "1111101" WHEN "1001",
                 "1001111" WHEN OTHERS;
END ARCHITECTURE selected;
```

Figure 4.9 Selected Signal Assignment with Others

4.2 Guarded Signal Assignments

The previous section discussed signal assignments with a right hand side that always drives its left hand side. We say that in the case of signal assignments, the right hand side is always connected to the left hand side. This section introduces guarded assignments that under certain conditions the right hand side is disconnected from its left hand side.

4.2.1 GUARD Signal and Expression

The keyword GUARDED can be used on the right hand side of the arrow in a concurrent signal assignment to make a guarded signal assignment. In this case an implicit or explicit GUARD signal controls the connection of the right hand side of the signal assignment to its left hand side.

As an example of an explicit GUARD signal consider the *explicit* architecture of *flipflop* shown in Figure 4.10. The code shown here is another architecture for the *flipflop* entity of Figure 4.5. The explicit BOOLEAN GUARD signal is declared in the declaration part of this architecture. In the statement part of this architecture, an expression

is assigned to the left hand side GUARD signal. This signal assignment is like any other signal assignment, and because it is assigned to GUARD, it is called a guard expression.

```
ARCHITECTURE explicit OF flipflop IS
   SIGNAL GUARD : BOOLEAN;
BEGIN
   GUARD <= clk = '1' AND NOT clk'STABLE;
   qout <= GUARDED '0' WHEN reset = '1' ELSE din;
END ARCHITECTURE explicit;
```

Figure 4.10 Explicit GUARD Signal

The GUARDED keyword on the right hand side of the assignment to *qout* in Figure 4.10, specifies that the right hand side waveform is only connected to *qout* when GUARD is TRUE. This signal assignment is called a guarded signal assignment and is controlled by the GUARD signal that is visible in the concurrent block that the assignment is in.

As shown, the guard expression in Figure 4.10 becomes true when *clk* is '1' and it has not been stable. Here we are using the 'STABLE signal attribute, the details of which are discussed in Chapter 6. This expression causes GUARD to be true on the rising edge of *clk*. When so, the right hand side of *qout* is connected to *qout* which puts a '0' into it when *reset* is '1', or puts *din* into it if reset is not '1'. Note that other than using the GUARDED signal, the right hand side of *qout* is no different than that of a conditional signal assignment. This signal assignment is sensitive to all its right hand side signals and the GUARD signal.

4.2.2 Block Statement

Semantically, the body of an architecture is considered a concurrent block. Similarly, the body of a generate statement described in the previous chapter is a block. A block can also be specified explicitly using a block statement. The body of a block statement is enclosed by BLOCK and END BLOCK keywords. A *block statement* can optionally include a guard expression that becomes the expression assigned to its implicit GUARD signal. This signal becomes visible in the *statement part* of the block statement.
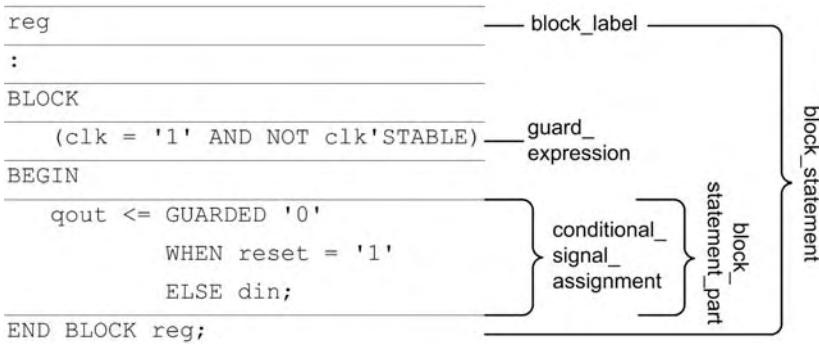
Figure 4.11 shows the *blocking* architecture of *flipflop* entity. This architecture is similar to the *explicit* architecture of Figure 4.10, exact that it does not explicitly define the GUARD signal.

Syntax details of a block statement are shown in Figure 4.12. The statement begins with a mandatory *block label* that is followed by the BLOCK keyword.

```
ARCHITECTURE blocking OF flipflop IS
BEGIN
   reg: BLOCK (clk = '1' AND NOT clk'STABLE) BEGIN
      qout <= GUARDED '0' WHEN reset = '1' ELSE din;
   END BLOCK reg;
END ARCHITECTURE blocking;
```

Figure 4.11 Block Statement with Guard Expression

Figure 4.12 Block Statement Syntax

The optional *guard expression* follows this keyword. The part of the block statement that is enclosed between the end of the guard expression and the BEGIN keyword contains the *block header* and *block declarative part*. These parts are optional and not included in this example.

For consistency with other VHDL structures, VHDL allows the use of the IS keyword after the guard expression. As with other constructs, the *statement part* of a block statement begins after the BEGIN keyword. This is a concurrent body of VHDL that can only contain concurrent statements. Options and other forms of block statements are illustrated in the examples that follow.

4.2.3 Block Statement Ports

The previous section discussed block statements from the point of view of being able to provide a guard expression. In addition to this, block statements can be used for grouping a series of concurrent statement. Statements grouped as such do not have to use the same guard expression, or any guard at all. This use of block statements is similar to describing an architecture and grouping concurrent statements in an architecture statement part. A component described as

such must be instantiated, whereas statements within a block just appear where they are used.

To better establish the use of a block statement as a structure for containing the description of a component, VHDL introduces the use of ports and port maps for the block statement. Figure 4.13 shows another architecture for the *flipflop* of Figure 4.5. This architecture uses a block similar to *blocking* architecture of Figure 4.11. The block statement in Figure 4.13, however, uses port specification to specify port signals used within the block.

```
ARCHITECTURE blockport OF flipflop IS
BEGIN
   reg: BLOCk (clk = '1' AND NOT clk'STABLE) IS
      PORT (r, d, c : IN BIT; q : OUT BIT);
      PORT MAP (reset, din, clk, qout);
   BEGIN
      q <= GUARDED '0' WHEN r = '1' ELSE d;
   END BLOCK reg;
END ARCHITECTURE blockport;
```

Figure 4.13 Block Statement with Ports

A *port map aspect* maps signals external to the block to the ports of the block. In our example, *r*, *d*, *c*, and *q* are local to the *reg* block and *reset*, *din*, *clk*, and *qout* are visible through out the *blockport* architecture. Local block ports cannot be used outside of the block, but signals visible in the architecture are still visible within the block.

The PORT and PORT MAP constructs shown in Figure 4.13 constitute the *block header* syntax construct of a block statement. In addition to this, the part of the block before the BEGIN keyword can include a *block declarative part*. Constants, signals, files, and other declarative items can appear in this part. Like block ports, declarations in this part are only local to the block.

4.2.4 Nested Block Statements

Because block statements are concurrent statements, and the statement part of a block statement is a concurrent body of VHDL, block statements can be nested. This means that we can use a block statement within another one.

When nesting block statements, it is important to note that the guard expression (if it exists) of an outer block does not automatically transfer to the guard expression of an inner block. Transferring an outer block guard expression to an inner block must be done by explicit use of the GUARD signal.

```
ENTITY latchflop IS
   PORT (din, clk : IN BIT; ql, qf : OUT BIT);
END ENTITY;
--
ARCHITECTURE nested OF latchflop IS
BEGIN
   lat: BLOCK (clk = '1') BEGIN
      ql <= GUARDED din;
      reg: BLOCK (GUARD AND NOT clk'STABLE) BEGIN
         qf <= GUARDED din;
      END BLOCK reg;
   END BLOCK lat;
END ARCHITECTURE nested;
```

Figure 4.14 Nested Block Statements

Figure 4.14 shows an entity-architecture pair of a hardware with a latch and a flip-flop output. The *ql* output of *latchflop* is a transparent latch output using *clk* as a level sensitive clock input, and the *qf* output of this structure the flip-flop output that is sensitive to the rising edge of the *clk* clock signal.

The guard expression of the *lat* block in Figure 4.14 is *clk*='1'. This expression is used for assigning *din* to *ql*. This expression transfers to the guard expression of the *reg* block by use of the GUARD signal in this guard expression. Recall that outside of the *reg* block and within, the *lat* block, GUARD refers to the guard expression of the *lat* block. When this expression is ANDed with NOT *clk*'STABLE, the guard expression of the *reg* block becomes the rising edge of *clk*.

4.2.5 Guarded Signals

VHDL has the concept of guarded signals that are used with guarded signal assignments that make blocks and guarded assignments even more useful for description of various hardware structures.

4.2.5.1 Resolved Signals. Recall from the last section of Chapter 2 that VHDL has resolved signals that can have multiple concurrent drivers. Associated with a resolved signal is a resolution function that resolves between multiple values assigned to a signal. More details of resolved signals, multiple drivers, and resolution functions will be discussed in Chapter 6.

For the discussion of guarded signals and guarded assignments, we use the VHDL predefined *std_logic* resolved type. In this discussion, we limit the use of resolved signals to having only one concurrent driver.

With a resolution function, there is a default resolved value. This value becomes the value of the resolved signal if the signal has no driver. This value is defined by the resolution function and may be different from the default value of signal. The default resolved value for *std_logic* type is 'Z'.

4.2.5.2 Guarded Signals. A resolved signal can be declared to have a kind. Kind specification follows the type mark of the resolved signal in its declaration.

Kind of a signal can be BUS or REGISTER. If undriven or if a disconnection occurs, a guarded signal of BUS kind receives its default resolved value, while an undriven REGISTER kind guarded signal retains its old value (the value before disconnection).

4.2.5.3 Assignment to BUS Kind Signals. Figure 4.15 shows a multiplexer with three-state output. This multiplexer uses the standard *std_logic* type which allows the use of the float value, 'Z'.

In this code, we have used *t* as an intermediate signal representing the tri-state output of the multiplexer (see Figure 4.16). This signal is declared as a guarded signal of *std_logic* resolved type and BUS kind. When *e* is '1' the AND-OR expression on the right hand side of *t* connects to *t*.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY multiplexer_en IS
   PORT (a, b, s, e : IN std_logic; w : OUT std_logic);
END ENTITY;
--
ARCHITECTURE blocking OF multiplexer_en IS
   SIGNAL t : std_logic BUS;
BEGIN
   tri: BLOCK (e='1') BEGIN
      t <= GUARDED (a AND NOT s) OR (b AND s);
      w <= t;
   END BLOCK tri;
END ARCHITECTURE blocking;
```

Figure 4.15 BUS Kind Guarded Signal
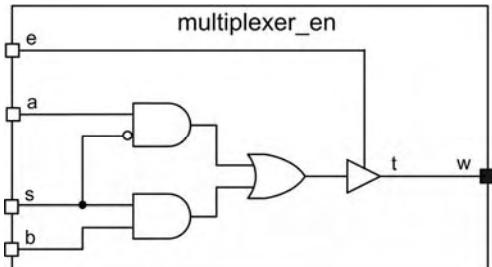
Figure 4.16 Hardware Corresponding to *multiplexer_en*

When *e* is '0', disconnection happens on *t* that because of its BUS kind makes it 'Z'. Signal *t* directly drives *w*. VHDL allows an output port to have a BUS kind specification. If this were done in our example, the use of *t* would not be necessary.

4.2.5.4 Assignment to REGISTER Kind Signals. Figure 4.17 shows a flip-flop description that uses the IEEE *std_logic* type for its inputs and outputs. The *blockport* architecture of this example uses a local signal *ff* of type *std_logic* and REGISTER kind to store flip-flop output values.

Assignment of '0' or *d* to *ff* takes place on the rising edge of *clk*. In the absence of this condition, the guarded assignment to *ff* disconnects the right hand side expression of *ff* from *ff*. The REGISTER kind causes a guarded signal to retain its old value when disconnection occurs. It is because of this property that the *ff* signal can be used as a flip-flop output.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY stdflop IS
   PORT (reset, din, clk, cen : IN std_logic:='0';
   qout : OUT std_logic);
END ENTITY;
--
ARCHITECTURE blockport OF stdflop IS BEGIN
   reg: BLOCK (clk = '1' AND NOT clk'STABLE)
      PORT (r, d, c : IN std_logic; q : OUT std_logic);
      PORT MAP (reset, din, clk, qout);
      SIGNAL ff : std_logic REGISTER;
   BEGIN
      ff <= GUARDED '0' WHEN r = '1' ELSE d;
      q <= ff;
   END BLOCK reg;
END ARCHITECTURE blockport;
```

Figure 4.17 REGISTER Kind Guarded Signal

An OUT port cannot be of a REGISTER kind. Therefore, in our *stdflop* example of Figure 4.17, the use of *ff* as an intermediate signal for the *q* output of the flip-flop is necessary.

The *reg* block in this example uses *ff*, and it is not necessary to access this signal outside of this block. Because of this, *ff* is declared in the declarative part of the *reg* block. In addition to being an example for the use of REGISTER kind guarded signals, the example of Figure 4.17 illustrates the use of block declarative items and is another example of block statements with ports.

Another point illustrated here, is the use of *cen* in the *stdflop* entity, but not in the *blockport* architecture. Since this signal is not being used, we like to have the option of associating OPEN with it when *stdflop* is instantiated. VHDL only allows an IN port to be OPEN if a default value is specified for it. The use of '0' default value in *stdflop* PORT (line 4) is for this purpose.

4.2.6 Timing Disconnections

While, the AFTER clause can be used to delay assignments to guarded signals, this construct cannot be used for disconnections. Instead, the DISCONNECT construct of VHDL can be used for this purpose. *Disconnection specification* is a block declarative item that can appear in a block statement or architecture declarative part. This construct uses a time value to delay disconnection to the specified guarded signal.

Figure 4.18 shows another architecture for the multiplexer of Figure 4.17. In the *timedisconnect* architecture, disconnections to *t* are delayed by 3.25 ns. This means that after *e* becomes '0', there is still a 1.25 ns time before *t* gets its default resolution value (i.e., 'Z').

```
ARCHITECTURE timedisconnect OF multiplexer_en IS
   SIGNAL t : std_logic BUS;
   DISCONNECT t : std_logic AFTER 3.25 NS;
BEGIN
   tri: BLOCK (e='1') BEGIN
      t <= GUARDED (a AND NOT s) OR (b AND s);
      w <= t;
   END BLOCK tri;
END ARCHITECTURE timedisconnect;
```

Figure 4.18 Multiplexer with Disconnect Time

Before we close this section, we use a simulation example to show a small testbench and at the same time show simulation results of the *timedisconnect* architecture.

Figure 4.19 shows our *timed* architecture for a testbench for the *multiplexer_en* entity. *UUT1* and *UUT2* in this architecture are instantiation of multiplexers of Figure 4.15 and Figure 4.18 respectively. The outputs of the multiplexers are *w1* and *w2*.

```
ENTITY multiplexer_en_tester IS
END ENTITY;
--
ARCHITECTURE timed OF multiplexer_en_tester IS
   SIGNAL a, b, s, e, w1, w2 : std_logic;
BEGIN
   UUT1: ENTITY WORK.multiplexer_en (blocking)
         PORT MAP (a, b, s, e, w1);
   UUT2: ENTITY WORK.multiplexer_en (timedisconnect)
         PORT MAP (a, b, s, e, w2);
   a <= '0', '1' AFTER 20 NS, '0' AFTER 45 NS,
        '1' AFTER 71 NS;
   b <= '1', '0' AFTER 11 NS, '1' AFTER 37 NS,
        '0' AFTER 79 NS;
   s <= '0', '1' AFTER 29 NS, '0' AFTER 67 NS,
        '1' AFTER 97 NS;
   e <= '0', '1' AFTER 33 NS, '0' AFTER 83 NS,
        '1' AFTER 99 NS;
END ARCHITECTURE timed;
```
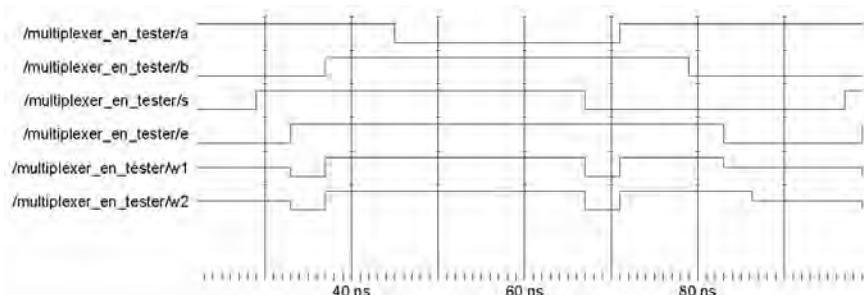
Figure 4.19 Comparing Disconnection Timing

In this testbench *e* becomes '0' causing disconnections in both multiplexer architectures at time 83 ns. The simulation result of the testbench of Figure 4.19 is shown in Figure 4.20. As shown here, as the result of *e* becoming '0', *w1* becomes 'Z' at 83 ns, while *w2* becomes 'Z' 3.25 ns later at 86.25 ns. The specified disconnection time does not affect the timing of values that are assigned to a signal.

Figure 4.20 Comparing Disconnection Times

4.3 Summary

The focus of this chapter was on concurrent bodies of VHDL. We discussed signal assignments, block statements, and guarded signals and guarded assignments. The constructs discussed here enable description of hardware at a level higher that what was discussed in the previous chapter. However, for more behavioral descriptions VHDL offers sequential statements within concurrent bodies that will be discussed in the next chapter.

Problems

4.1   Use guarded signal assignments to describe a simple latch with q and NOT q outputs that function the same as a latch formed by cross-coupled NOR gates with clocked inputs. Use reasonable delay values.

4.2   Use two of the latches in Problem 4.1 and necessary logic operations to describe a master-slave JK flip-flop.

4.3   Use guarded block statements to describe an 8-bit shift register. The structure has a serial input for right shifting the data and a single serial output. All activities are synchronized with the leading edge of the clock.

4.4   Use guarded block statements to describe an 8-bit up-down counter with a 2-bit mode select input, an 8-bit parallel data input, and an 8-bit data output. The unit performs an increment by 1 if the mode is 01, decrement by 1 if the mode is 10, and a parallel load of the eight bit input if the mode is 11. All activities are synchronized with the leading edge of the clock.

4.5   Write a description for a universal 8-bit shift register with a 2-bit mode select input, an 8-bit parallel data input, and an 8-bit data output. The unit performs a right shift if the mode is 01, left shift if the mode is 10, and a parallel load of the eight bit input if the mode is 11. All activities are synchronized with the leading edge of the clock. Use BLOCK statements or conditional signal assignments.

4.6   Write a description for a clocked T-type flip-flop. If T is '1' on the rising edge of the clock, the outputs of the flip-flop toggle.

4.7   Write a VHDL description for a rising edge trigger D-type flip-flop with asynchronous set and reset inputs and two outputs. Label

the data, clock, set and reset inputs *d*, *c*, *s* and *r*, respectively. Active *s* or *r* input override the clocked values on the *d* input; *s* and *r* cannot simultaneously be active. Changes on *d* without the rising edge of *c* have no effect on the *q* and *qb* outputs of the flip-flop. Use delay parameters *sq_delay*, *rq_delay*, and *cq_delay* for setting, resetting, and clocking the flip-flop, respectively. Develop a test bench for testing this flip-flop. Generate a simple periodic clock using a conditional signal assignment.

4.8   Using guarded block statements write a VHDL description for a falling edge trigger D-type flip-flop with a synchronous set input and a *q* output. Label the data, clock, and set inputs *d*, *clk*, *s*, respectively. Use delay parameters *sq_delay*, and *cq_delay* for setting and clocking the flip-flop, respectively. Develop a test bench for testing this flip-flop. Generate a simple periodic clock using a conditional signal assignment.

4.9   Using guarded block statements write a VHDL description for a falling edge trigger D-type flip-flop with an asynchronous set input, a clock-enable input, and a *q* output. Develop a test bench for testing this flip-flop. Generate a simple periodic clock using a conditional signal assignment.

4.10 Show the complete description of a datapath with 3 rising edge trigger 8-bit registers (*r1*, *r2*, *r3*), 3 internal three-state 8-bit busses (*rbus*, *obus*, *addbus*) a three-state input bus, *inbus*, an 8-bit *outbus* that connects to *obus*, and an ALU capable of either adding or subtracting two 8-bit numbers. The control signals coming from the controller are *inbus_on_rbus*, *alu_on_rbus*, *r2_on_obus*, *r3_on_obus*, *load_r1*, *load_r2*, *load_r3*, *sel_add*, *sel_sub*. The function of each of these control lines is as its name implies; *sel_add* and *sel_sub* control the functions of the ALU. Considering the ENTITY declaration shown below, complete the description of this data path. Use BLOCK statements or conditional signal assignments.

```
ENTITY datapath IS
   PORT (inbus : IN std_logic_vector (7 DOWNTO 0);
         outbus : OUT std_logic_vector (7 DOWNTO 0);
         clk : IN std_logic;
         inbus_on_rbus, alu_on_rbus, r2_on_obus,
         r3_on_obus : IN std_logic;
         load_r1, load_r2, load_r3,
         sel_add, sel_sub : IN std_logic);
END ENTITY;
```

Suggested Reading

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL Hardware Description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Perry, Douglas L., *VHDL: Programming by Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

5

Sequential Constructs for RT Level Descriptions

VHDL sequential constructs refer to the language constructs that execute in sequence. This means that while at the top-level a hardware component is described by interconnection of concurrent sub-components, an individual component may be described by the use of sequential statements. The group of sequential statements used for this description executes concurrent with other sub-components of the hardware system. Because human think sequential, VHDL sequential bodies provide a convenient means for describing components at the behavioral level.

This chapter focuses on VHDL sequential bodies and constructs. We start with the main sequential body of VHDL, that is a process statement, and continue our discussion with functions and procedures. The same constructs are used in process statements, functions and procedures. The last part of this chapter is dedicated to design organization, packaging parts and utilities, and VHDL libraries.

5.1 Process Statement

A signal assignment in the statement part of an architecture is a *process* which is always active and executes concurrent with other processes within the same architecture. This process has a single target, and executes when an event occurs on one of the signals on its right hand side. Therefore, it is said to be sensitive to signals on the right hand side of the signal assignment. A different kind of a process is a process statement which is also active at all times, executing con-

currently with other processes, but can be made sensitive to selected signals.

A process statement can assign values to more than one signal and can contain sequential statements. This statement begins with the PROCESS keyword and ends with END PROCESS. As shown in Figure 5.1, a process statement has a declarative part and a statement part. All constructs allowed in the declarative and statement parts of subprograms can also be used in process statements.
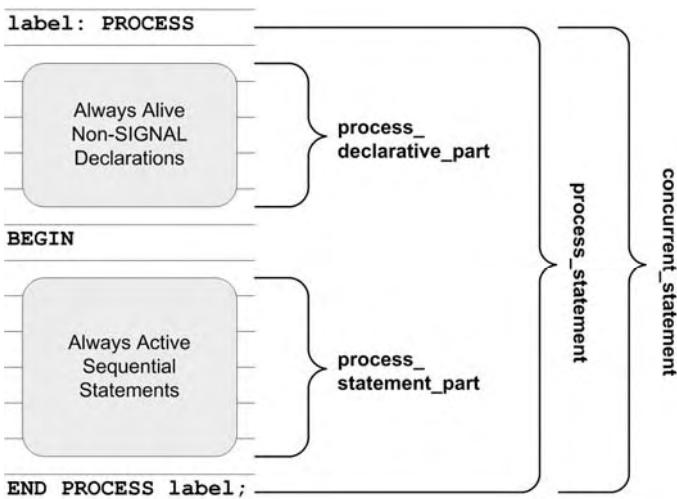
Figure 5.1 A Process Statement Block Diagram

5.1.1   Declarative Part of a Process

Variable, *file*, and *constant* objects can be declared in the declarative part of a process. Such objects are only visible to the process within which they are declared. Signals and constants declared in the declarative part of an architecture that encloses a process statement can be used inside a process. Such signals are the only means of communication between different processes.
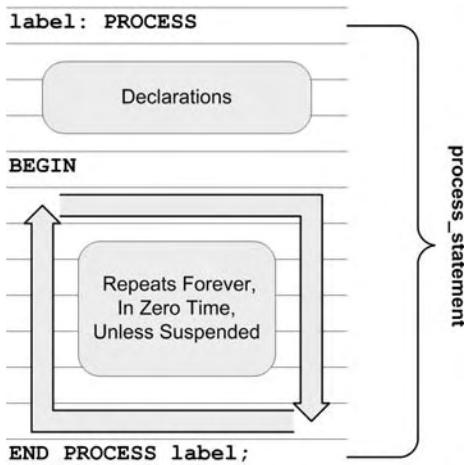
Initialization of objects declared in a process is done only once at the beginning of a simulation run. These objects stay alive for the entire simulation run. This way a variable declared in a process can be used to hold memory status or the internal state of a hardware system.

5.1.2 Statement Part of a Process

The statement part of a process is sequential, always active, and it executes in zero time. The following paragraphs and examples elaborate on these concepts.

Only sequential statements are allowed in the statement part of a process. These statements provide high level program flow control for assignment of values to signals and variables. For selection and assignment of values to signals, *if*, *loop* or *case statements* can be used. Although the syntax of several concurrent and sequential statements is the same, concurrent statements are not allowed in the statement part of a process statement. Conditional and selected signal assignments are strictly concurrent, and they cannot be used in a process statement.

As depicted in Figure 5.2, program flow inside a process starts at the beginning of its statement part and proceeds toward the end of this part. Statements reached by the program flow are executed sequentially in zero time. Unless suspended by implicit or explicit wait statements, a process repeats forever.

Figure 5.2 A Process Runs in Zero Time, Repeats Forever, Unless Suspended

Consider for example the partial code in Figure 5.3. In this example, the assignment of a to x is executed before assigning b to y. Both assignments schedule values to their left hand side targets, which will appear one *delta* time later. In order for these values to take place, the process must be suspended at least for one delta time. Otherwise, since it takes zero time to reach the end of the process and loop back to its beginning, by the time the program flow reaches the

first statement of the process, new values assigned in the previous iteration will not show on the left hand side signals.

```
ARCHITECTURE sequentiality_demo OF partial_process IS
BEGIN
   PROCESS
   BEGIN
     ...
     x <= a;
     y <= b;
     ...
   END PROCESS;
END sequentiality_demo;
```

Figure 5.3 Zero Distance Signal Assignments

A partial code, demonstrating the availability of data assigned to signals, is shown in Figure 5.4. In this figure, we assume that the value of x is '0' before the flow of the program reaches the signal assignment that uses x on the left hand side. Execution of this statement causes '1' to be scheduled for the x target after a δ delay. The if statement in this figure is executed immediately after the execution of the signal assignment. Since these two statements are executed during the same simulation cycle (in zero time), the new value of x is not available when the if statement is executed.

```
ARCHITECTURE data_availability_demo OF partial_process IS
    SIGNAL x : BIT :='0';
BEGIN
   PROCESS BEGIN
     ...
     x <= '1';
     IF x = '1' THEN
        Perform_action_1
     ELSE
        Perform_action_2
     END IF;
     ...
   END PROCESS;
END data_availability_demo;
```

Figure 5.4 Partial Code for Demonstrating Delay in Assignment of Values to Signals

The condition of this statement, therefore, will not be satisfied and *action_2* is performed. Had x been a variable, the symbol := would have to be used to assign a value to it, and its new value, '1', would be available when the if statement is executed. In that case, *action_1* would have been performed.

5.1.3   Process Sensitivity List

A process statement is always active and executes at all times if not suspended. A mechanism for suspending and subsequently conditionally activating a process is the use of sensitivity list. Following a PROCESS keyword, a list of signals in parentheses can be specified; this list is called the sensitivity list, and the process is activated when an event occurs on any of these signals. When the program flow reaches the last sequential statement, the process becomes suspended, although alive, until another event occurs on a signal that it is sensitive to.

Regardless of the events on the sensitivity list signals, a process is executed once at the beginning of a simulation run.

Behaviorally, the example description depicted in Figure 5.5 is the same as that of Figure 4.1 of the previous chapter. In both cases, an event on *a*, *b*, or *s* wakes up the process and depending on the value of *s*, *a* or *b* is assigned to *w*. Another process that is behaviorally equivalent to the process statement of Figure 5.5 is the conditional signal assignment of Figure 4.3.

```
ENTITY multiplexer IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE processing OF multiplexer IS
BEGIN
   com: PROCESS (a, b, s) BEGIN
      IF s='0' THEN w <= a AFTER 1.4 NS;
      ELSE w <= b AFTER 1.5 NS;
      END IF;
   END PROCESS com;
END ARCHITECTURE processing;
```

Figure 5.5 Multiplexer Described Using a Process with Sensitivity List

As another process statement with sensitivity list, consider the *synch_process* architecture of *flipflop* of Figure 5.6. The *reg* process shown here wakes up when an event occurs on *clk*. With this event if *clk* is '1', a '0' or *din* is assigned to *qout*. This behavior implements a rising edge D-type flip-flop with a synchronous *reset* input.

Figure 5.7 shows the syntax details for the process statement used in the description of Figure 5.6. The label of a process, which precedes this statement, is optional. If used, it can also be placed at the end of the process statement. The sensitivity list is also optional and can contain any number of signals that are visible outside of the process. Objects declared inside a process cannot be used in its sensitivity list for that process.
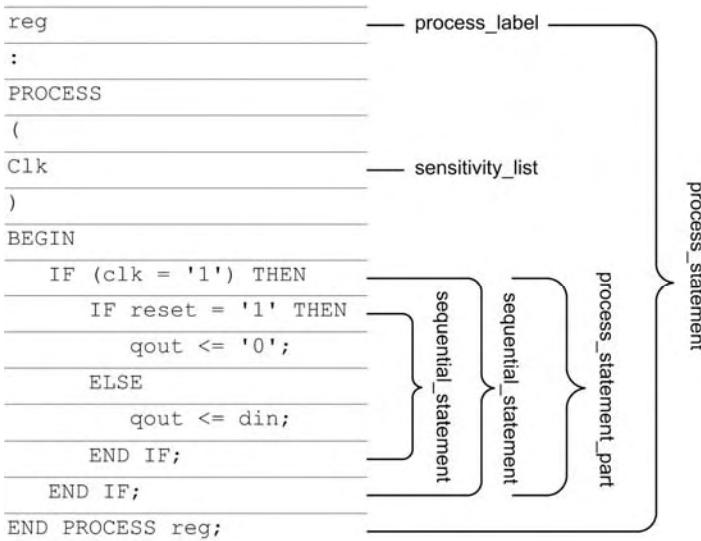
```
ENTITY flipflop IS
   PORT (reset, din, clk : IN BIT; qout : OUT BIT);
END ENTITY;
--
ARCHITECTURE synch_process OF flipflop IS BEGIN
   reg: PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF reset = '1' THEN qout <= '0';
         ELSE qout <= din;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE synch_process;
```

Figure 5.6 *flipflop* **Using Process with Sensitivity List**

As shown in Figure 5.7, only sequential statements are allowed in the statement part of a process, whereas, the process statements themselves are considered concurrent statements. This implies that processes cannot be nested. Where nesting of behavioral sequential bodies is necessary, procedures can be called from processes. It is, of course, possible to nest procedures since procedure calls are both concurrent and sequential statements. Procedures are discussed in Section 5.3.

Figure 5.7 Syntax of Process with Sensitivity List

The sequential statement used in the body of the *reg* process is an if statement that contains another if statement. The condition of the outer if statement is (*clk* = '1') and the condition of the enclosed if statement is *reset* = '1'. Any number of sequential statements with any level of nesting can be used in the statement part of a process statement.

As another example of a process statement, consider the *async_process* architecture of Figure 5.8. Unlike the process statement of Figure 5.6 that only used the clock, the process statement shown here uses *clk* and *reset* in its sensitivity list. Another difference between these two examples is that the *reset = '1'* condition precedes the clock condition in Figure 5.8. This means that when an activity occurs on *reset*, the process of Figure 5.8 wakes up and it first checks the value of *reset* regardless of *clk*. This behavior makes the *asynch_process* architecture of *flipflop* a D-type flip-flop with an active high asynchronous *reset* input.

```
ARCHITECTURE asynch_process OF flipflop IS
BEGIN
   reg: PROCESS (clk, reset) BEGIN
      IF reset = '1' THEN
         qout <= '0' AFTER 1.2 NS;
      ELSIF (clk = '1' AND clk'EVENT) THEN
         qout <= din AFTER 1.3 NS;
      END IF;
   END PROCESS reg;
END ARCHITECTURE asynch_process;
```

Figure 5.8 Process Statement Implementing Asynchronous Control

The statement part of the process of Figure 5.8 contains an if statement with ELSIF part. The syntax used here is different than that of Figure 5.6 in that no nesting of if statements is done here.

5.1.4 Postponed Processes

In the multiplexer example shown in Figure 5.5, if in a real time point but at different delta time points, *a*, *b* and *s* change, the *com* process will be activated at each delta time. At each such activation, the process is executed, values calculated, and reports will be made if there are any. The very last activation, however, will overwrite all calculated values and values for that real time frame will become stable.

Two problems can be caused by multiple activation of a process statement in a given real time point. One is that of generating false reports based on transitional non steady-state input data, and the

other is unnecessary executions of the process statement leading to slower simulation runs.

A *postponed process* can remedy these problems by activating a process statement only once per real-time point after all sensitivity list signals have become stable. Figure 5.9 shows the timing of the *com* process in Figure 5.5. For the timing shown in this figure, the non-postponed process is activated four times during the *t1* real time. The first time because *s* is '0', and *a* has become '1', it schedules a '1' into *w*. The second time *s* becomes '1' and it schedules the '0' value of *b* into *w*. The third time that *s* becomes 0, it reads *a* again and puts a '1' into *w*. The last time that the process wakes up is when *b* becomes '1' at time *t1*+4δ. At this time, because *s* is '0', the value of *a* that is a '1' will be again scheduled for *w*. The end result is that *w* becomes '1' after four times execution of this process.
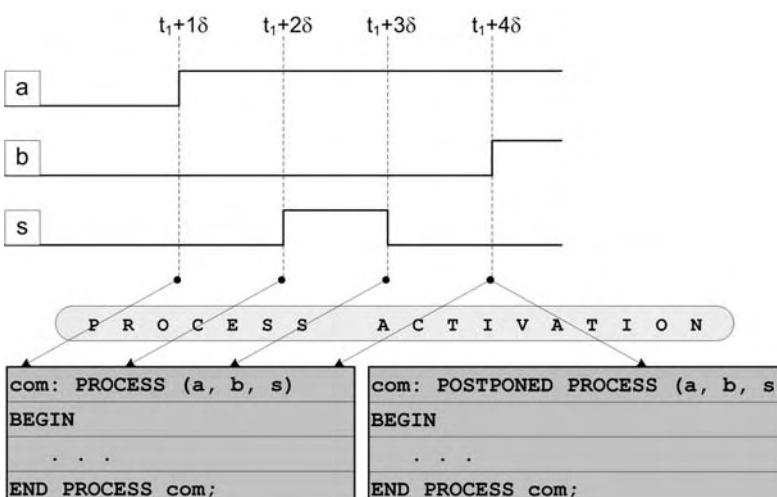
Figure 5.9 Activation of a Postponed Process

The *com* postponed process statement, the header of which is shown on the right in Figure 5.9, only wakes up once during *t1* and that is after all but the last simulation cycle has completed. As with the other process, this postponed process still result in a '1' in w. For this process, this value is calculated once without unnecessary executions of the process statement.

Concurrent statements are sensitive to activities on their right hand sides much the same way as processes are sensitive to their sensitivity list. Such statements may also be postponed by using the POSTPONED keyword on the left hand side of the statement.

The following statement, is processed only once during a real time point after all right hand side signals become stable.

```
concurrent1: POSTPONED w <= (a AND NOT s) OR (b AND s);
```

5.1.5 Passive Processes

A process statement or a concurrent statement that does not include any signal assignment is a *passive process*. Such a process may be used in the statement part of an entity declaration. For demonstration of this concept we add several timing checks to the entity description of *flipflop* of Figure 5.6.

Figure 5.10 shows the *flipflop* entity that has the *timing* process in its statement part. Several timing variables for keeping track of events and durations on *clk* are defined in the declarative part of this process. Note in this process statement that no signal assignments are taken place. Because these timing checks are included in the statement part of an entity, they apply to all architectures that are associated with it.

```
ENTITY flipflop IS
   PORT (reset, din, clk : IN BIT; qout : OUT BIT);
BEGIN
   timing: PROCESS (clk, reset, din)
      VARIABLE t_clk1, t_clk0 : TIME := 0 NS;
      VARIABLE t_clkon, t_clkoff : TIME := 0 NS;
   BEGIN
      IF clk'EVENT THEN
         IF clk = '1' THEN --rising edge
            t_clk1 := NOW;
            t_clkoff := t_clk1 - t_clk0;
         ELSE               --faling edge
            t_clk0 := NOW;
            t_clkon := t_clk0 - t_clk1;
         END IF;
      END IF;
      IF t_clkon /= t_clkoff THEN
         REPORT "Not 50% duty cycle: On:"
                & TIME'IMAGE(t_clkon) & "Off:"
                & TIME'IMAGE(t_clkoff);
      END IF;
      IF clk = '1' AND din'EVENT THEN
         REPORT "The din input changed while clk was '1'";
      END IF;
   END PROCESS timing;
END ENTITY;
```

Figure 5.10 Entity Statement with a Passive Process

In the first part of this process statement, times at which transitions of *clk* take place are recorded. For this purpose VHDL's NOW function that returns the simulation real time is used. Using these time marks, on and off durations of *clk* are calculated. In the second part of the process duty cycles that are not 50% of the clock cycle are reported using the *report statement*. The report statement begins with the REPORT keyword and it is a sequential statement.

We use the 'IMAGE construct to report the clock on and off times. When used with a type mark, 'IMAGE returns the value of its argument in the string form. Concatenation of strings corresponding to the on and off times with a message string are displayed if *t_clkon* and *t_clkoff* are different. Another if statement, near the end of the *timing* process, reports changes on *din* that occur while *clk* is '1'.

5.2 Sequential Wait Statements

The *wait statement* is a highly behavioral construct for modeling delays, handshakings, and hardware dependencies. This statement comes in four different forms and can be used only in procedures and processes that do not have the optional sensitivity list. When a program flow reaches a wait statement, the process or procedure that encloses it is suspended. The sequential body resumes after the conditions specified by the wait statement are met.

Five forms of the wait statement are shown here:

```
WAIT FOR waiting_time;
WAIT ON waiting_sensitivity_list;
WAIT UNTIL waiting_condition;
WAIT FOR 0 any_time_unit;
WAIT;
```

WAIT FOR causes suspension of a sequential body until the *waiting_time* elapses. The suspension caused by WAIT ON is resumed when an event occurs on any of the signals in the *waiting_sensitivity_list*. If a sequential body is suspended by a WAIT UNTIL, that body is resumed when an event occurs on the *waiting_condition* and the value of this condition is TRUE after the event. If the flow of program reaches a WAIT UNTIL when the *waiting_condition* is TRUE, suspension occurs, and resumption does not occur until an event causes the evaluation of condition. The forth form of wait statement, WAIT FOR 0 *time*, suspends a process for exactly one delta time (one simulation cycle). Counting delta times is possible by using this statement in a loop. Any valid time unit such as FS or NS can be used for the 0 time value. Finally, the last form of wait statement, WAIT, suspends a process forever.

Wait statements were presented here as individual constructs. Although, it is easier to describe and use them as such, VHDL formally defines WAIT as a construct with sensitivity clause, condition clause and timeout clause as shown below:

```
WAIT ON sensitivity_list UNTIL condition FOR time;
```

When a process is suspended by a wait statement, the resumption of the process occurs when an event occurs on a signal in the sensitivity list, and the condition is true and the specified time elapses.

If any of the wait clauses is not specified, that requirement for resumption of the process will be removed. The three clauses act independently with the exception of the condition clauses, which when specified adds implicit sensitivity signals from signals in its condition to a sensitivity clause. For example assuming *a_signal* is a signal and *a_variable* is a variable, the statement,

```
WAIT UNTIL a_signal AND a_variable = '1';
```
becomes

```
WAIT ON a_signal UNTIL a_signal AND a_variable = '1';
```

while,

```
WAIT UNTIL a_variable = '1';
```

remains as such with no sensitivity requirement. A process suspended by this construct resumes when *a_variable* becomes '1'.

The description here explains the earlier simpler description of WAIT UNTIL when we discussed it independent of the sensitivity clause.

The sensitivity list of a process statement is equivalent to a WAIT ON statement placed at the end of the statement part of the process. The *waiting_sensitivity_list* in this statement is the list of signals that would appear in the process sensitivity list. Therefore, the sensitivity list provides a simple, yet limited way of suspending and activating a process; the more general method is the use of wait statements. The two methods cannot be combined. For example the process statement,

```
PROCESS (a, b, c) . . .
```

is equivalent to one without the *a*, *b*, *c* sensitivity list, and with

```
WAIT ON a, b, c;
```

as its last sequential statement. No other wait statement can appear in the former process.

As our first wait statement example consider the *process_wait* architecture of *multiplexer* shown in Figure 5.11. This description is equivalent to the *processing* architecture of Figure 5.5. At time 0, this process executes once and goes into suspension when the WAIT ON statement is reached. An event on *a*, *b*, or *s* wakes up this process and runs its body once.

```
ARCHITECTURE process_wait OF multiplexer IS
BEGIN
   com: PROCESS
   BEGIN
      IF s='0' THEN
         w <= a AFTER 1.4 NS;
      ELSE
         w <= b AFTER 1.5 NS;
      END IF;
      WAIT ON a, b, s;
   END PROCESS com;
END ARCHITECTURE process_wait;
```

Figure 5.11 Process with WAIT

Another example of a process statement with WAIT is the *synch_waituntil* architecture of *flipflop* shown in Figure 5.12. This architecture implements a D-type flip-flop with a synchronous *reset* similar to the architecture of Figure 5.7. Using the WAIT UNTIL instead of sensitivity list allows the use of other WAIT statements in this architecture. As shown, we have used a WAIT FOR statement for adjusting the timing of this flip-flop. Because of this statement, a rising edge of *clk* that is distanced from its previous one less than 1.5 ns, is ignored.

```
ARCHITECTURE synch_waituntil OF flipflop IS
BEGIN
   reg: PROCESS
   BEGIN
      IF reset = '1' THEN
         qout <= '0' AFTER 1.2 NS;
      ELSE
         qout <= din AFTER 1.3 NS;
      END IF;
      WAIT FOR 1.5 NS;
      WAIT UNTIL clk = '1';
   END PROCESS reg;
END ARCHITECTURE synch_waituntil;
```

Figure 5.12 Multiple WAIT Statements

5.3 VHDL Subprograms

VHDL *subprogram*s can be used in designs for defining hardware sub-modules, as well as utility functions for applications such as type conversions and input/output. This section discusses definition and usage of subprograms in VHDL.

In many programming languages, subprograms are used to simplify coding, modularity, and readability of descriptions. VHDL uses subprograms for these applications as well as for those that are more specific to hardware descriptions. Furthermore, VHDL subprograms are used for adaptation of various data types to standard VHDL operators. Regardless of the application, behavioral softwarelike constructs are allowed in subprograms. VHDL allows two forms of subprograms, *functions* and *procedures*. Functions return values and cannot alter the values of their parameters. A procedure, on the other hand, is used as a statement and can alter the values of its parameters.

5.3.1 Function Definition

Many of the features of the VHDL language that are related to functions can be demonstrated by the *mux* function of Figure 5.13. This function performs multiplexing and will be used later for developing another architecture for our multiplexer.

```
FUNCTION mux
   (databits : BIT_VECTOR; sel : BIT_VECTOR)
RETURN BIT IS
   VARIABLE selint : INTEGER := 0;
BEGIN
   FOR i IN sel'LENGTH - 1 DOWNTO 0 LOOP
      IF sel (i) = '1' THEN
         selint := selint + 2**i;
      END IF;
   END LOOP;
   RETURN databits (selint);
END FUNCTION mux;
```

Figure 5.13 A Simple Function Definition

Syntax details of this function are shown in Figure 5.14. The discussion below refers to this figure. As shown here, the function designator is *mux*, its formal parameters are *databits* and *sel*, and its return type mark is BIT. The type for *databits* and *sel* is BIT_VECTOR. Dimensions for these parameters are not explicitly specified, and will be known when the function's formal parameters are associated with the actual parameters when the function is invoked.
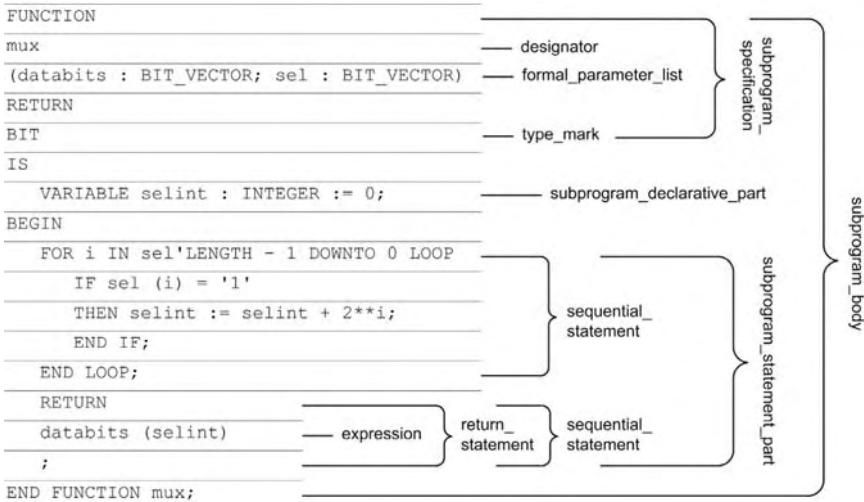
Figure 5.14 Function Syntax Details

The function declarative part includes declaration of *selint* vari-able. Variables but not signals can be declared in a declarative part of a function.

The statement part of this function consists of sequential state-ments. The two sequential statements used here are a sequential for-loop and a return statement. The index variable of the for-loop state-ment is *i* and looping is done for length of *sel* down to 0. The 'LENGTH attribute reads the length of the array it is applied to. The for-loop statement encloses another sequential statement. This statement is an if-then statement that calculates the integer value of *sel* and places it in *selint*. The return statement in the statement part of *mux* returns the bit of *databits* selected by *selint*.

Figure 5.15 shows how function *mux* of Figure 5.13 is called. As shown, in order to be able to use *mux*, it has been placed in the de-clarative part of *functional* architecture of *multiplexer*. The last line in the statement part of this architecture is a signal assignment that uses *mux* on its right hand side. When a function is called, it returns its calculated value in zero time. Because of this, and in order for this description to better represent the actual circuit, the value returned by the function is delayed 8 ns before it is assigned to the output of the circuit.

```
ARCHITECTURE functional OF multiplexer IS

    FUNCTION mux (databits : BIT_VECTOR; . . .
        .
        .
        .
    END FUNCTION mux;

    SIGNAL sel : BIT_VECTOR (0 DOWNTO 0);
BEGIN
    sel(0) <= s;
    w <= mux ((a,b), sel) AFTER 8 NS;
END ARCHITECTURE functional;
```

Figure 5.15 Calling a Function

5.3.2 Procedure Definition

We use the *consecutive_data* procedure of Figure 5.16 to illustrate how procedures are defined and utilized. This simple example shows the use of declarations, assignments, and sequential statements in procedures.

```
PROCEDURE consecutive_data
    (SIGNAL target : OUT BIT_VECTOR;
     CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
    VARIABLE data : BIT_VECTOR (target'RANGE);
    VARIABLE sum, carry : BIT;
BEGIN
    FOR i IN 1 TO n LOOP
        carry := '1';
        FOR j IN data'REVERSE_RANGE LOOP
            sum := data (j) XOR carry;
            carry := data (j) AND carry;
            data (j) := sum;
        END LOOP;
        target <= TRANSPORT data AFTER ti * i;
    END LOOP;
END PROCEDURE consecutive_data;
```

Figure 5.16 Procedure Definition

The *consecutive_data* procedure has an output signal of type BIT_VECTOR. This output is *target* that is a parameter of the procedure. All other parameters of this procedure are inputs. When called, n consecutive binary numbers starting from 0 are placed on *target*. The n data sets are distanced in time by *ti*. This procedure is useful in developing testbenches.

The declarative part of *consecutive_data* declares *data*, *sum*, and *carry* intermediate variables. The type of the *data* variable is BIT_VECTOR, and its size is taken from that of *target*. The 'RANGE attribute, that is applied to *target*, reads the range of the actual signal that will be associated with *target* when the procedure is called.

A for-loop in the statement part of this procedure loops *n* times, each time putting *data* into *target*. The delay mechanism for placing data into the *target* signal is TRANSPORT. This mechanism allows new data scheduled for this signal to be appended to the existing data.

The inner loop in the statement part of *consecutive_data* procedure adds a 1 to *data* each time it is executed. Looping is done for all bits of *data* starting from its right most bit. If the actual parameter of this procedure that is associated with *target* is an *m*-bit vector declared from *m*-1 to 0, then *target* and *data* will take the same range. In this case, 'REVERSE_RANGE of *data* becomes 0 to *m*-1. The inner loop of *consecutive_data* adds a 1 to *data*. Adding and retaining carry begins from the least significant bit of *data*. Addition is done this way because VHDL does not define the add operation for objects of the BIT_VECTOR type. Various libraries and packages exist that facilitate arithmetic and logical operations for several standard types. Libraries and packages are discussed in the next section, and details of some of the standard packages are included in this book as appendices.

Three instances of calling the *consecutive_data* procedure of Figure 5.16 are shown in Figure 5.17. The architecture shown applies consecutive data to *a*, *b* and *s* inputs of *multiplexer8*. The *multiplexer8* entity was presented in Chapter 3. This is an octal 2-to-1 multiplexer. The data inputs of this circuit are 8-bit *a* and *b* vectors, and its select input is *s*. In the testbench, data on *a*, *b*, and *s* are distanced in time by 123 ns, 79 ns and 119 ns, respectively. Since *consecutive_data* requires a BIT_VECTOR to be associated with *target*, and *s* is a scalar, we have used the one bit long *sel* of BIT_VECTOR type to associate with *target*.

Procedures may be called from concurrent or sequential VHDL bodies. A sequential procedure call is executed when the program flow reaches it. On the other hand a concurrent procedure call, like those of Figure 5.17 is executed once at the very beginning of simulation and after that when an event occurs on one of its input signals. Since *consecutive_data* does not have any input signal, its invocations in Figure 5.17 are only executed at the beginning of simulation after the initialization phase.

```
ARCHITECTURE procedural OF multiplexer8_tester IS

   PROCEDURE consecutive_data
      (SIGNAL target : OUT BIT_VECTOR;
       CONSTANT ti : TIME; CONSTANT n : INTEGER)
   IS
      .
      .
      .
   END PROCEDURE consecutive_data;

   SIGNAL a, b, w2 : BIT_VECTOR (7 DOWNTO 0);
   SIGNAL s : BIT;
   SIGNAL sel : BIT_VECTOR (0 TO 0);
BEGIN
   UUT2: ENTITY WORK.multiplexer8 (direct)
         PORT MAP (a, b, s, w2);

   consecutive_data (a, 123 NS, 6);
   consecutive_data (b, 79 NS, 6);
   consecutive_data (sel, 119 NS, 8);

   s <= sel(0);
END ARCHITECTURE procedural;
```

Figure 5.17 Concurrent Procedure Calls

As another example of a procedure, consider the *onehot_data* procedure of Figure 5.18. This procedure is similar to *consecutive_data* except that is places a walking-1 on its *target* output. *onehot_data* uses a *while-loop* instead of the for-loop of Figure 5.16. The ROR operation rotates contents of *data* one places to the right with each iteration of the while-loop.

```
PROCEDURE onehot_data
   (SIGNAL target : OUT BIT_VECTOR;
    CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
   VARIABLE data : BIT_VECTOR (target'RANGE);
   VARIABLE i : INTEGER := 0;
BEGIN
   data (0) := '1';
   WHILE i < n LOOP
      data := data ROR 1;
      target <= TRANSPORT data AFTER ti * i;
      i := i + 1;
   END LOOP;
END PROCEDURE onehot_data;
```
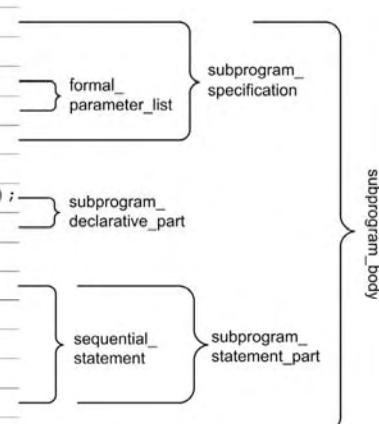
Figure 5.18 Using While Loop

5.3.3 Language Aspects of Subprograms

The general structure of subprograms was discussed in relation to the *mux* function in Figure 5.13. The *consecutive_data* procedure is a larger example, and it contains other language features of subprograms. As shown in Figure 5.19, the procedure begins with a subprogram specification. This part specifies the formal parameter list, which is syntactically similar to an interface list. The first parameter in this list indicates that *target* is a BIT_VECTOR signal with OUT mode which can only be written into. The mode of the next two parameters is IN, and they are constants of TIME and INTEGER types, respectively. If the class of an object, for example, SIGNAL or CONSTANT, for the parameters of a procedure is not specified, CONSTANT is assumed for the IN mode parameters and VARIABLE is assumed for the OUT mode parameters.

Figure 5.19 Details of a Subprogram Body

In the subprogram declaration in Figure 5.19, a BIT_VECTOR type variable and two BIT types are declared. Variables are initialized to their initial values each time a subprogram is called. If initial values are not specified, default initial values, which depend on the type of an object, are used. In this case data is initialized to all 0's, and *sum* and *carry* are initialized to the '0' default. The body of the *consecutive_data* subprogram consists of a single loop statement. Details of this statement are discussed later in this chapter.

5.3.4 Nesting Subprograms

Functions and procedures are sequential bodies that can be used as utilities for testbench and hardware development, or for description

of hardware units. These subprograms can be invoked from concurrent or sequential bodies of VHDL. Being themselves sequential bodies, subprograms can be invoked from other subprograms. This section shows several examples of subprograms invoking other subprograms.

The *mux* function discussed before (Figure 5.13) has a part that converts its *sel* input to integer. A better design is to separate this conversion part into an integer conversion function and use this function in *mux*. Figure 5.20 shows an *int* function and its utilization in our new version of *mux*. With this scheme, *int* becomes a function for converting any BIT_VECTOR type to integer, and can be used in concurrent or sequential bodies. The *mux* function in Figure 5.20 uses *int* to index *databits* to select the bit of this vector that is addressed by *sel*. In order to make it a general purpose function, *int* uses the 'LENGTH attribute to read the size of its input BIT_VECTOR.

```
FUNCTION int (invec : BIT_VECTOR) RETURN INTEGER IS
   VARIABLE tmp : INTEGER := 0;
BEGIN
   FOR i IN invec'LENGTH - 1 DOWNTO 0 LOOP
      IF invec (i) = '1' THEN
         tmp := tmp + 2**i;
      END IF;
   END LOOP;
   RETURN tmp;
END FUNCTION int;

FUNCTION mux (databits : BIT_VECTOR; sel : BIT_VECTOR)
RETURN BIT IS
BEGIN
   RETURN databits (int(sel));
END FUNCTION mux;
```

Figure 5.20 Using a Function in Another

Another example of using a subprogram in another is shown in Figure 5.21. In this code the *inc* procedure is used for incrementing its *invec* parameter. This parameter is declared as an INOUT variable in order to be used both as input and output. The *inc* procedure uses a loop for *sum* and *carry* calculations in order to perform bit-by-bit incrementing of *invec*. The 'REVERSE_RANGE attributes causes incrementing to be done starting from the least significant bit of *invec*.

The *consecutive_data* procedure shown in Figure 5.21 uses *inc* to increment its *data* variable. Calling *inc* in *consecutive_data* is a sequential procedure call, whereas, invocations of *consecutive_data* in

multiplexer8_tester in Figure 5.17 are considered to be concurrent procedure calls.

```
PROCEDURE inc (VARIABLE invec : INOUT BIT_VECTOR) IS
   VARIABLE sum, carry : BIT;
BEGIN
   carry := '1';
   FOR j IN invec'REVERSE_RANGE LOOP
      sum := invec (j) XOR carry;
      carry := invec (j) AND carry;
      invec (j) := sum;
   END LOOP;
END PROCEDURE inc;
--
PROCEDURE consecutive_data
   (SIGNAL target : OUT BIT_VECTOR;
    CONSTANT ti : TIME; CONSTANT n : INTEGER) IS
    VARIABLE data : BIT_VECTOR (target'RANGE);
BEGIN
   FOR i IN 1 TO n LOOP
      inc (data);
      target <= TRANSPORT data AFTER ti * i;
   END LOOP;
END PROCEDURE consecutive_data;
```

Figure 5.21 Using a Procedure in Another

A utility hardware function for binary decoding is shown in Figure 5.22. This function uses the *int* function that was developed for implementing *mux*. Variable *tmp* represents the output of the decoder. A variable assignment in the body of *dcd* sets all bits of *tmp* to '0'. The assignment that follows this puts a '1' in the *tmp* location that is addressed by the decoder's *bin* input. The *int* function generates an integer for indexing *tmp*.

```
FUNCTION dcd (bin : BIT_VECTOR) RETURN BIT_VECTOR IS
   VARIABLE tmp : BIT_VECTOR(0 TO 2**bin'LENGTH - 1);
BEGIN
   tmp := (OTHERS => '0');
   tmp (int(bin)) := '1';
   RETURN tmp;
END FUNCTION dcd;
```

Figure 5.22 Using *int* Function

A 3-to-8 decoder using the *dcd* function of Figure 5.22 is shown in Figure 5.23. The decoder has an enable input (*en*) that makes all decoder outputs zero when it is '0'. This decoder can be adjusted to any size when instantiated.

```
ENTITY decoder IS
   PORT (bin_in : IN BIT_VECTOR; en : IN BIT;
         dcd_ou : OUT BIT_VECTOR);
END ENTITY decoder;
--
ARCHITECTURE functional OF decoder IS
BEGIN
   dcd_ou <= dcd (bin_in) WHEN en = '1'
                          ELSE (OTHERS => '0');
END ARCHITECTURE functional;
```

Figure 5.23 Using *dcd* Function in a Concurrent Statement

5.4 VHDL Library Structure

The VHDL language uses libraries for design organization, user re-sources, vendor specific information, and adaptation of the language to the standards. Contents of a VHDL library are design units such as entity-architecture pairs, configurations, and packages.

5.4.1 Creating Libraries

When a design unit is being compiled, by default it is placed in a li-brary called WORK. This library is automatically created when a de-sign project is defined.

The built-in standard VHDL library is STD. STD is internal to the language and is not available for user manipulation. VHDL has other system libraries that are used for adapting the language to cer-tain standards. An example is the IEEE library that includes type and operator definition for various logic value systems, math pack-ages and other standard utilities. System libraries are created when a VHDL simulator is installed.

Vendor libraries in VHDL include parts, timing files, cell librar-ies, and vendor specific types and definitions. These libraries are cre-ated when a certain chip manufactures is specified during installa-tion of a VHDL simulator.

Users can also create their own libraries. A VHDL simulation environment provides a mechanism for creation, deletion, and ma-nipulation of user libraries. The language has no provisions for ma-nipulation of libraries. When a design is being compiled, the user specifies if it is to be compiled into a certain user library or the de-fault WORK library.

5.4.2 Using Libraries

Except for the STD and WORK libraries, the uses of which are implied, a design unit must always specify the library that is being used. The VHDL construct for this purpose is the *library* construct that uses the LIBRARY keyword followed by the library name. For example, for a design to use utility package of the IEEE library, the following statement that is referred to as a *library clause* must appear in the corresponding design file.

```
LIBRARY IEEE;
```

VHDL has no language support for creating, deleting, moving, or archiving libraries. Library management is generally done within a VHDL based software environment and the related tools. A certain software environment may use the name of a design project as the library name, while another vendor may use command line instructions for library management. In the section that follows we show how libraries are used. We assume two user libraries are defined. One is the *utilities* library for organizing utility types and subprograms, and the other is the *components* library for organizing commonly used hardware components. For accessing elements of these libraries the following statements must be used.

```
LIBRARY utilities;
LIBRARY components;
```

5.5 Packaging Utilities and Components

In the parts library of a hardware designer, gates or components are grouped according to their technology, physical characteristics, cost, or complexity or simply according to their availability. A designer chooses a certain group of components based on specific design requirements. In VHDL, packages can be used for this grouping of components. The VHDL package constructs can also be used for packaging commonly used user-defined types and subprograms.

A VHDL package has a *package declaration* and a *package body*. The package declaration contains the declaration of entities (types, components, subprograms, etc.) that are to be accessible by entities using the package. The package body has the description of subprograms declared in the package declaration, and subprograms and types that are used local to the package body.

5.5.1 A Package of Utilities

A package for utilities such as type declarations and subprograms requires a package declaration and a package body. An example package declaration for a utility package is shown in Figure 5.24. This includes declaration of *int*, *mux*, *bin*, *dcd*, *inc* and *consecutive_data* subprograms. These subprograms were discussed in the previous section. As shown, a subprogram declaration is basically the same as a subprogram header (see corresponding codes of Section 5.3) except that it does not contain the IS keyword. The declarations here make these subprograms visible to the package user.

```
PACKAGE BasicUtilities IS
   FUNCTION int (invec : BIT_VECTOR) RETURN INTEGER;
   FUNCTION mux (databits : BIT_VECTOR;
                       sel : BIT_VECTOR) RETURN BIT;
   FUNCTION bin (inint, size : INTEGER)
                                    RETURN BIT_VECTOR;
   FUNCTION dcd (bin : BIT_VECTOR) RETURN BIT_VECTOR;
   PROCEDURE consecutive_data
      (SIGNAL target : OUT BIT_VECTOR;
       CONSTANT ti : TIME; CONSTANT n : INTEGER);
END PACKAGE BasicUtilities;
```

Figure 5.24 An Example Package Declaration

Figure 5.25 shows a package body that is associated with the *BasicUtilities* package declaration of Figure 5.24. This body contains descriptions for subprograms declared in Figure 5.24. The descriptions are those of the previous section with the addition of *bin* that is shown in this figure. These subprograms are all compiled at the same time when the package is compiled. Figure 5.26 shows the syntax of a package declaration and a package body.

Subprograms of this package can only be accessed by design units that makes contents of *BasicUtilities* visible by the use of the *use clause*. Furthermore, the package itself must become visible by the use of the *LIBRARY clause* for the library that the package is compiled in. For example, suppose that our *BasicUtilities* is compiled in the *utilities* library. For a design unit to use the subprograms of Figure 5.24, the following statements must appear in the corresponding design file:

```
LIBRARY utilities;
USE utilities.BasicUtilities.All;
```

```
PACKAGE BODY BasicUtilities IS

FUNCTION int: see Figure 4.37

FUNCTION mux: see Figure 4.31

FUNCTION bin (inint, size : INTEGER) RETURN BIT_VECTOR IS
   VARIABLE tmpi : INTEGER := inint;
   VARIABLE tmpb : BIT_VECTOR (size - 1 DOWNTO 0);
BEGIN
   tmpb := (OTHERS => '0');
   FOR i IN 0 TO size - 1 LOOP
      IF ((tmpi MOD 2) = 1) THEN
         tmpb(i) := '1';
      END IF;
      tmpi := tmpi / 2;
   END LOOP;
   RETURN tmpb;
END FUNCTION bin;

--   PROCEDURE inc: see Figure 5.21

--   PROCEDURE consecutive_data: see Figure 5.21

END PACKAGE BODY BasicUtilities;
```
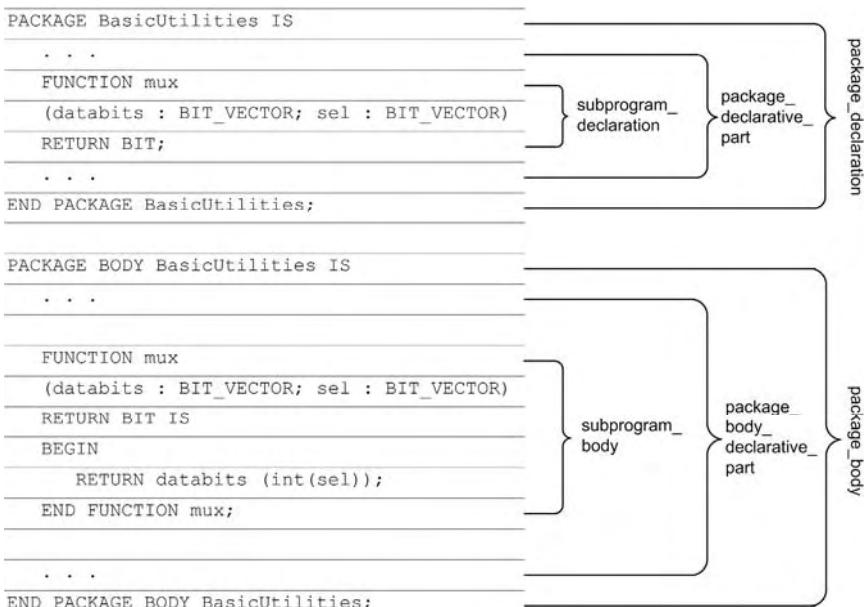
Figure 5.25 Subprogram Definition in Package Body

Figure 5.26 Package Declaration and Body Syntax

The first statement makes visible everything that is in the *utilities* library, and the second statement makes *all* declarations of *BasicUtilities* visible. To avoid name conflicts and to eliminate those declarations that are not being used, individual declarations that are being used could be named instead of using the ALL keyword. For example if we only need *mux* and *dcd* from the above package, the following statement would provide the necessary visibility.

```
USE utilities.BasicUtilities.mux,
    utilities.BasicUtilities.dcd;
```

In the following discussion we assume that the *utilities* library has been created and *BasicUtilities* is compiled in this library.

5.5.2 A Package of Components

While the above discussion concentrated on the use of packages for organizing utility functions, the discussion here focuses on use of packages and libraries for organizing components described by entity-architecture pairs or configuration declarations.

```
LIBRARY utilities;
USE utilities.BasicUtilities.ALL;

ENTITY alu4function IS
   PORT (ai, bi : IN BIT_VECTOR;
         mode : IN BIT_VECTOR (1 DOWNTO 0);
         aluout : OUT BIT_VECTOR);
END ENTITY;
--
ARCHITECTURE customizable OF alu4function IS
   CONSTANT size : INTEGER := ai'LENGTH;
BEGIN
   PROCESS (ai, bi, mode) BEGIN
      CASE BIT_VECTOR (mode) IS
         WHEN "00" =>
            aluout <= bin(int(ai) + int(bi), size);
         WHEN "01" =>
            aluout <= bin(int(ai) - int(bi), size);
         WHEN "10" =>
            aluout <= ai AND bi;
         WHEN "11" =>
            aluout <= ai OR bi;
         WHEN OTHERS =>
            aluout <= bin(0, size);
      END CASE;
   END PROCESS;
END ARCHITECTURE customizable;
```

Figure 5.27 A Design Unit Compiled in our *GenericParts* Library

Consider, for example, the *alu4function* design unit shown in Figure 5.27. Because this design uses the *bin* function, the design file includes the library and use clauses shown. This ALU is a four-function ALU with generic size inputs and outputs. Suppose that this, the generic *dregister* of Figure 5.28, and several other generic designs discussed earlier in this chapter are compiled in a library named *components*. Usually, a VHDL based tool has a compile option for specifying the library that a design is compiled into.

```
LIBRARY utilities;
USE utilities.BasicUtilities.ALL;

ENTITY dregister IS
   PORT (rst, clk : IN BIT; regin : IN BIT_VECTOR;
         regout : OUT BIT_VECTOR);
END ENTITY;
--
ARCHITECTURE synchronous OF dregister IS
   CONSTANT size : INTEGER := regin'LENGTH;
BEGIN
   reg: PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF rst = '1' THEN regout <= bin (0, size);
         ELSE regout <= regin;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE synchronous;
```

Figure 5.28 D-Register Compiled in *GenericParts*

With the compilations named above, our components library will include *decoder*, *alu4function*, *multiplexer_n*, *ssd*, and *dregister*. If a VHDL description of a component is to use any of these design units, the library clause shown below must be included in the design file. In addition, a use clause must be used to make design units of this library visible. This is also shown below:

```
LIBRARY components;
USE components.All;
```

Furthermore to make proper binding to one of the components of this library, e.g., *alu4function*, the following configuration specification must be used.

```
FOR ALL: alu_instances
USE ENTITY components. Alu4function (customizable);
```

A design unit wanting to use all of the components of our *components* library must individually declare all components. This situation can be eased by using a package declaration to contain declarations for these components. Figure 5.29 shows the *GenericParts* package that we compile in our *components* library.

```
PACKAGE GenericParts IS
   COMPONENT dec_n PORT
      (bin_in : IN BIT_VECTOR; en : IN BIT;
       dcd_ou : OUT BIT_VECTOR);
   END COMPONENT;
   COMPONENT alu_n PORT
      (ai, bi : IN BIT_VECTOR;
       mode : IN BIT_VECTOR (1 DOWNTO 0);
       aluout : OUT BIT_VECTOR);
   END COMPONENT;
   COMPONENT mux_n PORT
      (ins : IN BIT_VECTOR;
       s : IN BIT_VECTOR; w : OUT BIT);
   END COMPONENT;
   COMPONENT ssd_f PORT
      (bcd : IN BIT_VECTOR (3 DOWNTO 0);
       display : OUT BIT_VECTOR (1 TO 7));
   END COMPONENT;
   COMPONENT dreg_n PORT
      (rst, clk : IN BIT; regin : IN BIT_VECTOR;
       regout : OUT BIT_VECTOR);
   END COMPONENT;
END PACKAGE GenericParts;
```

Figure 5.29 A Package of Component Declarations

This is only a package of declarations and does not require an associated package body. For a design unit to take advantage of these declarations, it must first use the library clause for visibility into the library. Then, a USE clause must be used to make the declarations of the package visible.

We will complete this section by showing a testbench for *alu4function* that we compiled in our *components* library. This example shows how our *utilities* and *components* libraries, as well as our *BasicUtilities* and *GenericParts* packages are used.

The *alu_n_tester* entity and architecture pair is shown in Figure 5.30. This design uses the *consecutive_data* procedure of the *BasicUtilities* package. Because this package is compiled in the *utilities* library, lines 1 and 2 in this figure are needed to make this procedure visible. Our testbench instantiates *alu_n* according to the declaration of Figure 5.29. Lines 3 and 4 of Figure 5.30 make this declaration visible. The configuration specification of Figure 5.30 binds the *alu_n*

instance to the *alu4function(customizable)* entity-architecture pair. This binding requires visibility to this entity-architecture pair that is satisfied by line 3 and 5 of Figure 5.30.

```
LIBRARY utilities;                                    -- Line 1
USE utilities.BasicUtilities.ALL;                     -- Line 2

LIBRARY components;                                   -- Line 3
USE components.GenericParts.ALL;                      -- Line 4
USE components.ALL;                                   -- Line 5

ENTITY alu_n_tester IS END ENTITY;
--
ARCHITECTURE timed OF alu_n_tester IS
   SIGNAL m : BIT_VECTOR (1 DOWNTO 0) := "00";
   SIGNAL li,ri,ao : BIT_VECTOR (7 DOWNTO 0) := "00000100";
   FOR UUT1 : alu_n USE ENTITY
      components.alu4function (customizable);
BEGIN
   UUT1: alu_n PORT MAP (li, ri, m, ao);
   consecutive_data (m, 123 NS, 13);
   consecutive_data (li, 223 NS, 9);
   consecutive_data (ri, 257 NS, 9);
END ARCHITECTURE timed;
```

Figure 5.30 Using Components and their Declarations

5.6 Sequential Statements

Statements used in processes, functions and procedures of this chapter are regarded as sequential statements. These statements execute when the program flow in a sequential body reaches them. Sequential statements are useful for behavioral descriptions of components and tasks. The syntax of sequential statements is similar to what is found in most software languages, and because of this similarity we will only briefly discuss their syntax here. For this discussion we use the examples presented in the earlier parts of this chapter.

5.6.1 If Statement

The multiplexer example of Figure 5.5 uses a simple form of an if statement. The else-part shown here may be eliminated if not needed. However, the use of keywords IF, THEN, and END IF are required in this statement. Figure 5.31 shows the syntax of the *if statement* used in the *multiplexer* architecture of Figure 5.5. Since this statement itself is a sequential statement, it can enclose another if statement and thus if statements can be nested.
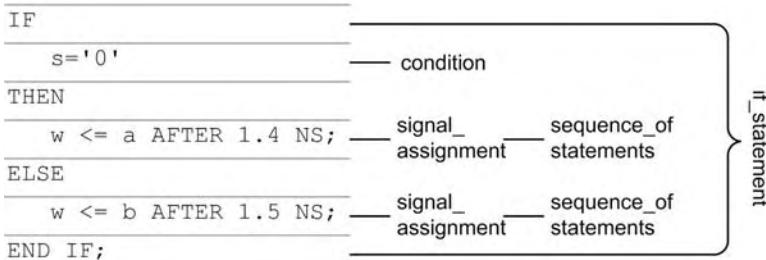
Figure 5.31 Simple if Statement Syntax

Figure 5.8 shows another use of an if statement. This example uses an ELSIF that is followed by a *condition*, the THEN keyword, and a sequential statement that assigns *din* to *qout*.

5.6.2 Loop Statement

VHDL allows *loop statements* in sequential bodies such as processes and subprograms. A loop statement can have an *iteration scheme* such as a *for* or *while* scheme, or it may be without any iteration scheme.

The procedure of Figure 5.16 contains a nesting of two for-loop statements. The outer loop contains a variable assignment, a loop statement, and a sequential signal assignment. The inner loop has three variable assignments. Figure 5.32 shows the syntax details of this statement. The iteration scheme used with this loop statement causes the statements within the loop to be executed n times, while *i* changes from 1 to n. The value of this identifier can be used within the loop and need not be declared.
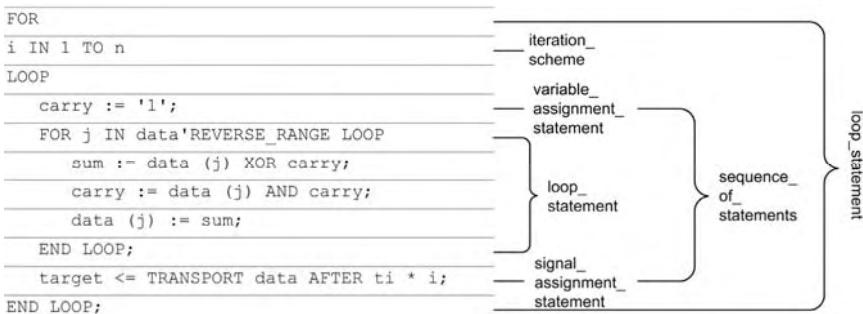
Figure 5.32 Loop Statement with a FOR Iteration Scheme

   VHDL also allows loop statements without an iteration scheme. Such a statement is an infinite loop. The only way to exit from this loop is to use an *exit statement*. For example, the loop shown in Figure 5.33 terminates only when *x* is equal to 25. If this condition does not occur, the looping continues indefinitely.

```
Long_runing : LOOP
   .
   .
   .
   IF x = 25 THEN
      EXIT;
   END IF;
   .
   .
   .
END LOOP long_runing;
```

Figure 5.33 Partial Code for Demonstrating Exiting from a Potentially Infinite Loop

   Next and *exit statements* can be used within loop statements. A next statement reached by the program flow within a loop causes the rest of the loop to be skipped and the next iteration to be taken. An exit statement causes the termination of the loop that it applies to. Both statements can be used optionally with a loop label and a condition, as presented here:

```
NEXT loop_label WHEN condition;
```

The if statement of Figure 5.33 can be replaced with the exit statement shown here:

```
EXIT WHEN x = 25;
```

If the optional *loop label* of the next or the exit statements is not included, the next or exit statements apply to their innermost enclosing loop. Inclusion of this label, however, enables the application of these statements to selected outer loops. Consider the partial code in Figure 5.34.

   While in *loop_2*, if after the execution of *sequential_statemet_4 condition_1* is TRUE, the next-statement causes the remainder of *loop_2* and *loop_1* loops to be skipped, and the next iteration of *loop_1* is taken. Therefore, the value of *i* is incremented and the *sequential_statement_1* is executed after the execution of the *next statement*.

```
loop_1 : FOR i IN 5 TO 25 LOOP
   . . .
   sequential_statement_1;
   . . .
   sequential_statement_2;
   . . .
   loop_2 : WHILE j <= 90 LOOP
   . . .
      sequential_statement_3;
      sequential_statement_4;
      . . .
      NEXT loop_1 WHEN condition_1;
      . . .
      sequential_statement_5;
      sequential_statement_6;
      . . .
   END LOOP loop_2;
   . . .
END LOOP loop_1;END LOOP long_runing;
```

Figure 5.34 Partial Code for Demonstrating Conditional Next Statements in a Loop

5.6.3 Case Statement

Another sequential flow control statement of VHDL is the *case statement*. This statement was used in the *alu4function* example of Figure 5.27. This statement has an optional *case label* followed by a colon that is not used in this example. The statement shown begins with the CASE keyword, followed by a *case expression*, and then several *case statement alternatives* follow. When all alternatives are listed, END CASE keywords and a semicolon end this statement.

Figure 5.35 shows the syntax of the case statement used in the *alu4function* example. As shown, the case expression is *mode* that is casted with BIT_VECTOR for its type to match the various choices that follow. There are five case statement alternatives here that begin with the WHEN keyword. Each alternative has a *choice* (or *choices*) followed by a right arrow and followed by *sequence of statements*.

The last case statement alternative uses OTHERS for its *choice*. Use of OTHERS is only mandatory if the preceding choices have not covered all cases of the *case expression*. Obviously, since in our example *mode* is a two-bit vector and we have covered all its choices, use of OTHERS is not necessary.

If the same group of statements need to execute for several choices, the choices can be ored together with vertical bar (|) characters. An example is shown below:

```
WHEN choice1 | choice2 => statement1; statement2;
```

```
CASE
BIT_VECTOR (mode)                  ──── expression
IS
    WHEN
        "00"
        =>
        aluout <= bin(int(ai) +
                    int(bi), size);
    WHEN "01" =>
        . . .
    WHEN "10" =>
        . . .
    WHEN "11" =>
        . . .
    WHEN
        OTHERS
        =>
        aluout <= bin(0, size);
END CASE;
```

sequence_of_ statements

case_ statement_ alternative

case_statement_alternative

case_statement_alternative

case_statement_alternative

case_statement_alternative
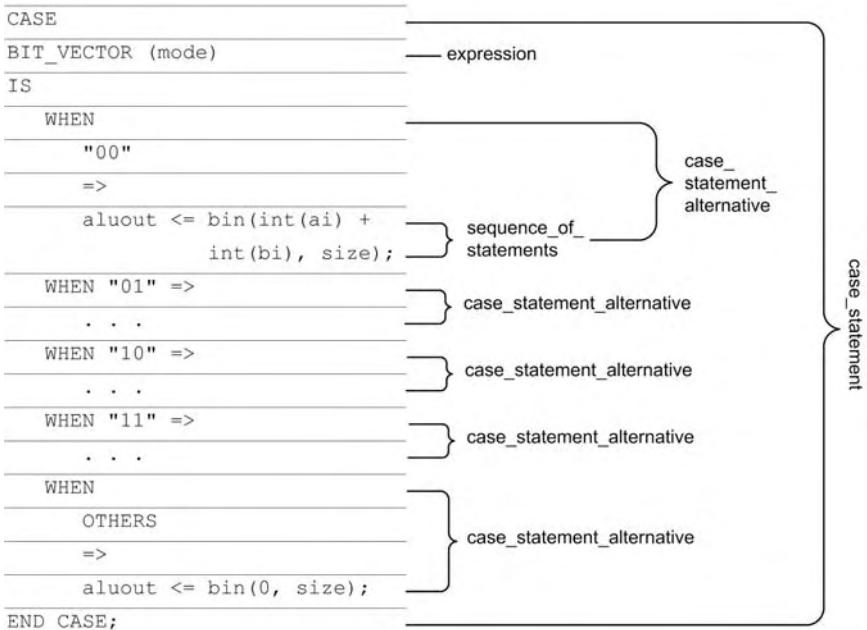
case_statement

Figure 5.35 Syntax Details of C*ase Statement*

5.6.4 Assertion Statement

The *assertion statement* is a useful statement for observing activity in a circuit or defining constraints or conditions in the way a circuit operates. The general format of this statement is:

```
ASSERT assertion_condition
REPORT "reporting_message" SEVERITY severity_level;
```

The statement is said to "occur" when the Boolean *assertion_condition* expression becomes FALSE. At this point, the *reporting_message* is issued, and the simulator takes the action specified by the *severity_level* parameter. The latter parameter can be NOTE, WARNING, ERROR or FAILURE. The ERROR or FAILURE severity levels causes the simulation to stop after issuing the *reporting_message* and a simulation error or failure message. The other two *severity_levels* cause appropriate massages to be issued and the simulation to continue. The REPORT keyword and its following *reporting_message*, as well as the SEVERITY keyword and *severity_level*, are optional parts of the assertion statement. If the REPORT is not present, only a system message is issued, and the absence of the SE-

VERITY keyword and its accompanying *severity_level* defaults to the ERROR severity level. The exact series of actions taken by different severity level parameters is simulation-dependent.

An assertion statement with a FALSE condition always occurs. Such a statement is equivalent to the sequential statement shown below:

```
REPORT "reporting_message" SEVERITY severity_level;
```

Sequential and concurrent VHDL bodies can use assertion statements. A sequential assertion statement issues the *reporting_message* if its *assertion_condition* is FALSE when the program flow reaches the statement. Figure 5.36 shows an example illustrating this use of the assertion statement.

```
ARCHITECTURE sync_timed OF dregister IS
   CONSTANT size : INTEGER := regin'LENGTH;

BEGIN
   reg: PROCESS (clk)
      VARIABLE last_edge, duration : TIME := 0 NS;
   BEGIN
      duration := NOW - last_edge;
      last_edge := NOW;
      ASSERT NOT (duration < 3 NS)
         REPORT "Clock Width Too Short"
         SEVERITY NOTE;
      IF (clk = '1') THEN
         IF rst = '1' THEN regout <= bin (0, size);
         ELSE regout <= regin;
         END IF;
      END IF;
   END PROCESS reg;
END ARCHITECTURE sync_timed;
```

Figure 5.36 Architecture for Dregister Using Sequential ASSERT

As shown in this figure, *sync_timed* is an alternative architecture for *dregister* of Figure 5.28. The *reg* process in this architecture uses *last_edge* and *duration* variables to keep track of pulse widths of *clk*. The *duration* variable is updated with every edge of the clock. After it is updated an assert statement issues a message if the value of this duration is less than 3 nanoseconds.

The use of NOT with the ASSERT makes the message to correspond to the condition that appears after NOT. So we can read it as: if duration is less than 3 ns, display "clock width too short". The following paragraph elaborates on this.

The expression "ASSERT condition ..." reads as "make sure that this condition is satisfied; otherwise, ...". Therefore, it is clear that a good case must be used as the condition of the statement. The problem arises, however, in the many situations where the good cases are too many to list, and it is easier to write the complement of the unwanted case. For checking errors, then, we will always use *ASSERT(NOT(unwanted_cases))*. After canceling the two negations, this is equivalent to *ASSERT(wanted_cases)*. For cases in which grouping good cases is as easy as grouping unwanted cases, this negation is not necessary. For example, the following statement triggers when *numb* becomes negative.

```
ASSERT numb >= 0;
```

For cases where violation of constraints must be continuously checked and reported, concurrent assertion statements should be used. These cases include checking of timing constraints such as pulse width, setup time, and hold time. A concurrent assertion statement can be placed in the statement part of an architecture or in the statement part of an entity declaration. In either case, it is observed at all times and it occurs when an event causes its condition to become FALSE. Examples of concurrent assertions for setup and hold time checks are presented in the next chapter.

5.7 Summary

The focus of this chapter was on description of hardware using sequential statements. A sequential statement offers a convenient way of describing behavior of a hardware component. VHDL bodies for inclusion of sequential statements are process statements and subprograms. Details of these constructs and various forms of their utilizations were discussed in this chapter. After a thorough treatment of this subject, the chapter discussed VHDL library structures and packages. We showed how packages can be used for inclusion of subprograms and component declarations.

Problems

5.1 In the testbench shown below, show waveforms applied to the CUT inputs up to 80 NS.

```
ENTITY tester IS END TESTER;
--
```

```
ARCHITECTURE data OF tester IS
    SIGNAL reset, clock : BIT := '1';
    SIGNAL xx : BIT_VECTOR ( 3 DOWNTO 0) := "1101";
    SIGNAL z : BIT;
BEGIN
    CUT : CircuitUnderTest PORT MAP( xx, reset, clock, z );
    PROCESS BEGIN
       reset <= '1';
       WAIT FOR 26 NS;
       reset <= '0';
       WAIT FOR 4 NS;
       WAIT;
    END PROCESS;
    --
    clock <=NOT clock AFTER 5 NS WHEN NOW <= 70 NS ELSE '0';
    --
    PROCESS BEGIN
       WAIT UNTIL clock = '1';
       WAIT FOR 3 NS;
       xx <= xx(0) & xx(3 DOWNTO 1);
    END PROCESS;
END ARCHITECTURE;
```

5.2 Show the result of the following code for 120 NS.

```
ENTITY top IS END ENTITY;
--
ARCHITECTURE tester OF top IS
    SIGNAL a, b, c : INTEGER;
BEGIN
    PROCESS
       VARIABLE vc : INTEGER := 0;
    BEGIN
       vc := a * 2 * b + 11;
       WAIT FOR 55 NS;
       WAIT;
    END PROCESS;
    PROCESS BEGIN
       WAIT FOR 20 NS;
       a <= ( a + 2 ) MOD 4;
    END PROCESS;

    PROCESS BEGIN
       b <= 1;
       WAIT FOR 40 NS; b <= 5;
       WAIT FOR 40 NS; b <= 7;
       WAIT FOR 40 NS; b <= 11;
       WAIT;
    END PROCESS;
```

```
   PROCESS (a)
      VARIABLE vc : INTEGER := 5;
   BEGIN
      vc := a * 2 * b;
      c <= vc + 1;
   END PROCESS;
   PROCESS (a, b, c) BEGIN
      ASSERT FALSE
      REPORT "At: " & TIME'IMAGE(NOW) &
             " a= " & INTEGER'IMAGE(a) &
             " b= " & INTEGER'IMAGE(b) &
             " c= " & INTEGER'IMAGE(c) SEVERITY NOTE;
   END PROCESS;

END ARCHITECTURE;
```

5.3 Given the following description show the waveforms on *out1* and *out2* outputs. The problem involves signals, sensitivity list, variable initialization, variable assignments, and scheduling.

```
ENTITY signals IS
   PORT (ain, bin, cin : IN BIT; out1, out2 : OUT BIT);
END signals;
   --
ARCHITECTURE nothing OF signals IS
   SIGNAL cx : BIT;
BEGIN
   PROCESS
      VARIABLE cv : BIT := '0';
   BEGIN
      CASE BIT_VECTOR'(ain, bin, cin) IS
         WHEN "001" =>
           cx <= NOT ain AFTER 10 NS;
           cv := cx;
          WHEN "010" =>
           cx <= NOT bin AFTER 10 NS;
           cv := cx;
        WHEN "100" =>
           cx <= NOT cin AFTER 10 NS;
           cv := cx;
        WHEN "101" =>
           cx <= cv AFTER 15 NS;
        WHEN "110" =>
           cx <= cin AFTER 10 NS;
        WHEN OTHERS => NULL;
      END CASE;
      out1 <= cv;
      out2 <= cx;
      WAIT ON ain, bin, cx;
   END PROCESS;
END nothing;
```

5.4   Write a function for the carry output of a full-adder. Write a function for the sum output of a full-adder.

5.5   Write a function, *inc_bits*, that returns the 4-bit increment of its 4-bit input vector. Write Boolean expressions for the four bits of the output.

5.6   Write a procedure, *apply_bit*, such that bits of a 24 bit wide string input to the procedure are applied to its target signal according to the specified time interval. Make sure no unnecessary transactions occur on the target of the procedure. A sample call to this procedure is shown here.

```
apply_bit (target, "110001000100001111001010", 300 NS);
```

5.7   Write an unconstrained odd parity checker function. The input is of *std_logic_vector* type and the output of *std_logic* type. Treat values 'Z' and 'X' as '1'. The function returns the XOR results of all its input bits.

5.8   Write a procedure that assigns consecutive binary numbers to its output BIT_VECTOR lines. The parameters of the procedure are an 8-bit target output and a time period. When called, it will assign sequential binary numbers from 0 to 255 to its target signal output. These numbers are distanced by the amount of the constant associated with the period parameter.

5.9   Write a procedure, *apply_bit*, such that bits of an unconstrained string input to the procedure are applied to its target signal according to the specified time interval. Make sure no unnecessary transactions occur on the target of the procedure. A sample call to this procedure is shown here:

```
apply_bit (target, "110001000100001111001010", 300 NS);
```

5.10 Write a procedure that issues a warning message if a positive pulse on its *go* output appears any sooner than 200 NS than a complete positive pulse on its *ready* input. Use assertion statements for issuing the error. The only acceptable case is *go* to appear at least 200 ns after *ready*. Any other case and any overlapping of positive pulses must be reported as a violation. This procedure is to be called from a concurrent body of VHDL. Use the standard *std_logic* type.

5.11 A signal is to be passed to a procedure. The procedure is to count all transactions that have occurred on the signal from time 0 to the time of calling of the procedure. The procedure returns this integer count via its second argument. Write the procedure using *sig* for input signal and *cnt* for output transaction count. The procedure may be called from a sequential body such as a process statement.

5.12 Write an assertion statement to issue a warning message if a negative pulse shorter than 1 us appears on the input clock.

5.13 Write an assertion statement to issue a warning message if the frequency of the observing clock is lower than 100 KHz. If the clock is too slow in some MOS circuit, the circuit loses information. Assume 50% duty cycle.

5.14 Show VHDL code for a 4-bit counter that counts the following sequence: 0100, 0001, 1011, 1010, 0110, 1111, 0111, 0000, 1000. After a clock pulse, a present count of 0001 results in the next count of 1011, and a present count of 1000 results in the next count of 0100 (roll over). Code this circuit by a memory, *mem*, that is initialized by reading a text file that contains the count sequence. You should read the contents of *mem.dat* file and load it into *mem*. Show the complete VHDL code including necessary declarations and contents of the memory file.

Suggested Reading

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL hardware description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Perry, Douglas L., *VHDL: Programming By Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

6

VHDL Language Utilities and Packages

The previous chapters discussed VHDL language constructs for structure and RT level descriptions. We discussed major concurrent and sequential bodies of VHDL. Many of the language issues like types, operators, and attributes were used in the discussions of the previous chapters, but we never discussed their language details and their variations. This chapter focuses of language utilities. We discuss types, operators, type and operator related utilities, and attributes. VHDL attributes perform certain functions on types, entities and objects of VHDL.

6.1 Type Declarations and Usage

VHDL is a strongly typed language. Type declarations must be used for definition of objects and their types. Operations in VHDL are defined for specific types of operands. The STANDARD package in the STD library defines basic types such as BIT or INTEGER; other types also can be defined. Basic operators also can be defined to perform operations on operands of these new types. The general classes of types we will discuss in this chapter include the *scalar*, *physical composite*, and *file* types.

6.1.1 Enumeration Type for Multi-Value Logic

The basic scalar type is enumeration. This is defined as a set of all possible values that such a type can have. The BIT type of the

161

STANDARD package is an enumeration of '0' and '1', and the BOO-LEAN type of this package is an enumeration of FALSE and TRUE. CHARACTER, also in this package, is defined as the set of 256 extended ASCII characters. Other enumeration types can be defined by the use of the type declaration construct.

The VHDL standard logic (*std_logic_1164*) package defines the *std_logic* type that takes any of the nine values, 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', and '-'. These values stand for undefined, unknown, pulled low, pulled high, float, weak unknown, weak low, weak high, and don't care. The *std_logic_1164* package uses the following for declaring this type.

```
TYPE std_logic IS('U','X','0','1','Z','W','L','H','-');
```

This type has nine enumeration elements. The position of the left-most element is 0 and other enumeration elements are numbered accordingly. The left-most element is the default value of an object that is declared of this type.
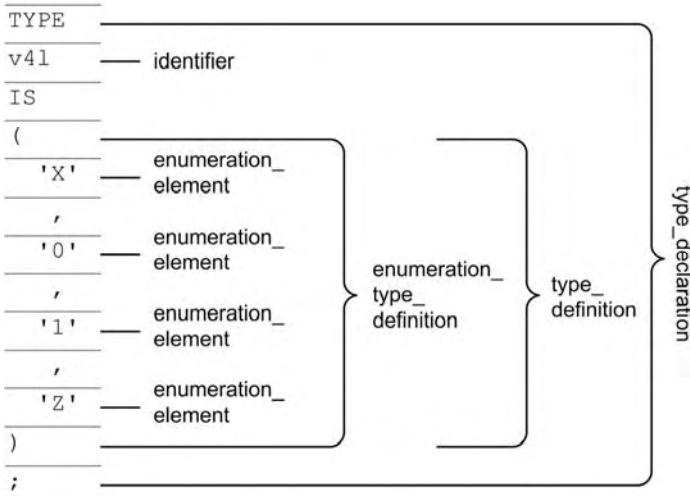
For illustrations of this chapter we use a four-value type that is compatible with the logic value system of the Verilog hardware description language. The type mark we use for this type is v4l that is for Verilog 4-value logic. The Verilog language uses a four-value system that consists of 'X', '0', '1', and 'Z'. In most cases, 'X' is considered the default value. Based on this, we defined our v4l as shown below.

```
TYPE v4l IS ('X','0','1','Z');
```

When this type is declared in the declarative part of an architecture, or if it is made visible to a design by use of a package, *v4l* can be used to declare objects that can assume any of the four values, 'X', '0', '1', or 'Z'. We assume the *VerilogLogic* package exists and it is compiled in the *utilities* library. This package includes the declaration of *v4l* type and other related utilities that we will discuss in this chapter.

Figure 6.1 shows syntax details of the above type declaration. Enclosed in parentheses are four enumeration elements that are separated by commas. This forms the enumeration type definition which, together with the TYPE keyword, the *v4l* identifier, and the IS keyword, forms a type declaration for declaring *v4l*.

Instead of using single characters enclosed in quotes, enumeration elements can be identifiers formed by a string of characters. For example, individual mnemonics of a processor can be used as enumeration elements to declare an instruction set in a computing machine.

Figure 6.1 Syntax Details of a Type Declaration

6.1.1.1 Modeling a four-value Inverter. In the v4l four-value logic system, '0' and '1' are for low and high logic values, respectively. The 'Z' value is for the high impedance or open, and the 'X' value is unknown or conflict. Input-to-output mapping of an inverter in this value system is shown in Figure 6.2.

Figure 6.2 Input-Output Mapping of an Inverter in *v4l* Logic Value System

As shown in the figure, inverting an unknown or high impedance input ('X' or 'Z') results in an unknown. Figure 6.3 shows the entity declaration and the architecture body of an inverter that uses this logic value system.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY vlog_inv IS
   GENERIC (tplh, tphl : TIME := 0 NS);
   PORT (w : OUT v4l; a : IN v4l);
END ENTITY vlog_inv;
--
ARCHITECTURE conditional OF vlog_inv IS
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE conditional;
```

Figure 6.3 VHDL Description of an Inverter in *v4l* Logic Value System

The *v4l* type is made visible to the description in Figure 6.3 when we specify the use of the package that contains it. This visibility enables us to use *v4l* for the type of the ports on the *v4l* entity. The architectural description in Figure 6.3 uses a conditional signal assignment statement to implement the table in Figure 6.2.

In this and other codes that we use *v4l*, we try to follow Verilog rules as closely as possible. In Verilog, the output of a primitive gate comes first in the list of ports. Thus we have listed *w* first. Also, since in Verilog default delays are 0, we have used 0 ns for our tplh and tphl generics.

6.1.1.2   Modeling a four-value NAND Gate. A two-input NAND gate in the *v4l* logic value system can be modeled according to the input-output mapping shown in Figure 6.4.

Figure 6.4 Input-Output Mapping of a NAND Gate in *v4l* Logic Value System

Figure 6.5 shows the VHDL description for an interface and an architecture of a *vlog_nand2* entity. The input and output ports of this entity are of the *v4l* type, whose declaration is included in the *VerilogLogic* package. A conditional signal assignment in the statement part of the *conditional* architecture of the *vlog_nand2* entity is used for implementing the table in Figure 6.4.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY vlog_nand2 IS
   GENERIC (tplh, tphl : TIME := 0 NS);
   PORT (w : OUT v4l; a, b : IN v4l);
END ENTITY vlog_nand2;
--
ARCHITECTURE conditional OF vlog_nand2 IS
BEGIN
   w <= '1' AFTER tplh WHEN (a='1') NAND (b='1') ELSE
        '0' AFTER tphl WHEN (a='1') AND (b='1') ELSE
        'X' AFTER tplh;
END ARCHITECTURE conditional;
```

Figure 6.5 VHDL Description of a NAND Gate in *v4l* Logic Value System

6.1.1.3   Initial Values of Enumeration Types. With declaration of objects, an initial value can optionally be specified using the initial value expression that follows the := symbol. If this symbol and its following expression are not present in the declaration of an object, a default initial value that depends on the type of the object is used. For the enumeration types, this value is the left-most enumeration element. For the gates in Figure 6.3 and 6.5, which is the left-most element of the *v4l* type, is the initial value for all the input and output ports. Had we used the 'Z', '0', '1', 'X' ordering for the definition of the v4l type, default initial values of all the objects of this type would have been 'Z'.

6.1.2 Using Real Numbers

Besides the enumeration type, other types of the scalar classes are the INTEGER and REAL types. Both of these types are defined in the STANDARD package. The exact range of these types is implementation dependent, but they generally range from a small negative number to a large positive number depending on the word size of the host machine. For the INTEGER type, these numbers are restricted to integers.

We use a load-dependent model of a CMOS inverter in order to demonstrate the use of REAL and INTEGER numbers and their rela-

tionship to each other. The inverter contains its own pull-up and pull-down resistance values and adjusts its delays according to the load capacitance at its output node. This capacitance value is passed to the inverter model by use of the generic parameters.

The VHDL model of a CMOS NOT gate is shown in Figure 6.6. The *cmos_not* entity contains a generic formal parameter of REAL type. The entity declaration part of this figure specifies the *rpu* and *rpd* constants and their values. Declaring these constants in the entity declaration causes them to be visible to all architectures that are written for this entity.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY cmos_not IS
   GENERIC (c_load : REAL := 0.066E-12); --Farads
   PORT (w : OUT v4l; a : IN v4l);
   CONSTANT rpu : REAL := 3000.0; --Ohms
   CONSTANT rpd : REAL := 2100.0; --Ohms
END ENTITY cmos_not;
--
ARCHITECTURE rc_timed OF cmos_not IS
   CONSTANT tplh : TIME :=
      INTEGER (rpu * c_load *1.0E15) * 3 FS;
   CONSTANT tphl :
      TIME := INTEGER (rpd * c_load *1.0E15) * 3 FS;
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE rc_timed;
```

Figure 6.6 An Inverter Model with *RC* Timing Parameters

The declarative part of the *rc_timed* architecture in Figure 6.6 defines the *tplh* and *tphl* constants in terms of the pull-up or the pull-down resistances and the load capacitance. The constructs used for these declarations are constant declarations that contain expressions for their values. Since all the generics and constants are defined at the initialization time, expressions based on these parameters can be used for the initial values of objects or for the values of other constants.

6.1.3 Type Conversions

The discussion in the following paragraphs refers to the *cmos_not* example of Figure 6.6 and focuses on type conversions. We show now

REAL, INTEGER, and physical type TIME are mixed and how conversions from one to the other are done.

Because of the types of the *tplh* and *tphl* in figure, the result of the evaluation of their constant value expressions must be of type TIME. A constant of type TIME can be formed by multiplying an integer number by a valid unit of this type. Since the resistance and capacitance values are of the REAL type, their multiplication result is a floating point number. A floating-point number must be converted to an integer and it must be given a unit of TIME in order to be used for an object of type TIME. For this reason, in the constant value expressions of the *rc_timed* architecture of *cmos_not*, we have used explicit REAL to INTEGER type conversion and have multiplied the resulting integer by an appropriate time unit. Explicit type conversions, such as those demonstrated here, can be done for closely related types.

Consider the constant expression for the *tplh* constant. This expression converts the multiplication of *rpu* and *c_load* to an integer by use of the explicit type conversion. Before this type conversion takes place, however, the *RC* product is multiplied by a factor of 1E15. This is done because normal pull resistance and load capacitance multiplications result in small fractions, and converting these small floating point numbers to the INTEGER type results in zero. The 1E15 factor is compensated for by using the femtosecond (FS = $1E^{-15}$ sec) time unit for the overall expression. Multiplication by a factor of 3 is also included in the constant expression of the *tplh* constant. This factor is used to account for the exponentiality of the waveforms. We are approximating the delay values that are based on exponential waveforms by linear *RC* equations. An exponential function takes about *3*RC* to complete its transition from one value to another.

6.1.4 Physical Types

Physical types in VHDL are another type in the scalar class. Values of a physical type are used with units defined in the type definition. Type TIME is a physical type that is defined in the STANDARD package, and it is used for measuring time. The units of this type have been defined as FS, PS, NS, US, MS, SEC, MIN, and HR. Other physical types for measuring other quantities such as distance, temperature, resistance, and capacitance can also be defined.

Figure 6.7 shows the definition of *capacitance* as a type for representing capacitance. This definition consists of the name of the physical type, a rang constraint, a base unit declaration (declaration of *ffr*), and several secondary unit declarations (declarations of *pfr* to *far*). The units for this type range from *ffr* (femtofarads) to *far* (far-

ads). The base unit is *ffr*, and all other units are defined in terms of this unit. Other units can be added to this type, provided they are multiples of the base unit. Only integer numbers can be used for the bounds of the range constraint of a physical type. Since we have specified 0 to INTEGER'HIGH for the range constraint, negative capacitance values cannot be assigned to an object of type *capacitance*, although larger values of this type may be used in expressions. Another example of a physical type definition is that of *resistance*, as shown in Figure 6.8. As defined by this type, an object of type *resistance* can have units ranging from *l_o* (milliohms = $10^{-3}\ \Omega$) to *g_o* (gigaohms = $10^9\ \Omega$).

```
TYPE capacitance IS RANGE 0 TO INTEGER'HIGH
   UNITS
     ffr;  -- Femto Farads (base unit)
     pfr = 1000 ffr;
     nfr = 1000 pfr;
     ufr = 1000 nfr;
     mfr = 1000 ufr;
     far = 1000 mfr;
   END UNITS;
```

Figure 6.7 Type Definition for Defining the C*apacitance* Physical Type

To illustrate the use of the *capacitance* and the *resistance* physical types, we show them in an alternative description for the *inv_rc* inverter. (The definitions for these types are assumed to be included in the *BasicUtilities* package; if they are included, the definitions are available to designs that have a use clause specifying application of the *BasicUtilities* package.)

```
TYPE resistance IS RANGE 0 TO INTEGER'HIGH
   UNITS
     l_o;  -- Milli-Ohms (base unit)
     ohms = 1000 l_o;
     k_o = 1000 ohms;
     m_o = 1000 k_o;
     g_o = 1000 m_o;
   END UNITS;
```

Figure 6.8 Type Definition for Defining the *Resistance* Physical Type

Figure 6.9 shows an entity declaration and a partial architectural description for the *cmos_not* that takes advantage of the *resistance* and the *capacitance* physical types. Except for the types of the generics and the constants, this description is the same as that in Figure 6.6. The *cmos_not* entity declaration in Figure 6.9 has a ge-

neric parameter of type capacitance. This declaration also defines the pull-up and pull-down resistances in terms of the resistance type.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
USE utilities.BasicUtilities.ALL;

ENTITY cmos_not IS
   GENERIC (c_load : capacitance := 66 ffr);
   PORT (w : OUT v4l; a : IN v4l);
   CONSTANT rpu : resistance := 3 k_o;
   CONSTANT rpd : resistance := 2.1 k_o;
END ENTITY cmos_not;
--
ARCHITECTURE rc_timed OF cmos_not IS
   CONSTANT tplh : TIME :=
      (rpu / 1 l_o) * (c_load / 1 ffr) * 3 FS / 1000;
   CONSTANT tphl : TIME :=
      (rpd / 1 l_o) * (c_load / 1 ffr) * 3 FS / 1000;
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE rc_timed;
```

Figure 6.9 Using *Resistance* and *Capacitance* Physical Types

The *rc_timed* architecture of the *cmos_not* uses two constant declarations for declaring propagation delay parameters and assigning constant values to them. The expression evaluating the constant value for *tplh* uses *rpu* and *c_load* to evaluate the low-to-high propagation delay. In the first set of parentheses in the constant expression of this parameter, the *rpu* parameters are divided by the base unit for the *resistance* physical type. This results in an integer representing the value of *rpu* in terms of *l_o* (10^{-3} Ω). Similarly, in the second set of parentheses, *c_load* is divided by the *capacitance* base unit. This converts any capacitance value that is associated with the *c_load* generic parameter to an integer representing the amount of the capacitance in *ffr* (10^{-15} F). Multiplying these two sets of parentheses results in an integer *RC* value which is scaled up by a factor of 10^{18}. Using FS (10^{-15} s) and dividing the *tplh* constant expression by 1000 compensates for the use of *l_o* and the *ffr* units. As shown in the description in Figure 6.9, a factor of 3 is used in order to account for the exponentiality of the waveforms. The constant expression for the *tphl* delay parameter is similar to that of the *tplh*. Dividing a physical type by one of its units removes the type from it and converts it to an integer. For best precision, the base unit should be used. Had we divided *rpu* by *k_o* instead of *l_o*, *rpu* values would have been rounded off to the smallest *k_o* values.

The VHDL multiplication operator is defined for multiplying integer and floating point numbers. It is also valid to multiply an integer or a floating-point number by a physical type. Multiplication of two physical types, however, is not defined for the standard multiplication operator. Dividing the *resistance* and *capacitance* physical types by their base units enabled the use of the standard multiplication operator in the timing equations in Figure 6.9.

6.1.5 Array Declarations

The VHDL language includes constructs that can be used to declare multidimensional array types; these array types can then be used to declare objects. Array elements must all be of the same type. Arrays can be indexed with the normal integer indexing or indexed using the elements of an enumeration type. Arrays can be unconstrained, meaning that their range can be left unspecified.

A VHDL array type declaration begins with the keyword TYPE. The declaration specifies the name of the type that is being declared, the range of the array, and the type of each element in the array. Figure 6.10 shows the declarations of *v4l_byte*, *v4l_word*, *v4l_4by8*, *v4l_1kbyte*, and *v4l_8cube*. The elements of *v4l_byte*, *v4l_word*, *v4l_4by8*, and *v4l_8cube* are of the previously defined *v4l* type, and the type of the eight elements of *v4l_1kbyte* is *v4l_byte* defined in the first line of the figure.

```
TYPE v4l_byte IS ARRAY (7 DOWNTO 0) of v4l;
TYPE v4l_word IS ARRAY (15 DOWNTO 0) of v4l;
TYPE v4l_4by8 IS ARRAY (3 DOWNTO 0, 0 TO 7) of v4l;
TYPE v4l_1kbyte IS ARRAY (0 to 1023) OF v4l_byte;
TYPE v4l_8cube IS ARRAY (0 TO 7, 0 TO 7, 0 TO 7) of v4l;
```

Figure 6.10 Declaring Array Types

Once types declared in Figure 6.10 become visible to a design, they can be used to declare objects. For example, an 8-bit *v4l* signal should be declared as:

```
SIGNAL s_byte : v4l_byte
             := "ZZZZZZZZ";
```

The initial values of the eight lines of *s_byte* signal are all 'Z'. For the individual bits of *s_byte*, the 'Z' value overrides 'X', which is the default initial value for the *v4l* type. Recall that the left-most element is the default initial value in an enumeration type, which was 'X' in the declaration of *v4l*.

Other forms of array value assignments or initial values, particularly useful for larger arrays, are also allowed in VHDL. An alternative to the above initialization of *s_byte* bits to 'Z' is using what is referred to as an array *aggregate*, which will be discussed in detail in the next section. In using this option, signal *s_byte* can be declared and initialized as:

```
SIGNAL s_byte : v4l_byte
              := ('Z','Z','Z','Z','1','1','1','1');
```

This statement initializes bits 7 down to 4 of *s_byte* to 'Z' and bits 3 down to 0 of this signal to '1'. This initialization uses an array aggregate with positional association, meaning that each value is associated with the array element in its corresponding position.

Assigning or initializing signals can also be done with named association. For example, array index 5 can be assigned value 'Z' if an array aggregate contains 5 => 'Z'. The keyword OTHERS can be used in array aggregate named association to refer to all indexes for which a value is not specified prior to the appearance of OTHERS. In the following declaration,

```
SIGNAL s_byte : v4l_byte
              := (5 => 'Z', OTHERS => '1');
```
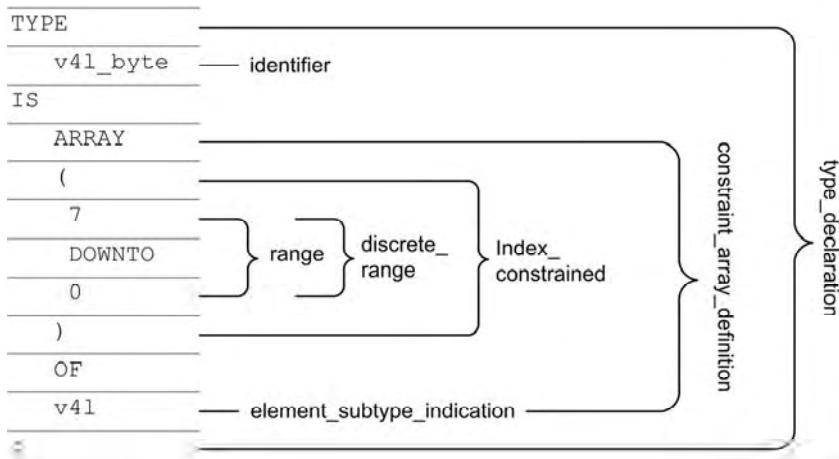
s_byte(5) is given the initial value of 'Z' and all other bits of *s_byte* are initialized to '1'. A range is also allowed in a named association. In this case, a range specification groups contiguous indexes of array elements that are to receive the same value. As shown below, bit 1 and 0 of *s_byte* are initialized to 'Z', and all other bits are given a '1' value.

```
SIGNAL s_byte : v4l_byte
              := (1 DOWNTO 0 => 'Z', OTHERS => '1');
```

Shown below is another example of array initialization. This statement initializes bits 1 and 0 of *s_byte* to 'Z', bits 3 and 4 to 'X', and all other bits (2, 5, 6, and 7) to '1'. This example indicates that the index range can be in any direction as long as it covers valid array indexes. Even though, *v4l_byte* is declared with a descending range (Figure 6.10), specifying array element values in an array aggregate with an ascending range (3 to 4) is allowed.

```
SIGNAL s_byte : v4l_byte
              := (1 DOWNTO 0 => 'Z',
                  3 TO 4 => 'X',
                  OTHERS => '1');
```

Figure 6.11 shows the syntax details of the type declaration used to define the *v4l_byte* type in Figure 6.10. This declaration specifies the range of the arrays and the type of its elements. The range defines the upper and the lower bounds of the array. The DOWNTO descending range specification causes the left-most bit and right-most bits of an object of type *v4l_byte* to have indices of 7 and 0, respectively.

Figure 6.11 Syntax Details of an Array Type Declaration

Figure 6.10 also shows the declaration of a two-dimensional array type, namely *v4l_4by8*. This declaration uses two range specifications separated by commas. The first is a descending range and the second is an ascending range. The index (3, 0) references the upper left bit of an object of type *v4l_4by8*.

Referencing an element or groups of elements in an array can be achieved by indexing or by using slice specifications. To reference an array element by indexing, an index for each of the ranges in the array must be specified. To reference an array slice, a discrete range should be specified. Figure 6.12 shows signal declaration and several valid assignments to signals of the types declared in Figure 6.10. The first declaration in Figure 6.12 declares *s* as a scalar of type *v4l*. The next two declarations define *s_byte* and *s_word* as one-dimensional arrays of *v4l*. The *s4by8* array is a 4-by-8 two-dimensional array of *v4l*, and the *s_1kbyte* signal is a one-dimensional array of size 1024, whose elements are the *v4l_byte* type. The last declaration shown here is *s_8cube* of type *v4l_8cube*. This is a three-dimensional array whose elements are of *v4l* type.

```
ARCHITECTURE assign OF array_test IS
   SIGNAL s : v4l;
   SIGNAL s_byte : v4l_byte;
   SIGNAL s_word : v4l_word;
   SIGNAL s_4by8 : v4l_4by8;
   SIGNAL s_1kbyte : v4l_1kbyte;
   SIGNAL s_8cube : v4l_8cube;
BEGIN
   SA1: s_byte <= v4l_byte ( s_word (11 DOWNTO 4) );
   SA2: s <= s_4by8 (0, 7);

   SA3: s_byte <= s_1kbyte (27);
   SA4: s <= s_1kbyte (23)(3);

   SA5: s_byte <= s_byte (0) & s_byte (7 DOWNTO 1);
   SA6: s_byte (7 DOWNTO 4) <=
           s_byte(2) & s_byte(3) & s_byte(4) & s_byte(5);

   SA7: s_byte (7 DOWNTO 4) <=
           (s_byte(2), s_byte(3), s_byte(4), s_byte(5));
   SA8: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <=
           s_byte (5 DOWNTO 2);

END ARCHITECTURE assign;
```

Figure 6.12 Signal Assignments Based on Signal Declarations.

The first signal assignment (statement labeled *SA1*) in Figure 6.12 assigns a slice of *s_word* to all of *s_byte*. The next assignment (*SA2*) indexes a bit of the *s_4by8* two-dimensional *v4l* array, and assigns this bit to the *s* signal. Multidimensional arrays such as *s_4by8* can only be indexed and cannot be sliced. Therefore, it is more appropriate to use one-dimensional arrays of vectors such as those of *v4l_1kbyte* type for declaring hardware memories.

SA3 signal assignment in Figure 6.12 assigns byte at location 27 of *s_1kbyte* to the left-hand-side signal *s_byte*.

Multi-indexing of arrays is allowed. *SA4* assignment in Figure 6.12 shows bit 3 of byte at location 23 of *s_1kbyte* is selected and assigned to *s*. The left-hand *s* signal is a scalar of type *v4l*.

Concatenations are allowed on the right hand sides of signal assignments. *SA5* uses indexing and slicing to form a vector to assign to *s_byte*. The result is right rotate of the *s_byte* vector. The assignment labeled *SA6* is another right-hand-side concatenation. This assignment puts bits 2, 3, 4 and 5 of *s_byte* into bits 7 down to 0 of this vector. Bits 2, 3, 4 and 5 had to be individually selected, because this ordering is ascending and *s_byte* is descending.

Demonstrated by *SA7* and *SA8* are the use of aggregate operation on the right and left of signal assignments. Unlike concatena-

tions that are only allowed on the right hand side, the aggregate operation is allowed on the right and left. An aggregate operation can select any bit of a vector in any order. However, only elements of a vector and not a slice of a vector can be selected. Figure 6.13 shows what happens to bits of *s_byte* after assignment *SA7* has taken place. Note that *SA6* results in the same assignment.


```
SA7: s_byte (7 DOWNTO 4) <=
        (s_byte(2), s_byte(3), s_byte(4), s_byte(5));
```
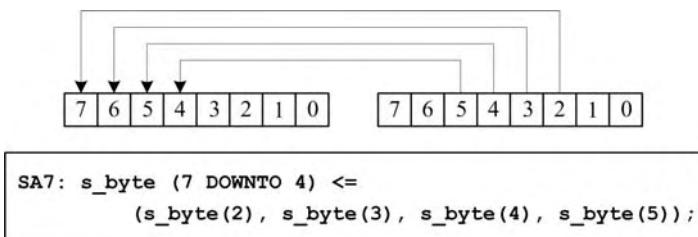
Figure 6.13 Reversing Bits of *s_byte*

Slicing an array with a range in the opposite direction of its declared range is considered to be a constraint error. Slicing a multi-dimensional array is not possible, but an array whose elements are themselves arrays, such as *s_1kbyte*, can be indexed and then sliced. The following selects bits 3 and 2 of byte 219 of the *s_1kbyte* array signal.

```
s_1kbyte (219) (3 DOWNTO 2)
```

6.1.5.1 Initializing Multidimensional Arrays. Initial values for a one-dimensional array type signal must be placed in a set of parentheses and should follow the : = symbol in the signal declaration. The initial values of individual array elements should be separated by commas. Nested sets of parentheses should be used for multi-dimensional arrays. In this case, the top level set of parentheses corresponds to the left-most range of the array.

```
SIGNAL s_4by8 : v4l_4by8 :=
        (
                ( '0', '0', '1', '1', 'Z', 'Z', 'X', 'X' ),
                ( 'X', 'X', '0', '0', '1', '1', 'Z', 'Z' ),
                ( 'Z', 'Z', 'X', 'X', '0', '0', '1', '1' ),
                ( '1', '1', 'Z', 'Z', 'X', 'X', '0', '0' )
        );
SIGNAL s_4by8 : v4l_4by8 := (OTHERS => "11000000");
SIGNAL s_4by8 : v4l_4by8 := (OTHERS => (OTHERS => 'Z'));
SIGNAL s_4by8 : v4l_4by8
            :=(OTHERS => (0 TO 1 => '1', OTHERS =>'0'));
```

Figure 6.14 Initializing a Two Dimensional Array

Figure 6.14 shows several mechanisms for the initialization of the *s_4by8* signal whose type is defined to be *v4l_4by8* in Figure 6.10. The *s_4by8* array is a 4-by-8 array of *v4l*. In the first declaration of Figure 6.14, the initial values are specified in a nesting of parenthesized sets of values. Shown in separate rows, the deepest level of nestings corresponds to the 0 TO 7 range of *s_4by8*. Since the left most range of the array is 3 DOWNTO 0, four such rows are needed to initialize all the elements in the array. The second statement in this figure uses named association for specifying initial values of this array. The keyword OTHERS covers all the left-most range of *s_4by8*, and each such index is initialized with an 8-bit vector with a value of "11000000".

The third declaration shown in Figure 6.14 initializes all 32 bits of *s_4by8* to 'Z'. The first OTHERS (reading from the left) covers the left-most range of *s_4by8* and the second OTHERS covers the right most range of this signal (0 to 7). The last declaration in Figure 6.14 has the same effect as the second one. Also, since the element type of all arrays declared here is *v4l*, with a default initial value of 'X' (that is because 'X' was used as the left-most enumeration element for this type), initializing *s_4by8* to all Xs by specifying

```
… := (OTHERS => (OTHERS => 'X'))
```

has the same effect as not specifying any initial values at all.

Although we have been emphasizing the value sets formed for initialization of arrays, the same mechanisms can be used on the right hand side of signal assignments. For example, in an architecture body, the statement,

```
s_4by8 <= (3 => (OTHERS => 'X'),
           0 => (OTHERS => 'X'),
           OTHERS => (0=> 'X', 7=> 'X', OTHERS =>'1');
```

writes an 'X'-box with '1'-fill into *s_4by8*. The reader is encouraged to verify this pattern.

6.1.5.2   Non Integer Indexing. VHDL allows the use of any type indication for index definition of arrays. In Figure 6.11, the 7 DOWNTO 0 range was used for the definition of the array in this figure. Instead of using a range, a type indication can be used for the discrete range of an array. If an enumeration type is used for the discrete range specification of an array, the array must be indexed using the enumeration elements of this type. As an example, consider the following declaration of the *v4l_2d* array.

```
TYPE v4l_2d IS ARRAY (v4l, v4l) OF v4l;
```

This is a two-dimensional array that has *v4l* type elements. The *v4l* type is also used as the two discrete ranges of this array. Therefore, the enumeration elements of *v4l* must be used to access elements in the *v4l_2d* array. The two-input NAND gate description in Figure 6.15 uses this array to describe a NAND gate in the *v4l* logic value system. For this and other examples in this chapter, we assume that the *v4l_2d* type is included in the *BasicUtilities* package and can become visible by application of the use clause as shown in Figure 6.15.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
USE utilities.BasicUtilities.ALL;

ARCHITECTURE tabular OF vlog_nand2 IS
   CONSTANT v4l_nand2_table : v4l_2d := (
       -- X   0   1   Z
        ('X','1','X','X'),   -- X
        ('1','1','1','1'),   -- 0
        ('X','1','0','X'),   -- 1
        ('X','1','X','X'));  -- Z
BEGIN
   w <= v4l_nand2_table (a, b) AFTER (tplh + tphl)/2;
END ARCHITECTURE tabular;
```

Figure 6.15 Enumeration Type for Discrete Range of a Two-Dimensional Array

Figure 6.15 also shows the tabular architecture of *vlog_nand2* of Figure 6.5. Types *v4l* and *v4l_2d* are visible to this architecture. Since *v4l* has four enumeration elements, the *v4l_2d* is a 4-by-4 array with its rows and columns indexed as 'X', '0', '1', and 'Z'. In the declarative part of the tabular architecture of *vlog_nand2*, the *v4l_nand2_table* is declared as a constant array of type *v4l_2d*, and it is initialized according to the two-input NAND gate input-output mapping shown in Figure 6.4. The statement part of this architecture consists of a signal assignment whose right-hand side is a look-up into the *v4l_nand2_table*.

Non-integer indexing can take advantage of named association by specifying the exact name of an index. The constant in Figure 6.15 can be initialized to the same values using the following declaration:

```
CONSTANT  v4l_nand2_table : v4l_2d := (
    '0' => (OTHERS => '1'),
    '1' => ('0' => '1', '1' => '0', OTHERS => 'X'),
    OTHERS => ('0' => '1', OTHERS => 'X') );
```

In this declaration, '0', '1', and 'X' values on the left-hand sides of the association symbol, =>, are the names of indexes, while those on the right-hand sides of this symbol are values for the constant array.

6.1.5.3 Unconstrained Arrays. VHDL allows the declaration of unconstrained arrays. This is particularly useful for developing generic descriptions or designs. The bounds of unconstrained arrays used for formal parameters are determined according to the actual parameters that are associated with them. The standard BIT_VECTOR is an unconstrained one-dimensional array of BITs. In the STANDARD package, this type is declared as shown here:

```
TYPE BIT_VECTOR IS
                 ARRAY (NATURAL RANGE <>) OF BIT;
```

This declaration defines BIT_VECTOR as an array with type BIT elements and specifies that it can be indexed by any range of natural numbers. NATURAL, also declared in the STANDARD package, is a type for numbers ranging from 0 to the largest allowable integer. Another unconstrained array in the STANDARD package is the STRING type. This type, shown below, is an unconstrained array of characters; when indexing it, positive numbers should be used.

```
TYPE STRING IS
                 ARRAY (POSITIVE RANGE <>) OF CHARACTER;
```

Similar to the declaration used in the standard package to define BIT_VECTOR, the *std_logic_1164* package defines an unconstrained array of *std_logic* as shown here:

```
TYPE std_logic_vector IS
                 ARRAY (NATURAL RANGE <>) OF std_logic;
```

Similarly, we define an unconstrained array for our Verilog compatible type, *v4l*.

```
TYPE v4l_vector IS
                 ARRAY (NATURAL RANGE <>) OF v4l;
```

Figure 6.16 shows syntax details for this type declaration. The index definition of this array (which is read as "natural range box") indicates that for a range specification of objects of this type or for other type declarations that are based on this type, any descending or ascending range of natural numbers can be used.
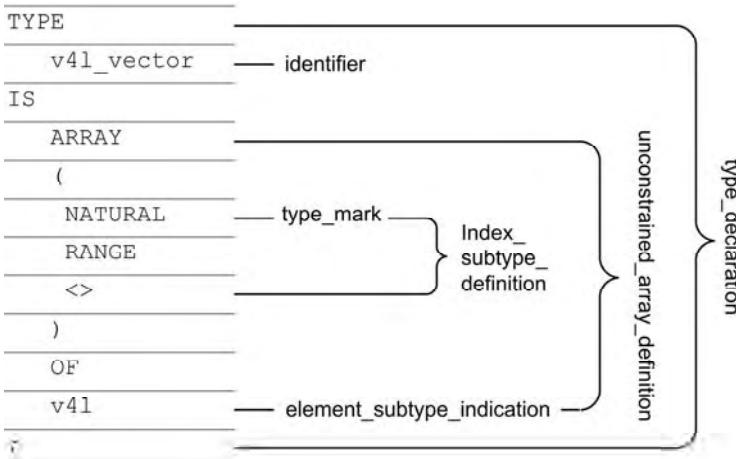
Figure 6.16 Syntax Details of an Unconstrained Array Declaration

To illustrate the use of unconstrained arrays, consider the *one-hot_data* procedure shown in Figure 6.17. The *target* signal in the formal parameters of this procedure is declared as an unconstrained *v4l* vector. This procedure places one-hot values that are distanced in time by *ti* on its *target* output. The number of data put on *target* is passed to this procedure by constant *n*.

```
PROCEDURE onehot_data
   (SIGNAL target : OUT v4l_vector;
    CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
   VARIABLE data : v4l_vector (target'RANGE);
   VARIABLE i : INTEGER := 0;
BEGIN
   data (0) := '1';
   WHILE i < n LOOP
      data := data(data'RIGHT) & data(data'LEFT DOWNTO 1);
         target <= TRANSPORT data AFTER ti * i;
         i := i + 1;
   END LOOP;
END PROCEDURE onehot_data;
```

Figure 6.17 A Generic Version of the *onehot_data* Procedure

The *target* parameter uses *v4l_vector* in Figure 6.16. To specify the range of the intermediate variable (*data*); we have used *target'RANGE*. The *data* variable stores the binary result of this procedure before assigning it to *target*. When an actual signal is associated with the *target* formal parameter, *target* becomes an array signal whose range is the same as that of the actual parameter. This range

is then used for declaring *data*, making it a variable whose range is the same as the range of the actual parameter associated with *target*.

Another use of unconstrained arrays is in the design of generic hardware structures. Figure 6.18 shows a memory structure with an unconstrained address space (*address*) and an unconstrained word size (*datain* or *dataout*). Memory data and address ports are of *v4l_vector* type that is an unconstrained array of *v4l*. In the *behavioral* architecture of *vlog_ram* the *mem* two-dimensional unconstrained array type is declared. The elements of this array are all of *v4l* type.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY vlog_ram IS
   PORT (address : IN v4l_vector;
         datain : IN v4l_vector; dataout : OUT v4l_vector;
         cs, rwbar : IN v4l; opr : IN BOOLEAN);
END ENTITY vlog_ram;
--
ARCHITECTURE behavioral OF vlog_ram IS
   TYPE mem IS ARRAY
               (NATURAL RANGE <>, NATURAL RANGE <>) of v4l;
BEGIN
   PROCESS
      CONSTANT memsize : INTEGER := 2**address'LENGTH;
      VARIABLE memory : mem (0 TO memsize-1, datain'RANGE);
   BEGIN
      id: IF opr'EVENT THEN
         IF opr=TRUE THEN init_mem (memory, "memdata.dat");
         ELSE dump_mem (memory, "memdump.dat"); END IF;
      END IF;
      wr: IF cs = '1' THEN
         IF rwbar = '0' THEN -- Writing
            FOR i IN dataout'RANGE LOOP
               memory (int(address), i) := datain (i);
            END LOOP;
         ELSE                -- Reading
            FOR i IN datain'RANGE LOOP
               dataout (i) <= memory (int(address), i);
            END LOOP;
         END IF;
      END IF;
      WAIT ON cs, rwbar, address, datain, opr;
   END PROCESS;
END ARCHITECTURE behavioral;
```

Figure 6.18 A Generic Memory Model

We are declaring a memory type that is an array of memory words. This memory must be flexible for any word size and address space. Because VHDL does not allow definition of an unconstrained array of unconstrained elements, we are defining our memory as a two-dimensional unconstrained array. The first range of this array is treated as the memory size, and the second range as the word size.

A process statement in the *behavioral* architecture of *vlog_ram* declares the *memory* variable having the two-dimensional *mem* type. This process handles writing and reading to and from this memory. The *memsize* constant is size of the memory and is calculated using the length of the input *address*.

The body of the process statement has memory initialization, dump, write, and read operations. The first if-statement (labeled *id*) calls *init_mem* when *opr* becomes true and calls *dump_mem* when *opr* becomes false. These procedures perform file I/O and will be explained in a later section in this chapter.

The if-statement in the body of the process statement of Figure 6.18 that is labeled *wr* handles write and read operations of the memory. For writing, *datain* is read bit-by-bit and the values are placed in second index of *memory*. The first index of this array is indexed by *address*. Reading from this memory is done in a similar fashion.

To keep this memory model general, all memory operations use. 'LENGTH and 'RANGE array attributes for looking up memory array length and range. For this memory model to function properly, *dataout'RANGE* and *datain'RANGE* used for writing and reading must be the same. The model can be made more robust by including an assert statement that reports any size difference or mismatch of memory ports.

```
FUNCTION int (invec : v4l_vector) RETURN INTEGER IS
   VARIABLE tmp : INTEGER := 0;
BEGIN
   FOR i IN invec'LENGTH - 1 DOWNTO 0 LOOP
      IF invec (i) = '1' THEN
         tmp := tmp + 2**i;
      ELSIF invec (i) = '0' THEN
         tmp := tmp;
      ELSE
         tmp := 0;
      END IF;
   END LOOP;
   RETURN tmp;
END FUNCTION int;
```

Figure 6.19 Unconstrained Function *int*

Because the type of *memory* that is *mem* is declared as having NATURAL type indexes, memory must be indexed as such. For this purpose we have used the *int* function that converts an unconstrained *v4l_vector* type vector to an integer. This unconstrained function is shown in Figure 6.19. We assume *int* is in the *VerilogLogic* package and is made available to our *vlog_ram* entity by use of the USE clause at the beginning of the code of Figure 6.18.

Partial code for a testbench for *vlog_ram* is shown in Figure 6.20. The architecture shown declares *addr* as a 6-bit vector, and *ramin* and *ramout* as 8-bit vectors. Associating these vectors with the ports of *vlog_ram* makes it memory of 32 8-bit words.

```
ENTITY vlog_ram_tester IS END ENTITY vlog_ram_tester;

ARCHITECTURE timed OF vlog_ram_tester IS
   SIGNAL ramin, ramout : v4l_vector (7 DOWNTO 0);
   SIGNAL addr : v4l_vector (5 DOWNTO 0);
   SIGNAL cs, rwbar : v4l;
   SIGNAL operate : BOOLEAN;
BEGIN
   UU1: ENTITY WORK.vlog_ram (behavioral)
       PORT MAP (addr, ramin, ramout, cs, rwbar, operate);
   operate <= TRUE AFTER 5 NS, FALSE AFTER 400 NS;
   cs <= '0', '1' AFTER 15 NS, '0' AFTER 337 NS;
   rwbar <= '1', '1' AFTER 190 NS;
   addr <= "101100" AFTER 020 NS, "101110" AFTER 040 NS,...
   ramin <= "11110001" AFTER 010 NS, . . .

END ARCHITECTURE timed;
```

Figure 6.20 Testbench Instantiating an Unconstrained Memory

6.1.6 File Type and External File I/O

Specifying files is a two-step process of file type declaration and file declaration. File declarations can be used to define a file type. Data is associated with an identifier that is defined as a file type. This data type is the type of the data contained in files of the specified type. The following statement declares *logic_data* as a file type whose contents are of the predefined CHARACTER type:

```
TYPE logic_data IS FILE OF CHARACTER;
```

This file type can be used in a file declaration to declare files of this type for read, write, or append operations.

Several options exist for declaring a file using a declared file type. Anywhere from specifying just a file type to opening a physical file can be specified with file declarations. Declarations of file types

and files of certain types may appear in declarative parts of concurrent or sequential VHDL bodies. For file declaration examples, consider the following:

```
FILE input_logic_value_file1 :
                           logic_data;
FILE input_logic_value_file2 :
                           logic_data IS "input.dat";
FILE input_logic_value_file3 :
                logic_data OPEN READ_MODE IS "input.dat";
```

The first statement specifies *input_logic_value_file1* as the logical name for a file of *logic_data* type. This file must be opened to be associated with a physical file. An explicit OPEN statement must be used for opening this file. The second statement specifies a logical file, associates it with a physical file on a host system, and opens the file in READ_MODE. The third statement provides an option for opening a file in READ_MODE, WRITE_MODE or APPEND_MODE.

If a file is to be opened in write or append modes, only the following two options exist. The first option requires an explicit open statement in which WRITE_MODE may be specified:

```
FILE output_logic_value_file1 :
                            logic_data;
FILE output_logic_value_file2 :
              logic_data OPEN WRITE_MODE IS "input.dat";
```

Data read or written to *input_logic_value_file* and *output_logic_value_file* files in the above examples are in the extended ASCII (elements of CHARACTER) form. An object of the CHARACTER type must be associated with the operand that is used for reading or writing these files.

6.1.6.1 Opening and Closing Files. A declared file, such as *output_logic_value_file1* that is not implicitly opened when it is declared can be opened with an open statement. The following open statements open files of the examples presented above. The READ_MODE is the default and may be dropped.

```
FILE_OPEN (input_logic_value_file1,
                           "input.dat", READ_MODE);
FILE_OPEN (output_logic_value_file1,
                           "output.dat", WRITE_MODE);
```

An extra parameter of FILE_OPEN_STATUS type may be included as the first parameter of the FILE_OPEN statement. Possible values that are returned via this parameter are:

```
OPEN_OK
STATUS_ERROR
NAME_ERROR
MODE_ERROR
```

The conditions under which these values are returned are self-explanatory.

FILE_OPEN statements are procedure calls that can be called from sequential or concurrent VHDL bodies. A FILE_OPEN statement in a sequential VHDL body is executed every time the program flow reaches it, while a concurrent FILE_OPEN statement is executed only once at the beginning of a simulation run.

The logical name of a file that is being opened must be visible to the open statement opening the file. When passing declared files, parameters of file type must be used. Similarly, the logical name of the file is what is used for read, write, append and close file operations.

A FILE_CLOSE statement closes a declared file. The file being closed must be visible in the region in which the close statement appears. Closing files for which open statements were used in the above examples is done as follows:

```
FILE_CLOSE (input_logic_value_file1);
FILE_CLOSE (output_logic_value_file1);
```

When a READ_MODE file closes, opening it again for reading causes reading to be done from the beginning of the file. When a WRITE_MODE file closes, opening it again for writing to it will start writing from the beginning, causing the previous contents to be overwritten. Opening a file in the APPEND_MODE enables writing to the end of the file.

6.1.6.2 File READ and WRITE Operations. VHDL provides three operations, READ, WRITE, and ENDFILE for the file types. READ takes a filename and an object of the file data type as its argument. It reads the next data from the file and places it in its data argument. The arguments of the WRITE operation are similar to those of the READ operation. This operation writes data into the specified file. The ENDFILE operation takes a filename as its argument and returns TRUE if a subsequent READ cannot be done from the file. READ and WRITE operations are procedure calls, while ENDFILE is a function call.

As an example of a file type declaration and external file I/O, consider the *init_mem* procedure shown in Figure 6.21. The first statement shown in this figure declares *v4lfiletype* as a file type of CHARACTER. This declaration must be made visible to any statement that declares a file of this type.

```
TYPE v4lfiletype IS FILE OF CHARACTER;

PROCEDURE init_mem
   (VARIABLE memory: OUT mem; CONSTANT datafile: STRING)
IS
   FILE v4ldata : v4lfiletype;
   VARIABLE v4lvalue : v4l;
   VARIABLE char : CHARACTER;
BEGIN
   FILE_OPEN (v4ldata, datafile, READ_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         READ (v4ldata, char);
         v4lvalue := chartov4l (char);
         memory (i,j) := chartov4l (char);
      END LOOP;
      READ (v4ldata, char);
      READ (v4ldata, char); -- read cr lf
   END LOOP;
END PROCEDURE init_mem;
```

Figure 6.21 Reading an External File

The *init_mem* procedure takes the memory to initialize and name of the physical file to read the initial (memory values) from. In the declaration part of this procedure *v4ldata* file of *v4lfiletype* type is declared. The FILE_OPEN procedure shown in this procedure opens the physical file whose name is passed to this procedure via *datafile*. Once the file is open, all characters are read from the file, they are converted to the *v4l* type and are placed in appropriate positions in the *memory* array. The `REVERSE_RANGE(2) attribute used for the index range of the inner, for-loop in this procedure reads the range of the second index (2) of *memory*.

Figure 6.22 shows the *dump_mem* procedure that is structured similar to *init_mem*. This procedure writes memory contents to the physical file passed to it via its *datafile* formal parameter. This procedure uses the WRITE procedure to write CHARACTER type data representing *v4l* values to the *v4ldata* logical file.

The two procedures discussed here are used in the memory model of Figure 6.18. In addition to being examples of using READ and WRITE file I/O procedures, these procedures are also good examples of unconstrained arrays. The unconstrained memory model of Figure 6.18 sizes its *memory* array according to address and word length as it is instantiated. This memory is passed to *init_mem* and *dump_mem* procedures that are also written as generic size procedures.

```
PROCEDURE dump_mem
   (VARIABLE memory: IN mem; CONSTANT datafile: STRING)
IS
   FILE v4ldata : v4lfiletype;
   VARIABLE v4lvalue : v4l; VARIABLE char : CHARACTER;
BEGIN
   FILE_OPEN (v4ldata, datafile, WRITE_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         v4lvalue := memory (i, j);
         WRITE (v4ldata, v4ltochar (v4lvalue));
      END LOOP;
      WRITE (v4ldata, cr);
   END LOOP;
END PROCEDURE dump_mem;
```

Figure 6.22 Writing into an External File

6.1.6.3 Passing Files. In the above examples we passed a filename to our read or write procedure and performed our entire reading or writing at once. Since the declarative part of a procedure renews each time the procedure is called, a file declaration in this part causes the file to initialize each time the procedure is called. This means that all read operations always start from the beginning of the file and all write operation always start a new file.

Instead of passing a filename and starting a new file each time a procedure is called, file declarations in the declarative part of a process statement are executed only once at the beginning of the simulation run and continuous read or write operations can be made into the same file. Similarly, a file that is declared and opened in the declaration part of an architecture opens only once from multiple reading or writing. For situations that the use of a procedure is designed for multiple readings or writings into the same file, a file object can be declared outside of the procedure and passed to the procedure for reading or writing. For example if the declaration,

```
FILE v4lDataVec : v4lfiletype IS "signaldata.dat";
```

appears in the declarative part of an architecture, for the procedure that is declared as shown below,

```
PROCEDURE AssignData
      (SIGNAL s : out v4l_vector; FILE f: v4lfiletype);
```

The following procedure call enables reading from *signaldata.dat* to be done from where a previous call left off.

```
AssignData (s => targetsignal; f => v4lDataVec);
```

File I/O discussion in this section concentrated on the use of basic file handling primitives of VHDL. While this is sufficient for simple read and write operations, more complex file handling can become very difficult using the basic primitives. The VHDL standard TEXTIO package provides several file types and their corresponding read and write procedures that can be used for more complex file I/O applications.

6.2 VHDL Operators

Types and operators are related issues. The previous section presented type declarations; this section presents operators that operate on operands of given types. The next section shows ways in which operators can be defined for operand types that they are not defined in the standard language.

As in any computer language, operators play a key role in functions performed in VHDL. Operators in VHDL are similar to what can be found in most software language, with the extra emphasis on operators performing operations on logical operands. This section assumes a general knowledge of programming languages; therefore many language details of operators will not be presented.

6.2.1 Logical Operators

Logical operators consist of AND, OR, NAND, NOR, XOR, XNOR, and NOT. The NOT operator is a unary operator, and all others use two operands (binary operators). Logical operators perform on predefined types BIT, BOOLEAN and BIT_VECTOR. When vectors are used, the number of bits of the two operands must be the same. The following statements show the valid use of operators, if operands in each statement are of the same type and size:

```
x <= a XNOR b;
x_vector <= a_vector AND b_vector;
```

Strings representing operator symbols can be used as function names for performing the same function as the operator they are representing. For example, the above statements can be written as:

```
x <= "XOR" (a, b);
x_vector <= "AND" (a_vector, b_vector);
```

In this case, the order of the operands must stay the same in either form of usage.

6.2.2 Relational Operators

Relational operators operate on operands of the same type and return a BOOLEAN TRUE or FALSE value. Operators in this group are =, /=, <, <=, >, and >= with equal, not equal, less than, less than or equal, greater than, and greater than or equal functionalities. The = and /= operators operate on operands of any type. The other relational operators perform their normal functions when used with scalar operands. When array operands are used with these operators (<, <=, >, and >=), they perform ordering operations and return TRUE or FALSE based on values of array elements starting from the left. The following paragraphs present relational and ordering examples.

For two integers of the same value, = returns TRUE. For integers i1>i2, the >, >=, and <= operations return TRUE, and the < operation returns FALSE. In all operations the resulting target must be of BOOLEAN type as shown below:

```
a_boolean <= i1 > i2;
b_boolean <= i1 /= i2;
```

Two one-dimensional arrays can be compared using relational operators. For this comparison, the relation between array elements in considered. For array elements of enumeration type, the left-most enumeration element has the smallest value and the right-most element has the largest value. For the *v4l* type defined earlier in this chapter, 'X' is the smallest, followed by '0', and '1', and 'Z' is considered the largest of these enumeration elements. For the predefined type BIT, '1' is greater than '0' since BIT is defined as:

```
TYPE BIT IS ('0', '1');
```

Assuming values of "00011" and "00100" for *a_bit_vector* and *b_bit_vector* arrays, the relation,

```
a_bit_vector < b_bit_vector
```

returns TRUE. Evaluation begins with the left-most bits. As soon as the third bit of *a_bit_vector* is found to have a smaller value than that of *b_bit_vector* ('0' is less than '1'), the operator < completes its operation and returns TRUE.

For arrays of differing sizes, the same evaluation method holds. If *a_v4l_vector* is "011Z" and *b_v4l_vector* is "0X", the relation,

```
a_v4l_vector < b_v4l_vector
```

returns FALSE.

6.2.3 Shift Operators

VHDL has shift operators for multiple shiftings with various shift functions. Shift operation can be *logical* or *arithmetic*, *left* or *right*, and *shift* or *rotate*. Figure 6.23 shows a complete list of shift operators and their operations.

	Shift/Rotate	*Left/Right*	*Logical/Arithmetic*
SLL	Shift	Left	Logical
SLA	Shift	Left	Arithmetic
SRL	Shift	Right	Logical
SRA	Shift	Right	Arithmetic
ROL	Rotate	Left	Logical
ROR	Rotate	Right	Logical

Figure 6.23 Shift Operators

These operators use two operands. The left operand is the array that is being shifted, and the right operand is a positive or negative integer specifying the number of shifts or rotate positions. Negative shift positions change the direction of the operation. For example, SLL with a negative number of shifts is equivalent to SRL with positive value of the same number for its shift positions. VHDL defines the shift operators for arrays with BIT or BOOLEAN element types. The next section shows how these and other operators can be defined for other types such as our own *v4l* and the IEEE standard *std_logic*. The result of a shift operation is an array of the same size of the left operand.

A logical shift right (left) operation shifts an array to the right (left), drops the right-most (left-most) element, and fills the left (right) side of the array with a fill value. The fill value is the left most enumeration element of the array element. For an array of BIT, the fill value is '0', and for *v4l* is 'X'.

An arithmetic shift right (left) operation shifts its array operand to the right (left) and uses the left-most (right-most) element for the left (right) fill.

Figure 6.24 shows application of various shift operations on an array of *v4l* (*v4l_vector*). The value of *av* that is the first operand of shift operations in this figure is "Z01XZ101"

Start with av=	Z	0	1	X	Z	1	0	1

av SLL 1	0	1	X	Z	1	0	1	X
av SLA 1	0	1	X	Z	1	0	1	1
av SRL 1	X	Z	0	1	X	Z	1	0
av SRA 1	Z	Z	0	1	X	Z	1	0
av ROL 1	0	1	X	Z	1	0	1	Z
av ROR 1	1	Z	0	1	X	Z	1	0

Figure 6.24 Application of Shift Operators

6.2.4 Adding Operators

Addition, subtraction, and concatenation form the adding group of operators. Add and subtract are defined for numeric types of INTEGER and REAL. Both operands of an adding operator must have the same type. Add and subtract are not defined for BIT or BIT_VECTOR types, but VHDL packages for defining such operations are available.

As with other operators, an adding operator can be used in the following two formats:

```
a + b
"+" (a, b)
```

Operands of a concatenation operator must be arrays or elements of the same type. Concatenating two scalars of the same type forms an array of size 2.

6.2.5 Sign Operators

Sign operators + and – are unary operators that apply to numeric types.

6.2.6 Multiplying Operators

The four multiplying operators are *, /, MOD, and REM. Multiplication and division have their conventional mathematical meanings and are defined for operands of the same type of INTEGER or REAL.

Both operands of MOD and REM operators must be of the INTEGER type. The remainder, REM, operator returns the remainder of integer division of the absolute value of its left operand by the absolute value of its right operand. The sign of the result is the same as

that of the left operand. The modulus, MOD, operator calculates the modulus of its left and right operands. The sign of the result is the same as that of the right operand.

6.2.7 Other Operators

The last group of operators fits in none of the above categories. At the present time, ** (exponential) and ABS (absolute value) are in this group. We leave this section open for any future operators that may be added to the language.

6.2.8 Aggregate Operation

An aggregate operation combines one or more values into a complex array or record type. We have used aggregates in this chapter for array initializations. Assuming *a* and *b* are objects of BIT type, the following expressions are equivalent:

```
(a, b)
 a & b
```

The first expression uses an aggregate operation to form a 2-bit vector, and the second expression concatenates *a* and *b* together. Aggregate operation can only be applied to elements of the same size and type. Concatenation, on the other hand, can be used to concatenate different-size arrays of the same element type.

An aggregate operation applies to records as well as arrays. The next section shows examples of record aggregates. An aggregate can be done on the left-hand side of a signal assignment. For example, the following are valid signal assignments:

```
(a, b) <= a2;
(a, b) <= "10";
(a, b) <= ('1', '0');
```

The third statement uses aggregate on the right- and left-hand sides of the signal assignment. The aggregate operation applies to elements of user defined vector types as well as VHDL predefined types.

6.3 Operator and Subprogram Overloading

In VHDL, subprograms with the same name and different types of parameters or results are distinguished from each other. A name used by more than one such subprogram is said to be overloaded. Overloading is a useful mechanism for using the same name for sub-

programs that perform the same operation on data of different types. VHDL allows overloading of user-defined subprograms, standard functions, and operators.

6.3.1 Operator Overloading

Our first examples for overloading show how to define the basic logical operators for the *v4l* type defined in the previous section. Figure 6.25 shows logic tables for AND, OR and inversion operations in the *v4l* logic value system. These tables are according to the Verilog 4-value logic operations.

   If one input of the "AND" function is '0', the output is '0' even if the other input is unknown ('X'). Similarly, if at least one input of the "OR" function is '1', the output becomes '1'. The "NOT" table is a repetition of the logic value table for the inverter in Section 6.1, and is shown here for completeness. Figure 6.26 shows the definition of the "AND" function according to Figure 6.25. Other logical functions for the *v4l* type are similarly defined. Such definitions overload the corresponding VHDL operators that are defined for the BIT and the BOOLEAN types.

Figure 6.25 Verilog 4-Value Logic Operations Used for *v4l*

   The description of Figure 6.26 uses the *v4l_2d* type definition of Section 6.1.5. This type definition and various overloading functions can become visible to designs using them by the use of packages. For example, we can make overloading of various logic functions available to our designs by including them in our *VerilogLogic* package.

   When a design uses AND, OR, or NOT operators with BIT or BOOLEAN operands, the standard VHDL operators are of the *v4l* type, overloaded functions similar to the "AND" of Figure 6.26 are used if they are visible to the design.

   As an example of using overloaded logical operators consider the multiplexer example of Figure 6.27. As shown, ports of the *multiplexer* entity are declared as having type *v4l*. This type and its related

overloaded operators are made visible to our design by the use of the *VerilogLogic* package. This is compiled in the utilities library.

```
FUNCTION "AND" (a, b : v4l) RETURN v4l IS
   CONSTANT v4l_and_table : v4l_2d := (
     'X' =>   ('X','0','X','X'),
     '0' =>   ('0','0','0','0'),
     '1' =>   ('X','0','1','X'),
     'Z' =>   ('X','0','X','X'));
BEGIN
   RETURN v4l_and_table (a, b);
END "AND";
```

Figure 6.26 Overloading AND Logical Function for the v4l Four Value Logic System

The *booleanlevel* architecture in Figure 6.27 uses AND, OR, and NOT operations. Note that the operands of these operators on the right hand side of *w* are all of type *v4l*. Because of this, the overloaded AND, NOT, and OR of *VerilogLogic* are used instead of the standard operators of VHDL. Recall that the standard operators of VHDL for the BIT type are in the *STD* package of the *STANDARD* library.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY multiplexer IS
   PORT (a, b, s : IN v4l; w : OUT v4l);
END ENTITY;
--
ARCHITECTURE booloeanlevel OF multiplexer IS
BEGIN
   w <= (a AND NOT s) OR (b AND s);
END ARCHITECTURE booloeanlevel;
```

Figure 6.27 Using Overloaded Operators

For the next example of overloading, consider the expressions used for calculating *tplh* and *tphl* delay parameters in *cmos_not* in Figure 6.9. Physical type-to-integer and integer-to-physical type conversions were done in these expressions because the VHDL multiplication operator is not defined for multiplying two physical types. By overloading this operator as shown in Figure 6.28, it can be made to accept *resistance* as the type of its first operand (left of the operator) and *capacitance* as the type of its second operand (right of the operator) and to produce results of type TIME.

The "*" overloading function uses the definition of *resistance* and *capacitance* physical types. In the statement part of this function, *resistance* and *capacitance* physical types are neutralized to equivalent

integers, and the result of multiplying these integers is multiplied by the appropriate unit of type TIME. This time expression is returned as the result of the function.

```
FUNCTION "*" (a : resistance; b : capacitance)
   RETURN TIME IS
BEGIN
   RETURN  ( ( a / 1 l_o) * ( b / 1 ffr ) * 1 FS ) / 1000;
END "*";
```

Figure 6.28 Overloading: Multiplying *Resistance* and *Capacitance* Resulting TIME

The *rc_timed* architecture of *cmos_not* shown in Figure 6.9 uses the overloaded multiplication operator for calculating *tplh* and *tphl* propagation delays. The first multiplication operator in the constant value expressions of *tplh* or *tphl* is associated with the function in Figure 6.28. The other multiplication operator uses the standard VHDL operator. This example assumes that type *v4l*, physical *resistance* and *capacitance* and the overloaded multiplication operator available in the *VerilogLogic* and *BasicUtilities* packages that are used in this design.

6.3.2 Subprogram Overloading

The final discussion in this section focuses on overloading system and user-defined subprograms. Examples of system subprograms are READ and WRITE procedures used in developing *dump_mem* (Figure 6.21) and *init_mem* (Figure 6.22) procedures for our memory model of Figure 6.18. These tasks can be overloaded for reading and writing any data type from the corresponding data files. Overloaded versions of these procedures are provided in various text I/O packages such as that of the VHDL standard and std-logic libraries. Chapter 8 discusses VHDL libraries and the use of such procedures.

As an example of overloading user subprograms, consider the *dump_mem* procedure in Figure 6.29. This procedure is the BIT version of *dump_mem* in Figure 6.22, and is used for writing contents of a memory to an external file. As shown in Figure 6.29 the type of *memory* passed to this version of *dump_mem* is two-dimensional array of bits. This formal parameter of the procedure distinguishes it from *dump_mem* that was developed for memories of *v4l* data type that was shown in Figure 6.22.

Using *init_mem* and *dump_mem*, a memory model for BIT type data, similar to that of Figure 6.18 is developed. This model uses the *int* function that is an overloaded version of this function shown in Figure 6.19. The *BasicUtilities* package that is included in the CD

that accompanies this book includes the complete code of memory and other utilities.

```
TYPE mem IS ARRAY (NATURAL RANGE <>,
                   NATURAL RANGE <>) of BIT;
TYPE bit_filetype IS FILE OF CHARACTER;

PROCEDURE dump_mem (VARIABLE memory : IN mem;
                    CONSTANT datafile : STRING) IS
   FILE BIT_data : BIT_filetype;
   VARIABLE BIT_value : BIT;
   TYPE BIT_char IS ARRAY (BIT) OF CHARACTER;
   CONSTANT BIT_tochar : BIT_char := ('0', '1');
BEGIN
   FILE_OPEN (BIT_data, datafile, WRITE_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         BIT_value := memory (i, j);
         WRITE (BIT_data, BIT_tochar (BIT_value));
      END LOOP;
      WRITE (BIT_data, cr);
   END LOOP;
END PROCEDURE dump_mem;
```

Figure 6.29 Overloaded Memory Dump Procedure

6.4 Other Types and Type-Related Issues

Subtypes, records, and aliases are type-related issues that can be used for hardware modeling and design. This section is devoted to the description of these topics.

6.4.1 Subtypes

For a previously defined type, subtypes consisting of the subsets of the values of the original type can be defined. The original type is called the base type, and it is fully compatible with all its subtypes. In VHDL, all types are subtypes of themselves. Because of this, the word *subtype* is used to refer to all declared types and subtypes.

Subtypes are used when a subset of a previously defined type is to be utilized and when compatibility with the base type is to be preserved. For example, consider defining *bit_compatible_byte* as:

```
SUBTYPE bit_compatible_byte IS BIT_VECTOR (7 DOWNTO 0);
```

This   declaration   makes   objects   that   are   declared   as *bit_compatible_byte* compatible with other objects whose base types

are BIT_VECTOR. As a counter example, consider the declaration of *byte_of_bits* as:

```
TYPE byte_of_bits IS ARRAY (7 DOWNTO 0) OF BIT;
```

If an object of type *byte_of_bits* is to be assigned to an 8-bit BIT_VECTOR object, or if such objects are to be used in an expression, the use of explicit type conversions from *byte_of_bits* to BIT_VECTOR or vice versa is required.

A subtype can be declared to have a range of enumeration elements of an enumeration type. For example, the following subtype declaration defines *bcd_numbers* as a subtype whose elements are integers between 0 and 9:

```
SUBTYPE bcd_numbers IS INTEGER RANGE 0 TO 9;
```

Objects of type *bcd_numbers* can be used in the same expressions with INTEGER type objects without requiring any form of a type conversion.

The definition of a general multi-level logic value system, on which the definition of other logic value systems can be based, is an important application of this concept. For example, based on our *v4l* type, subtypes such as *v3l* and *v2l* can be defined as shown here:

```
SUBTYPE v3l IS v4l RANGE '0' TO 'Z';
SUBTYPE v2l IS v4l RANGE '0' TO '1';
```

According to these declarations, the *v3l* type is a three-value logic system that contains enumeration elements of '0', '1', and 'Z', and the *v2l* type is a two-value logic which contains '0' and '1'. The base type for both of these subtypes is *v4l*.

Assigning an object of a smaller subtype, e.g., *v2l*, to an object of a larger subtype, e.g., *v4l*, can be done directly, and there is no need for any type conversion. The opposite is also possible, except that if an out-of-range value is assigned to the object of the smaller subtype, a simulation warning message will be issued.
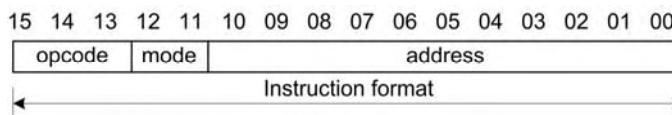
In our example, the *v2l* subtype contains all the enumeration elements of the predefined BIT type. These two types, however, are not compatible, so type conversion by means of a user function is required for assignments or operations that involve these two types.

6.4.2 Record Types

Arrays are composite types whose elements are all of the same type. Records are also of the composite class, but they can consist of elements of different types. A record type definition consists of the decla-

ration of the elements of the record that is bracketed between the RECORD keyword and END RECORD keywords. Each record element declaration declares one or more identifiers and their types.

For example, consider an instruction format for a simple computer that has eight operations, four addressing modes, and an address space of 2^{11} words. Figure 6.30a shows the instruction format and the type declarations for these three fields. The opcode is an enumeration type whose elements are the instruction mnemonics, the addressing mode is an integer ranging from 0 to 3, and the address is an 11-bit BIT_VECTOR.

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
opcode			mode		address										

Instruction format

```
TYPE opcode IS (sta, lda, add, sub, and, nop, jmp, jsr);
TYPE mode IS RANGE 0 TO 3;
TYPE address IS BIT_VECTOR (10 DOWNTO 0);
```
(a)

```
TYPE instruction_format IS RECORD
   opc : opcode;
   mde : mode;
   adr : address;
END RECORD;
```
(b)

```
SIGNAL instr : instruction_format
             := (nop, 0, "00000000000");
```
(c)

```
instr.opc <= lda;
instr.mde <= 2;
instr.adr <= "00011110000";
```
(d)

```
instr <= (adr => (OTHERS => '1'), mde => 2, opc => sub)
```
(e)

Figure 6.30 Record Type, (a) Three Instruction Fields, (b) Declaration of Instruction Format, (c) A Signal of Record Type, (d) Referencing Fields of a Record Type Signal, (e) Record Aggregate.

In Figure 6.30b, the *instruction_format* type is declared as a record that contains three fields of *opc*, *mde*, and *adr* of types *opcode*, *mode*, and *address*, respectively. A signal of type *instruction_format*, shown in Figure 6.30c, is declared, and the fields of this signal (*instr*)

are initialized to *nop*, 0, and "00000000000". Figure 6.30d shows three signal assignments assigning values to the *instr* signal field. Finally, Figure 6.30e shows assignments to *instr* using a record aggregate with named association. In this assignment, the *adr* field receives the "11111111" value, *mde* becomes 2, and *opc* receives *sub*. Association by position can also be done for this assignment.
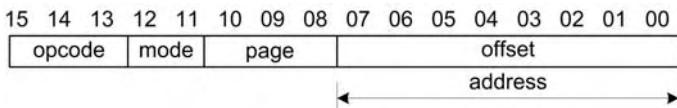
6.4.3 Alias Declaration

An object, an indexed part of it, or a slice of it can be given alternative names by using an alias declaration. This declaration can be used for signals, variables, or constants, and it can define new identifiers of the same class and type.

As an example, consider a flag register that is declared as a 4-bit BIT_VECTOR with a descending 3 DOWNTO 0 range. Starting with the most-significant bit, the bits of this register are carry, overflow, negative, and zero flags. The declarations shown below specify aliases for each of the bits in the flag register:

```
ALIAS c_flag : BIT IS flag_register(3);
ALIAS v_flag : BIT IS flag_register(2);
ALIAS n_flag : BIT IS flag_register(1);
ALIAS z_flag : BIT IS flag_register(0);
```

With these declarations, the equivalent identifiers can be used instead of indexing the *flag_register*. An alias declaration specifies an identifier, its type, and the name of an object the identifier becomes an alias of.

(a)

```
ALIAS page :
        BIT_VECTOR (2 DOWNTO 0) IS instr.adr (10 DOWNTO 8);
ALIAS offset :
        BIT_VECTOR (7 DOWNTO 0) IS instr.adr (7 DOWNTO 0);
```

(b)

```
page <= "001";
offset <= X"F1";
```

(c)

Figure 6.31 Alias Declaration, (a) Page and Offset Addresses, (b) Alias Declaration for the Page and Offset Parts of the Address, (c) Assignments to Page and Offset Parts of Address

For an example of using an alternative name for a slice of an array, consider the address field of the *instruction_format* in Figure 6.30. This 11-bit address can consist of a 3-bit page address and an 8-bit offset address, as shown in Figure 6.31a. The alias declarations in Figure 6.31b equate *page* to the 3 most-significant bits of the address field of an instruction and *offset* to its 8 least-significant bits. Figure 6.31c shows signal assignment to *page* and *offset* aliases. These assignments result in assigning an 11-bit address to the *adr* field of *instr*.
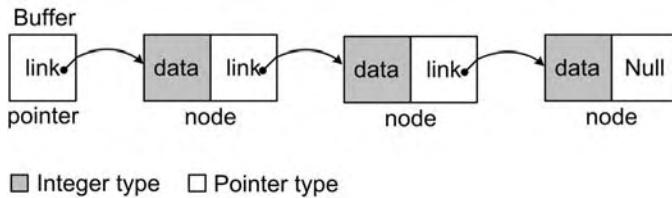
6.4.4 Access Types

Some of the material presented in this chapter thus far may appear to be irrelevant to a hardware design environment and may seem to be at a much higher level than a hardware designer ever needs. On the other hand, in the earlier chapters we mentioned that VHDL is a language not only for the description of hardware, but for modeling a complete hardware design, test and verification environment. Certain hardware/software applications can also use VHDL for evaluation of hardware algorithms that run along side with software applications.

A high level utility in the VHDL language is the ACCESS type for definition of pointers and generation of linked lists. Examples of applications of linked lists are in large-memory modeling, test bench generation, dynamic modeling of FIFOs and stacks, and dynamic fault lists. In this section, we show VHDL declarations necessary for defining a linked list and example procedures for manipulating the list.

6.4.4.1 Linked-list Definition. An example linked list with a *node* containing an integer *data* and a *link* for pointing to the next node is shown in Figure 6.32a. This list is identified by *Buffer* which is the pointer that points to the first *node*. For an empty list *Buffer* is NULL. Each node has a *data* part and a *link* part. An integer type is assumed for the data in this example. The first pointer that identifies the list is considered the head of the list.

Figure 6.32 shows the graphical representation of a linked list and its corresponding VHDL declarations. The first statement is an *incomplete type declaration* that defines *node* as a type to which a pointer will be assigned. The next statement is an ACCESS type declaration, defining *pointer* for accessing elements of type *node*. The last statement is a record type declaration that specifies the type of *node* as a record containing *data* and a *link* that has the type of *pointer*. This same type is also used for the head of the list. The *pointer* type used within this node is the required type for a *node* to point to an element of this same type.

```
TYPE node;
TYPE pointer IS ACCESS node;
TYPE node IS RECORD
   data : INTEGER;
   link : pointer;
END RECORD;
```

Figure 6.32 Linked List Graphical Representation and Definition in VHDL

6.4.4.2   Using a Linked List. Example procedures for entering data, removing data, and clearing a linked list are presented here. A linked list based on declarations of Figure 6.32 is identified by a variable of type *pointer*. Such a variable must have an initial value of NULL. The following shows declaration of *Buff* as the head of a linked list to be created.

```
VARIABLE Buff : pointer := NULL;
```

The following assignment creates the first new node and assigns it to *Buff*.

```
Buff := NEW node;
```

The next new *node* is linked to *Buff* by the use of the following assignment:

```
Buff.link := NEW node;
```

Figure 6.33 shows a procedure that takes an existing linked list as input and appends an integer to the end of it. We assume the pointer of the linked list is of *pointer* type above, and its nodes are of the *node* type declared above. As specified in the declarations, data types in this linked list are of the INTEGER type.

```
PROCEDURE insert
   (VARIABLE head : INOUT pointer; din : INTEGER)
IS
   VARIABLE t1 : pointer;
BEGIN
   -- Insert a node with value din
   IF head=NULL THEN
      head := NEW node;
      head.data := din;
      head.link := NULL;
      REPORT "The List was originally empty!";
   ELSE
      t1 := head;
      WHILE t1.link /= NULL LOOP
         t1 := t1.link;
      END LOOP;
      t1.link := NEW node;
      t1 := t1.link;
      t1.data := din;
      t1.link := NULL;
   END IF;
REPORT "Value:"&INTEGER'IMAGE(din)&" inserted!";
END insert;
```

Figure 6.33 Creating a linked list and entering data

The head of the linked list has the type of pointer and is passed to this procedure. If the linked list is empty, the head is Null and if it already contains data it is a pointer to a node. The while-loop shown in Figure 6.33 starts from the head of the linked list and finds the first null pointer. When done, *t1* is the first Null pointer. Following this, a NEW node is created and assigned to *t1*. The *data* field of *t1* gets the new integer (*din*) and its pointer gets Null. Assignment of *din* and Null to *t1* can be done by aggregating them as shown below:

```
t1 := (din, Null);
```

Figure 6.34 shows another procedure for the linked list that we are discussing. This procedure removes the link with value *v* from the linked list. The head of the list is passed to the *remove* procedure via its *head* argument. In the body of procedure, the node with value *v* is searched for, and when found, the pointer to it is set to the node that it points to. When this rearrangement of pointers takes place, the node with value *v* is freed by use of DEALLOCATE.

```
PROCEDURE remove
   (VARIABLE head : INOUT pointer; v : IN INTEGER)
IS
   VARIABLE t1, t2 : pointer;
BEGIN

   t1 := head;
   t2 := head;
   IF head /= NULL THEN
      IF head.data = v THEN
         head := head.link;
         REPORT "Value:"&INTEGER'IMAGE(v)&
                " was in the head and removed!";
      ELSE
         WHILE t1 /= NULL LOOP
            IF t1.data = v THEN
               t2.link := t1.link;
               REPORT "Value:"&INTEGER'IMAGE(v)&
                      " removed!";
               EXIT;
            ELSE
               t2 := t1;
            END IF;
            t1 := t1.link;
         END LOOP;
      END IF;

      IF (t1 = NULL) THEN
         REPORT "Value: "&INTEGER'IMAGE(v)&
                " doesn't exist in the list!";
      END IF;
   ELSE
      REPORT "The List is empty! Can't remove.";
   END IF;

END remove;
```

Figure 6.34 Removing an Item From a Linked List

Figure 6.35 shows the clear procedure that frees all nodes of the linked list that is passed to it. As with other procedures discussed, the head of the linked list is passed to this procedure via the INOUT head argument. Initially *head* is set to Null and then all nodes are deallocated until a node is found that has a Null pointer.

```
PROCEDURE clear (VARIABLE head : INOUT pointer) IS
   VARIABLE t1, t2 : pointer;
BEGIN
   -- Free all the linked list
   t1 := NEW node;
   t1 := head;
   head := NULL;
   WHILE t1 /= NULL LOOP
      t2 := t1;
      t1 := t1.link;
      DEALLOCATE (t2);
   END LOOP;
   REPORT "The List cleared successfully!";
END clear;
```

Figure 6.35 Freeing a Linked List

With the above linked list utilities, new linked lists of this format can be declared using the variable declaration shown below. For appending a data element to the end of the list the *insert* procedure must be called. The example shown below adds a *node* to *fifo* and sets its value to 25. For removing the node, the *remove* procedure must be called. The example shown below removes node with value 26 from *fifo*. For clearing *fifo* the *clear* procedure must be called.

```
VARIABLE fifo, stack : pointer := Null;
. . .
insert (fifo, 25);
remove (fifo, 26);
clear (fifo);
```

6.4.5 Global Objects

A signal declared in a package can be written to or read by all VHDL bodies that the package is visible to. Concurrent writing to a shared signal will be possible only if the signal is resolved, as will be discussed in the next chapter. A function for resolving multiple driving values is defined for resolved signals.

Because of concurrency in the language, conflicts and indeterminacy may be caused with shared or global variables. In spite of this, VHDL allows shared variables to be declared in packages and architectures. A shared variable declared in a package is accessible to all bodies that use the package. The scope of shared variables declared in an architecture is only within the body of the architecture. An example shared variable declaration is:

```
SHARED VARIABLE dangerous : INTEGER := 0;
```

Shared variables are not protected against multiple simultaneous read and write operations. However, signal semaphores for creating such a protection can be done in VHDL.

6.4.6 Type Conversions

For cases that types of operands or expressions are not known from the context in which they are used in, or in cases when a nondefault conversion is to be enforced, type conversions may be used. VHDL offers two mechanisms for this purpose.

6.4.6.1 Qualifiers. The first mechanism for type adaptation in VHDL is using qualifiers and the other is using explicit conversions.

An expression or an aggregate may be qualified to a specific type using a type qualifier. Consider the last signal assignment (labeled *SA8*) in Figure 6.12 (repeated below for reference). The right-hand side of this statement clearly specifies the type and values of the right-hand side. However, in the statement shown below (labeled *SA9*) with the same left-hand side as *SA8*, the type of the right hand side cannot uniquely be determined. The right-hand-side aggregate may be 'X' values of *v4l* type, or may be 'X' values of the predefined CHARACTER type.

```
SA8: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <=
                              s_byte (5 DOWNTO 2);

SA9: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <=
                                  (OTHER => 'X');
```

Because of this ambiguity, a qualifier such as that shown in the following is necessary on the right hand side of the signal assignment. In this statement 'X's are qualified for size and element type.

```
SA9: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <=
                    v4l_byte'(OTHER => 'X');
```

6.4.6.2 Explicit Type Conversions. The other mechanism for type adaptation is using explicit conversions.

Explicit type conversions perform conversions between closely related types. INTEGER and REAL types are closely related. Array types with same type elements or array types with same elements of the same base type are also considered to be closely related. Consider, for example, the following type and signal declarations:

```
TYPE v4l_byte IS ARRAY (7 DOWNTO 0) of v4l;
TYPE v4l_octal IS ARRAY (7 DOWNTO 0) of v4l;
```

```
   . . .
   SIGNAL sb : v4l_byte;
   SIGNAL so : v4l_octal;
```

Types of *sb* and *so* are different and, therefore, the assignment shown below is not allowed:

```
   sb <= so;
```

On the other hand, an explicit type conversion can be used to convert the type of *so* to that of *sb*. The first statement shown below converts *so* of type *v4l_octal* to its closely related type of *v4l_byte* and assigns it to *sb* of type *v4l_byte*. The second statement converts *sb* of type *v4l_byte* to its closely related type of *v4l_octal* and assigns it to *so* of type *v4l_octal*.

```
   sb <= v4l_byte (so);
   so <= v4l_octal (sb);
```

6.4.7 Standard Nine-Value Logic

Chapter 5 presented basics of the 1164 IEEE standard. With the material presented in this chapter we are now able to present some other details related to the *std_logic* type. The package is compiled in the IEEE library, and it is named *std_logic_1164*. The following statements make all *std_logic* utilities available to a design:

```
   LIBRARY IEEE;
   USE IEEE.std_logic_1164.ALL;
```

The standard defines *std_logic* as an enumeration type with nine logic values. Since 'U' is defined as the first enumeration element, this value is considered as the default initial value. The *std_logic_vector* is an unconstrained array of *std_logic*. Arrays of this type can be declared with any range or size.

All logical and shift operators are overloaded for *std_logic* and *std_logic_vector*. Therefore, no change in operator names needs to be made when going from BIT to the more general *std_logic* logic value system.

The standard logic package includes subtypes for smaller logic value sets. Figure 6.36 shows *std_logic* subtypes and their enumeration elements. Conversion functions for all subtypes and the BIT type to and from *std_logic* are included in the package. For example, the *To_StdLogicVector* function converts its BIT_VECTOR operand to *std_logic_vector*. Appendices of this book include a listing of *std_logic* utilities, and other related packages.

TYPE	
X01	'X', '0', '1'
X01Z	'X', '0', '1', 'Z'
UX01	'U', 'X', '0', '1'
UX01Z	'U', 'X', '0', '1', 'Z'

Figure 6.36 *std_logic* Sub-types

6.5 Predefined Attributes

Predefined attributes in VHDL provide functions for more efficient coding or mechanisms for modeling hardware characteristics. Attributes can be applied to arrays, types, signals, and entities, and they have the format shown below. When reading this, the single quote (') is read as *tick*.

```
array_or_type_or_signal_or_entity_name'ATTRIBUTE_NAME
```

This section discusses array, type, signal and entity attributes. All the predefined attributes are listed and categorically discussed. Examples are shown only for key attributes; other attributes are used in the examples in the chapters that follow.

6.5.1 Array Attributes

Array attributes are used to find the range, length, or boundaries of array objects. These attributes can only be used with array objects, and do not apply to scalars. In order to present examples of these attributes we use the two-dimensional *s_4by8* signal of Figure 6.12. The type and declaration of this signal are repeated here for reference.

```
TYPE v4l_4by8 IS ARRAY (3 DOWNTO 0, 0 TO 7) of v4l;
SIGNAL s_4by8: v4l_4by8;
```

As shown, *s_4by8* is a two-dimensional array with two range specifications. The first range specification (referred to as (1)) is descending with 3 being its left index and 0 its right index. The second range specification (2) is an ascending range. Figure 6.37 shows examples of the predefined array attributes as they are applied to the *s_4by8* signal. A number in parentheses can follow an array attribute. For multi-dimensional arrays, this number indicates the index range of the array.

Attribute	Description	Example	Result
'LEFT	Left bound	s_4by8 'LEFT	3
'RIGHT	Right bound	s_4by8 'RIGHT s_4by8 'RIGHT(2)	0 7
'HIGH	Upper bound	s_4by8 'HIGH(2)	7
'LOW	Lower bound	s_4by8 'LOW(2)	0
'RANGE	Range	s_4by8 'RANGE(2) s_4by8 'RANGE(1)	0 TO 7 3 DOWNTO 0
'REVERSE_RANGE	Reverse range	s_4by8 'REVERSE_RANGE(2) s_4by8 'REVERSE_RANGE(1)	7 DOWNTO 0 0 TO 3
'LENGTH	Length	s_4by8 'LENGTH	4
'ASCENDING	TRUE If Ascending	S_4by8 'ASCENDING(2) s_4by8 'ASCENDING(1)	TRUE FALSE

Figure 6.37 Predefined Array Attributes.

The examples in the previous chapters used 'RANGE and 'LENGTH attributes. More examples, based on *array_x* being an array object with ascending range, are shown here:

The following condition is true:
```
condition array_x'LEFT < array_x'RIGHT is true.
```

The two ranges shown in the following two lines are equivalent:
```
array_x'LEFT TO array_x'RIGHT
array_x'RANGE
```

The two expressions shown in the following two lines are equivalent:
```
array_x'HIGH - array_x'LOW + 1
array_x'LENGTH
```

The two expressions shown in the following two lines are equivalent:
```
array_x (array_x'LOW)
```
Value of element of *array_x* at the location of its lowest index.

6.5.2 Type Attributes

Type attributes are used for accessing elements of defined types and are only valid for scalar types. Although several type and array attributes use the same names, it is important to realize that their meanings may be different. For example, when applied to an enumeration type, the 'RIGHT type attribute results in the right-most enumeration element of that type. Attributes 'BASE, 'LEFT, 'RIGHT, 'HIGH, and 'LOW can be applied to any scalar type, while attributes

'POS, 'VAL, 'SUCC, 'PRED, 'LEFTOF, and 'RIGHTOF can only be used with an integer type, an enumeration type, or a physical type. For example, using 'VAL(2) with an enumeration type results in the enumeration element in position 2 for the enumeration type. Enumeration elements are numbered from left to right starting with 0.

Figure 6.38 shows VHDL type attributes and presents an example for each one. The examples refer to *v4l*, *v3l*, *v2l*, and *opcode* types presented earlier in this chapter and are repeated here for reference.

```
TYPE v4l IS ('X', '0', '1', 'Z');
SUBTYPE v3l IS RANGE '0' TO 'Z';
SUBTYPE v2l IS RANGE '0' TO '1';
TYPE opcode IS (sta, lda, add, sub, and, nop, jmp, jsr);
```

Attribute	Description	Example	Result
'BASE	Base of type	v3l'BASE	v4l
'LEFT	Left bound of type or subtype	v3l'LEFT v4l'LEFT	'0' 'X'
'RIGHT	Right bound of type or subtype	v3l'RIGHT v4l'RIGHT	'Z' 'Z'
'HIGH	Upper bound of type or subtype	INTEGER'HIGH v3l'HIGH	Large 'Z'
'LOW	Lower bound of type or subtype	POSITIVE'LOW v4l'LOW	1 'X'
'POS(V)	Position of value V in *base* of type.	v4l'POS('Z') v3l'POS('X')	3 0
'VAL(P)	Value at Position P in *base* of type.	v4l'VAL(3) v3l'VAL(3)	'Z' 'Z'
'SUCC(V)	Value, after value V in *base* of type.	v3l'SUCC('1')	'Z'
'PRED(V)	Value, before value V in base of type.	v3l'PRED('1')	'0'
'LEFTOF(V)	Value, left of value V in *base* of type.	v3l'LEFTOF('1') v3l'LEFTOF('X')	'0' Error
'RIGHTOF(V)	Value, right of value V in *base* of type.	v3l'RIGHTOF('1') v3l'RIGHTOF('X')	'Z' '0'
'ASCENDING	TRUE if range is ascending	v4l'ASCENDING	TRUE
'IMAGE (V)	Converts value V of type to string.	v4l'IMAGE('Z') opcode'IMAGE(lda)	"Z" "lda"
'VALUE(S)	Converts string S to value of type.	opcode'VALUE("nop")	nop

Figure 6.38 Predefined Type Attributes.

The results of attributes 'LEFT, 'RIGHT, 'HIGH, 'LOW, and 'AS-CENDING correspond to the values of the types or subtypes that they are applied to, while the 'IMAGE, 'VALUE, 'POS, 'VAL, 'SUCC, 'PRED, 'LEFTOF, and 'RIGHTOF attributes perform the specified functions on the base of the subtype. For example, v3l'POS('X') results in 0, which is the position of 'X' in the *v4l* type. Notice that 'X' is not even contained in the *v3l* subtype.

6.5.3 Signal Attributes

Signal attributes are used for objects in the signal class of any type. Such attributes are used for finding events, transactions, or timings of events and transactions on signals. These attributes are most useful for modeling hardware properties.

Attributes 'STABLE, 'EVENT, 'LAST_EVENT, and 'LAST_VALUE deal with events occurring on a signal. For example, when 'EVENT is used with a signal, the result is true (BOOLEAN TRUE) when an event occurs on that signal, that is, when the value of the signal changes. Attributes 'QUIET, 'ACTIVE, 'LAST_ACTIVE, and 'TRANSACTION have to do with the transactions that occur on a signal. For example, *s_signal*'ACTIVE is true when a transaction occurs on the *s_signal*, even if the transaction does not cause a change of value on this signal.

The results of the attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION are signals and can be used like signal objects. For example, *s_signal*'DELAYED' STABLE is only valid because *s_signal*'DELAYED results in a signal to which the 'STABLE signal attribute can be applied.

Figure 6.39 presents a list of the signal attributes and gives a simple example for each one, showing the kind and type of the result. Also shown in this figure is a box indicating whether the attribute deals with transactions or events on a signal. The *s1* signal used in the examples is assumed to be a scalar signal of type BIT. Signal attributes are time-dependent, which means that their values may change continuously during simulation.

Figure 6.40 shows an example waveform on *s1* and the result of using various attributes with this BIT type signal. The waveform shown here includes transactions and events. Each transaction is indicated by a rectangular block (□) of δ duration on the waveform. Events occur when a transaction causes the value of *s1* to change. Those attributes whose results are of the type of signal that they are applied to (type BIT for signal *s1*) are shown by logical waveforms, and those with BOOLEAN or BIT results (independent of type of *s1*) are indicated with shaded blocks. Attributes that result in a signal are shown in the **bold** font.

Attribute	T/E	Example	Kind	Type
Attribute description for the specified example				
'DELAYED	-	s1'DELAYED (5 NS)	SIGNAL	As s1
A copy of *s1*, but delayed by 5 NS. If no parameter or 0, delayed by delta. Equivalent to TRANSPORT delay of *s1*.				
'STABLE	EV	s1'STABLE (5 NS)	SIGNAL	BOOLEAN
A signal that is TRUE if *s1* has not changed in the last 5 NS. If no parameter or 0, the resulting signal is TRUE if *s1* has not changed in the current simulation time.				
'EVENT	EV	s1'EVENT	VALUE	BOOLEAN
In a simulation cycle, if *s1* changes, this attribute becomes TRUE.				
'LAST_EVENT	EV	s1'LAST_VALUE	VALUE	TIME
The amount of time since the last value change on *s1*. If *s1*'EVENT is TRUE, the value of *s1*'LAST_VALUE is 0.				
'LAST_VALUE	EV	s1'LAST_VALUE	VALUE	As *s1*
The value of *s1* before the most recent event occurred on this signal.				
'QUIET	TR	s1'QUIET (5 NS)	SIGNAL	BOOLEAN
A signal that is TRUE if no transaction has been placed on *s1* in the last 5 NS. If no parameter or 0, the current simulation cycle is assumed.				
'ACTIVE	TR	s1'ACTIVE	VALUE	BOOLEAN
If *s1* has had a transaction in the current simulation cycle, *s1*'ACTIVE will be TRUE for this simulation cycle, for delta time.				
'LAST_ACTIVE	TR	s1'LAST_ACTIVE	VALUE	TIME
The amount of time since the last transaction occurred on *s1*. If *s1*'ACTIVE is TRUE, *s1*'LAST_ACTIVE is 0.				
'TRANSACTION	TR	s1'TRANACTION	SIGNAL	BIT
A signal that toggles each time a transaction occurs on *s1*. Initial value of this attribute is not defined.				
'DRIVING	-	s1'DRIVING	VALUE	BOOLEAN
If *s1*is being driven in a process, *s1*'DRIVING is TRUE in the same process.				
'DRIVING_VALUE	-	s1'DRIVING_VALUE	VALUE	As *s1*
The driving value of *s1* from within the process this attribute is being applied.				

Figure 6.39 Predefined Signal Attributes. Signal *s1* is of Type BIT
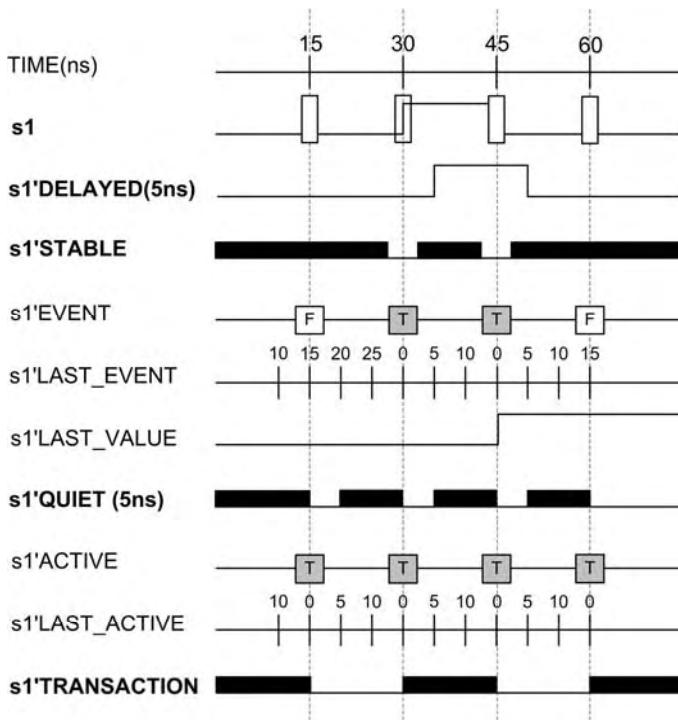
Figure 6.40 Results of Signal Attributes when Applied to the BIT Type Signal, *s1*

Common applications of signal attributes include edge detection, pulse-width verification, glitch detection, and level mode analysis. For example, the 'EVENT attribute in an edge-trigger flip-flop can check for a change in the value of the clock, that is, an edge of a clock. Let us consider the description of the falling-edge D-type flip-flop in Figure 6.41. The statement part of this description consists of a conditional signal assignment which conditionally assigns the D-input to *q*. The condition of the conditional signal assignment becomes TRUE if (1) *c* is *zero* and (2) *c* has had an *event* during the current simulation cycle. In other words, the condition is TRUE if *c* changes, and this change causes it to be '0'. Clearly, this condition detects the falling edge of *c* in the current simulation cycle.

Although c'EVENT and NOT c'STABLE are equivalent in many cases, because 'STABLE is itself a signal, different results can be produced when using 'EVENT or 'STABLE in concurrent statements. Consider, for example, the partial codes shown in Figure 6.42 for assignments to *qf* and *ql*. The *qf* signal is a flip-flop output while *ql* is a latch output.

```
ENTITY brief_d_flip_flop IS
   PORT (d, c : IN BIT; q : OUT BIT);
END brief_d_flip_flop;
--
ARCHITECTURE falling_edge OF brief_d_flip_flop IS
   SIGNAL tmp : BIT;
BEGIN
   q <= d WHEN (c = '0' AND c'EVENT);
END falling_edge;
```

Figure 6.41 A Simple Falling Edge Flip-Flop Using Signal Attributes

```
FF: BLOCK (c = '0' AND NOT c'STABLE) BEGIN
    qf <= GUARDED din;
END BLOCK FF;
--
LT: BLOCK (c = '0' AND c'EVENT) BEGIN
    ql <= GUARDED din;
END BLOCK LT;
```

Figure 6.42 Comparing 'STABLE and 'EVENT

Assignment to *qf* in Figure 6.42 is a guarded assignment using a guard expression that involves the 'STABLE attribute. When *c* makes a transition to '0', the guard expression becomes true. Since *c* is a signal, the change in the value of *c* causes the guarded signal assignment to *qf* to wakeup. And since this change makes the guard expression to become TRUE, *din* connects to *qf*. At this exact simulation time, the value of NOT c'STABLE is also true. In the next simulation cycle, that *c* has already been '0' for one simulation cycle, *c* is considered to be stable. Because of this, the NOT c'STABLE expression becomes false. In this simulation cycle, since c'STABLE itself is a signal, its change in value causes the guard expression of *qf* to see an event. This event causes the evaluation of the right hand side of *qf*. In this evaluation, the guard expression is false and, therefore, causes *din* to disconnect from *qf*. The end result is that *din* is connected to *qf* for only one simulation cycle. Since this time coincides with the falling edge of *c*, *qf* becomes the output of a falling edge D-flip-flop.

The situation with the *ql* output shown in Figure 6.42 is different than that of *qf*. In the first simulation cycle that *c* changes to '0' the guard expression of *ql* sees an event and at the same time evaluates to TRUE. This causes *din* to connect to *ql*. In the next simulation cycle, *c* has not had an event and therefore c'EVENT is FALSE. However, because c'EVENT is not a signal, the c'EVENT expression becoming FALSE does not cause the expression on the right hand side of *ql* to evaluate. This means that the TRUE value of the guard expression remains TRUE until another event occurs on one of the sig-

nals involved in this expression, i.e., *c* or *din*. This is a level sensitive behavior that causes *ql* to work as a transparent latch output.

As another example for the use of signal attributes, consider the *brief_t_flip_flop* in Figure 6.43. This is the description of a toggle flip-flop that toggles only when a positive pulse longer than 20 ns appears on its *t* input. A conditional signal assignment assigns one of the two possible values to the *tmp* signal, which is then assigned to the output. The *tmp* signal, which stores the internal state of the flip-flop, is assigned to the complement of itself when two conditions are TRUE. The first condition is the falling edge of *t* (t='0' AND t'EVENT), which is the same as the condition in the example in Figure 6.41. The second condition is TRUE if *t*, before this last fall, has been stable for 20 ns. The operation t'DELAYED evaluates to a signal that is delayed from *t* by 0 ns, and therefore, it does not include the change that just occurred on it. If this delayed signal has been stable for at least 20 ns, we can conclude that the width of the positive pulse on *t* has been at least 20 ns.

```
ENTITY brief_t_flip_flop IS
   PORT (t : IN BIT; q : OUT BIT);
END brief_t_flip_flop;
--
ARCHITECTURE toggle OF brief_t_flip_flop IS
   SIGNAL tmp : BIT;
BEGIN
   tmp <= NOT tmp WHEN (
       (t = '0' AND t'EVENT) AND (t'DELAYED'STABLE(20 NS))
                    ) ELSE tmp;
   q <= tmp AFTER 8 NS;
END toggle;
```

Figure 6.43 A Simple Toggle Flip-Flop Using Signal Attributes

Attributes 'DRIVING and 'DRIVING_VALUE are used for reading driver specifications of signals. In a process statement, *s* 'DRIVING returns TRUE if signal *s* has at least one driver in that region. Since an out port of a component cannot be read from within the component, the 'DRIVING_VALUE provides a convenient mechanism for reading out port values. In an INOUT port, where multiple drivers can exist for a port, the 'DRIVING_VALUE reads the value given to such a port in the process statement to which this attribute is being applied. As will be discussed in Chapter 7, reading a resolved signal with multiple drivers results in a value that is the resolution of all its drivers. To read only the driver of such a signal, the 'DRIVING_VALUE attribute can be used.

6.5.4 Entity Attributes

Entity attributes may be used to generate a string corresponding to the name of signals, components, architectures, entities, or other members of what is referred to as entity class in VHDL. The VHDL language construct *entity_class* consists of entities, architectures, configurations, procedures, functions, packages, types, subtypes, constants, signals, variables, components, labels, literals, units, groups, and files. In a design unit, in order to extract a string representing any of these named entities, an entity attribute may be used.

Attributes 'SIMPLE_NAME, 'PATH_NAME and 'IN-STANCE_NAME provide hierarchical path information to named entities. The simplest of these attributes is 'SIMPLE_NAME and provides a string representing the simple name of the named entity. In an architecture of an entity, the use of this attribute results in a string representing the entity name.

The 'PATH_NAME, when used in an architecture at a low level of hierarchy, generates a string containing entity names and instantiation labels from the top of hierarchy leading to the named entity that the attribute is being applied to.

```
ENTITY multiplexer_n_tester IS END ENTITY;
--
ARCHITECTURE timed OF multiplexer_n_tester IS
   SIGNAL a : BIT_VECTOR(7 DOWNTO 0);
   SIGNAL s : BIT_VECTOR(2 DOWNTO 0);
   SIGNAL w1 : BIT;
   FOR UUT1: mux_n
      USE ENTITY components.multiplexer(customizable);
BEGIN
   UUT1: mux_n PORT MAP (a, s, w1);
   onehot_data (a, 123 NS, 9);
   consecutive_data (s, 79 NS, 11);
END ARCHITECTURE timed;

ENTITY multiplexer IS
   PORT (ins: IN BIT_VECTOR; s: IN BIT_VECTOR; w: OUT BIT);
END ENTITY multiplexer;
--
ARCHITECTURE customizable OF multiplexer IS BEGIN
   ASSERT FALSE
   REPORT customizable'SIMPLE_NAME SEVERITY NOTE;
   ASSERT FALSE
   REPORT customizable'PATH_NAME SEVERITY NOTE;
   ASSERT FALSE
   REPORT customizable'INSTANCE_NAME SEVERITY NOTE;
   w <= mux(ins, s);
END ARCHITECTURE customizable;
```

Figure 6.44 Applying Entity Attributes

The string generated by the 'INSTANCE_NAME, when applied to a named entity, contains entity, architecture, and instantiation labels leading to the design entity from the top of hierarchy in a hierarchical design.

These attributes are especially useful when reporting certain events on signals and/or components. A signal, for example, may be unambiguously identified from inside of a component deep down in hierarchy in a hierarchical design.

Figure 6.44 shows a multiplexer description and its corresponding testbench. This design was discussed in Chapter 5. In the code shown in *customizable* architecture version of this chapter (Figure 6.44) we have added assert statements reporting 'SIMPLE_NAME, 'PATH_NAME, and 'INSTANCE_NAME of the *customizable* entity class.

Figure 6.45 shows results produced by the ASSERT statements of Figure 6.44. As shown, the 'SIMPLE_NAME only produces the name of the architecture. 'PATH_NAME produce the name of the entity that *customizable* is instantiated from and its instance name. The 'INSTANCE_NAME attribute produces all entity, architecture, and instance names from where architecture is instantiated leading to the architecture itself.

```
--# ** Note: customizable

--# ** Note: :multiplexer_n_tester:uut1:

--# ** Note: :multiplexer_n_tester(timed)
            :uut1@multiplexer(customizable):
```

Figure 6.45 Entity Attribute Examples

6.5.5 User-Defined Attributes

In addition to the predefined attributes, VHDL allows definition and use of user-defined attributes. Such attributes do not have simulation semantics, so it is up to the user to define them and use them in accordance with the way they are defined.

User-defined attributes may be applied to the elements of an entity class, as described in the previous section. Before an attribute can be used, it has to be declared using an attribute declaration. An attribute declaration identifies a name as an attribute with a given type. For example, the following declaration declares *sub_dir* as an attribute that can take values of STRING type:

```
ATTRIBUTE sub_dir : STRING;
```

If the above declaration is made visible to a description, it can be associated with any of the elements of the entity class mentioned above, i.e., entity, architecture, configuration, etc. For example, in order to associate the *sub_dir* attribute with the *multiplexer* entity in Figure 6.44, this attribute specification must appear in the declarative part of that entity:

```
ATTRIBUTE sub_dir OF multiplexer: ENTITY IS "/user/vhdl";
```

The expression *multiplexer'sub_dir*, anywhere in an architecture of the *multiplexer* entity, evaluates to "*/user/vhdl*".

Figure 6.46 shows *delay* and *sub_dir* attribute definitions. The type of the *delay* attribute is *timing* that is also defined in this figure. The *sub_dir* attribute has a STRING type. Using the *utility_attributes* package makes these attributes visible to designs.

```
PACKAGE utility_attributes IS
   TYPE timing IS RECORD
      rise, fall : TIME;
   END RECORD;
   ATTRIBUTE delay : timing;
   ATTRIBUTE sub_dir : STRING;
END utility_attributes;
```

Figure 6.46 Attribute Definitions

The entity declaration of *multiplexer* shown in Figure 6.47 uses the attributes of the *utility_attributes* package. In the declarative part of this entity, the attributes are associated with the entity itself and with the output of this entity. In the statement part of the *customizable* architecture of the *multiplexer* entity, the *rise* and *fall* fields of the *delay* attribute of the *w* output are used in calculating the delay values on this output.

An attribute specification for associating a user-defined attribute with an entity class can appear in any declarative part in which the attribute and the entity it is being applied to are visible. For example, in Figure 6.47, the attribute specification that associates *delay* with the *w* output could appear in the declarative part of the *customizable* architecture. In the same example, it is also worthwhile noting that if the *delay* attribute is to be applied to other signals, those signals should be listed along with *w*, separated by commas. If an attribute is to be applied to all visible signals, the keyword ALL can replace the list of individual signals. The keyword OTHERS can also be used to apply the attribute to all entity classes that have not been specified above it.

Values of user-defined attributes can be used in expressions or on the right-hand side of assignments, but no assignments can be made to them.

```
USE WORK.utility_attributes.ALL;

ENTITY multiplexer IS
   PORT (ins: IN BIT_VECTOR; s: IN BIT_VECTOR; w: OUT BIT);
   ATTRIBUTE sub_dir OF multiplexer : ENTITY
                                      IS "/user/vhdl";
   ATTRIBUTE delay OF w : SIGNAL IS (8 NS, 10 NS);
END ENTITY multiplexer;
--
ARCHITECTURE customizable OF multiplexer IS BEGIN
   w <= '1' AFTER w'delay.rise WHEN mux(ins, s) = '1' ELSE
        '0' AFTER w'delay.fall;
END ARCHITECTURE customizable;
```

Figure 6.47 Using Attributes

6.6 Standard Libraries and Packages

Adaptation of the VHDL language to new technologies and new design methodologies is done via its libraries and packages. With the original definition of the language the STD library that included the STANDARD and TEXTIO packages were defined. Later on, the IEEE library was defined that included new types and utilities based on the industry standard *std_logic* type. Contents of these packages and their utilization are described here. The complete packages of the important libraries are included in the CD that accompanies this book, and declarations of key IEEE packages are included in appendices in this book. This section gives a brief overview of package contents and shows examples where needed.

6.6.1 STANDARD Package

The STANDARD package is in the STD library. This package in an internal language package and its body does not exist as a VHDL code. Basic types such as BIT, BIT_VECTOR, INTEGER, and other types discussed thus far in this book are included in the STANDARD package.

VHDL logical, relational, and arithmetic operations are over loaded for basic types of this package. For example the AND logical operation is defined for BIT and BIT_VECTOR so that two BIT type operands or two BIT_VECTOR operands of the same size can be ANDed.

While arithmetic operations are defined for INTEGER and REAL types, this package does not overload these operations for the BIT and BOOLEAN types. If needed, users can develop their own binary arithmetic functions. The standard numeric package (IEEE 1076.3) is the NUMERIC_BIT package that contains overloading of arithmetic operations for the BIT and BIT_VECTOR types.

6.6.2 TEXTIO Package and ASCII I/O

Basic unformatted input-output to external files was described earlier in this chapter. The methods described used primitive VHDL file operations and can be used with any data type. VHDL also supports a TEXTIO package which includes types and procedures for ASCII line-oriented input or output. This package is in the STD library and is shown in an appendix in this book.

The TEXTIO package defines the LINE type as a pointer to STRING, which is used for all text file readings and writings. The file type provided by this package is TEXT, and it defines files of ASCII strings. Procedures defined in this package for handling input-output are READ, READLINE, WRITE, and WRITELINE. In addition, function ENDFILE provides a mechanism for checking the status of a file. These subprograms can be used only if a TEXT type file is declared and opened using language utilities described in Section 6.1.6.

For reference, file declarations, file open, and file close statements are shown here for an example TEXT file. The following statements are alternatives for opening file *f* of type TEXT.

```
FILE f: TEXT;
FILE f: TEXT IS "input.txt";
FILE f: TEXT OPEN READ_MODE IS "input.txt";
```

If a file is declared using the first declaration alternative, it must be opened before it can be read from or written into. The following procedure calls open file *f* in three possible read, write, and append modes:

```
FILE_OPEN (f, "input.txt", READ_MODE);
FILE_OPEN (f, "output.txt", WRITE_MODE);
FILE_OPEN (f, "output.txt", APPEND_MODE);
```

The procedure call shown below closes file *f* after it has been opened for read and write operations:

```
FILE_CLOSE (f);
```

The READLINE (f, l) procedure reads a line of file *f* and places it in buffer *l* of type LINE. The READ (l, v, …) reads a value *v* of its type form *l*. The WRITE (l, v, …) writes the value *v* to LINE *l* and WRIELINE (f, l) writes *l* to file *f*. Function ENDFILE (f) returns TRUE if the end of file f is reached.

READ and WRITE procedures are valid for values of types BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING, and TIME. Other parameters of these procedures include orientation, size, and unit if *v* is of type TIME.

For reading from a file, after READLINE reads a line, data can be extracted from the line (or buffer) using READ. This can continue until the entire buffer is consumed. For writing to a file, a LINE type variable is filled with data using WRITE, and the line is written to the file using WRITELINE.

6.6.2.1 TEXTIO Reading. Figure 6.48 shows a procedure that uses the TEXTIO package for reading standard type data from a file. The *GetData* procedure takes a target signal, *s*, and a file input as its arguments. Data read from file input *f* will be scheduled on *s*. The file input is a declared file that has been opened outside of the procedure. As shown, the type of this file is TEXT that is defined in the TEXTIO package.

The declarative part of *GetData* declares *lbuf* of the predefined LINE type. This variable is used to read a line from file *f*. In the body of *GetData*, a line is read from *f* using READLINE. Following this, two invocations to the READ procedure read time and data. Type of *t* is TIME and type of *d* is BIT_VECTOR. Data extracted from *lbuf* are according to argument type of the READ procedure thus, time is read into *t* and BIT values are read into *d*. Data extracted into *d* are scheduled into the target *s* signal.

```
PROCEDURE GetData
    (SIGNAL s : OUT BIT_VECTOR; FILE f : TEXT)
IS
    VARIABLE lbuf : LINE;
    VARIABLE t : TIME;
    VARIABLE d : BIT_VECTOR (s'RANGE);
BEGIN
    WHILE NOT ENDFILE (f) LOOP
        READLINE (f, lbuf);
        READ (lbuf, t);
        READ (lbuf, d);
        s <= TRANSPORT d AFTER t;
    END LOOP;
    FILE_CLOSE (f);
END PROCEDURE GetData;
```

Figure 6.48 Reading a TEXTIO File

Figure 6.49 shows a sample input file illustrating the format of data expected by the *GetData* procedure. Time and its unit come first in each line, followed by a white-space and then followed by eight BIT type values.

```
  0 ns 00111000
 10 ns 00101111
 35 ns 10110000
 45 ns 11101010
 50 ns 01100001
 55 ns 00101110
 95 ns 11100011
110 ns 00011100
```

Figure 6.49 Sample Data File

6.6.2.2 TEXTIO Writing. We show a testbench for testing an 8-bit 2-TO-1 multiplexer for demonstrating TEXTIO writing. Figure 6.50 shows a complete text input and output testbench for our *multiplexer8* entity. The multiplexer has two 8-bit inputs and one is selected for the *w1* output.

```
USE STD.TEXTIO.ALL;

ENTITY multiplexer8_tester IS END ENTITY;
--
ARCHITECTURE timed OF multiplexer8_tester IS
   SIGNAL a, b, w1 : BIT_VECTOR (7 DOWNTO 0);
   SIGNAL s : BIT := '0';
   FILE Ain : TEXT OPEN READ_MODE IS "Ain.dat";
   FILE Bin : TEXT OPEN READ_MODE IS "Bin.dat";
BEGIN

   UUT1: ENTITY WORK.multiplexer8 (conditional)
         PORT MAP (a, b, s, w1);
   PROCESS (w1)
      FILE Wout : TEXT OPEN WRITE_MODE IS "Wout.dat";
      VARIABLE lbuf : LINE;
   BEGIN
      WRITE (lbuf, NOW, RIGHT, 8, NS);
      WRITE (lbuf, w1, RIGHT, 9);
      WRITELINE (Wout, lbuf);
   END PROCESS;

   GetData (a, Ain);
   GetData (b, Bin);
   s <= NOT s AFTER 25 NS WHEN NOW <= 140 NS ELSE '0';
END ARCHITECTURE timed;
```

Figure 6.50 Using Text Data for Input and Output

The testbench shown uses the STD.TEXTIO utilities as illustrated by the first line in Figure 6.50. The declarative part of the *multiplexer8_tester* declares *Ain* and *Bin* logical files of type TEXT and associates these logic files with *Ain.dat* and *Bin.dat* physical files. Using two invocations of *GetData* of Figure 6.48, data from *Ain* and *Bin* are read and scheduled into *a* and *b*, respectively.

Writing of multiplexer result is done by the PROCESS statement shown in the architecture body of our testbench. This process is sensitive to the multiplexer output, *w1*. The *Wout* logical file of TEXT type is declared, opened, and associated with the *Wout.dat* physical file. Since this file statement appears in the declarative part of a process statement, it is executed only once at the beginning of the simulation. Subsequent writings are done to the end of the file.

For writing into *Wout*, two WRITE statements write to *lbuf*, and WRITELINE writes *lbuf* to *Wout*. The first WRITE writes the current time (NOW). The time is written using an 8 character field, right justified, and uses the NS time unit. A nine character field in dedicated to the *w1* output, allowing a white-space and eight BIT type data. The format of the data written into *Wout* is the same as input files illustrated in Figure 6.49.

6.6.2.3 Std_logic TEXTIO. The *std_logic* package provides utilities for the standard nine-value logic. For TEXTIO in this type, the *std_logic_TEXTIO* package is introduced. This package is in the IEEE library and is used in conjunction with the standard TEXTIO package. The *std_logic_TEXTIO* adds *std_logic* read and write functions to the existing ones in TEXTIO.

In order to be able to read and write in *std_logic*, the library and package use statements shown in Figure 6.51 must be included in a corresponding design file. With the inclusion of what is shown here, READ and WRITE procedures work for *std_logic* types as well as for those of the standard package.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_TEXTIO.ALL;
```

Figure 6.51 std_logic TEXTIO Package

6.6.3 Std_logic_1164 Package

The standard *std_logic_1164* package is in the IEEE library and defines types and utilities for the standard industry standard nine-value logic system. We have been using the *v4l* type in this chapter to

illustrate what is needed in a package for definition and utilization of a logic value system. Utilities in *std_logic_1164* package are similar to those we explained in *VerilogLogic* except that those of the former are more extensive and more complete.

6.6.3.1 Type Definition. The *std_logic_1164* defines *std_ulogic* and *std_logic* types. The *std_logic* is the resolved version of *std_ulogic* and will be explained in the next chapter. These types are enumeration types and have nine values as shown in Figure 6.52.

```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak      Unknown
                     'L',  -- Weak      0
                     'H',  -- Weak      1
                     '-'   -- Don't care
                   );
```

Figure 6.52 *Std_logic* Enumeration Values

In addition, vector versions of these types are also defined as unconstrained arrays. These types are *std_logic_vector* and *std_ulogic_vector*.

Common subsets of *std_ulogic* for various logic systems are also defined and shown in Figure 6.53. As discussed in relation to subsets, objects declared as subset of a type are fully compatible with these of the base type.

```
SUBTYPE X01    IS resolved std_ulogic
          RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z   IS resolved std_ulogic
          RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01   IS resolved std_ulogic
          RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z  IS resolved std_ulogic
          RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
```

Figure 6.53 Subtypes of the *std_logic* Type

6.6.3.2 Overloaded Logical Operators. The standard logic package contains overloaded logic operators for "AND", "NAND", "OR", "NOR", "XOR", "XNOR" and "NOT" operators. These operators are overloaded for *std_ulogic*, and because *std_logic* is considered a subtype of *std_ulogic*, they also work for the *std_logic* type.

In addition, the standard logical operators named above, are overloaded for *std_logic_vector* and *std_ulogic_vector* and combinations of the two arrays. With these overloadings, a designer can simply use *std_logic* instead of BIT to take advantage of this nine-value system.

6.6.3.3 Conversion Functions. The *std_logic_1164* package contains functions for conversions to and from BIT and *std_logic* and its subsets. For example *To_StdLogicVector* converts *BIT_VECTOR* or *std_ulogic_vector* to *std_logic_vector*. Another example is function *TO_X01* that converts BIT, *std_logic*, *std_ulogic* and their vectorized versions to *X01* and *std_logic_vector*.

6.6.3.4 Edge Detection. Edge detection functions *rising_edge* and *falling_edge* are included in the standard logic package. These functions are recognized by most synthesis tools for flip-flop clock edge detection.

6.6.4 Std_logic_arith Package

The IEEE standard arithmetic package is an important package that eases the use of the VHDL language for arithmetic and logical functions, in spite of the language's complex typing system.

The *std_logic_arith* defines *SIGNED* and *UNSIGNED* unconstrained arrays of *std_logic*. It then overloads all arithmetic and relational operators of VHDL, 1) for SIGNED, INTEGER, and NATURAL types, and 2) for UNSIGNED, INTEGER, and NATURAL. With this overloading, we can use "+" for adding a signed or an unsigned *std_logic_vector* with an integer. Similarly we can mix a signed or unsigned vector with an integer in relational operations, i.e., ">", "<", "<=", ">=", "=", and "\".

6.6.4.1 The UNSIGNED Package. The use of *std_logic_arith* requires specification of all objects as SIGNED or UNSIGNED. Once an object is declared as either of these types, conversion to the other and conversion to *std_logic* becomes difficult. The *std_logic_unsigned* package sits on top of the *std_logic_arith* package, it assumes all *std_logic_vector* declarations are unsigned and overloads all arithmetic and relational operators for unsigned numbers declared as *std_logic_vector*. The unsigned package already includes the arithmetic package. To use all unsigned arithmetic on the *std_logic_vector* types, library and package use clauses shown in Figure 6.54 must be used.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_UNSIGNED.ALL;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_SIGNED.ALL;
```

Figure 6.54 Using Unsigned and Signed

6.6.4.2 The SIGNED Package. The *std_logic_signed* package sits on top of the *std_logic_arith* package and forces all logical and relational operators to treat their operands as signed 2's complement numbers. The type mark recognized in this package is *std_logic_vector* that is treated as a signed type. Figure 6.54 shows library and use statements for utilizing utilities of the *std_logic_SIGNED* package.

If a design requires both signed and unsigned arithmetic, the *std_logic_arith* or *NUMERIC_STD* must be used. SIGNED and UNSIGNED packages only allow signed or unsigned arithmetic.

Other packages include MATH_REAL, MATH_COMPLEX, and other fixed and floating point packages. Standard packages and some of the proposed ones appear in the CD that accompanies this book.

6.7 Summary

This chapter focused on linguistics aspects of VHDL and discussed types, operators, overloading, and related matters. Details of type definitions, overloading and attributes were discussed. At the same time we used a simple type definition to illustrate concepts discussed. The last section introduced standard libraries that define standard types and operators. Use of libraries and standard packages simplifies the use of VHDL for design or description of hardware based on standard technologies. A lot of times, use of packages eliminates the need for understanding many of difficult language constructs. However, for a better understanding of the languages and with a look into future technologies, the issues discussed in the earlier part of chapter become important.

Problems

6.1 Using concurrent statements, write a VHDL code that sets an error signal (*error*) to ERROR if the period of an incoming clock signal

ever drops below 1 US. At all other times the *error* signal value stays at GOOD. Declare necessary types.

6.2 A) Write a T flip-flop description with a clock (*clk*) and a *t* input. Toggling is done on the rising edge of *clk* when *t* is '1'. Include a generic parameter for the flip-flop delay, one for the minimum pulse width on *t*, and one for the flip-flop identification number. Each flip-flop should be able to report an error message (use assertion statements) if a glitch of less than the specified parameter is detected on its *t* input. The message should include the flip-flop identification number (use 'IMAGE to convert integer to string). B) Write a generate statement for generating an unconstrained counter using T flip-flops of Part A. Do not specify binding, generate a unique identification number for each flip-flop. C) Write a configuration declaration for binding the counter to its flip-flops and specifying its generic values.

6.3 VHDL attributes are applied to generate waveforms on seven signals. Show values that appear on these signals considering all delta delays.

```
PACKAGE types_and_values IS
   TYPE four IS ('0', b, c, '1');
   CONSTANT d1 : TIME := 100 US;
   CONSTANT d2 : TIME := 050 US;
END types_and_values;
--
USE WORK.types_and_values.ALL;
--
ENTITY type_test IS END ENTITY;
--
ARCHITECTURE attributes OF type_test IS
   SIGNAL a1, a2, a3, a4, a5, a6, a7 : four;
BEGIN
   a1 <= four'RIGHTOF(a1) AFTER d1 WHEN a1 /= four'RIGHT
                ELSE '0' AFTER d1;
   a2 <= a1'DELAYED(d2);
   a3 <= a1 AFTER d2;
   a4 <= '1' WHEN a2 = a3 ELSE '0';
   a5 <= '1' WHEN a1'STABLE ELSE '0';
   a6 <= '1' WHEN NOT a1'EVENT ELSE '0';
   a7 <= '1' WHEN a5'QUIET = a6'QUIET ELSE '0';
END attributes;
```

6.4 A signal is to be passed to a procedure. The procedure is to count all transactions that have occurred on the signal from time 0 to the time of calling of the procedure. The procedure returns this IN-

TEGER count via its second argument. Write the procedure using *sig* for input signal and *cnt* for output transaction count. The procedure may be called from a sequential body such as a process statement.

6.5   Is the statement shown in the statement part of the architecture shown below a valid VHDL statement? Does this architecture simulate as it is shown here? If not, write a function so that when used by the description shown here, the signal assignment in this description becomes a valid simulatable VHDL statement. What you write the function to do is not important; just write a valid function to return any thing that makes the statement simulatable.

```
ENTITY WhatIsThis IS
   --FUNCTION ???
        . . .
   --END ???
END WhatIsThis;
--
ARCHITECTURE behavioral OF WhatIsThis IS
   SIGNAL a, b, c, d : INTEGER := 0;
BEGIN
   a <= (b <= c) <= d;
END behavioral;
```

6.6   Write a VHDL architecture with an enable input and 3 data inputs a, b, and c. When enable becomes '1', events on a, b, and c are counted and will be reported to the output when enable becomes 0. The output is a 4-bit binary number that can only keep modulo-16 counts of the input events. Consider simultaneous events on the inputs.

Suggested Reading

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Barnes, John, *Programming in Ada 2005*, 2006, Addison Wesley, ISBN: 0321340787.

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

Burns, Alan, Andy Wellings, and John Barns, *Concurrency in Ada, 2nd edition*, 1998, Cambridge University Press, ISBN: 052162911X.

Chu, Pong, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press, ISBN: 0471720925.

Cohen, Norman, *Ada As A Second Language, 2nd edition*, 1995, McGraw-Hill Science/Engineering/Math, ISBN: 0070116075.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL Hardware Description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Perry, Douglas L., *VHDL: Programming by Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

7 VHDL Signal Model

The previous chapters focused on the use of VHDL in design and discussed most of its constructs. The syntax and semantics of most language constructs have been discussed up to this point in the book.

The focus of this chapter is on the VHDL simulation model; we discuss how VHDL deals with timing and concurrency which are the two most important factors that distinguish a hardware language from a software one. The first part of the chapter deals with the VHDL delay modeling. In this part we discuss how multiple sequential values interact when placed on the driver of a signal. The second part of the chapter deals with concurrency modeling. This part shows how multiple concurrent drivers of a signal interact to generate a value for the signal. Together, multiple sequential values into a driver and multiple drivers for a signal, provide a complete signal model that models most properties of actual hardware signals.

7.1 Characterizing Hardware Languages

Timing and concurrency are the main characteristic of an HDL. These features are instrumental in the correct description of hardware components at various levels of abstraction.

7.1.1 Timing and Concurrency of Operations

We use a small example to illustrate how timing and concurrency of operations help correct modeling of hardware. The example is a 2-to-1 multiplexer shown in Figure 7.1. Depending on the value of s, input a or b is selected to appear on w.

In the circuit shown, numbers inside gates represent delay values of these gates in nanoseconds. We will examine this circuit for the situation shown in this figure in which s changes from '1' to '0' at time $t0$. As shown in the timing diagram, because of time delay differences in the two paths from s to w, 8 ns after the change on s, a 4 ns glitch appears on w. Timing and concurrency of operations in VHDL allow VHDL model for this circuit to simulate exactly as shown in Figure 7.1.
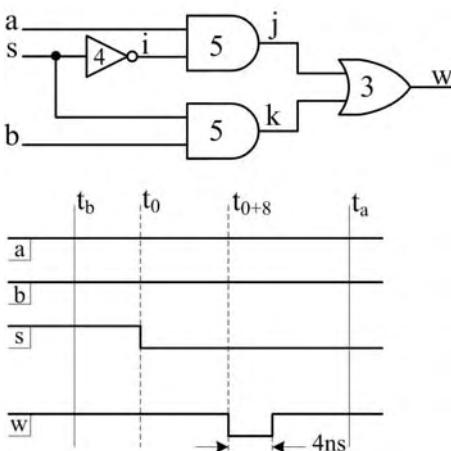
Figure 7.1 Illustrating Timing and Concurrency

7.1.1.1 Sequential Modeling. Figure 7.2 shows a sequence of four statements for modeling the multiplexer of Figure 7.1. The sequentiality in execution of these statements is the way a software language operators. Evaluating these statements at time t_b and t_a, before and after s changes, results in correct values for w at these times. However the model of Figure 7.2 cannot properly model the 8 ns delay and the 4 ns glitch that appears on s starting at time t_0+8.

```
i := NOT s;
j := a AND i;
K := s AND b;
W := j OR k;
```

Figure 7.2 Modeling a Multiplexer with Sequential Statements

7.1.1.2   Concurrent Modeling.   Unlike the sequential model dis-
cussed above, modeling the circuit of Figure 7.1 with the statements
that contain timing and execute concurrently can generate the de-
signed timing of Figure 7.1. The VHDL model of the multiplexer is
shown in Figure 7.3.
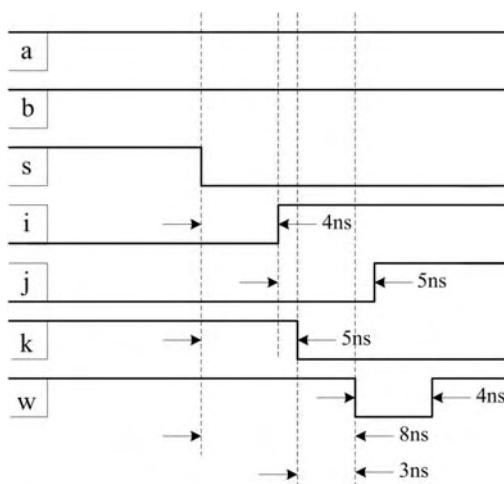
```
ENTITY mux IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE concurrent of mux IS
   SIGNAL i, j, k : BIT;
BEGIN
   i <= NOT s AFTER 4 NS;
   j <= a AND i AFTER 5 NS;
   k <= b AND s AFTER 5 NS;
   w <= j OR k AFTER 3 NS;
END ARCHITECTURE concurrent;
```

Figure 7.3   Modeling a Multiplexer with Concurrent Statements

Unlike the statements in Figure 7.2, these in Figure 7.3 do not
execute in the order that they appear. Statements in Figure 7.3 are
sensitive to their right hand sides and when an event occurs on a sig-
nal on the right hand side of a statement, e.g., *s* changes from '1' to
'0', the corresponding statement executes and schedules a value to its
left hand side signal. A scheduled value on a signal becomes the value
of the signal when its time arrives. For example, due to the change in
s at t_0 a '1' will be scheduled for *i* to occur at $t_0 + 4$ ns.

Figure 7.4   Timing of Signals of Concurrent Description of Figure 7.3

The sequence of events and values scheduled for various signals of Figure 7.3 are shown in Figure 7.4. When k changes at t_0+5 ns, the right hand side of w sees a change causing a '0' to be scheduled for this signal 3 ns later. At time t_0+9 ns, j changes that causes another scheduling on w. The scheduling on w cause the 4 ns glitch to appear on w as the result of the execution of concurrent assignments of Figure 7.3.

Evaluation of the sequential and concurrent models of circuit of Figure 7.1 illustrate that concurrency and timing are needed for proper modeling of hardware constructs. In the sections that follow we discuss semantics of VHDL statements and concurrency in this language that make behaviors such as that of Figure 7.4 to happen.

7.2 Signal Assignments

Assignment of values to signals in sequential and concurrent bodies is an important issue in VHDL. Concepts related to delay modeling will be discussed in this section. Section 7.3 continues this topic as it relates to concurrency. Issues related to multiple concurrent assignments to the same signal are covered in Section 7.4.

In its simplest form, a signal assignment consists of a target signal on the left hand side of a left arrow and an expression for defining a waveform on the right hand side (with no AFTER clause). Such an assignment specifies that the right hand side be assigned to the left hand side a delta time later. Physically this time is 0 s, but it has nonzero scheduling significance. For example, an assignment that is scheduled to occur two delta times later will be done after an assignment that is scheduled to occur after one delta, and the result of the later assignment will not be available for the earlier assignment. Both assignments, however, occur before the smallest physical time unit. This will become clearer when we discuss concurrency in Section 7.3.

Optionally, a signal assignment can include an AFTER clause specifying that a physical time delay is to occur before the assignment to the left hand takes place. If this time delay is zero, the simple form described above will apply.

Signal assignments can have *inertial* or *transport* delays. Inertial delays can have an additional *reject* specification. The way delays are handled in signal assignments is referred to as delay modeling. The delay mechanism for a signal assignment may be specified on the right hand side of the assignment. The default mechanism is inertial. Transport delay mechanism must be explicitly specified.

The *delay* architecture of the *example* entity in Figure 7.5 includes assignments with inertial, inertial with reject, and transport

mechanisms. This architecture also includes an assignment for creating a reference signal, *waveform*. The waveform on *waveform* consists of positive and negative pulses of 5 ns, 4 ns, 3 ns, and 2 ns that are distanced by 6 ns inactive periods.

In the first assignment, the INERTIAL keyword is optional. The second assignment specifies a reject value for an inertial delay mechanism. In this assignment, both REJECT and INERTIAL keywords are mandatory. The third assignment has a transport delay mechanism.

```
ENTITY example IS END ENTITY;
--
ARCHITECTURE delay OF example IS
   SIGNAL waveform : BIT;
   SIGNAL target1, target2, target3 : BIT;

BEGIN
   -- Inertial delay
   target1 <= waveform AFTER 5 NS;

   -- Inertial with reject
   target2 <= REJECT 3 NS INERTIAL waveform AFTER 5 NS;

   -- Illustrating transport delay
   target3 <= TRANSPORT waveform AFTER 5 NS;

   -- Creating waveform (not shown)
   waveform <= -- P5, N6, P4, N6, P3, N6, P2, P6,
               -- N5, P6, N4, P6, N3, P6, N2, N6;

END delay;
```
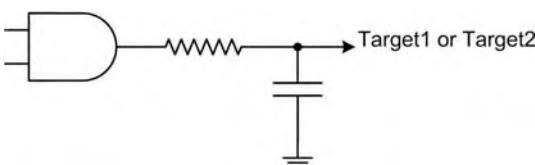
Figure 7.5 VHDL Description for the Demonstration of Delay Mechanisms

7.2.1 Inertial Delay Mechanism

Inertial delays can be used to model capacitive networks, such as the one shown in Figure 7.6, which corresponds to the first assignment of Figure 7.5. If a pulse whose width is less than 5 ns occurs on *waveform*, it will be rejected and does not appear on *target1*. A pulse of exactly 5 ns will not be rejected.

Figure 7.6 The RC Delay is Best Represented by an Inertial Delay Mechanism

Delays through capacitive networks and through gates with threshold values can be more accurately modeled with an inertial delay mechanism and a pulse rejection value that is less than the value of the inertial delay. The second assignment of Figure 7.5 is an inertially delayed signal assignment with a reject value of 3 ns. As long as pulses on the right hand side waveform signal are larger than 3 ns, they appear on *target2* with 5-ns delay. A pulse that is less than, or equal to, 3 ns will not appear on *target2*.

7.2.2 Transport Delay Mechanism
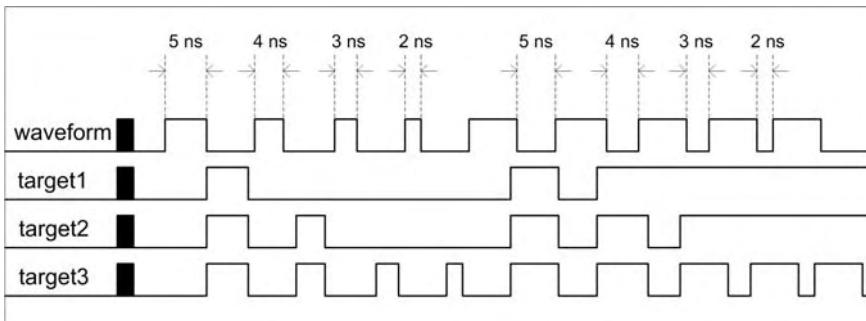
Delays through transmission lines and networks with virtually infinite frequency response can be modeled by the transport delay mechanism. Regardless of the width of a pulse, a signal assignment with a transport delay mechanism delays the right hand side signal by the specified transport delay value. The third assignment of Figure 7.5 causes *target3* to be a 5-ns-delayed duplicate of *waveform*.

7.2.3 Comparing Inertial and Transport

For observing the differences between the three delay mechanisms, Figure 7.5 generates a waveform on its *waveform* signal that has pulses of different durations. On the right hand side of the *waveform* signal, a waveform consisting of four positive and four negative pulses of 5, 4, 3, and 2 ns are produced. Positive pulses are separated by 6-ns negative pulses, and negative pulses are separated by 6-ns positive pulses. The first line in Figure 7.7 shows the waveform produced on *waveform*.

The same *waveform* signal appears on the right hand side of three signal assignments to *target1*, *target2*, and *target3* signals. Delay values for these three assignments are 5 ns. Assignments to *target1* and *target2* are inertial, while assignment to *target3* is of the transport mechanism. Assignment to *target2* specifies a reject value of 3 ns.

As shown in Figure 7.7, positive or negative pulses of 5 ns or more appear on all three targets. A 4-ns pulse (positive or negative) appears on *target2* because it is larger than the reject value of this assignment, and it appears on *target3* because of the transport mechanism. The two 4-ns pulses on *waveform* are rejected by the inertially delayed assignment to *target1*. Figure 7.7 also shows that pulses that are less than 3 ns only appear on *target3*, which has a transport delay mechanism on its right hand side.

Figure 7.7 Illustrating Differences between Delay Mechanisms in VHDL

Figure 7.7 shows that the difference between *target1* and *target2* is only when a 4 ns pulse (larger than reject and smaller than inertial delay) appears on *waveform*. Differences between left hand side *target1* and *target3* signals occur when pulses less than the inertial delay value of 5 ns appear on *waveform*.

The effect of a delay mechanism can be produced by using other mechanisms. For example, an inertial delay with a delay value *d* and a reject value of 0 ns is equivalent to a transport delay with delay *d*. Other forms of pulse delay and/or pulse rejection can be generated by Boolean combination of signals with different delay formats.

7.3 Concurrent and Sequential Assignments

Basic concepts of concurrency and VHDL bodies for sequential and concurrent statements have been discussed earlier in this book as well as in Section 7.1. This section discusses assignments to signals in sequential and concurrent bodies. This discussion explains the language mechanisms that make it implement various delay mechanisms as discussed in the previous section.
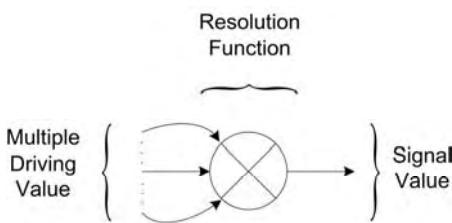
7.3.1 Concurrent Assignments

In an architectural body, all signal assignments are concurrent. When the value of a signal on the right hand side of an assignment changes, the entire right hand side waveform is evaluated, and the result is assigned to the left hand side target. For example, in Figure 7.5 if the value of *waveform* changes, the new value will be scheduled for assignment to *target1* after 5 ns. From time 0 to 85 ns, *waveform* changes between '1' and '0' a total of 18 times. Because this signal appears on the right hand sides of *target1*, *target2* and *target3* sig-

nals, each of these signal assignments becomes active 18 times in the 0 to 85 ns time interval.

As a result of the execution of a concurrent signal assignment, a value will be scheduled for the left hand side signal. When this happens, it is said that a value is placed on the *driver* of the left hand side signal.

Multiple concurrent assignments to a signal produce multiple drivers for that signal. A signal can only have multiple drivers if, along with the declaration of the signal, the name of a function for resolving a single value from multiple driving values is specified. In this text, resolution functions, multiple driving values of a signal, and the single resolved signal value are represented as shown in Figure 7.8. Signals with the declaration of which a resolution function is specified are said to be resolved signals. These signals, resolution functions, and issues regarding connection and disconnection of signal drivers will be discussed in Section 7.4.

Figure 7.8 Resolving a Single Value from Multiple Driving Values

A sequential body of VHDL that contains multiple assignments to a certain signal can only produce a single driver for that signal. All such signal assignments affect the same driver. Assignments to a signal in multiple sequential bodies are considered concurrent, and each produces a driver for that signal. In this case a resolution function is required for the signal.

7.3.2 Events and Transactions

Events and transactions are often referenced when discussing signal assignments. When a waveform causes the value of the target signal to change, an *event* is said to have occurred on the target signal. When a value is scheduled to be assigned to a target signal after a given time, a *transaction* is said to have been placed on the *driver* of the target signal. A transaction that does not change the value of a signal is still a transaction, but it does not cause an event on the signal. A transaction is represented by a value-time pair in parentheses.

The value is the current value if the time element is zero; otherwise, the value is the future value for the driver of the signal.

Execution of a signal assignment (sequential or concurrent) causes value v calculated from the right hand side and the delay d specified with the signal assignment to be placed on the driver of the left hand side signal as a (v, d) transaction. As shown in Figure 7.9, when placement of this transaction (tr_1) takes place at time t, during the current simulation time (now), the initial time component of the transaction becomes d. At a later time when the now time changes to $t+t0$, the time component of the tr_1 transaction becomes $d-t0$. As simulation progresses, the time component of this transaction approaches 0. When now becomes $t+d$, the time component of the tr_1 transaction reaches 0, and the transaction is said to be $expired$.

Transaction time component
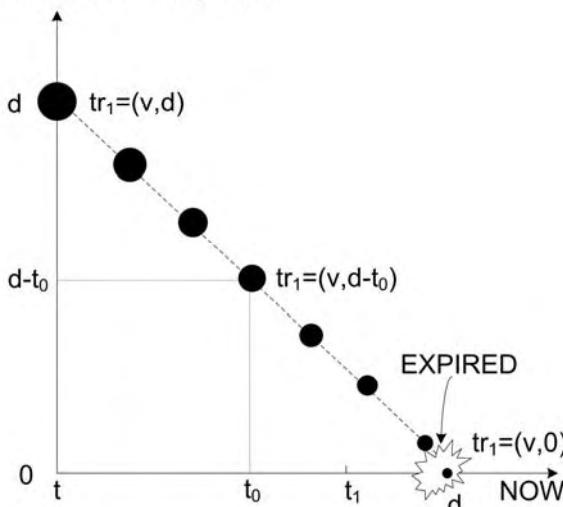
Figure 7.9 A Transaction, from Creation to Expiration

As an example, consider the description in Figure 7.10. The declaration of signals a, b, and c causes the creation of these signals with '0' initial values. The initial value of each of these signals appears as though it has been the value of the signal for an infinitely long time prior to the start of simulation. Creation of the signals is referred to as *elaboration*, and assigning their initial values to them is called *initialization*.

```
ARCHITECTURE demo OF example IS
   SIGNAL a, b, c : BIT := '0';
BEGIN
   a <= '1' AFTER 15 NS;
   b <= NOT a AFTER 5 NS;
   c <= a AFTER 10 NS;
END demo;
```
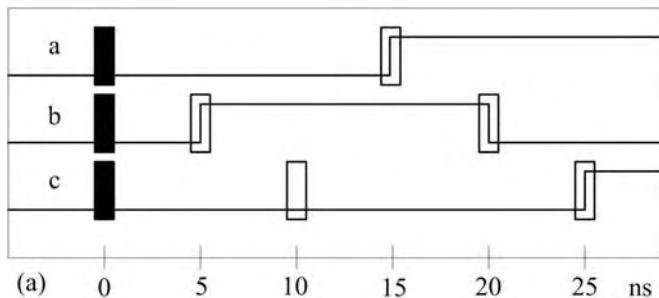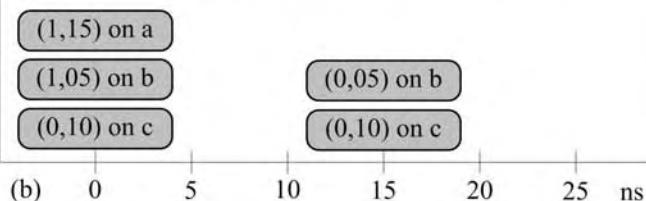
Figure 7.10 A Simple Description for Illustrating Events and Transactions

After the initialization phase, values of a, b, and c signals are all zero. At the start of the simulation, at time 0 (this and other time values in the discussion of this example are in nanoseconds), the value of '0' on a causes a ('1', 5 ns) transaction to be scheduled for b, and a ('0', 10 ns) transaction for c. Also at this time, a '1' is scheduled for the a signal after 15 ns, causing a ('1', 15 ns) transaction on the driver of this signal. At time 5, the time element of the scheduled transaction for b becomes zero, and its value becomes current (transaction expires), which causes the value of this signal to change from '0' to '1'. This change of value is an event on b. Five nanoseconds later, at time 10, the scheduled transaction on c becomes current (its time element becomes zero), causing a driving value of '0' on this signal. Since at this time the value of c is already '0', this transaction does not cause an event on c.
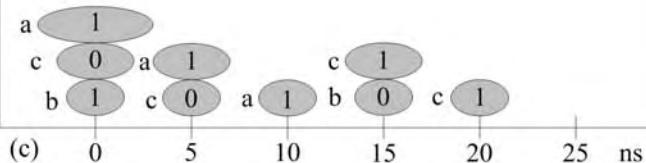
Figure 7.11 shows events and transactions on signals in the description in Figure 7.10. Hollow rectangles on waveforms signify transactions, and black rectangles signify initial values. Those transactions with a transition inside are considered events, e.g., b at 5 ns. Figure 7.11a shows the resulting timing diagram, with rectangles the transactions when they become current. Figure 7.11b shows the transactions that are placed on the a, b and c signals at the time this placement takes place. Figure 7.11c shows the transactions that exist on the signals before they become current. When a transaction is placed on the driver of a signal, it stays there and its time value decrease linearly with time until it becomes current. Figure 7.11c only shows transactions at 5 ns intervals. A transaction is represented by a circle; the size of the circle signifies the value of the time element of the transaction. Finally Figure 7.11d shows all transactions from when they come to exist to when they expire.
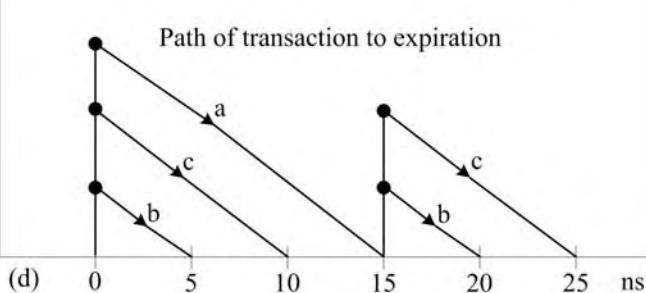
Figure 7.11   Events and Transactions that Occur on Signals in Figure 7.10. (a) The Resulting Timing Diagram Showing Transactions when they become Current;   (b) Transactions when they are Placed on Signals;   (c) Transactions as their Time Values Approach Zero to Become Current;   (d) Transactions from Creation to Expiration

No new transactions are placed on the signals in Figure 7.10 until time 15. At this time the transaction that was placed on the drive of *a* at time 0 becomes current and changes the value of this signal to '1'. This event causes a ('0', 5 ns) transaction on *b* and *a* ('1', 10 ns) transaction on *c*. When these transactions become current (i.e., their time elements become 0 at time 15+5 ns and 15+10 ns, respectively), the values of *b* and *c* are the opposite of the values of their corresponding transaction. Therefore, both transactions cause events on these signals, at time 20 on *b* and at time 25 on *c*.

In the example of Figure 7.10, we carefully avoided placing a transaction on the driver of a signal when there is still another pending transaction on that driver. If this happens, a decision has to be made as to whether both or only one of the transactions remain on the driver of the signals. Issues related to this matter will be treated in Section 7.3.4 when we discuss sequential placement of transactions on drivers of signals.

7.3.3 Delta Delay

In addition to real time delay, VHDL also defines a simulation cycle as an internal delay, referred to as *delta* delay (δ). The primary use of this delay is in modeling hardware concurrency. Consider the description of Figure 7.12 in which two concurrent signal assignments appear in the *delta* architecture of timing. When an event occurs on inputs *a* or *b*, *a* AND *b* is evaluated and scheduled to occur on the *z* output after 10 ns. The complement of *z* is then placed on the driver of *z_bar* with zero delay. Based on concurrency defined in the language, it is expected that *a* AND *b* and its complement appear on *z* and *z_bar* exactly at the same time.

```
ENTITY timing IS
   PORT (a, b : IN BIT; z, zbar : BUFFER BIT);
END ENTITY;
--
ARCHITECTURE delta of timing IS
BEGIN
   z_bar <= NOT z;
   z <= a AND b AFTER 10 NS;
END delta;
```

Figure 7.12 Demonstrating Need for Delta Delay

On the other hand, NOT *z* cannot be evaluated until *z* receives its new value. The new value on *z* causes an event on this signal, and this event causes the right hand side of *z_bar* to be evaluated and new value assigned to *z_bar*. Obviously, evaluation of the right hand

side of *z_bar* must wait for the new value of *z*. In order to hide this waiting, and make it appear as if both signals receive values at the same *real* time, VHDL introduces the concept of delta delay time.

The *z* signal receives its new value 10 ns after an event on *a* or *b*, and *z_bar* receives its new value a delta time after *z* receives its value (10 ns +δ). Because delta time is only an internal simulation cycle and does not contribute to real-time values, it appears as if *z* and *z_bar* have received their values at the same time (that is, the same *real* time).

In the example presented in Figure 7.13, nonzero delays were used. For the purpose of demonstrating the concept of delta time delay, we consider another version of the circuit description of Figure 7.1. The description in Figure 7.13 uses zero delay assignments for the internal nodes of the circuit and a delayed assignment, with accumulative worst-case delay of 12 ns, for the *w* output.

```
ENTITY mux IS
   PORT (a, b, s : IN BIT; w : OUT BIT);
END ENTITY;
--
ARCHITECTURE concurrent of mux IS
   SIGNAL i, j, k : BIT;
BEGIN
   i <= NOT s;
   j <= a AND i;
   k <= b AND s;
   w <= j OR k AFTER 12 NS;
END ARCHITECTURE concurrent;
```

Figure 7.13 VHDL Description for Demonstrating the Delta Delay

In our analysis of this description we use the same input values used in the analysis that led to the timing diagram in Figure 7.4. The new timing diagram, showing the events and transactions (vertical rectangles), is shown in Figure 7.14. As before, we assume that *a*, *b*, and *s* external signals are initialized to '1'; that is, their values prior to time zero and at time zero are '1'. We also assume that, external to this description, a '0' is assigned to the *s* signal at time zero. Since *s* is a signal, this new value appears on it a delta time later at time 1δ. One delta time after input *s* changes, nodes *i* and *k* receive their new values; that is, *i* becomes '1' (value of NOT *s*) and *k* becomes '0' (value of *a* AND *b*) at time 2δ. The event on *k* causes a zero value to be scheduled for output *w* after 12 ns, causing a ('0', 12 ns) transaction of driver of *w*. The event on *i* causes the expression for *j* to be evaluated, and as a results the value at node *j* changes one delta time after *i* changes; it changes from '0' to '1' at time 3δ. The event on *j* then

causes the output expression to be evaluated, which again results in scheduling a new value on the output 12 ns after this event, placing a ('1', 12 ns) transaction on the driver of *w*.
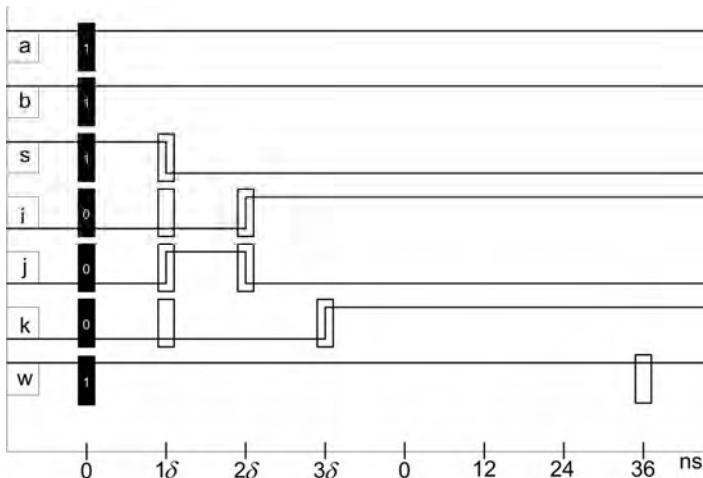
Figure 7.14 Timing Diagram for the Description of Figure 7.13, Showing Delta Delays

The second transaction on the driver of the *w* output, ('1', 12 ns), overwrites the first, ('0', 12 ns), and since the value of *w* is already a '1', the dominant transaction causes no event on this line. Although the steady state value on *w* is correct, the intermediate values on *w* are not modeled according to the actual circuit. Note that transactions on *k* and *j* at time zero do not translate to transactions on the *w* output 12 ns later. This is because the physical time delay absorbs all delta delays. Notice a glitch on *j* at zero time. The explanation of this glitch is left as an exercise.

Another example of delta time, transactions, and concurrency is shown in Figure 7.15. This description is for a chain of two zero-delay inverters, with *a* being the first input, *c* the output, and *b* the mid point. Signals *a*, *b*, and *c* are initialized to '0'. A '1' is assigned to *a*, causing a transaction on *b*, which in turn causes transaction on *c*.

```
ARCHITECTURE concurrent OF timing_demo IS
   SIGNAL  a, b, c : BIT := '0';
BEGIN
   a <= '1';
   b <= NOT a;
   c <= NOT b;
END concurrent;
```

Figure 7.15 Chain of Two Inverters, Delta Time, Transactions, and Concurrency

The timing diagram in Figure 7.16 indicates that all transactions occur at time zero between $0 + 1\delta$ and $0 + 3\delta$. Every transaction in this analysis causes an event to occur. At time zero, a, b, and c signals have values that are specified in the declarations of the signals. At this time, a '1' is scheduled for a, and the complement of a, whose value is still '0' at time zero, is scheduled for b; therefore, both a and b will have ('1', 0 ns) transactions on their drivers. Also at time zero, the complement of signal b, whose value is '0' at this time, is scheduled for the c signal, causing the placement of ('1', 0 ns) transaction on the driver of c. One delta time later at 1δ, signals a, b, and c receive their new values, which are all '1's. The new value on a causes a transaction on b one delta time later at 2δ, which results in an event that changes b to '0'. Similarly, the value of b at 1δ causes an event on c at 2δ. The event on b at time 2δ causes the right hand side of assignment to c to be evaluated, which causes another event on c one delta time later at 3δ.
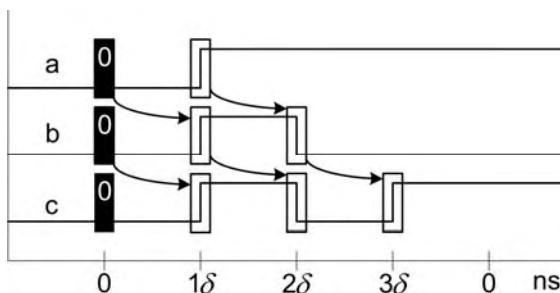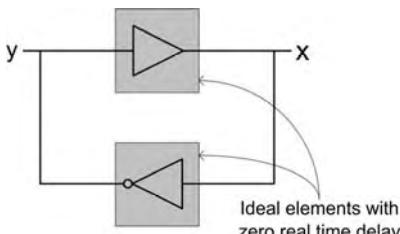
Figure 7.16 Timing Diagram for the *timing_demo* Description of Figure 7.15

As the last example for demonstration of delta delay, consider the circuit in Figure 7.17a and its corresponding VHDL description in Figure 7.17b. If the inverter and the buffer have zero delay values, this circuit should oscillate in zero time forever. The waveform extracted from simulation of the *oscillating* (*forever*) architecture (shown in Figure 7.17c) shows this oscillation. In zero real time (between 0 and 0 femtoseconds) values of x and y toggle for as long as the simulation run is allowed to continue. The simulation time does not advance beyond 0 fs.
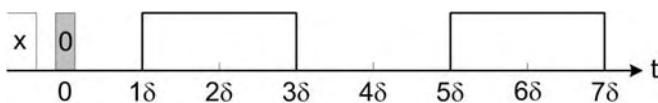
(a)

```
ARCHITECTURE forever OF oscillating IS
   SIGNAL  x: BIT := '0';
   SIGNAL  y: BIT := '1';
BEGIN
   x <= y;
   y <= NOT x;
END forever;
```

(b)

(c)

Figure 7.17 Oscillation in Zero Real Time. (a) Circuit to Model; (b) VHDL Representation; (c) Signal Waveforms

7.3.4 Sequential Placement of Transactions

Assignments to signals in the sequential bodies of VHDL, for example, in the body of process statements, are done sequentially. This means that the order in which signal assignments appear is important, and it is legal to make multiple assignments to simple nonresolved signals. Multiple sequential assignments cause new transactions to be placed on the driver of a signal. New transactions may be placed while there are transactions on the driver of the signal that are still not expired. When a new transaction is to be placed on the driver of a signal, the transactions that are already scheduled for that signal will then be considered. The new transaction will either overwrite the previous transactions or be appended to them, depending on the timing of the new transaction and the type of the assignment.

As an example for sequential placement of transactions in sequential bodies of VHDL, consider the code shown in Figure 7.18. The first assignment to *x* places the (*v1*, *t1*) transaction on the driver of *x*. Immediately following this assignment a new transaction (*v2*, *t2*) is to be placed on the driver of *x*. This new transaction may overwrite the existing transaction or may be appended to it in the driver of *x*. Conditions under which appending or overwriting transactions occur will be explained later in this section.

The WAIT statement at the end of the process statement causes the process to suspend forever, allowing enough time for all pending transactions to expire.

```
ARCHITECTURE sequential OF sequential_placement IS
    . . .
BEGIN
   PROCESS
      x <= v1 AFTER t1;
      x <= v2 AFTER t2;
      WAIT;
   END PROCESS;
END sequential;
```

Figure 7.18 Sequential Placement of Transactions in a Sequential Body of VHDL

Sequential placement of transactions on the driver of signals can also be done in concurrent VHDL bodies. Consider, for example, the code shown in Figure 7.19. Initially, (*v1*, 0) is placed on the driver of *a*, and immediately after that the (*v2*, *t1*) transaction is appended to it. Assuming initial value of *a* is not *v1*, signal *a* changes at time 0 to *v1* and then to *v2* at time *t1*.

```
ARCHITECTURE concurrent OF sequential_placement IS
   . . .
BEGIN
   a <= v1, v2 AFTER t1;
   x <= a AFTER t2;
END concurrent;
```

Figure 7.19 Sequential Placement of Transactions in a Concurrent Body of VHDL

At time 0, when an event occurs on *a* that is on the right hand side of the assignment to *x*, the (*v1*, *t2*) transaction (*v1* being the value of *a* at this time) will be placed on the driver of *x*. At time *t1*, the time component of this transaction changes to *t1-t2*, making the driver of *x* contain the (*v1*, *t1-t2*) transaction. Meanwhile, at this time, signal *a* changes to *v2*, causing another event on the right hand side of *x*. This event causes (*v2*, *t2*) to be considered for appending or

overwriting the existing (*v1*, *t1-t2*) transaction on *x*. Appending or overwriting depends on the values of *v1*, *v2*, *t1*, and *t2* and the delay mechanisms.

7.3.4.1 Signal Drivers. A signal has a driving value, and there may be several pending transactions on this signal waiting to become current. When a transaction becomes current, its value becomes the driving value of the signal. A driver for a scalar signal is represented by a *projected output waveform*. We will represent a projected output waveform as a queue of transactions. The tip of the queue is the expired transaction, which constitutes the driving value of the signal. Figure 7.20 shows this representation.
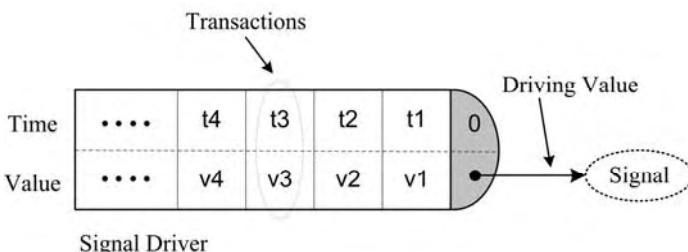
Figure 7.20 Projected Output Waveform

Transactions are queued in the order of their time components. In the figure shown, *t1<t2<t3 …* . As time advances, all transaction time components (t_i) are decremented. When the time component of a certain transaction reaches 0, all transactions move forward in the queue, and the one with the 0 time component replaces the driving value of the signal.
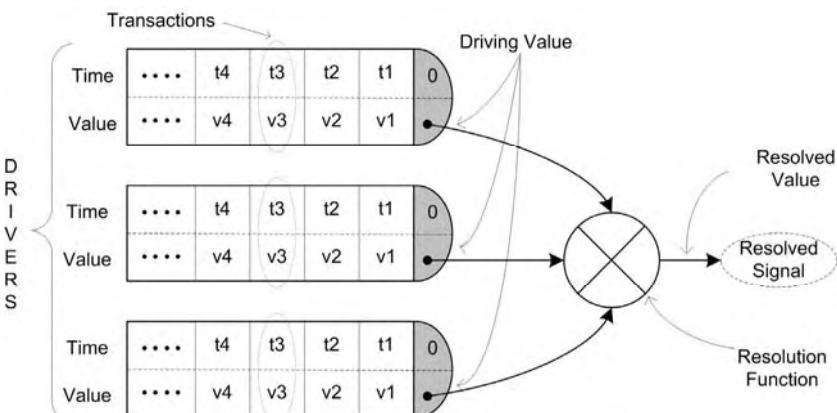
Figure 7.21 Multiple Drivers of a Resolved Signal
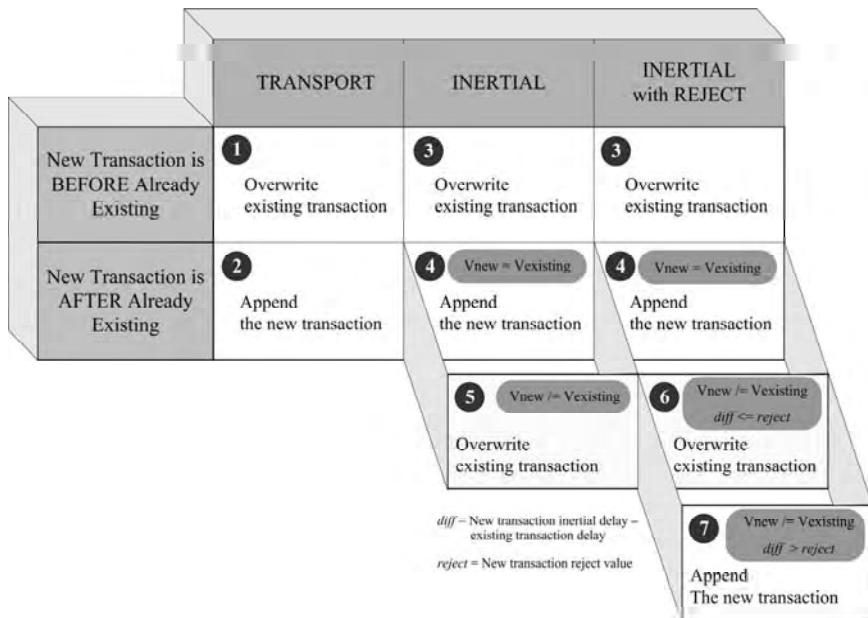
A process statement (such as that shown in Figure 7.18) or a concurrent signal assignment (such as that shown in Figure 7.19) creates only one driver for a signal. Assignments to a signal in multiple concurrent bodies create multiple drivers for the signal. Such a signal must be resolved, and a resolution function must exist to resolve a value from multiple driving values. This situation is represented by multiple projected output waveforms in Figure 7.21.

The discussion of this section focuses on a single driver signal, as shown in Figure 7.20. The multiple driver case depicted in Figure 7.21 will be discussed in the next section.

7.3.4.2 Transaction Appending Rules. An incoming *new* transaction is characterized by its value, time, and delay mechanism. Transactions on the driver of a signal are characterized by their value and time only. A new transaction is compared with those already on the driver of a signal (referred to as *existing*). Value and time of the transactions and the delay mechanism of the incoming transaction decide on the resulting transactions in the projected output waveform of a signal.

A new transaction on the driver of a signal scheduled *before* an already existing transaction always overwrites the existing transaction. A new transaction on the driver of a signal scheduled *after* an already *existing* transaction is appended to the existing transaction if the delay is of the transport type.

For the inertial delays, inertial with reject and simple inertial are handled slightly different and are discussed separately. For the simple inertial, new transaction scheduled after the existing transaction appends to the existing transaction unless the values of the transactions are different and the difference between the time of the new transaction and the existing transaction is less than the inertial delay value. In this case, the new transaction overwrites the existing transaction. For the inertial delay with a reject value, the new transaction scheduled after the existing transaction appends to the existing transaction unless the values of transactions are different and the difference between the time of the new transaction and the existing one is less than *or equal to* the inertial reject value. In this latter case, the new transaction overwrites the existing transaction. Figure 7.22 summarizes this discussion of the resulting transaction on the driver of a signal. Various situations of overwriting or appending transactions will be illustrated in several examples.

Figure 7.22 Effective Transactions on the Driver of a Signal when Multiple Transactions Are Sequentially Placed on the Signal Driver
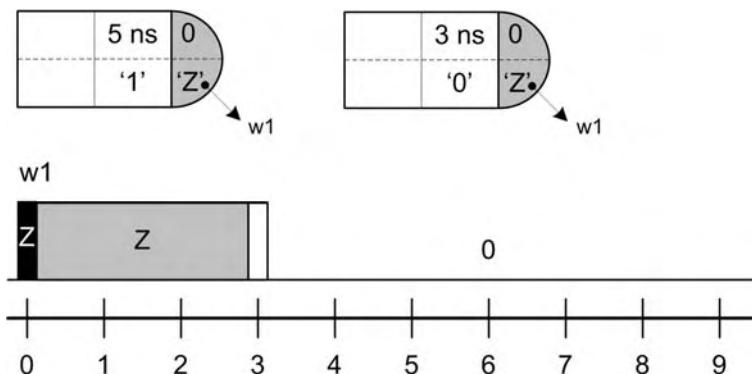
In the examples that follow, we use a process statement to place a transaction on the driver of a signal. In this same process, another signal assignment with a certain delay mechanism places a second transaction on the driver of this signal. A WAIT statement at the end of the process statement suspends the process and stops further placement of transactions on the signal. The signal used for this study is of type *v4l*, which can take 'X', '0', '1' or 'Z' values.

Case 1: When a sequential signal assignment statement is executed, a (*v1*, *t1*) transaction will be placed on the driver of the target signal. If another signal assignment with a transport delay mechanism causes a second transaction (*v2*, *t2*) to be placed on the driver of the same signal, the first transaction will be overwritten by the second transaction if *t2* is less than *t1*. For example, in the process of Figure 7.23, the ('1', 5 ns) transaction is discarded, resulting in the timing diagram shown in this figure. This figure also shows the driver of *w* before and after the placement of ('0', 3 ns) transaction. The initial value of *w1* is 'Z'. At time 3 ns, the ('0', 3 ns) transaction expires, which causes the driving value of *w1* to become '0'. This example illustrates box 1 of Figure 7.22.

```
case1: PROCESS BEGIN -- Transport, Before
   w1 <= '1' AFTER 5 NS;
   w1 <= TRANSPORT '0' AFTER 3 NS;
   WAIT;
END PROCESS case1; -- Overwrites existing
```
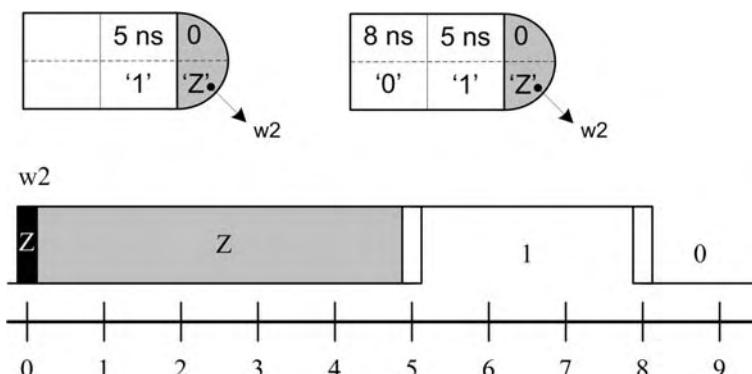

Figure 7.23 Transport Delay, Before Existing Transactions

Case 2: On the other hand, if *t2* of the new transaction is greater than *t1* of the existing transaction and transport delay mechanism is used for placement of the new transaction, the situation shown in box 2 of Figure 7.22 occurs. In this case the new transaction will be appended to the existing one.

```
case2: PROCESS BEGIN -- Transport, After
   w2 <= '1' AFTER 5 NS;
   w2 <= TRANSPORT '0' AFTER 8 NS;
   WAIT;
END PROCESS case2; -- Appends to existing
```


Figure 7.24 Transport Delay After Existing Transaction

Figure 7.24 shows an example for this case. The ('1', 5 ns) transaction remains on the driver of *w2* and expires at time 5 ns. The second transaction, ('0', 8 ns), expires 3 ns later, causing the value of *w* to change to '0'. Figure 7.24 also shows the driver of *w2* before and after placement of ('0', 8 ns).

Case 3: Figure 7.25 is used to demonstrate the interaction of transactions when the new transaction, i.e., (*v2*, *t2*), is being placed inertially, and its time component, (*t2*), is less than the time component, (*t1*), of the existing transaction, i.e., (*v1*, *t1*). This case is represented in boxes 3a and 3b of Figure 7.22. For the inertial and inertial with reject delay mechanism the second inertially delayed assignment in Figure 7.25 sequentially places the ('0', 3 ns) transaction on the driver of *w3a* or *w3b*. This new transaction overwrites the existing ('1', 5 ns) transaction. The waveform in this figure shows ('0', 3 ns) expiring at 3 ns, causing the current driving value of *w3a* or *w3b* to become '0' at this time. The initial value of *w3a* or *w3b* is assumed to be 'Z'.

```
case3a: PROCESS BEGIN -- Inertial, Before
   w3a <= '1' AFTER 5 NS;
   w3a <= INERTIAL '0' AFTER 3 NS;
   WAIT;
END PROCESS case3a; -- Overwrites existing
--
case3b: PROCESS BEGIN -- Reject, Before
   w3b <= '1' AFTER 5 NS;
   w3b <= REJECT 3 NS INERTIAL '0' AFTER 3 NS;
   WAIT;
END PROCESS case3b; -- Overwrites existing
```
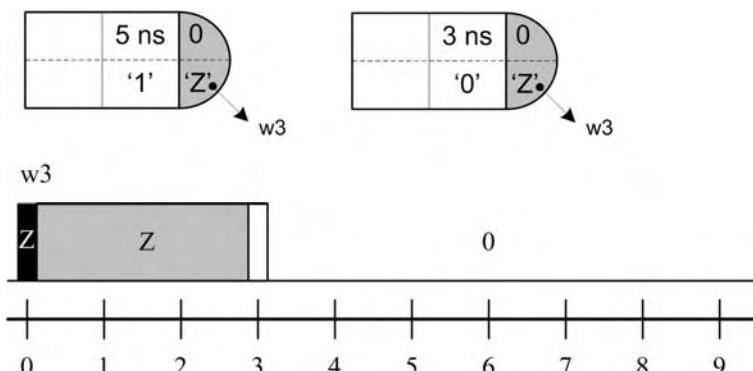

Figure 7.25 Inertial or Inertial with Reject, Before Existing

Case 4: If the new transaction is being placed inertially (with or without reject), values of new and existing transactions play a role in deciding whether the new transaction appends or overwrites the existing one. If *v2* and *v1* (values of existing and new transactions, respectively) are equal, the new inertially placed transaction appends to the already existing one. This situation is represented in boxes 4a or 4b of Figure 7.22. Example processes and their corresponding timing diagram are shown in Figure 7.26. As shown, the existing ('0', 5 ns) and the new ('0', 8 ns) transactions expire and become driving values for *w4a* or *w4b*.

```
case4a: PROCESS BEGIN -- Inertial, After, Vn=Ve
   w4a <= '0' AFTER 5 NS;
   w4a <= INERTIAL '0' AFTER 8 NS;
   WAIT;
END PROCESS case4a; -- Appends to existing
--
case4b: PROCESS BEGIN -- Reject, After, Vn=Ve
   w4b <= '0' AFTER 5 NS;
   w4b <= REJECT 8 NS INERTIAL '0' AFTER 8 NS;
   WAIT;
END PROCESS case4b; -- Appends to existing
```
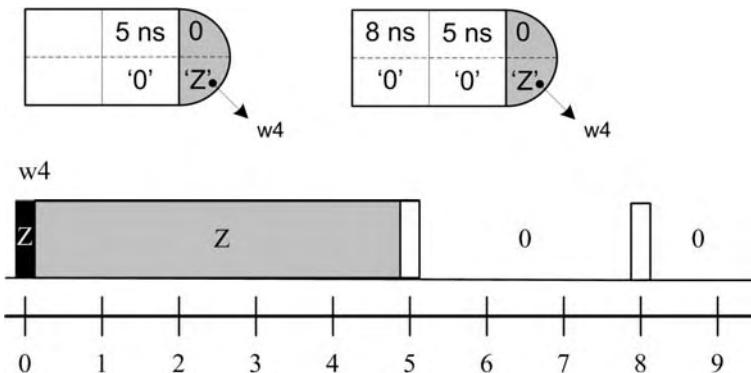

Figure 7.26 Inertial or Inertial with Reject, Same Value Transaction After Existing

Shown in Figure 7.26, when *t2* is larger than *t1* and *v2 = v1*, specifying a reject value that is different from the default inertial delay does not influence the final projected output waveform of the signal being assigned.

Case 5: Our next example concerns an inertial delay mechanism for which the new transaction is after the existing, and unlike case 4, the values of the new transaction and the existing one are different. As
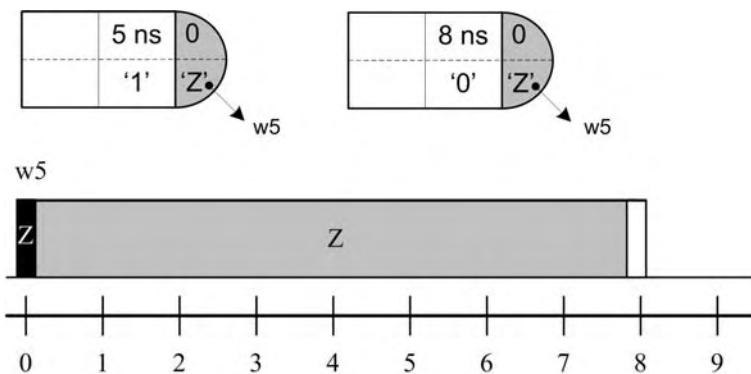
illustrated in Figure 7.27, the existing transaction value is '1', and that of the new one is '0'. In this case, the new transaction overwrites the existing, causing $w5$ to remain 'Z' until 8 ns that the ('0', 8 ns) expires. When this happens, $w5$ becomes '0' at 8 ns.

The initial value of $w5$ of Figure 7.27 is 'Z'. The pending transaction ('1', 5 ns) is placed on $w5$, but is later overwritten by ('0', 8 ns). The final value of $w5$ becomes '0' at 8 ns.

```
case5: PROCESS BEGIN -- Inertial, After, Vn/=Ve
   w5 <= '1' AFTER 5 NS;
   w5 <= INERTIAL '0' AFTER 8 NS;
   WAIT;
END PROCESS case5; -- Overwrites existing
```


Figure 7.27 Inertial Delay, Different Value Transaction After Existing

Case 6: The next case deals with the inertial delay mechanism with a reject value. As in case 5, the values of the existing and new transaction are different. In this case (as shown in Figure 7.28), the difference between the time of the new transaction and that of the existing one is calculated. If this difference is less than or equal to the reject delay value, e.g., 8 ns – 5 ns < 4 ns, then the new transaction overwrites the existing one.

As shown in Figure 7.28, this case produces the same result as in case 5, in which the reject value is considered to be the same as the inertial delay value.

Case7: Our final example illustrates the case of a new transaction with a different value than the existing one, with a reject value that is less than the difference of timing of the new and existing transactions. This situation is similar to the previous one (case 6), except for the reject value. As shown in Figure 7.29, a new transaction of the type discussed appends to the existing transaction of the left hand

side signal. As shown in this figure, the difference between the time of the new transaction and that of the existing, i.e., 8 ns − 5 ns, is larger than the reject value that is 2 ns.

```
case6: PROCESS BEGIN
   -- Reject, After, Vn/=Ve, Diff <= Reject
   w6 <= '1' AFTER 5 NS;
   w6 <= REJECT 4 NS INERTIAL '0' AFTER 8 NS;
   WAIT;
END PROCESS case6; -- Overwrites existing
```
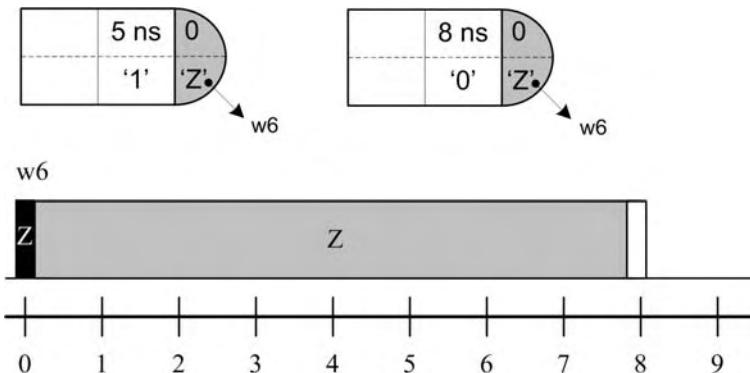

Figure 7.28 Inertial with Reject, Different Values, After Existing, Reject Occurs

```
case7: PROCESS BEGIN
   -- Reject, After, Vn/=Ve, Diff > Reject
   w7 <= '1' AFTER 5 NS;
   w7 <= REJECT 2 NS INERTIAL '0' AFTER 8 NS;
   WAIT;
END PROCESS case7; -- Appends to existing
```
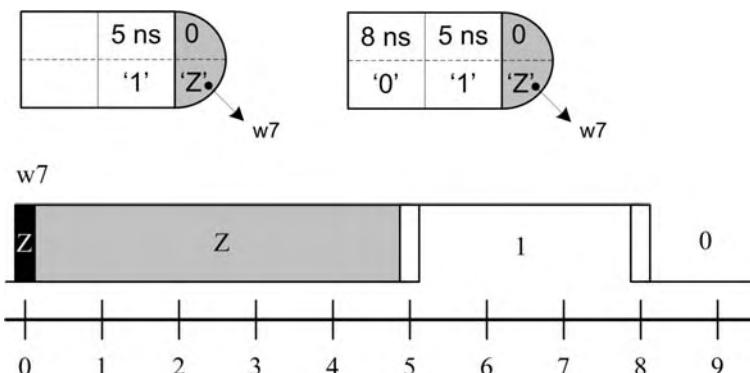

Figure 7.29 Inertial with Reject, Different Values, After Existing, Reject Doesn't Occur

7.3.4.3 Pulse Rejection. Section 7.2 presented pulse rejection and delay modeling at an abstract level. Based on the delay model and pulse width, we discussed conditions under which a pulse is delayed or removed. Waveforms in Figure 7.5 presented several examples for various delay mechanisms. At a lower abstraction level, pulse delays and rejections are explained in terms of sequential placement of transactions on signal drivers. To demonstrate this, we consider transactions and events that occur on the three target signals in Figure 7.5 between times 0 and 40 ns. Assignments for producing events in this time interval and resulting waveforms are repeated here in Figure 7.30. This figure also shows transactions and events for the target signals.
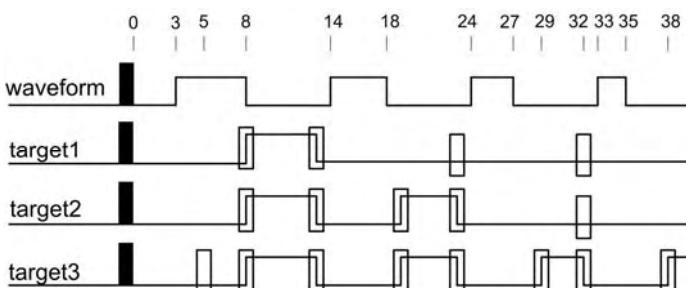
Figure 7.30 Pulse Rejection in Inertial, Reject, and Transport Delay

Signals *waveform*, *target1*, *target2*, and *target3* are all initially zero. At time 0, all three assignments to target signals are executed, which cause ('0', 5 ns) transactions to be placed on the drivers of all three *target* signals. Note that the value of this transaction is the initial value of *waveform*. Transactions and activities on *target* signals between times 0 and 40 ns are shown in Figure 7.31. At time 3 ns the initial transactions on *target1*, *target2* and *target3* have changed to ('0', 2 ns). At this time, waveform changes from '0' to '1', causing the placement of the ('1', 5 ns) transaction on the driver of target signals. This new transaction overwrite the ('0', 2 ns) transactions on *target1* and *target2*, according to the rules shown in boxes 5 and 6 in Figure 7.27. On the other hand, the ('1', 5 ns) transaction appends to the existing ('0', 2 ns) transaction on driver of *target3*. This is according to the rule specified in box 2 in Figure 7.22. Two nanoseconds later, at time 5 ns, the time component of ('0', 2 ns) on *target3* is reduced to 0. The shaded area on *target3* at time 5 ns (Figure 7.30) shows expiration of this transaction. At 8 ns, the ('1', 5 ns) transactions placed on *target* signals at 3 ns expire, causing the value of these signals to change to '1'. Also at this time, *waveform* changes to '0', causing ('0', 5 ns) transaction to be placed on the drivers of the three *target* signals. When these transactions expire, the values of *targets* change to '0' at

time 13 ns. The result is that the 5 ns pulse that started on *waveform* at time 3 ns appears on all *target* signals with 5 ns of delay.
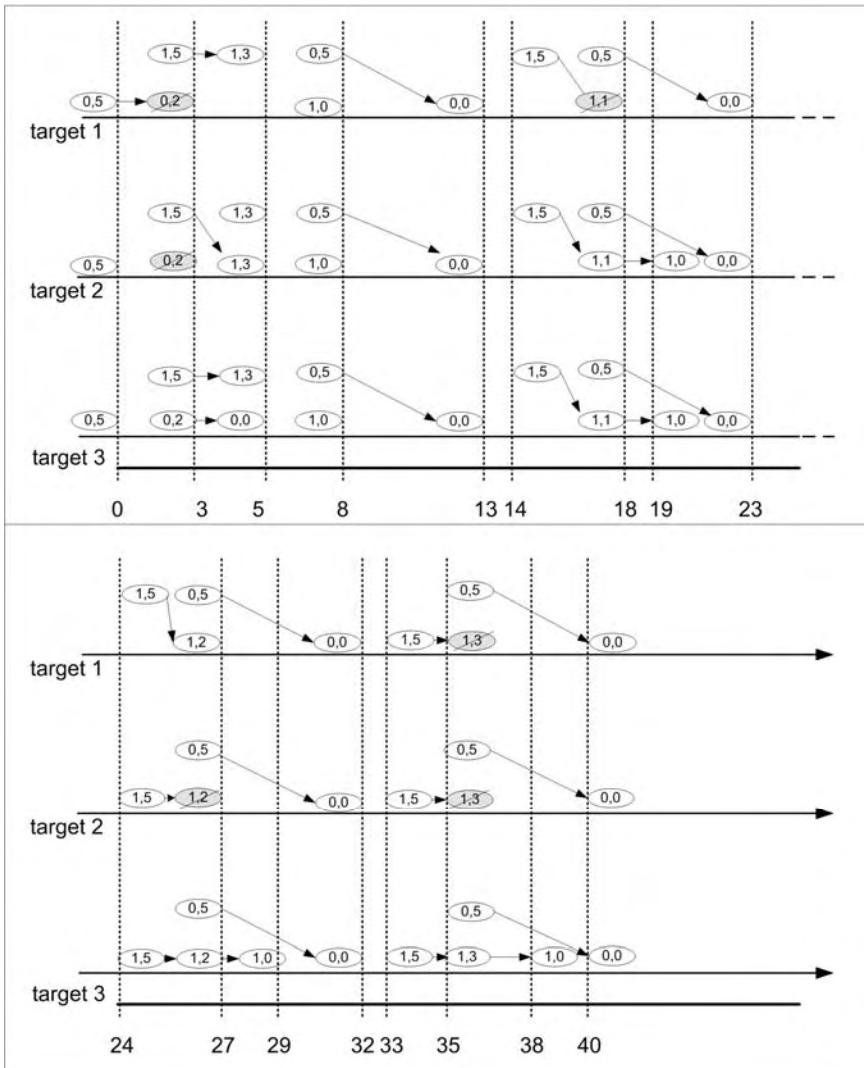
Figure 7.31 New, Pending, and Expired Transactions on the Targets of Figure 7.30

A second pulse starts at time 14 ns on *waveform*. At this time, the ('1', 5 ns) transaction is placed on the drivers of *target* signals. At time 18 ns (4 ns after placement of the original transactions), time components of existing transactions on *target* signals are reduced to 1 ns, making them ('1', 1 ns). At this time, '1' to '0' event on *waveform* causes ('0', 5 ns) transactions on drivers of *target* signals. This new

transaction overwrites the existing transaction on *target1* according to box 5 of Figure 7.22 (5 ns - 1 ns is less than 5 ns, the inertial delay value of *target1* assignment). The placement of this transaction on *target2* and *target3* follows rules specified in boxes 7 and 2 of Figure 7.22, respectively, which result in appending this transaction to the existing one for each signal. Existing transactions on *target2* and *target3*, i.e., ('1', 1 ns) and ('0', 5 ns), expire at time 19 ns and 23 ns, respectively, completing positive pulses on these signals. On the other hand, only one driver is left on the drive of *target1*, which expires at time 23 ns, causing no change in the value of this signal.

A third pulse 3 ns long appears on *waveform* starting at time 24 ns. Figure 7.31 shows transactions that are placed on *target* signals of Figure 7.30. At time 27 ns, ('0', 5 ns) transactions overwrites existing transactions on *target1* and *target2* according to boxes 5 and 6 of Figure 7.22. On the other hand, according to box 2 of this figure, the ('0', 5 ns) transaction appends to the existing transaction on the driver of *target3* at time 27 ns. Other activities caused by pulses on *waveform* up to 40 ns are shown in Figure 7.31. The resulting waveforms on *target* signals are shown in Figure 7.30.

7.3.4.4 Placing Waveform Elements.   Before we end this topic, we present another example to focus on two important issues related to the sequential placement of transactions. The First issue is that, as a result of an event on the right hand side of a concurrent signal assignment all right hand side waveform elements will be sequentially scheduled on the left hand side signal. The second issue is that the specified delay mechanism only applies to the first waveform element, and all other waveform elements are treated as transport. Activities on signals *a* and *b* are shown in Figure 7.32. At time 0, the ('1', 5 ns) transaction is to be placed on the driver of *a* with an inertial mechanism, while ('0', 10 ns) is to be placed on the driver of this signal with a transport delay mechanism. Because ('0', 10 ns) will be appended to ('1', 5 ns), a positive pulse appears on *a*. Every time an event occurs on *a*, the ('0', 0 ns) transaction with inertial mechanism and the (*a*, 3 ns) transaction with transport mechanism are sequentially placed on the driver of *b*.

Consider for example time 10 ns when *a* makes a '1' to '0' transition. The value of signal *b* at this time is '1'. As the result of the event on *a*, ('0', 0 ns) and ('0', 3 ns) are placed on the driver of *b*. Placement of ('0', 0 ns) follows the overwriting rule specified in box 3 of Figure 7.22 for an inertially placed transaction. Placement of ('0', 3 ns) follows the rule specified in box 2 of Figure 7.22 for a transport delay mechanism. At time 13 ns, the last transaction on the driver of *b* expires with no change in the value of signal.

```
ARCHITECTURE delay OF example IS
   SIGNAL a, b, BIT := '0';
BEGIN
   a <= '1' AFTER 5 NS, '0' AFTER 10 NS, '1' AFTER 15 NS;
   b <= '0', a AFTER 3 NS;
END ARCHITECTURE delay;
```
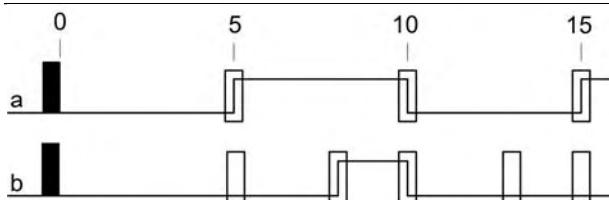

Figure 7.32 Sequential Placement of Transactions by Concurrent Assignments

7.4   Multiple Concurrent Drivers

The discussion of the previous section concentrated on a single driver
of a signal and discussed how various transactions interact to produce
a signal value. In situations that a signal appears on the left hand
sides of multiple concurrent assignments, or multiple process assign
values to the signal, multiple drivers will be created for the signal.
While placement of values in each driver remains as discussed in Sec-
tion 7.3, the final value of the signal will be determined by all signal
drivers. Figure 7.21 shows how multiple driving values are resolved
to produce a value for a signal with multiple assignments.

7.4.1   Resolving between Multiple Driving Values

Figure 7.33 shows a multiplexer circuit that uses NMOS pass transis-
tors to select either of the a, or b inputs. The transistors act as unidi-
rectional pass gates that cause node y to be driven by a or b depend-
ing on the value of s.
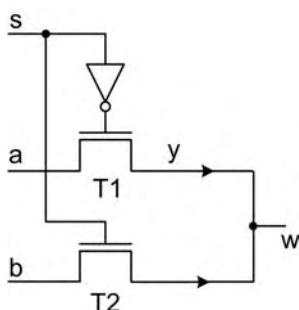
Figure 7.33 Pass Transistor Based Multiplexer

Figure 7.34 shows a VHDL description that is intended to model the circuit Figure 7.33. Type of signals, and in particular, the type of *y*, is *v4l* which is the four-value Verilog equivalent type discussed in Chapter 6. Corresponding to each transistor in Figure 7.33, there is a signal assignment that either puts the value of an input (*a* or *b*), or puts 'Z' on *y*. This means that each signal assignment continuously drives its left hand side with a value. Since the two signal assignments are concurrent, *y* has two drivers at all times. If *s* is '0', the *t1* assignment drives *y* with the value of *a*, and the *t2* assignment drives *y* with 'Z'. The description shown in this figure does not specify how these two values are used to form the final value of *y*. This description does not compile and produces an error indicating that *y* cannot have two drivers.

```
ENTITY multiplexer IS
   PORT (a, b, s : IN v4l; w : OUT v4l);
END ENTITY;
--
-- Does not compile
ARCHITECTURE wired OF multiplexer IS
   SIGNAL y : wiring v4l;
BEGIN
   T1: y <= a WHEN s='0' ELSE 'Z';
   T2: y <= b WHEN s='1' ELSE 'Z';
   w <= y;
END ARCHITECTURE wired;
```

Figure 7.34 Multiplexer Circuit, Two Concurrent Assignments (Does Not Compile)

In order to remedy this problem, we have to do in VHDL what is being done in the actual circuit of Figure 7.33. In the transistor circuit, since only one transistor conducts at any one time, one side of the wiring at *y* is always 'Z'. An actual value ('0' or '1') coming from the conducting transistor overrides the 'Z' value coming from the other side, and causes *y* to take the value of data input of the conducting transistor. In other words, node *y* has a resolution that resolves between multiple driving values.

7.4.1.1 Resolving a Pair of Values. In order to define a resolution in VHDL, we first have to decide how two driving values are to resolve. For example, corresponding with the wiring of Figure 7.33, '0' and 'Z' resolve to '0', and '1' and 'Z' resolve to '1'. Figure 7.35 shows a wiring function that specifies resolutions for every pair of values of the *v4l* type.

```
FUNCTION wire (a, b : v4l) RETURN v4l IS
   CONSTANT v4l_wire_table : v4l_2d := (
    'X' => ('X', 'X', 'X', 'X'),
    '0' => ('X', '0', 'X', '0'),
    '1' => ('X', 'X', '1', '1'),
    'Z' => ('X', '0', '1', 'Z'));
BEGIN
   RETURN v4l_wire_table (a, b)
END wire;
```

Figure 7.35 Resolving Every Pair of Values of *v4l* Type

The *wire* function specifies resolutions of values of the *v4l* type in a tabular fashion. The type of *v4l_wire_table* constant is *v4l_2d* that was discussed in Chapter 6. The function return value is decided by a lookup in this table. As shown, for example, a '0' and '1' resolve to 'X' indicating a short in hardware.

7.4.1.2 Resolving Multiple Driving Values. In order to be able to apply the wire resolution of Figure 7.35 to node *y* of the VHDL code of Figure 7.34, we have to form a function that applies the wire function to all possible drivers of *y*. This means that, although *y* in this example has only two drivers, the resolution of multiple driving values has to allow for any number of drivers, 0 to n.

The *wiring* function shown in Figure 7.36 is a resolution function that applies the *wire* function of Figure 7.35 to every pair of values in the *drivers* input of this function. The *drivers* input of *wiring* is of the vector type of its *v4l* return value. As shown in Figure 7.36, the for-loop in the body of the *wiring* function uses the *accumulate* variable to accumulate the *wire* result of all *drivers* elements. If *drivers* has a null range, i.e., an array of no elements, the default value of accumulate that is 'Z' will be returned.

```
FUNCTION wiring ( drivers : v4l_vector) RETURN v4l IS
   VARIABLE accumulate : v4l := 'Z';
BEGIN
   FOR i IN drivers'RANGE LOOP
      accumulate := wire (accumulate, drivers(i));
   END LOOP;
   RETURN accumulate;
END wiring;
```

Figure 7.36 Wiring Resolution Function, an Array Version of *Wire*

7.4.1.3 Resolution Function. The *wiring* function is a resolution function that can be applied to signal *y* of Figure 7.34. Before we show how this is done, there are certain properties of resolution functions

that must be discussed. A resolution function has a return type and an argument that is an unconstrained array of its return type. Recall from our discussion in Chapter 6 that *v4l_vector* was defined in our *VerilogLogic* package as an unconstrained array of *v4l*. Furthermore, in order for a function to qualify as a resolution function, the array type argument must be its only argument.

A function that does not read a global variable is called a PURE function. VHDL requires that a function that is to be used as a resolution function must be *pure*. Therefore, shared variables, variables visible in a declarative part that a resolution function is placed, and external file data cannot be read in a resolution function.

The vector argument of a resolution function allows any number of drivers including zero. A resolution function should provide a default value that will be returned if the resolution function is called with null drivers. Having null drivers occurs if all drivers of a signal are disconnected. We will describe driver disconnection in a later section in this chapter. The value returned by our *wiring* resolution function if it is called with null drivers is 'Z'.

The format of the resolution function of Figure 7.36 can be used for developing many gate level resolution functions like wired-AND and wired-OR. In such functions, the corresponding logic function must be applied to all bits of the drivers input of the function. VHDL, however, does not limit resolution functions to gate level applications. A resolved type of a record, or any complex type, can be defined in VHDL. Furthermore, the body of a resolution function is not limited to performing a uniform operation on all drivers like that shown for the *wiring* function or what should be done for implementing a wired-AND or wired-OR function. A resolution can be a voter for reliability applications, an accumulative adder for load calculations, or it can describe a routing algorithm for a network or NoC (Network on a Chip) applications.

Another property of resolution functions that is important to mention is the ordering and index of the active drivers of a resolution function in its array argument. The VHDL language guarantees that all active drivers of a resolution function form a contiguous array that becomes the argument of the resolution function. The order of these elements, their range, and starting value of their index are not known and not specified in the VHDL language.

7.4.1.4 Applying a Resolution Function. A function written as specified above qualifies as a resolution function. Such a function can be applied to any signal whose type is the same as the return type of the resolution function. Figure 7.37 shows he *wired* architecture of the *multiplexer* that compiles and implements a 2-to-1 one-bit multiplexer. As shown in this figure, signal *y* is declared as *wiring v4l*. The

use of *wiring* in this declaration refers to the wiring resolution function of Figure 7.36. This function must be made visible to this architecture in order to be used in its declarations. This visibility requirement can be achieved by placing the *wiring* function in a package, or directly using the function in the architecture declarative part. We assume that this function is in the *VerilogLogic* package that is used in this architecture.

The type of *y* in the architecture of Figure 7.37 is defined as a subtype of *v4l*. The use of *wiring* before the actual type mark (*v4l*) specifies this subtype. The assignment of *y* to *w* is easily done, since the base of *y* is *v4l* that is the same as that of *w*. A signal defined as such is referred to as a resolved signal. We say that *y* is a resolved signal using the *wiring* resolution function.

```
ARCHITECTURE wired OF multiplexer IS
   SIGNAL y : wiring v4l;
BEGIN
   T1: y <= a WHEN s='0' ELSE 'Z';
   T2: y <= b WHEN s='1' ELSE 'Z';
   w <= y;
END ARCHITECTURE wired;
```

Figure 7.37 Working Architecture for Multiplexer

7.4.1.5 Resolution Package. The above example showed how a one-bit signal could be declared as having a certain resolution function. In order to be able to declare multi-bit resolved busses, a vector type having resolved element types must be defined. Figure 7.38 shows declarations necessary for defining resolved signals and busses.

```
FUNCTION wiring ( drivers : v4l_vector) RETURN v4l;
SUBTYPE wired_v4l IS wiring v4l;
TYPE wired_v4l_vector IS
                 ARRAY (NATURAL RANGE <>) OF wired_v4l;
```

Figure 7.38 Resolution Related Declarations

Declarations shown in Figure 7.38 must be placed in a package to be used by designers wanting to use the *wiring* resolution function. The first declaration is the declaration of the resolution function itself. The second declaration specifies *wired_v4l* as a *wiring* subtype of *v4l*. This subtype indication can be used for declaring scalar signals. For example in Figure 7.37 instead of using *wiring v4l* for declaring *y*, we could use *wired_4vl*. The third declaration in Figure 7.38 declares an array type of resolved elements. The *wired_v4l_vector* is an uncon-

strained array whose elements are of *wired_4vl* type. Since *wired_4vl* is a resolved type, *wired_4vl_vector* becomes the type mark for declaring resolved busses.

Figure 7.39 shows an n-bit 4-to-1 multiplexer that is described by four concurrent three-state bus connections. All signal types in this description are of the resolved array type, *wired_v4l_vector*. This description assumes that the wire function of Figure 7.35, the *wiring* resolution function of Figure 7.36, and the declarations of Figure 7.38 exist in the *VerilogLogic* package, and this package is compiled in the *utilities* library.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY multiplexer_n4 IS
   PORT (a, b, c, d : IN wired_v4l_vector;
         s : IN wired_v4l_vector (1 DOWNTO 0);
         w : OUT wired_v4l_vector);

END ENTITY;
--
ARCHITECTURE wired OF multiplexer_n4 IS
BEGIN -- Four Bus Connections

   BC1: w <= a WHEN s="00" ELSE
            (a'RANGE => 'Z');
   BC2: w <= b WHEN s="01" ELSE
            (b'RANGE => 'Z');
   BC3: w <= c WHEN s="10" ELSE
            (c'RANGE => 'Z');
   BC4: w <= d WHEN s="11" ELSE
            (d'RANGE => 'Z');

END ARCHITECTURE wired;
```

Figure 7.39 Using Resolved Multi-bit Busses

7.4.1.6 A Resolution Package. As discussed above, several types and functions are needed for expressing a resolution function. We discuss a wired-OR resolution package to further clarify topics discussed in the previous sub-sections.

For a wired-OR resolution, the function that applies to every pair of drivers is OR. For a given type, e.g., *v4l*, if the OR operation is already overloaded, a function such as the *wire* function of Figure 7.35 is not needed. With this assumption, the declaration and body of the *VerilogLogic* package that is to contain the *oring* related definitions is shown in Figure 7.40. Note here that the result value of oring for null drivers is '0', which is the non-controlling value for an OR function.

```
PACKAGE VerilogLogic IS
FUNCTION oring ( drivers : v4l_vector) RETURN v4l;
   SUBTYPE ored_v4l IS oring v4l;
   TYPE ored_v4l_vector IS
                          ARRAY(NATURAL RANGE<>)OF ored_v4l;
   . . .
END PACKAGE VerilogLogic;

PACKAGE BODY VerilogLogic IS
   FUNCTION oring ( drivers : v4l_vector) RETURN v4l IS
      VARIABLE accumulate : v4l := '0';
   BEGIN
      FOR i IN drivers'RANGE LOOP
         accumulate := accumulate OR drivers(i);
      END LOOP;
      RETURN accumulate;
   END oring;
   . . .
END PACKAGE BODY VerilogLogic;
```

Figure 7.40 Package Description for *Oring* Resolution Function

7.4.1.7 Relation to Sequential Transactions. We use the notation show in Figure 7.41 to represent resolution functions, signal drivers, and driving a resolved left hand side signal. This notation elaborates the fact that sequential placement of transactions on a signal driver affect values that become resolution function inputs after they expire. While not expired, a signal transaction remains pending and maybe overwritten or appended to according to rules specified in Section 7.3. The timing of a resolution function is 0 delta and 0 real time. This means that the resolution function always provides a value resolved from driving values of its drivers even though the assignment of this value to the left hand side signal is always timed.
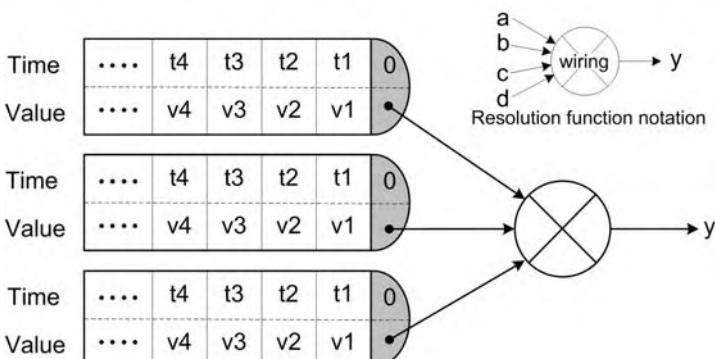
Figure 7.41 Projected Output Waveforms of Resolution Function

7.4.2 Resolutions with Guarded Assignments

The topic of guarded signals, guarded signal assignments, and use of guarded signal in block statements with a guard expression were discussed in Chapter 4. In this section we revisit this topic and explain the role of resolution functions and resolved signals in guarded assignments.

A guarded signal assignment uses the GUARDED keyword on the right hand side of the assignment arrow. Connection and the disconnection of the right hand side of such an assignment are controlled by a guard expression represented by an implicit or explicit guard signal.

Figure 7.42 shows a selection-logic with n inputs and n select lines. An active select line causes its corresponding input to be connected to y. We have used guarded signal assignments for describing connections, and have generated n such assignments using a for-generate statement. This description corresponds to the transistor circuit shown in Figure 7.43, and will be used as an example in the discussions that follow.

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY selection_1ofn IS
   PORT (ins : IN wired_v4l_vector;
         sel : IN wired_v4l_vector;
         w : OUT wired_v4l);
END ENTITY;
--
ARCHITECTURE wired OF selection_1ofn IS
   SIGNAL y : wired_v4l BUS;
BEGIN
   Mi: FOR i IN ins'RANGE GENERATE
      Ti: BLOCK (sel(i) = '1') BEGIN

         y <= GUARDED ins (i);

      END BLOCK Ti;
   END GENERATE Mi;
   w <= y;
END ARCHITECTURE wired;
```

Figure 7.42 1-of-n Selection Logic Assign Guarded Assignments
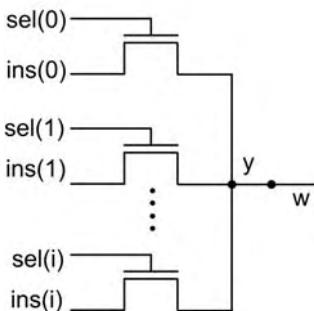
Figure 7.43 NMOS Transistor Based Selection Logic

If multiple guarded assignments are made to a left hand side signal, the signal must be resolved. When the guard expression of a signal assignment becomes false, the right hand side of the signal assignment disconnect from its left hand side. In case of a resolved left hand side (multiple guarded assignments to a signal), disconnection stops placement of values into the disconnected driver, and affects the way an expired driver value contributes to its resolution function.
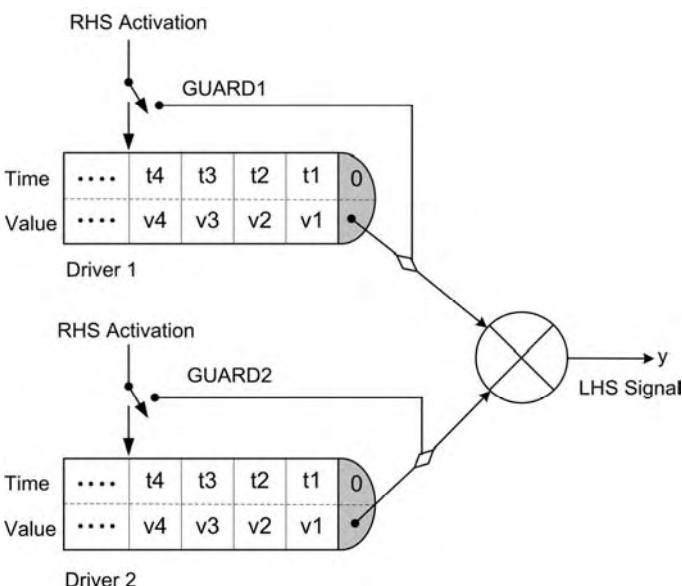
Figure 7.44 Guarded Signal Assignments into Resolved Signals

7.4.2.1 Guarded Assignments. Figure 7.44 shows concurrent guarded signal assignments made to a left hand side resolved signal. As shown, each driver is controlled by a guard expression. As shown, and as described in the above paragraph, a driver guard affects a left hand side signal in two ways. One is the disconnection from the right hand side as illustrated by the switches in Figure 7.44, and the other is the contribution of the resolution function as shown by the diamond boxes. The disconnection of the right hand side from the driver is independent of the kind (BUS or REGISTER) of the left hand side signal. Actually, for this disconnection, the left hand side signal does not even have to be a guarded signal (no kind is required). However, the kind of a signal, decides how an expired value affects the resolution function. See Figure 7.42 as an example description.

7.4.2.2 BUS Kind Resolved Signal. When a disconnection occurs from a left hand side signal of Bus kind, the right hand side no longer contributes to the resolution function, and is removed from the driver array argument of the resolution function. The resolution function is then called to calculate the new signal value without the disconnected value. The same happens when the last driver is disconnected from a Bus kind signal. With the last disconnection, the resolution function is called with null drivers, which causes the default resolution function value to be returned as the signal value. In the *selection_1ofn* circuit of Figure 7.42, the last disconnection from *y*, causes the *wiring* resolution function of Figure 7.36 to be called with null drivers, causing return of the 'Z' value.

7.4.2.3 REGISTER Kind Resolved Signal. Like a Bus kind guarded signal, disconnection of a driver of a Register kind, stops further transactions from being placed in the disconnected driver, and removes that driver from the resolution function argument. However, unlike the Bus kind that the last disconnection removes the driver and then calls the resolution function with null drivers, the last disconnection from a Register kind only removes the driver contribution and does not call the resolution function. This causes a Register kind guarded signal to retain its last value, implying a register.

Figure 7.45 shows a transistor selection circuit with an inverted output. Because transistors of the output inverter store charge in their gate capacitances, when all select inputs are '0', node *y* retains its old value. This circuit is modeled by the architecture shown in Figure 7.46.
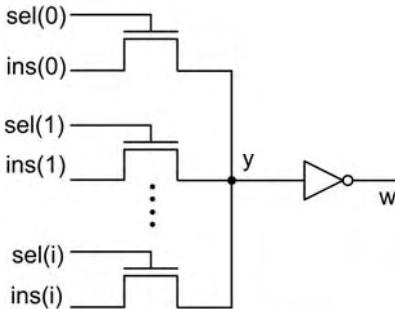
Figure 7.45 NMOS Half-Register with Selection Logic

```
ARCHITECTURE wired_reg OF selection_1ofn IS
   SIGNAL y : wired_v4l REGISTER;
BEGIN
   Mi: FOR i IN ins'RANGE GENERATE
      Ti: BLOCK (sel(i) = '1') BEGIN
         y <= GUARDED ins (i);
      END BLOCK Ti;
   END GENERATE Mi;
   w <= NOT y;
END ARCHITECTURE wired_reg;
```
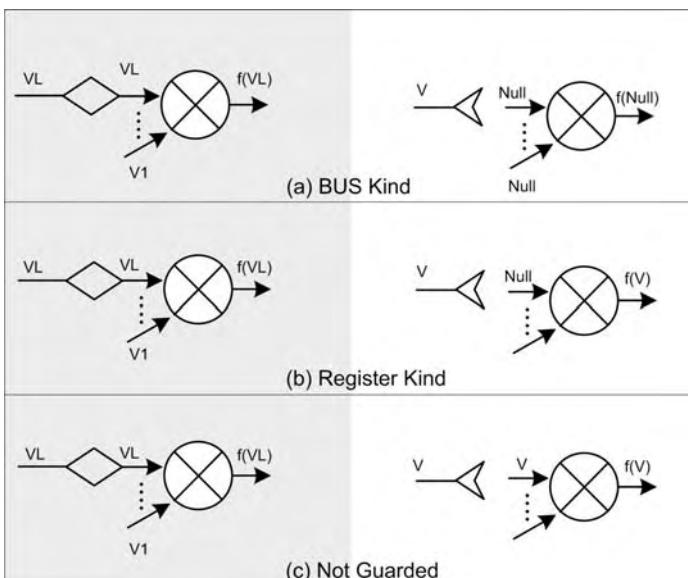
Figure 7.46 Using REGISTER Kind for Selection Logic with Half-Register

7.4.2.4 No Kind Resolved Signal. Disconnection of a driver from a resolved signal for which a kind is not specified, i.e., signal is not guarded, stops further transactions from being placed in the signal driver, and disconnects the driver from the resolution function. Such a disconnected driver retains its value at the time of disconnection and continues to contribute to the resolution function.

The IEEE standard for VHDL synthesis does not allow guarded assignments with signals that are not guarded. In this book, we have avoided making multiple assignments to such signals. With only one driver, a non-guarded signal behaves similar to a guarded signal with the REGISTER kind.

7.4.2.5 Comparing Disconnections. The difference between BUS and REGISTER kinds of guarded signal can also be explained by the use of the notation used in Figure 7.44. Figure 7.47 shows the last disconnection from (*a*) a guarded signal of kind BUS, (*b*) a guarded signal of kind REGISTER, and (*c*) a non-guarded signal. This figure shows values contributing to a resolution function before and after the last driver is disconnected. The left column shows the values before disconnection, and the right column shows the resolved values after the last disconnection occurs on the three kinds of signals.

Figure 7.47a shows that the last disconnection from a bus replaced the driving value of the disconnected driver with NULL and the resolution function is called to calculate the new value using NULL input. In Figure 7.47b, for a REGISTER kind, the last driver value is replaced with NULL, but the resolution function is not called to calculate a new value based on NULL input. For the last case of disconnection from a non-guarded signal, the driving value is not removed, and the resolution function continues to provide its result based on the value before disconnection.

Figure 7.47 Last Disconnections. (a) BUS Kind; (b) REGISTER Kind; (c) no Kind

7.4.3 Resolving INOUT Signals

Using a resolved signal on the right and left hand sides of a signal assignment does not necessarily refer to the same signal value. For example the assignment

```
a <= a AND b AFTER delay;
```

uses a on the right and left hand sides of a signal assignment. The value used on the right hand side contributes to the AND function which provides a value to the resolution function of signal a. However, the value assigned to a is the value out of the resolution function and not the value that a inputs to the resolution function. This situation is graphically presented in Figure 7.48. Since INOUT ports are bi-directional lines, an implicit IN and an implicit OUT port exist

for each INOUT line. Because INOUT ports provide outputs as well as inputs, connecting them requires resolution functions. When several INOUT ports are connected, the resolution function of the resolved intermediate signal provides a value that will be read from each INOUT port when used on the right hand side of an expression. Figure 7.49 shows a partial VHDL code and its corresponding graphical notation for connecting INOUT lines.

Since signal w is driven by the OUT side of INOUT of *one* and *two*, it is declared as a resolved signal of type *oring v4l*. The resolved value of w is given to the IN sides of both component INOUT ports. Reading x or y in component *one* or *two* (using it on a right hand side) reads the resolved w value. On the other hand, making an assignment to x or y (from component *one* or *two*, respectively) provides an input value for the *oring* resolution function. The input value provided as such, contributes to the value of the w signal.
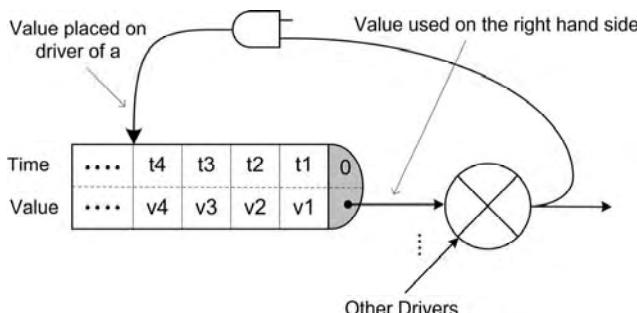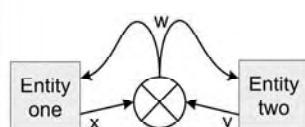
Figure 7.48 Resolved Signals on Right- and Left-Hand Sides

(a)

(b)

Figure 7.49 Connecting INOUT Ports Require Resolved Signals. (a) VHDL Code; (b) Graphical Notation

7.4.4 Standard Resolution

The IEEE standard logic (*std_logic_1164*) package discussed in the previous chapters provides a resolution function and utilities and type conversions based on this function.

7.4.4.1   Standard Type.   The actual 9-value type defined in the *std_logic_1164* package is *std_**u**logic* type. The "***u***" in this type mark stands for unresolved. This type mark has 'U', 'X', '0', '1', 'Z', 'L', 'H', 'W', and '-' values. The IEEE 1164 standard defines the *std_ulogic_vector* unconstrained array type and overloads various logic operations and functions and procedures for *std_ulogic* and *std_ulogic_vector*.

7.4.4.2   Resolution Function.   The standard IEEE 1164 resolution function is called *resolved*. The *resolved* function implements a wire function that is similar to the *wiring* function that we developed for our *v4l* type. The *resolved* function uses the *resolution_table* lookup table to lookup the resolved value of a pair of wired std-ulogic values. Details of this function can be found in the CD that accompanies this book.

7.4.4.3   Resolved Type.   The *std_logic_1164* package defines the *std_logic* type as its resolved type. This is a *resolved* subtype of *std_ulogic*. All assignments to objects of *std_logic* type are generated by the *resolved* resolution function.

The *std_logic_1164* package contains overloaded functions and operators for *std_logic*, *std_logic_vector* and their combinations with *std_ulogic* and *std_ulogic_vector*. Overloadings are done such that the resolved *std_logic* type can be used throughout a design. However, using this impedes VHDL's capability to detect and issue messages for those cases that unwanted multiple assignments are made to non-shared busses. To take advantage of this VHDL's capability, it is recommended to use *std_ulogic* for single source busses and *std_logic* for multi-source shared busses. Converting between the vector forms of these types is easily done since they are considered as closely related types.

7.5 Summary

We used "VHDL Signal Model" for the title of this chapter in order to be able to discuss sequential and concurrent assignments of values to signals all in one chapter. The first part of this chapter took a limited look of a single driver and only discussed how sequential transactions affect a signal driver. In the second part, we showed how multiple

driving values interact for resolving a value for a signal with multiple concurrent drivers. The topics of sequential placement of transactions and resolution functions were the two key topics that we discussed.

With this chapter, we have completed all language related topics and VHDL simulation model. The chapters that follow merely focus on design of components using VHDL.

Problems

7.1   In the VHDL code shown below four processes generate drivers on *w*, *x*, *y*, and *z* signals. Show transactions that expire on the drivers of each signal.

```
PACKAGE placing IS
  TYPE four IS (a, b, c, d);
END placing;
--
USE WORK.placing.ALL;
--
ENTITY placement IS END ENTITY;
--
ARCHITECTURE sequential OF placement IS
  SIGNAL w, x, y, z : four;
BEGIN

  P1: w <=
      four'RIGHTOF(w) AFTER 100 NS WHEN w /= four'RIGHT
      ELSE a AFTER 100 NS;
  P2: x <=
      four'LEFTOF(x) AFTER 80 NS WHEN x /= four'LEFT
      ELSE d AFTER 80 NS;

  P3: y <= a AFTER 20 NS, w AFTER 30 NS, x AFTER 40 NS;

  P4: PROCESS (x) BEGIN
    z <= TRANSPORT w AFTER 20 NS;
  END PROCESS;

END sequential;
```

7.2   Given the following signal assignments, show all transactions that are placed on the driver of each signal. At each event, show transactions that are appended, overwritten, and are expired. Show resulting waveforms on each signal; include transactions that expire even if they do not result in an event.

```
ARCHITECTURE dataflow OF signals IS
   SIGNAL a, b, c, d : v4l := '0';
BEGIN
   a <= NOT a AFTER 8 NS WHEN NOW <= 30 NS;
   b <= 'Z', a AFTER 25 NS, '0' AFTER 35;
   c <= '1', a AFTER 5 NS, b AFTER 20 NS;
   d <= a AFTER 5 NS, b AFTER 15 NS, c AFTER 25 NS;
END dataflow;
```

7.3   Given the following guarded signal assignments, show wave-forms that appear on *t1*, *t2*, and *t3* as results of waveforms that are generated on *a1*, *a2*, *b1*, and *b2* signals. Show real time events only.

```
ENTITY block_test IS END ENTITY;
--
ARCHITECTURE guarded_blocks OF block_test IS
   SIGNAL a1, a2 : BIT;
   SIGNAL b1, b2 : BIT;
   SIGNAL t1, t2, t3 : BIT;
BEGIN
   a1 <= NOT a1 AFTER 100 US;
   a2 <= NOT a2 AFTER 150 US;
   b1 <= NOT b1 AFTER 350 US;
   b2 <= NOT b2 AFTER 225 US;
   outter : BLOCK (a1 = '1' AND a1'EVENT)
   BEGIN
      t1 <= GUARDED b1 AFTER 10 US;
      inner : BLOCK (a2 = '0' AND NOT a2'STABLE AND GUARD)
      BEGIN
         t2 <= GUARDED b2 AFTER 10 US;
      END BLOCK;
   END BLOCK;
   separate : BLOCK (a2 = '1')
   BEGIN
      t3 <= GUARDED NOT t3
              WHEN a1'STABLE(50 US)'DELAYED(75 US)'EVENT;
   END BLOCK;
END guarded_block;
```

7.4   In test applications it becomes necessary to inject a fault into a circuit line. The fault overrides the normal value of the line and sets it to '1' or '0' depending on stuck-at-1 or stuck-at-0 faults that are be-ing injected. We are to develop utilities to enable us to inject faults into lines of *std_logic* type. For this purpose, every circuit line be-comes a record with a normal *std_logic* logic value and a field that specifies the fault if there is one. This field takes *sa1*, *sa0* and *nofault* values for stuck-at-1, stuck-at-0 and non-faulty values. This value decides if a line takes its normal value or a faulty '1' or '0'. A) Show

std_ulogic_faultabe type of record as described above. B) Show utilities for generating a resolution function to resolve between multiple drivers for lines of type *std_ulogic_faultabe* C) Write a resolution function that generates a normal *std_logic* value if no fault is injected on a line, or it generates '1' or '0' if a fault is injected on one of the drivers of a line. For this you can use the *resolution_table* and/or the *resolved* function of the *std_logic_1164* package.

7.5   Given the following signal assignments, show transactions and events that expire on each signal. Show timing diagram for time 0 to 350 NS.

```
ENTITY HLsignals IS END signals;
--
ARCHITECTURE dataflow OF HLsignals IS
   TYPE HL IS ('L', '0', '1', 'H');
   TYPE HL_vector IS ARRAY (HL) OF HL;
   CONSTANT inverse : HL := ('H', '1', '0', 'L');
   SIGNAL a, b : HL;
   SIGNAL c : HL := '0';
BEGIN
   a <= '0', '1' AFTER 20 NS,
        '0' AFTER 120 NS, 'H' AFTER 150 NS;
   b <= '0', 'L' AFTER 65 NS, inverse(c) AFTER 90 NS;
   c <= '1', b AFTER 30 NS, a AFTER 50 NS;
END dataflow;
```

7.6   Write a resolution function for an *n* input voter, voting on *n* positive bytes inputs. The output will take the byte value that appears on the majority of the inputs, with a tolerance of *t*. The tolerance allows inputs to be different by this value and still be considered as equal. This constant is defined in the resolution function.

7.7   Use the *f4l* type to write a resolution function that is similar to the wired-or resolution function of Verilog. In addition to the logic values, wire strengths also play a role in determination of a resolved value. Strength values that you will use from strongest to weakest are **Supply**, **Strong**, **Pull** and **Weak**. Between every two drivers, if strength values are the same, the ORing resolution is used and the strength of the result becomes that of the two drivers. If the strengths are not the same, the logic value of the stronger signal and its strength value becomes the resulting resolved value. The OR function for *f4l* is defined as the usual OR with 'Z' treated as logic '1'. Write all types, functions, and arrays that are needed for this resolution. Write the complete code of this resolution function.

7.8   A resolution function is to add integer drivers that are at least 20 ms old, and at most 50 ms old. The signal, *time_stamp*, receives integers from different sources. Integers placed on this signal must be timed and the resolved value of the signal must be updated every 1 ms or when a new value is assigned to the signal. All assignments are done with 0 delay values, no AFTER clause. When updating is to occur, a signal value that is less than 20 ms old, or one that is older than 50 ms will be ignored and not participate in the adding result of the resolution function. A) Declare the type you want to use for this resolved signal. B) Write all types, vectors and declarations necessary for developing a resolution function for this type. C) Write the *add_between_time* resolution function. D) Show examples of assignments to the *time_stamp* signal or any other signal that uses this resolution function. E) Show the mechanism for waking up the resolution function every 1 ms.

Suggested Reading

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Barnes, John, *Programming in Ada 2005*, 2006, Addison Wesley, ISBN: 0321340787.

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

Burns, Alan, Andy Wellings, and John Barns, *Concurrency in Ada, 2nd edition*, 1998, Cambridge University Press, ISBN: 052162911X.

Chu, Pong, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press, ISBN: 0471720925.

Cohen, Norman, *Ada As A Second Language, 2nd edition*, 1995, McGraw-Hill Science/Engineering/Math, ISBN: 0070116075.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL Hardware Description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Perry, Douglas L., *VHDL: Programming by Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

8 Hardware Cores and Models

The previous chapters presented the VHDL language syntax, semantics, and simulation model. We presented simple VHDL examples with the main purpose of demonstrating language constructs or simulation semantics. Combining VHDL language constructs and utilizing powerful language features for design and description of hardware is the mission of this chapter.

This chapter focuses on component descriptions. We discuss coding styles for simulation, synthesis, and modeling hardware modules. The styles presented are useful for design of cores and components of embedded systems. For the sake of scalability, and being of a general purpose nature, we develop most of the examples of this chapter with unconstrained input and output types. VHDL attributes and type declarations become useful for such applications.

Unless specified, models discussed in this chapter are synthesizable. In the first few examples we demonstrate and discuss rules for synthesis, and will follow these rules in the later examples.

8.1 Synthesis Rules and Styles

This section presents several combinational and sequential examples. The circuits are synthesizable, and related synthesis rules are presented as models are discussed.

8.1.1 Combinational Cores

VHDL constructs for concurrent assignments, sequential bodies, and component instantiations can be used for describing combinational circuits. Examples in this section cover these constructs.

8.1.1.1 Concurrent Assignments. A combinational circuit can be described by various forms of concurrent signal assignments, i.e., simple assignments, conditional assignments, and selected signal assignments. In the latter two cases, for combinational circuits, we should make sure that the assignments do not use the UNAFFECTED waveforms or leave conditions unspecified. If so, the unspecified cases cause the left hand side of the signal assignment to retain its old value, and thus, creating a latch.

Figure 8.1 shows an unconstrained magnitude comparator with equal, greater and less than outputs. Three concurrent conditional signal assignments handle the three outputs. The assignments assign '1' or '0' to their left hand sides according to their conditions. The VHDL language allows an ELSE part to be eliminated. If done so, the negation of the conditional of the if-statement causes the left hand side value to stay the same, which implies a latch. Since we are dealing with the combinational circuits in this section, obviously a latch is not desired.

The VHDL code shown in Figure 8.1 does not specify the ranges of it's a and b inputs. These will be determined when the comparator is instantiated in an upper-level structure. A more robust model of the comparator would include an assertion statement that would issue a message if the model is instantiated with different size inputs.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY comparator IS
   PORT (a, b: IN std_logic_vector;
         gt, eq, lt: OUT std_logic);
END ENTITY comparator;
--
ARCHITECTURE expression OF comparator IS
BEGIN
   eq <= '1' WHEN (a=b) ELSE '0';
   gt <= '1' WHEN (a>b) ELSE '0';
   lt <= '1' WHEN (a<b) ELSE '0';
END ARCHITECTURE expression;
```

Figure 8.1 Synthesizable Magnitude Comparator

Figure 8.2 shows the entity declaration of the comparator with an assertion for input size mismatch. If this happens, the comparator outputs its path name and a corresponding message. Obviously, this statement does not synthesize, and a synthesis tool either ignores it, or generator a warning or error message.

```
ENTITY comparator IS
      PORT (a, b: IN std_logic_vector;
            gt, eq, lt: OUT std_logic);
BEGIN
   ASSERT a'LENGTH = b'LENGTH
   REPORT comparator'PATH_NAME & ". Operand size mismatch";
END ENTITY comparator;
```

Figure 8.2 Entity Declaration with Assertion

8.1.1.2 Combinational Process Statements. Figure 8.3 shows an ALU with several arithmetic and logical operations. As with the comparator, this circuit is also unconstrained and adjusts itself to its instantiation from an upper-level structure. All operations of this circuit are unsigned as indicated by the *std_logic_unsigned* package.

The *procedural* architecture of *alu* shown in Figure 8.3 includes a process statement that functions as a combinational logic. Operations of the *alu*, that are *a+b*, *a-b*, *a*-input, *a* AND *b*, and NOT *a*, are handled by this process statement.

The *std_logic_unsigned* includes overloaded add and subtract operations that are used in this design. In order to capture the carry output of the arithmetic operations, *result*, that is one bit longer than the actual result needed, is declared and used on the left hand side of the ALU operations. In case of arithmetic operations, the left-most bit of *result* contains the carry. In order to satisfy right and left size requirements of VHDL, '0's are concatenated to the left of *a* and *b* operands to extend their sizes to that of *result*.

The body of the process statement of Figure 8.3 has a case statement that takes care of the operations of the ALU. The last case alternative is OTHERS that is there to account for unused choices of the *func* case expression. Note that since we are using the *std_logic* value system, many choices remain unspecified (i.e., those with 'U', 'X', etc). VHDL requires all choices to be specified in a case statement.

8.1.1.3 Process Combinational Style. In order to make sure a process statement that is intended to model a combinational circuit never implies a latch, two rules must be followed. A process statement following these rules always synthesizes to a combinational circuit.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY alu IS
   PORT (a, b : IN std_logic_vector;
         add_sub : IN std_logic;
         func : IN std_logic_vector (1 DOWNTO 0);
         y : OUT std_logic_vector;
         gt, eq, lt, co : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF alu IS
   SIGNAL result : std_logic_vector (a'LENGTH DOWNTO 0);
BEGIN
   PROCESS (a, b, add_sub, func) BEGIN
      CASE func IS
         WHEN "00" =>
            IF (add_sub = '1') THEN
               result <= ('0'&a) - ('0'&b);
            ELSE
               result <= ('0'&a) + ('0'&b);
            END IF;
         WHEN "01" => result <= '0'&a;
         WHEN "10" => result <= ('0'&a) AND ('0'&b);
         WHEN "11" => result <= NOT '1'&a;
         WHEN OTHERS => result <= (OTHERS => '0');
      END CASE;
   END PROCESS;

   UUT1: ENTITY WORK.compartor(expression)
         PORT MAP (a, b, gt, eq, lt);

   y <= result (a'LENGTH - 1 DOWNTO 0);
   co <= result (result'LEFT);
END ARCHITECTURE procedural;
```

Figure 8.3 Mixed Level ALU

Rule one has to do with inputs of a process statement. Inputs of a process are those signals that are used on the right hand side of assignments or are read by conditional expressions. The input rule says that all process inputs must appear on the sensitivity list of the process statement. The language semantics of this rule matches perfectly with the way a combinational logic block works. Namely, a combinational block always processes its body (gates) when an event occurs on one of its inputs.

Rule two of combinational process synthesis has to do with process left-hand-sides or outputs. Process left-hand-sides are those objects that receive some value within the process. This rule says that,

all left-hand side objects must receive some value regardless of the conditions of the conditional expression, i.e., no signal or variable should retain its old value when a process is executed from begin to end. This has to happen no matter what process input conditions are. There are two ways to implement this rule. One is to check every case and if-then statement and make sure all left-hand sides (process output signals and variables) always receive some value. Obviously for large combinational blocks this is a cumbersome task. Alternatively, we can indiscriminately set all process left-hand sides to their inactive values at the beginning of the process statement. This way, those signals or variables that receive a value in the body of the process statement will overwrite their inactive values, and those that do not receive a value take their inactive values. Therefore no object retains its old values. Retaining values synthesizes to latches.

The process statement in Figure 8.3 synthesizes to a combinational logic block. By extending the right hand sides of all assignments in this process (concatenating with '0'), we have made sure that all *result* bits receive some value regardless of their flow into the process statement. The process statement of Figure 8.4 uses the alternative implementation of the process left-hand side rule. In this process statement, we have set all bits of *result* to 0 at the beginning of the process statement. This way we can avoid extending bits of logical operations, (e.g., AND and NOT). The left-most bit of *result* that does not get a value for logical operations always gets a '0' at the beginning of the process statement.

```
PROCESS (a, b, add_sub, func) BEGIN
   result <= (result'RANGE => '0');
   CASE func IS
      WHEN "00" =>
         IF (add_sub = '1') THEN
            result <= ('0'&a) - ('0'&b);
         ELSE
            result <= ('0'&a) + ('0'&b);
         END IF;
      WHEN "01" => result (7 DOWNTO 0) <= a;
      WHEN "10" => result (7 DOWNTO 0) <= a AND b;
      WHEN "11" => result (7 DOWNTO 0) <= NOT a;
      WHEN OTHERS => result <= (OTHERS => '0');
   END CASE;
END PROCESS;
```

Figure 8.4 Inactive Process Output Setting

It is the semantics of the VHDL language that if a signal or a variable does not receive a value in a process statement, it retains its old value. It is also the property of a digital circuit that a latch on a

signal causes it to retain its value. Therefore, to the rule two of synthesis of combinational circuits avoids latches in the synthesized circuits.

8.1.1.4 Instantiation of Other Components.   In addition to the process statement discussed above, the ALU of Figure 8.3 includes other concurrent statements that are also part of the combinational *alu* entity. One such statement is the instantiation of the comparator of Figure 8.1. This example demonstrates that a hierarchical synthesizable combinational circuit can be made of synthesizable process statements, instantiation of other synthesizable parts, and synthesizable assignment statements.

8.1.2 Sequential Cores

Synthesizable sequential cores have very simple styles that can be combined together and with combinational synthesizable styles to form very complex synthesizable design cores. This section presents several simple, and yet general, styles for synthesizing sequential circuits.

8.1.2.1 Using Block Statements.   Figure 8.5 shows an n-bit up-down counter with a clock enable (*cen*) input and an asynchronous reset (*rst*) input. The counter uses the *std_logic* and the unsigned arithmetic packages. Guarded block statements are used for describing this counter.

The *counter* entity specifies counter inputs and outputs in *std_logic* type. Like other examples in this chapter, the output is an unconstrained array that will get its size when instantiated.

In the *blockbased* architecture of *counter*, signal *cnt_reg* is declared to hold the counter's internal count. This signal is needed since the mode of the *q* output is *output* that cannot be used on the right hand side of any expression. We are keeping the counter's state in *cnt_reg* to perform incrementing and decrementing on this signal. The declaration of *cnt_reg* uses the range of *q*, and is declared as a REGISTER kind guarded signal.

The body of the *blockbased* architecture has two nested block statements. The outer statement handles clocking and is labeled *cl*. The guard expression of this block is the positive edge of the *clk* signal. The inner block statement handles clock enabling and asynchronous reset. This block is labeled *en*. The enable signal is *cen* and is ANDed with the clock edge guard expression. The reason for this ANDing is clear, and it is because an active high clock enable signal enables the clock when it is '1' and disables it when '0'.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY counter IS
   PORT (clk, cen, rst, u_d : IN std_logic;
         q : OUT std_logic_vector);
END ENTITY ;

ARCHITECTURE blockbased OF counter IS
   SIGNAL cnt_reg : std_logic_vector(q'RANGE) REGISTER;
BEGIN
   cl: BLOCK (clk = '1' AND NOT clk'STABLE) BEGIN
      en: BLOCK ((cen = '1' AND GUARD) OR rst = '1') BEGIN
         cnt_reg <= GUARDED
               (cnt_reg'RANGE => '0') WHEN rst = '1' ELSE
                cnt_reg + 1 WHEN u_d = '1' ELSE
                cnt_reg - 1;
      END BLOCK;
   END BLOCK;
   q <= cnt_reg;
END ARCHITECTURE;
```

Figure 8.5 Counter Using Block Statements

The guard expression of the *en* block also includes the asynchronous *rst* signal. This signal is ORed with the rest of the guard expression since it acts independent of the clock (asynchronous). As a result, the expression of the guard signal within the *en* block becomes the edge of the clock AND with clock enable ORed with the reset. Therefore a guarded signal assignment in this block connects if *rst* is '1' or if clock edge is seen and *cen* is '1'. Clearly, this is asynchronous reset behavior as planned.

The guarded signal assignment in the *en* block sets *cnt_reg* to zero when *rst* is '1', and it counts up or down when rst is not active.

The code discussed above is synthesizable and follows styles presented in the VHDL 1076.6 RTL synthesis subset.

If *OR rst = '1'* is removed from the guard expression of the *en* block, the counter reset becomes synchronous. An asynchronous or synchronous *preset* signal can be added in a similar fashion. A synchronous parallel load feature can also be added to this style by including it in the condition that comes on the right-hand side of *cnt_reg*. In this example we have used two nested blocks. Perhaps a better style would have been to write a third block for *rst*, in which the guard signal would be ORed with *rst*. The choice of combining block statements or writing one for every feature of the counter or register structure is entirely upon the designer and his or her taste.

Before we leave this topic, it is worth spending a few minutes on the use of (OTHERS => '0') versus (xarray'RANGE => '0') for expanding an enumeration element, e.g., '0' to the size of an array, e.g., xarray. These constructs have been used in several occasions in the examples that we have discussed. When used on the right-hand side of an assignment, the type of the former construct will be determined based on that of the left-hand side. However, if this is used in an expression, the VHDL compiler will not be able to determine its type. On the other hand the latter construct (*xarray'RANGE* => '0') can be used on the right hand side of an assignment, in an expression, or any where a typed object may appear. Although the 'RANGE construct does not have the type mismatch issues OTHERS might have, it cannot include other enumeration elements and cannot be combined with other named associations as OTHERS can.

8.1.2.2 Process Statement for Sequential Logic. Another method of describing sequential circuits is by use of process statements. Figure 8.6 shows the VHDL description of a universal shift-register. The circuit has a parallel load (*ld*), synchronous reset (*rst*), left and right shift (*l_r*), shift enable (*shen*), serial input (*s_in*), and tri-state output enable (*oe*). The *dio* port of the *shift_reg* entity is the circuits bidirectional input and output port. The VHDL description of this shift-register is unconstrained, uses *std_logic*, and uses the unsigned package of the IEEE arithmetic package.

The *synch* architecture of *shift_reg* uses *parout* signal to hold the register contents of the shift register. This signal is declared as an *std_logic* array of the same size as the shift-register input-output port, *dio*. Within the process statement, *parout* is set to 0, loaded with *dio*, or shifted right or left.

The process statement shown in Figure 8.6 uses *clk* in its sensitivity list. The first statement in this sequential body detects the rising edge of the *clk* signal. Since the sensitivity list already detects *clk* events, the use of *clk'EVENT* in the condition of the if-statement is not essential. However, for compatibility with some older synthesis tools and flexibility of the model, use of this expression is recommended.

Alternatively, *rising_edge* or *falling_edge* functions of the *std_logic_1164* package can be used for the clock condition. Since all active assignments to *parout* take place within the if-statement with the clock edge condition, all shift-register activities are synchronous.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY shift_reg IS
   PORT (clk, ld, rst, l_r, shen, s_in, oe : IN std_logic;
         dio : INOUT std_logic_vector);
END ENTITY;
--
ARCHITECTURE synch OF shift_reg IS
   SIGNAL parout : std_logic_vector (dio'RANGE)
                 := (OTHERS => '0');
BEGIN
   PROCESS (clk)
      CONSTANT li : INTEGER := dio'LEFT;
   BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF (rst = '1') THEN
            parout <= (OTHERS => '0');
         ELSIF (ld = '1') THEN
            parout <= dio;
         ELSIF (shen = '1') THEN
            IF (l_r = '1') THEN
               parout <= parout (li-1 DOWNTO 0) & s_in;
            ELSE
               parout <= s_in & parout (li DOWNTO 1);
            END IF;
         ELSE
            parout <= parout; -- Not needed
         END IF;
      END IF;
   END PROCESS;
   dio <= parout WHEN (oe='1') AND (ld='0') ELSE
          (dio'RANGE => 'Z');
END ARCHITECTURE synch;
```

Figure 8.6 Shift-register Using Process Statement

The last statement before the end of the architecture of Figure 8.6, is a conditional signal assignment that puts *parout* shift-register contents on the *dio* if *oe* is '1' and *ld* is '0'. We are making sure that the *ld* input is '0', because if *dio* is externally driven for loading, we want to make sure that our architecture does not drive this same bidirectional port. When not driven with *parout*, the conditional signal assignment mentioned above drives *dio* with all 'Z's.

8.1.2.3 Asynchronous Control. As mentioned, the process in Figure 8.6 implements an asynchronous reset input, because all activities are controlled by the clock edge condition. Figure 8.7 shows the *asynch* architecture of *shift_reg* in which the process statement uses

rst as an asynchronous reset input. The two differences between this architecture and that of Figure 8.6 are: 1) the *rst* reset signal is included in the process sensitivity list, and 2) the *rst* condition check is precedes the clock edge detection. The combination of these two causes the process to wake up when *rst* changes, and when such happens and *rst* has become '1', *parout* is set to OTHERS => '0'.

```
ARCHITECTURE asynch OF shift_reg IS
   SIGNAL parout : std_logic_vector (dio'RANGE)
                := (OTHERS => '0');
BEGIN
   PROCESS (clk, rst)
      CONSTANT li : INTEGER := dio'LEFT;
   BEGIN
      IF (rst = '1') THEN
            parout <= (OTHERS => '0');
      ELSIF (clk = '1' AND clk'EVENT) THEN
         IF (ld = '1') THEN
            parout <= dio;
         ELSIF (shen = '1') THEN
            IF (l_r = '1') THEN
               parout <= parout (li-1 DOWNTO 0) & s_in;
            ELSE
               parout <= s_in & parout (li DOWNTO 1);
            END IF;
         ELSE
            parout <= parout; -- Not needed
         END IF;
      END IF;
   END PROCESS;
   dio <= parout WHEN (oe='1') AND (ld='0') ELSE
         (dio'RANGE => 'Z');
END ARCHITECTURE asynch;
```

Figure 8.7 Asynchronous Reset

8.1.2.4 Process Statement Sequential Circuit Synthesis. The above presented an example using a process statement representing a sequential circuit. Although the shift-register circuit is not a very complex hardware, the style we discussed applies to general sequential circuits. What follows summarizes the style used for the shift-register into a set of guidelines for sequential circuit synthesis.

A process statement that is to synthesize to a sequential circuit must include the clock signal in its sensitivity list. For asynchronous control, the corresponding signals must also appear in the process sensitivity list. A process statement for sequential circuit synthesis must include a conditional statement using the clock edge as its condition. For asynchronous control, conditions based on asynchronous signals must come before the clock condition. The body of the process

statement must include assignments to the register output of the process statement. It is recommended that a process statement only includes one left hand side signal. Such a signal becomes the register output of the process statement. If multiple register outputs (two left-hand sides) are to be used, they should be independent. That is, one should not appear in an expression on the right hand side of another.

The above guidelines are simple to follow and are unambiguous as far as the circuit that is implied. The standard VHDL 1076.6 reference discusses other more general synthesis styles.

8.1.3 Finite State Machines

Coding styles presented in Sections 8.1.1 and 8.1.2 for combinational and sequential circuits can be combined to describe finite state machines and controller circuits. Since sequence detectors are good examples of controllers and finite-state machines (FSM), the examples in this section are various forms of sequence detectors.

8.1.3.1 Moore Machines. A Moore machine is a state machine in which all outputs are fully synchronized with the circuit clock. In the state diagram form, each state of the machine specifies its output(s) independent of circuit inputs. In the VHDL code of a Moore machine, only circuit states participate in the output expression of the circuit.

Figure 8.8 shows a **101** Moore sequence detector with its corresponding block diagram related to its VHDL coding. The dark gray box signifies a process statement and the lighter box represents a concurrent signal assignment. The machine searches for **101** on its input and when received, the output of the circuit becomes **1** and remains at this level for a complete clock period. As shown in the state diagram, when the machine reaches the *got101* state, its output becomes **1**.
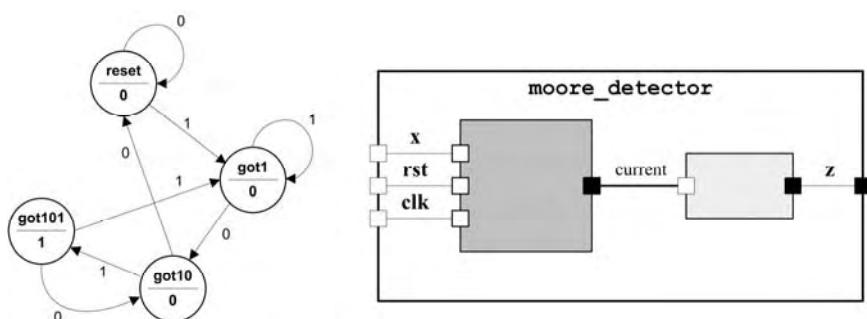
Figure 8.8 A Moore 101 Detector

In the VHDL code of this machine a process statement handling state transitions and clocking, generates *current* state of the machine. This variable is used by a concurrent signal assignment that generates the *z* output of the circuit.

Figure 8.9 shows the VHDL code of *moore_detector*. We have used an enumeration type declaration to declare the states of the machine. Type *state* has *reset, got1, got10,* and *got101* enumeration elements. The *current* signal that holds the state of our state machine is declared as a signal of type *state* and is initialized to *reset*. The *reset* initial state is only for simulation and has no synthesis significance. Since we have four states, a VHDL synthesis tool will use two state variables (i.e., two flip flops) for the states of the machine.

```
ENTITY moore_detector IS
   PORT (x, rst, clk : IN std_logic; z : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF moore_detector IS
   TYPE state IS (reset, got1, got10, got101);
   SIGNAL current : state := reset;
BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN
         current <= reset;
         ELSE
            CASE current IS
               WHEN reset =>
                  IF x = '1' THEN current <= got1;
                  ELSE current <= reset; END IF;
               WHEN got1 =>
                  IF x = '0' THEN current <= got10;
                  ELSE current <= got1; END IF;
               WHEN got10 =>
                  IF x = '1' THEN current <= got101;
                  ELSE current <= reset; END IF;
               WHEN got101 =>
                  IF x = '1' THEN current <= got1;
                  ELSE current <= got10; END IF;
               WHEN OTHERS => current <= reset;
            END CASE;
         END IF;
      END IF;
   END PROCESS;
   z <= '1' WHEN current = got101 ELSE '0';
END ARCHITECTURE;
```

Figure 8.9 Moore Machine VHDL Code

The process statement in Figure 8.9 implements a positive edge trigger sequential block with a synchronous reset (*rst*) input. If *rst* is active, *current* is set to *reset*, otherwise, a case statement assigns next state values to *current*. Next states of the machine are decided by the *current* state that is the case expression, and input values.

Each state of the machine is implemented by a case alternative, and its next state transitions are implemented by if statements conditioned by the *x* input of the circuit. Figure 8.10 shows a correspondence between the *got10* state of the machine and its VHDL code. This state branches out to *got101* or *reset* depending on *x*. The output of the circuit is implemented by a separate concurrent conditional signal assignment statement that puts a '1' on *z* when *current* is *got101*.
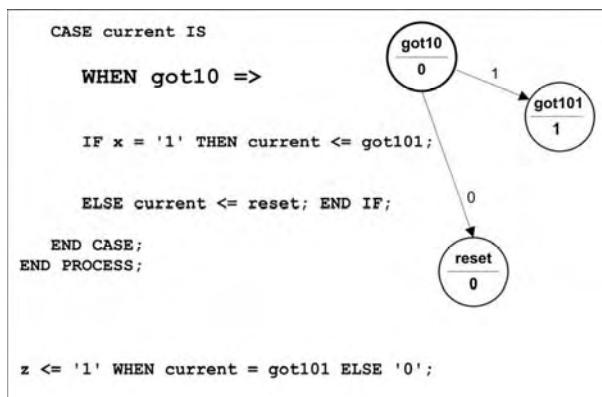
Figure 8.10 VHDL Code Correspondence with the *got10* State

Because this is a Moore machine, the condition for asserting the output of the circuit only includes the *current* variable, and circuit input(s) are not included.

Figure 8.11 shows another Moore machine example. This machine searches for **110** or **101** sequences on its *x* input. The search allows overlapping sequences.

8.1.3.2 Mealy Machines. A Mealy machine is different from a Moore machine in that its output depends on its current state and inputs while in that state. State transitions and clocking and resetting the machine are no different from those of a Moore machine, and the same coding techniques are used for describing them.

Figure 8.11 shows a **101** Mealy sequence detector and its corresponding VHDL code block diagram. This circuit has an asynchronous *rst* input that resets the machine to its reset state.
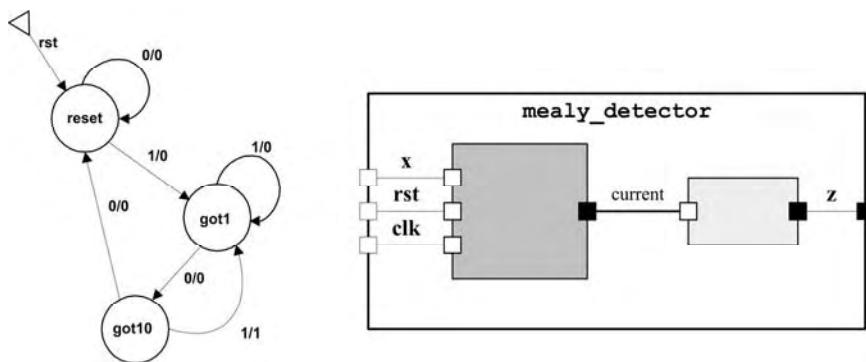
Figure 8.11 A 101 Mealy Machine

```
ENTITY mealy_detector2 IS
   PORT (x, clk, rst : IN std_logic; z : OUT std_logic);
END ENTITY mealy_detector2;
--
ARCHITECTURE procedural OF mealy_detector2 IS
   TYPE state IS (reset, got1, got10);
   SIGNAL current : state := reset;
BEGIN
   PROCESS (clk, rst) BEGIN
      IF rst = '1' THEN
         current <= reset;
      ELSIF (clk = '1' AND clk'EVENT) THEN
         CASE current IS
            WHEN reset =>
               IF x = '1' THEN current <= got1;
               ELSE current <= reset; END IF;
            WHEN got1 =>
               IF x = '1' THEN current <= got1;
               ELSE current <= got10; END IF;
            WHEN got10 =>
               IF x = '1' THEN current <= got1;
               ELSE current <= reset; END IF;
            WHEN OTHERS => current <= reset;
         END CASE;
      END IF;
   END PROCESS;
   z <= '1' WHEN (current = got10 AND x = '1') ELSE '0';
END ARCHITECTURE;
```

Figure 8.12 Mealy Machine VHDL Code

The VHDL code of Figure 8.12 corresponds to this Mealy machine. Type *state* is declared as an enumeration type having *reset*, *got1* and *got10* enumeration elements. This type is used to declare *current* that represents the state of the machine. For simulation, since current can only take reset, *got1* and *got10*, the use of OTHERS => as the last case alternative is not essential. However, after synthesis regardless of the coding technique for state assignments, e.g., encoded, binary, or one-hot, there will be unused states. Because of the use of OTHERS => ..., the synthesis tool synthesizes the circuit so that if an invalid state occurs, the *current* state always becomes reset.

The use of OTHERS => … as the last case alternative for state machine coding is always recommended. In simulation, this last case alternative never happens. In the actual circuit (post synthesis), the last case alternative makes provisions so that if an unspecified state occurs, the circuit always reset to its reset state. The default case alternative does not result in extra hardware in the synthesized circuit, if the machine does not have unused states.

The coding of the states and output of this machine are illustrated in Figure 8.13. Each state is specified by a case alternative of a case statement for which *current* is its case expression. Transitions to the next states of the machine are handled by if-then-else statements. The output of the machine is set to '1' using a conditional signal assignment. The condition part of this assignment uses the circuit's input as well as the current state of the machine.
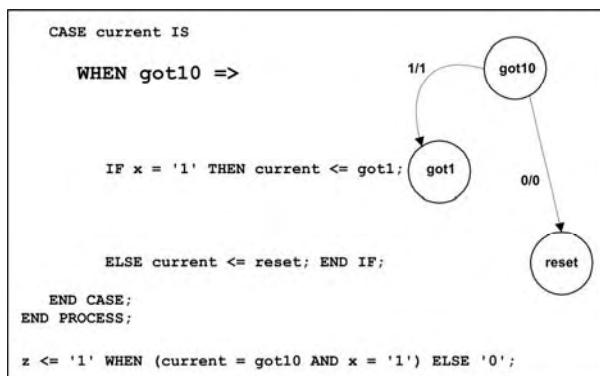
Figure 8.13 Mealy State and Output Coding

8.1.3.3 Huffman Coding Style. The Huffman model for a digital system characterizes it as a combinational block with feedbacks through an array of registers. According to the Huffman model, VHDL coding of digital systems uses a process statement for describing the register part and another concurrent statement for describing the combina-

tional part. This coding style and the Moore machine example that we will use in this section are shown in Figure 8.14. As shown, the *combinational* block uses *x* and *p_state* as input and generates *z* and *n_state*. The *register* block clocks *n_state* into *p_state*, and resets *p_state* when *rst* is active.
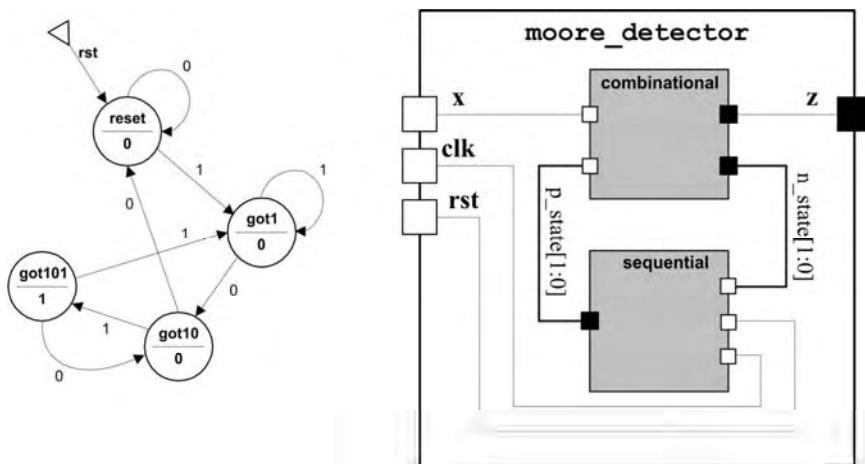
Figure 8.14 Huffman Style of Coding a State Machine

Figure 8.15 shows the VHDL code of the state diagram of Figure 8.14 according to the partitioning shown. In this code, type *state* is declared to represent the states of the machine. Following this declaration, *n_state* and *p_state* variables are declared as signals of type *state*. The *p_state* signal holds the present state of the machine and corresponds to state register outputs of the circuit. The *n_state* signal represents the input of the state register.

The *combinational* process statement appears in the *procedural* architecture of our *moore_detector4* entity. Since this is a purely combinational block, it is sensitive to all its inputs, namely *x* and *p_state*. Immediately following the process BEGIN, *n_state* and *z* are set to their inactive or reset values. This is done so that these variables are always refreshed with new values and never retain their old values. As discussed before, retaining old values implies latches, which is not what we want in our combinational block.

The body of the *combinational* process of Figure 8.15 contains a case-statement that uses the *p_state* input of the process for its case-expression. This expression is checked against the states of the Moore machine. As in the other styles discussed before, this case-statement has case-alternatives for *reset*, *got1*, *got10*, and *got101* states. In addition, an OTHERS case-alternative sets *n_state* to reset.

```
ENTITY moore_detector4 IS
   PORT (x, rst, clk : IN std_logic; z : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF moore_detector4 IS
   TYPE state IS (reset, got1, got10, got101);
   SIGNAL p_state, n_state : state;
BEGIN
   combinational: PROCESS (p_state, x) BEGIN
      n_state <= reset;
      z <= '1';
      CASE p_state IS
         WHEN reset =>
            IF x = '1' THEN n_state <= got1;
            ELSE n_state <= reset; END IF;
            z <= '0';
         WHEN got1 =>
            IF x = '0' THEN n_state <= got10;
            ELSE n_state <= got1; END IF;
            z <= '0';
         WHEN got10 =>
            IF x = '1' THEN n_state <= got101;
            ELSE n_state <= reset; END IF;
            z <= '0';
         WHEN got101 =>
            IF x = '1' THEN n_state <= got1;
            ELSE n_state <= got10; END IF;
            z <= '1';
         WHEN OTHERS =>
            n_state <= reset;
            z <= '0';
      END CASE;
   END PROCESS combinational;

   sequential: PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN p_state <= reset;
         ELSE
            p_state <= n_state;
         END IF;
      END IF;
   END PROCESS sequential;
END ARCHITECTURE;
```

Figure 8.15 Moore Detector VHDL Code According to Huffman Model

In the code portion corresponding to a particular case-alternative, based on input values, values are assigned to *n_state* and *z* output. Unlike the other styles where *current* is used for the present and next states, here we use two different variables, *p_state* and *n_state*.

The sequential process shown in Figure 8.15 handles the register part of the Huffman model of Figure 8.14. In this part, *n_state* is treated as the register input and *p_state* as its output. On the positive edge of the clock, *p_state* is either set to the *reset* state or is loaded with contents of *n_state*. Together, *combinational* and *sequential* blocks describe our state machine in a very modular fashion.

The advantage of this style of coding is in its modularity and defined tasks of each process. State transitions are handled by the *combinational* process and clocking is done by the *sequential* process. Changes in clocking, resetting, enabling or presetting the machine only affect the coding of the *sequential* process. If we were to change the synchronous resetting to asynchronous, the only change we had to make was adding *rst* to the sensitivity list of the *sequential* process.

8.1.3.4 A More Modular State Machine Coding Style. For a design with more input and output lines and more complex output logic, the *combinational* process may further be partitioned into a process for handling transitions and another for assigning values to the outputs of the circuit. For coding both of these processes, it is necessary to follow the rules discussed for combinational processes earlier in this section. Figure 8.16 shows a block diagram of this style and a Mealy sequence detector that we will use for illustrating this coding style.
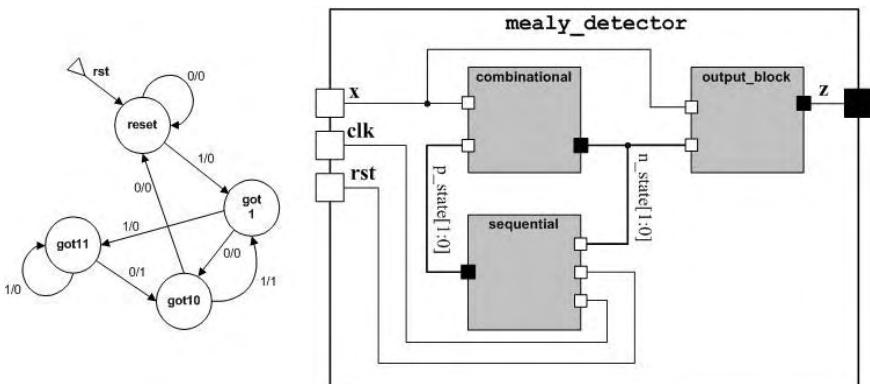
Figure 8.16 Using Three Separate Blocks for Describing a State Machine

Figure 8.17 shows the VHDL code for a Mealy machine that detects a sequence of **110** on its *x* input. This code uses two separate processes for assigning values to *n_state* and the *z* output. In a situation like what we have in which the output logic is fairly simple, a simple concurrent assignment could replace the *output_block* combinational process statement.

```
ENTITY mealy_detector6 IS
   PORT (x, en, clk, rst : IN std_logic;
         z : OUT std_logic);
END ENTITY mealy_detector6;
--
ARCHITECTURE procedural OF mealy_detector6 IS
   TYPE state IS (reset, got1, got11, got10);
   SIGNAL p_state, n_state : state;
BEGIN
   PROCESS (p_state, x) BEGIN
      CASE p_state IS
         WHEN reset =>
            IF x = '1' THEN n_state <= got1;
            ELSE n_state <= reset; END IF;
         WHEN got1 =>
            IF x = '1' THEN n_state <= got11;
            ELSE n_state <= got10; END IF;
         WHEN got11 =>
            IF x = '0' THEN n_state <= got10;
            ELSE n_state <= got11; END IF;
         WHEN got10 =>
            IF x = '1' THEN n_state <= got1;
            ELSE n_state <= reset; END IF;
         WHEN OTHERS => n_state <= reset;
      END CASE;
   END PROCESS;

   PROCESS (p_state, x) BEGIN
      CASE p_state IS
         WHEN reset => z <= '0';
         WHEN got1 => z <= '0';
         WHEN got11 => IF x = '1' THEN z <= '0';
                       ELSE z <= '1'; END IF;
         WHEN got10 => IF x = '1' THEN z <= '1';
                       ELSE z <= '0'; END IF;
         WHEN OTHERS => z <= '0';
      END CASE;
   END PROCESS;

   PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN p_state <= reset;
         ELSE
            IF en = '1' THEN
               p_state <= n_state;
            END IF;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;
```

Figure 8.17 A Mealy Machine Using Three Sequential Processes

The examples discussed above, in particular, the last two styles, show how combinational and sequential coding styles can be combined to describe very complex digital systems.

8.2 Memory and Queue Structures

This section discusses VHDL code for general purpose stack and FIFO cores. We start with a general purpose unclocked memory structure and then use this memory in a stack. Next we show design of a FIFO using a clocked register file. The techniques developed in this section can be used for development of various memory and data structures such as dual port memories and content addressable cores.

8.2.1 Generic RAM Core

Section 6.1 presented an unconstrained RAM with procedure for reading and dumping from and to external files. We used our example type *v4l* and file handling primitives for the memory model of Chapter 6. A general purpose model using *std_logic* and TEXTIO packages is presented here. We rely on the discussion of Chapter 6 for the details of this model.

Figure 8.18 shows *init_mem* and *dump_mem* procedures that use TEXTIO read and write utilities. The memory passed to these procedures in a two-dimensional array of *std_logic*. File type used by these procedures is TEXT. The *init_mem* procedure reads a *std_logic_vector* of the size of the second index of the memory from its *stddata* file and places the vector data into the memory one bit at a time. For writing, the *dump_mem* procedure reads data bits from the memory into *stdvalue* variable, and writes them into LINE type buffer, *l*. When all bits of word are written, a call to WRITELINE writes *l* to the external file.

```
PROCEDURE init_mem (VARIABLE memory: OUT mem;
                    CONSTANT datafile: STRING) IS
   FILE stddata : TEXT;
   VARIABLE l : LINE;
   VARIABLE data : std_logic_vector(memory'RANGE(2));
 BEGIN
   FILE_OPEN (stddata, datafile, READ_MODE);
   FOR i IN memory'RANGE(1) LOOP
      READLINE (stddata, l); READ (l, data);
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         memory (i,j) := data(j);
      END LOOP;
   END LOOP;
END PROCEDURE init_mem; --                            Continued
```

```
PROCEDURE dump_mem (VARIABLE memory: IN mem;
                      CONSTANT datafile: STRING) IS
   FILE stddata : TEXT;
   VARIABLE stdvalue : std_logic;
   VARIABLE l : LINE;
BEGIN
   FILE_OPEN (stddata, datafile, WRITE_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         stdvalue := memory (i, j);
         WRITE (l, stdvalue);
      END LOOP;
      WRITELINE (stddata, l);
    END LOOP;
END PROCEDURE dump_mem;
```

Figure 8.18 TEXTIO Based Memory init and dump Procedure

```
USE IEEE.std_logic_TEXTIO.ALL;
ENTITY std_logic_ram IS
   PORT (address : IN std_logic_vector;
         datain, dataout : OUT std_logic_vector;
         cs, rwbar : IN std_logic; opr : IN BOOLEAN);
END ENTITY std_logic_ram;
--
ARCHITECTURE behavioral OF std_logic_ram IS
   TYPE mem IS ARRAY (NATURAL RANGE <>,
                      NATURAL RANGE <>) of std_logic;
BEGIN
   PROCESS
      CONSTANT memsize : INTEGER := 2**address'LENGTH;
      VARIABLE memory : mem (0 TO memsize-1, datain'RANGE);
   BEGIN
      id: IF opr'EVENT THEN
         IF opr=TRUE THEN init_mem (memory, "memdata.dat");
         ELSE dump_mem (memory, "memdump.dat"); END IF;
      END IF;
      wr: IF cs = '1' THEN
         IF rwbar = '0' THEN -- Writing
            FOR i IN dataout'RANGE LOOP
               memory(conv_integer(address),i):=datain (i);
            END LOOP;
         ELSE                -- Reading
            FOR i IN datain'RANGE LOOP
               dataout(i)<=memory(conv_integer(address),i);
            END LOOP;
         END IF;
      END IF;
      WAIT ON cs, rwbar, address, datain, opr;
   END PROCESS;
```

Figure 8.19 Std_logic Unconstrained Memory

Figure 8.19 shows our general purpose RAM model using the *std_logic* package. The process statement in the body of the *behavioral* architecture of *std_logic_ram* has a sequential statement that is labeled *id*. This statement is responsible for initialization and dump, the *wr* statement in this process statement performs writing and reading of the memory.

8.2.2 Synthesizable Push-Pop Stack

Figure 8.20 shows a stack controller that together with instantiation of the memory of Figure 8.19 becomes a generic size general purpose synthesizable stack core model.

```
ENTITY stack IS
   GENERIC ( max: std_logic_vector := "101111");

   PORT (STin : IN std_logic_vector;
         clk, push, pop : IN std_logic; opr : IN BOOLEAN;
         STout : OUT std_logic_vector;
         empty, full : OUT std_logic);
END ENTITY stack;
--
ARCHITECTURE behavioral OF stack IS

   SIGNAL ramin, ramout : std_logic_vector (STin'RANGE);
   SIGNAL ramaddr, pntr : std_logic_vector (max'RANGE)
                       := (OTHERS => '0');
   SIGNAL cs, rwbar, full_temp : std_logic:= '0';
   SIGNAL empty_temp : std_logic:= '1';

BEGIN

   -- UPDATING PNTR
   -- POP/PUSH
   -- INSTANTIATE MEMORY
   -- HANDLING EMPTY AND FULL

END ARCHITECTURE behavioral;
```

Figure 8.20 Stack Controller Outline

The stack controller takes push and pop commands and prepares its *pntr* to point to locations in its instantiated RAM. As shown in Figure 8.20, our stack has an unconstrained *STin* input and a *max* generic parameter that sets the maximum size of the stack. The stack architecture defines RAM input and output (*ramin* and *ramout*) as *std_logic_vectors* of the size and range of *STin*. The address size of the RAM is set as the range of the *max* generic parameter. The value of

the *max* generic parameter specifies the size of the stack memory, and its range defines the memory block size used for the stack. Specifying a number that is not all 1's for the value of *max* means that not all words of the memory block allocated for the stack will be used. The *max* parameter value is used to set the stack *full* flag.

The *behavioral* architecture of the *stack* shown in Figure 8.20 has two process statements for updating the stack pointer, and pop and push operations. It has an instantiation statement for instantiating the memory, and a part for handling empty and full flags.

Figure 8.21 shows the *update_pntr* process statement. This is a sequential clocked process statement that decrements or increments *pntr* depending on *pop* and *push* inputs. The *pntr* register has a synchronous *rst* reset input.

```
-- UPDATING PNTR
Update_pntr: PROCESS (clk)
BEGIN
   IF (clk = '1' AND clk'EVENT) THEN
      IF pop = '1' THEN
         IF empty_temp /= '1' THEN
            pntr <= pntr - 1;
         END IF;
      ELSIF push = '1' THEN
         IF full_temp /= '1' THEN
            pntr <= pntr + 1;
         END IF;
      END IF;
   END IF;
END PROCESS;
```

Figure 8.21 Stack Pointer Update

Figure 8.22 shows a process statement for issuing memory read and write signals, and for setting proper data and address on the memory data and address ports. Since this is a combinational process, all process inputs are included on the sensitivity list, and all process outputs are set to their inactive values at the beginning of the process. When *pop* is issued, the memory is read from *pntr_1* and is made available on *STout*. While *pop* is active, the next clock edge changes the *pntr* value, that makes the stack output for pop invalid. An external device using our stack core should read the stack output for pop on the first clock edge after *pop* is issued.

When *push* is issued, *pntr* value goes on *ramaddr* and memory write is done. While *push* is active the next clock edge changes the pointer value. The device using our stack core must deassert the *push* input of the stack immediately after the first clock edge. The timing

requirement of *push* and *pop* operations could be more relaxed if input and output of the memory, or its address bus were registered.

```
-- POP/PUSH

pop_push: PROCESS (pop, push ,STin, ramout, pntr)
BEGIN
   ramaddr <= (OTHERS => '0');
   cs <= '0';
   rwbar <= '1';
   ramin <= (OTHERS => '0');
   STout <= (STin'RANGE => '0');
   IF (pop = '1' AND empty_temp = '0') THEN
      ramaddr <= pntr - 1;
      cs <= '1';
      rwbar <= '1';
      STout <= ramout;
   ELSIF (push = '1' AND full_temp = '0') THEN
      ramaddr <= pntr;
      cs <= '1';
      rwbar <= '0';
      ramin <= STin;
   END IF;
END PROCESS pop_push;
```

Figure 8.22 *Pop_push* Process

Figure 8.23 shows the last part of the code of the stack controller. In this code, an instantiation statement instantiates the *std_logic_ram* entity, and concurrent assignment statements issue *empty* and *full* stack output flags.

```
-- INSTANTIATE MEMORY

   UU1: ENTITY WORK.std_logic_ram (behavioral)
        PORT MAP (ramaddr, ramin, ramout, cs, rwbar, opr);

-- HANDLING EMPTY AND FULL

   empty_temp <= '1' WHEN (pntr = (pntr'RANGE => '0'))
                               ELSE '0';
   full_temp <= '1' WHEN (pntr = max) ELSE '0';

   empty <= empty_temp;
   full <= full_temp;
```

Figure 8.23 RAM Instantiation and *empty* and *full* Flags

8.2.3   Synthesizable Circular FIFO

Figure 8.24 shows a circular FIFO. This memory structure has read and write pointers that keep track of data in FIFO. When read and write pointers are equal, FIFO is empty. A read operation causes the read pointer to be incremented. Writing into FIFO, causes the write pointer to be incremented. When the write pointer is one position behind the read pointer, the stack is full. Finally, while incrementing the read or the write pointer, if it reaches its maximum count (2^n-1 for an n-bit counter), it rolls over and becomes 0. This effectively makes it looks like the FIFO is going around a circle, thus a circular FIFO. We use a FIFO size that is a power-of-two in order to make the rollover implement easier.
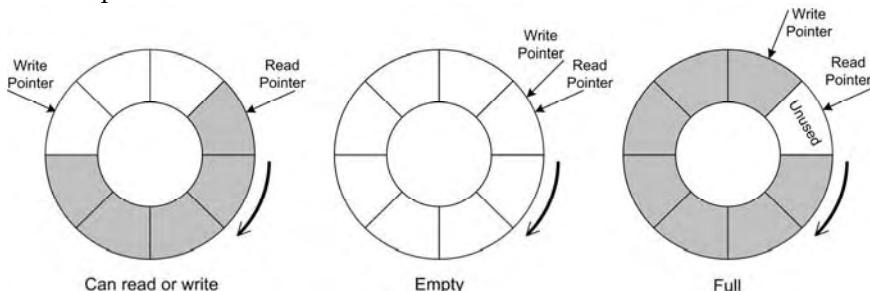
Figure 8.24 Circular FIFO

The outline of the VHDL code of FIFO is shown in Figure 8.25. The size of the FIFO is given to it as generic parameters. We have used a binary number for the size of the FIFO, and since this number must be a power of two, only its most significant bit is '1'. The size of the FIFO read and write pointers are calculated from the length of the *fifo_size* parameter.

The *procedural* architecture of *fifo_unconst* calculates *fsz*, *asz*, and *wsz* for the FIFO size, address size, and word size, respectively. In this architecture, the *fifo_ram* signal represents the FIFO register structure. This is a clocked memory of *fsz* size with *wsz* bits wide words. Signals *rd_ptr* and *wr_ptr* are declared for the read and write pointers. The size of these pointers is *asz* that is the size of the address bus of FIFO.

The body of the FIFO architecture of Figure 8.25 has three processes for *write*, *read*, and *pointer* setting. In addition, there are signal assignments for setting *empty* and *full* flags. The three processes are clocked and described below. Figure 8.26 shows a block diagram of this FIFO showing *read*, *write*, and *pointer* processes. We are also showing the *fifo_ram* and a block for *empty* and flags.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY fifo_unconst IS

   GENERIC (fifo_size : std_logic_vector := "1000");

   PORT (data_in : IN std_logic_vector;
         clk : IN std_logic;
         rst, rd, wr : IN std_logic;
         empty, full : OUT std_logic;
         data_out : OUT std_logic_vector);

END ENTITY ;
--
ARCHITECTURE procedural OF fifo_unconst IS

   CONSTANT fsz : INTEGER := conv_integer (fifo_size);
   CONSTANT asz : INTEGER := fifo_size'LENGTH - 1;
   CONSTANT wsz : INTEGER := data_in'LENGTH; --word_size;

   TYPE memory IS ARRAY (NATURAL RANGE <>) OF
                        std_logic_vector (wsz-1 DOWNTO 0);
   SIGNAL fifo_ram : memory (0 TO fsz-1);

   SIGNAL rd_ptr, wr_ptr : std_logic_vector(asz-1 DOWNTO 0)
                        := (OTHERS => '0');
   SIGNAL full_temp, empty_temp : std_logic;
BEGIN


-- WRITE

-- READ

-- POINTER

   empty_temp <= '1' WHEN ( rd_ptr=wr_ptr) ELSE '0';
   full_temp <= '1' WHEN ( rd_ptr=wr_ptr + 1) ELSE '0';

   empty <= empty_temp;
   full <= full_temp;


END ARCHITECTURE;
```

Figure 8.25 FIFO VHDL Code Outline
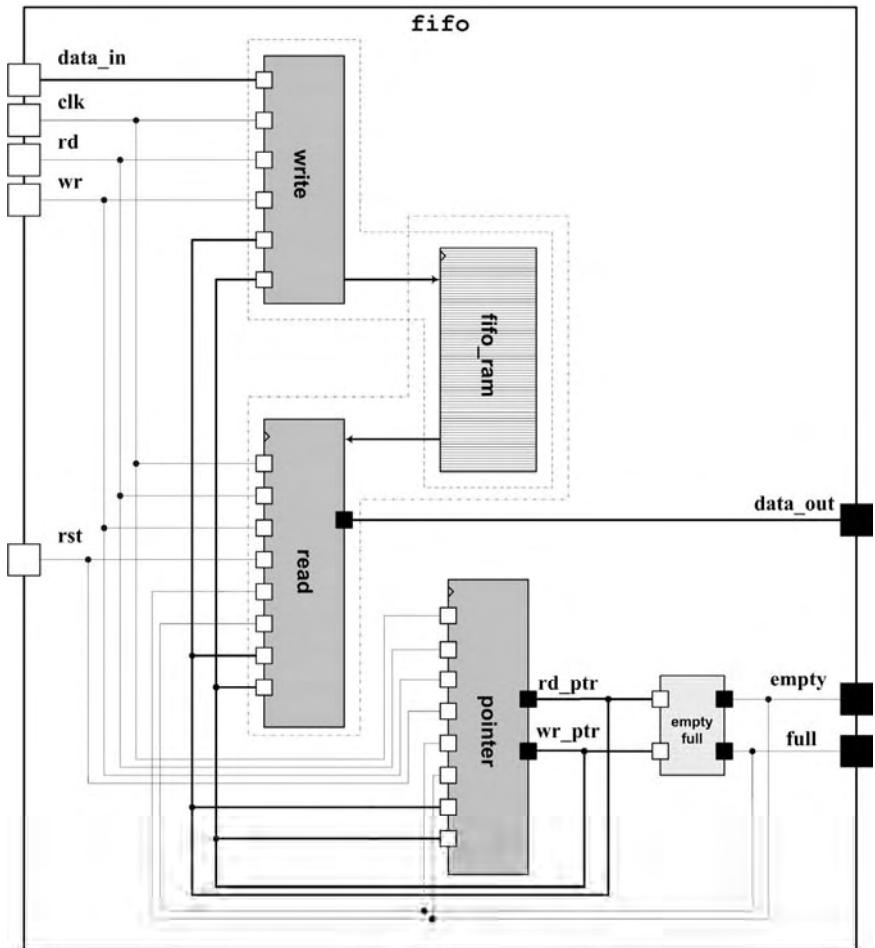
Figure 8.26 FIFO Block Diagram

8.2.3.1 Clocked Memory Writing. The *write* process shown in Figure 8.27 handles writing of data into the FIFO clocked memory. Because *fifo_ram* is assigned data within a process statement that is sensitive to the clock and the clock edge is detected, it synthesizes as a clocked memory. Writing into *fifo_ram* is done at the location of *wr_ptr*.

8.2.3.2 Clocked Memory Reading. The *read* process shown in Figure 8.28 reads *fifo_ram* at *rd_ptr* locations and loads them into *data_out*. Obviously, reading *fifo_ram* does not require a clock. However, because the read operations take place in a clocked process, the left hand side to which the memory is assigned, i.e., *data_out*, becomes a clocked register.

```
write : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF (wr='1' AND full_temp='0') THEN
         fifo_ram (conv_integer (wr_ptr)) <= data_in;
      ELSIF (wr='1' AND rd='1') THEN
         fifo_ram (conv_integer (wr_ptr)) <= data_in;
      END IF;
   END IF;
END PROCESS;
```

Figure 8.27 Clocked Writing: *fifo_ram*

```
read : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF (rd='1' AND empty_temp='0') THEN
         data_out <= fifo_ram (conv_integer (rd_ptr));
      ELSIF (rd='1' AND wr='1' AND empty_temp='1') THEN
         data_out <= fifo_ram (conv_integer (rd_ptr));
      END IF;
   END IF;
END PROCESS;
```

Figure 8.28 Clocked Reading: *fifo_ram*

```
pointer : PROCESS (clk) BEGIN

   IF (clk='1' AND clk'EVENT) THEN
      IF rst='1' THEN
         wr_ptr <= (OTHERS => '0');
         rd_ptr <= (OTHERS => '0');
      ELSE
         IF (wr='1' AND full_temp='0') OR
            (wr='1' AND rd='1') THEN
            wr_ptr <= wr_ptr+1;
         ELSE
            wr_ptr <= wr_ptr;
         END IF;
         IF (rd='1' AND empty_temp='0') OR
            (wr='1' AND rd='1') THEN
            rd_ptr <= rd_ptr+1;
         ELSE
            rd_ptr <= rd_ptr;
         END IF;
      END IF;
   END IF;
END PROCESS;
```

Figure 8.29 Updating FIFO Pointers

8.2.3.3   Multiple Clocked Register Process.   The *pointer* process shown in Figure 8.29 handles *rd_ptr* and *wr_ptr* counters. The *pointer* process is referenced as a comment in Figure 8.25 and as a block in the diagram of Figure 8.26. This process is a clocked process that increments *wr_ptr* and/or *rd_ptr* depending on the states of *wr* and *rd* inputs. Normally, for synthesis, we would use two separate processes for the two counters. However, since these counters are independent, combining them into a single process does not cause a problem or ambiguity in synthesizing the process.

This concludes the discussion of our synthesizable FIFO description. Note that we tried to design this FIFO as generic as possible. However, it is possible that same synthesis tools cannot recognize the generic parameters and the use of these parameters in type definitions. In such situations, constants should be used instead of the generic parameters.

8.2.4   Dynamic Access Type FIFO

In system level designs, and higher than RT level designs, it may be necessary to use a memory structure or arithmetic unit for evaluation of system performance. In such applications, the exact size of a hardware part may not be known initially, and depending on the required performance, the size and configuration of the structure and its RT level details may be decided.

A specific example of this scenario is the FIFOs in a transaction level modeling (TLM), or FIFOs in Network-on-Chip (NoC) switches. An NoC consists of an array of processing elements and switches. Network switches communicate data packets between processing elements. A switch uses FIFOs to buffer input and output data and to route data to switch ports where data is intended. An NoC design may involve configuration of switch FIFOs so that switch traffic is minimized and network performance is optimized.

In such applications, we may want to use a high level FIFO description that can easily be optimized, and that will not impose any size limitations on the overall system design. In a similar situation an arithmetic unit may need to be evaluated and its exact configuration determined in a high level system design environment, before deciding on its RT level details.

For scenarios like what is described above, VHDL access types can be used for describing dynamic structures such as FIFOs and arithmetic or floating point processors. This section discusses an access type FIFO. The FIFO length is dynamic and as it is used in a system level design, it reports its size as it grows and shrinks. A system designer can look at these statistics and decide the exact RTL memory buffer size for his or her application.

8.2.4.1 Dynamic FIFO Structure. Access type declarations for the
FIFO elements, the FIFO structure, its pointer, and *head* and *tail*
variables are shown in Figure 8.30. Variables *head* and *tail* keep
track of the two ends of the FIFO. The FIFO data is an 8-bit
std_logic_vector type array, but it can be set to any complex con-
strained type.

```
TYPE fifo_element;
TYPE pointer IS ACCESS fifo_element;
TYPE fifo_element IS RECORD
      data : std_logic_vector (7 DOWNTO 0);
      link : pointer;
END RECORD;
SHARED VARIABLE head, tail : pointer := NULL;
```

Figure 8.30 Dynamic FIFO Structure

8.2.4.2 Writing to Dynamic FIFO. Writing to the FIFO structure of
Figure 8.30 is done by the *write_fifo* procedure shown in Figure 8.31.
This procedure takes *head* and *tail* pointers as input. If *head* is null,
FIFO is empty and *data_in* is written to the *head*, and *tail* is set ac-
cordingly. If FIFO is not empty, a NEW link is created at the *tail*, and
data is written to it. Writing takes place at the *tail*.

```
PROCEDURE write_fifo(
      VARIABLE head, tail : INOUT pointer;
      data_in : IN std_logic_vector (7 DOWNTO 0)) IS
BEGIN

   IF (head=NULL) THEN
      head := NEW fifo_element;
      head.data := data_in;
      head.link := NULL;
      tail := head;
   ELSE
      tail.link := NEW fifo_element;
      tail := tail.link;
      tail.data := data_in;
      tail.link := NULL;
   END IF;

fifo_cnt := fifo_cnt+1;
REPORT "FIFO SIZE : "&INTEGER'IMAGE(fifo_cnt);

END write_fifo;
```

Figure 8.31 FIFO Write Procedure

8.2.4.3   Reading from Dynamic FIFO. Reading from the dynamic structure of Figure 8.30 is done by the *read_fifo* procedure of Figure 8.32. Inputs of this procedure are *head* and *tail* of the FIFO and its output is *data_out*. When reading, if *head* and *tail* are the same, then there is only one element in the FIFO that is read into data-out. However, if there are more than one element is the FIFO, the *head* is read, and *head* is set to *head.link*. In general, reading takes place at the head.

```
PROCEDURE read_fifo (
   VARIABLE head, tail : INOUT pointer;
   SIGNAL data_out : OUT std_logic_vector (7 DOWNTO 0)) IS
BEGIN

   IF (head=NULL) THEN
      REPORT "FIFO IS EMPTY!";
   ELSIF (head=tail) THEN
      REPORT "HEAD=TAIL CASE";
      data_out <= head.data;
      head := NULL;
      tail := NULL;
      fifo_cnt := fifo_cnt - 1;
   ELSE
      REPORT "ELSE CASE";
      data_out <= head.data;
      head := head.link;
      fifo_cnt := fifo_cnt - 1;
   END IF;

   REPORT "FIFO SIZE : "&INTEGER'IMAGE(fifo_cnt);

END read_fifo;
```

Figure 8.32 FIFO Read Procedure

8.2.4.4 FIFO with RT Level Interface.   Although our FIFO uses an access type memory structure, for RT level applications where a FIFO needs to be configured, an RT level interface is needed for the VHDL description of this hardware. Figure 8.33 shows the general outline of our *fifo_access* with RT level interface.

The declarative part of the architecture of *fifo_access* contains declarations and definitions shown in Figure 8.30, Figure 8.31, and Figure 8.32. In addition, *fifo_cnt* variable is declared to keep track of the size of FIFO. Ports and signals of this FIFO are the same as those of the RTL synthesizable FIFO of the previous section. The *fifo_access* architecture has *write* and *read* clocked processes that make writing and reading the access type FIFO look like clocked operations.

```
ENTITY fifo_access IS

   PORT (data_in : IN std_logic_vector (7 DOWNTO 0);
         clk : IN std_logic;
         rst, rd, wr : IN std_logic;
         empty, full : OUT std_logic;
         data_out : OUT std_logic_vector (7 DOWNTO 0));

END ENTITY ;
--
ARCHITECTURE procedural OF fifo_access IS

   -- FIFO structure declaration . . .
   SHARED VARIABLE fifo_cnt : INTEGER;
   SIGNAL full_temp, empty_temp : std_logic;

   -- FIFO write procedure definition . . .
   -- FIFO read procedure definition . . .

BEGIN

   write : PROCESS (clk) BEGIN
   . . .
   END PROCESS;
   --

   read : PROCESS (clk) BEGIN
   . . .
   END PROCESS;
   --

   full_temp <= '0';

   empty <= empty_temp;
   full <= full_temp;

END ARCHITECTURE;
```

Figure 8.33 fifo_access Interface and Outline

Figure 8.34 shows clocked processes for writing and reading our dynamic FIFO structure. These processes call *write_fifo* and *read_fifo* procedures of Figure 8.31 and Figure 8.32 instead of explicitly writing into a FIFO memory as was done in the synthesizable FIFO of the previous section.

```
write : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF rst='1' THEN
         fifo_cnt := 0;
      ELSE
         IF (wr='1' AND full_temp='0') THEN
            write_fifo(head, tail, data_in);
         ELSIF (wr='1' AND rd='1') THEN
            write_fifo(head, tail, data_in);
         END IF;
      END IF;
   END IF;
END PROCESS;
--
read : PROCESS (clk) BEGIN
   IF (clk='1' AND clk'EVENT) THEN
      IF rst='1' THEN
         fifo_cnt := 0;
      ELSE
         IF (rd='1' AND empty_temp='0') THEN
            read_fifo(head, tail, data_out);
         ELSIF (rd='1' AND wr='1' AND empty_temp='1') THEN
            read_fifo(head, tail, data_out);
         END IF;
      END IF;
      IF fifo_cnt=0 THEN
         empty_temp <= '1';
      ELSE
         empty_temp <= '0';
      END IF;
   END IF;
END PROCESS;
```

Figure 8.34 Writing and Reading ACCESS Type FIFO Structure

8.3 Arithmetic Cores
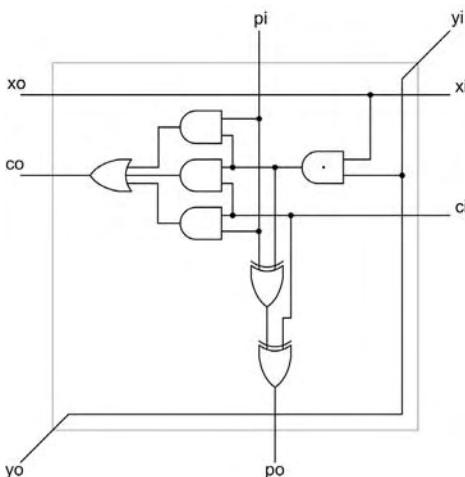
In this section we present several combinational and sequential arithmetic structures that can be used as embedded cores in an embedded design. The cores discussed are an array multiplier, a carry-lookahead adder, and a sequential multiplier. As the other cores in this chapter, *std_logic* type will be used for description of these components, and designs are done for configurability. Therefore unconstrained arrays are used where possible, otherwise generic parameters are used for configurability of designs.

8.3.1 Array Multiplier

Multiplication of binary operands in a digital system can be performed in a variety of ways that are different in speed and gate count. A sequential multiplier requires a clock and performs partial multiplications with each clock, while a combinational multiplier performs its task of multiplication without requiring a clock and it has a higher gate count. A combinational n×n multiplier requires an array on n×n multiplier cells each of which is responsible for multiplying a bit of multiplier with a bit of multiplicand and adding the result with the product bit coming from a previous multiplication stage.
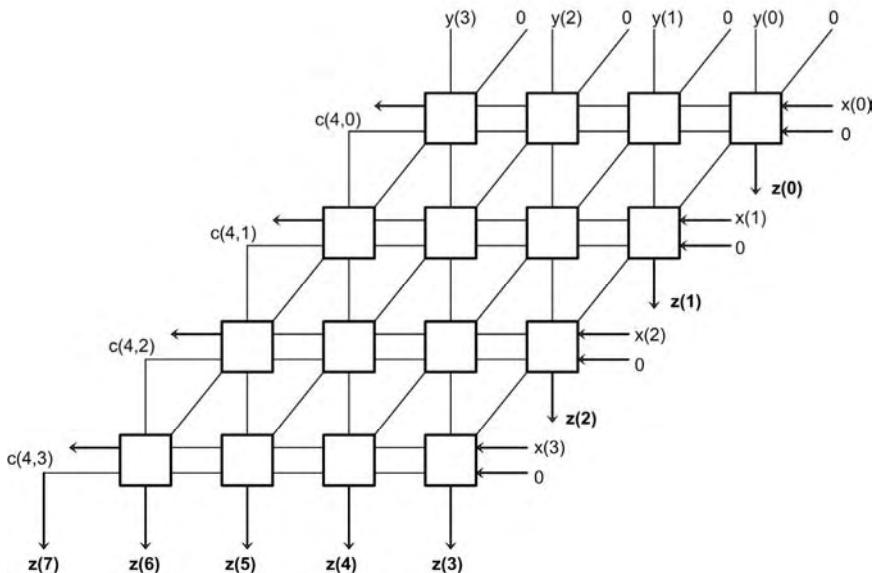
Therefore, an AND gate for 1×1 multiplication and a full-adder for the add operation constitute the required multiplier cell hardware, as shown in Figure 8.35.

The cell shown in Figure 8.35 multiplies its x_i and y_i inputs using the AND gate that is marked with a dot, and adds this result with its input partial product p_i, using its carry input c_i. This cell generates a partial product p_o, a carry output c_o, and passes x_i and y_i inputs on to its outputs (x_o and y_o).

Figure 8.35 Multiplier Cell

Figure 8.36 shows a 4×4 array multiplier that uses 16 of the multiplier cells of Figure 8.35. A 32-bit multiplier requires 1024 such cells. Figure 8.37 shows the VHDL code of a multiplier cell *bit_multiplier* corresponding to the diagram of Figure 8.35.

Figure 8.36   4×4 Multiplier

The *logical* architecture of *bit_multiplier* has a full-adder, an AND gate and pass-through wires connecting inputs x_i and y_i to their corresponding outputs, x_o and y_o.

```
ENTITY bit_multiplier IS
   PORT (xi, yi, pi, ci : IN std_logic;
         xo, yo, po, co : OUT std_logic);
END bit_multiplier;

ARCHITECTURE logical OF bit_multiplier IS
   SIGNAL xy : std_logic;
BEGIN
   xy <= xi AND yi;
   co <= (pi AND xy) OR (pi AND ci) OR (xy AND ci);
   po <= pi XOR xy XOR ci;
   xo <= xi;
   yo <= yi;
END logical;
```

Figure 8.37   One-bit Multiplier

Figure 8.38 shows the VHDL code of an n×n unconstrained array multiplier. In the declarative part of this architecture *xv*, *yv*, *cv*, and *pv* n×n arrays are declared. These signals are used for connecting signals of the bit-multipliers of the array multiplier to their adjacent bit-multiplier. These arrays are for connecting intermediate *x* inputs,

y inputs, carry, and partial products. For example xv(i, j) connects the x_o of a bit-multiplier at location *i, j* to *xi* of the bit-multiplier to its left.

```
ARCHITECTURE iterative OF array_multiplier IS
   COMPONENT bit_multiplier
      PORT (xi, yi, pi, ci : IN std_logic;
            xo, yo, po, co : OUT std_logic);
   END COMPONENT;
   CONSTANT n : INTEGER := x'LENGTH - 1;
   TYPE pair IS ARRAY (n+1 DOWNTO 0, n+1 DOWNTO 0) OF
                       std_logic;
   SIGNAL xv, yv, cv, pv : pair;
BEGIN
   rows: FOR i IN x'RANGE GENERATE
      cols: FOR j IN y'RANGE GENERATE
         cell: bit_multiplier PORT MAP
            (xv(i,j),   yv(i,j),   pv(i,j+1), cv(i,j),
             xv(i,j+1), yv(i+1,j), pv(i+1,j), cv(i,j+1));
      END GENERATE;
   END GENERATE;
   sides: FOR i IN x'RANGE GENERATE
      xv(i, 0) <= x(i);
      cv(i, 0) <= '0';
      pv(0, i+1) <= '0';
      pv(i+1, n+1) <= cv(i, n+1);
      yv(0, i) <= y(i);
      z(i) <= pv(i+1, 0);
      z(i + n+1) <= pv(n+1, i+1);
   END GENERATE;
END iterative;
```

Figure 8.38 Array Multiplier VHDL Description

The body of the iterative architecture of Figure 8.38 has *rows* and *cols* nested generate statements that wire n×n multiplier cells (*bit_multiplier* entities) together. This architecture also contains the *sides* generate statement that is responsible for wiring the inputs, the outputs, and the carry outputs (*cv*) to partial product inputs (*pv*). As shown in Figure 8.36, carry output of the left-most *bit_multiplier* of a row of the array connects to the product input of the left-most *bit_multiplier* of the next row.

8.3.2 Carry-Lookahead Adder

This section discusses a behavioral VHDL description of a carry-lookahead adder. The purpose of this VHDL description is for illustrating how carry-lookahead adders work, and not necessarily for synthesis. In a synthesis environment users select the add operation for synthesizing an adder. Depending on time and space constraints,

as well as the synthesis target library, synthesis tools decide on the type of adders to use.

In a ripple-carry adder, an adder cell (one-bit adder) uses the carry output from its previous cell to generate its sum and its carry output. This is unlike a carry-lookahead adder that generates its carry bits based on the bits of the data inputs (add operands), instead of relying on carry bits from the previous bits. A carry-lookahead adder is faster than a ripple-carry adder because independence of a carry bit on its previous bits eliminates the otherwise carry generation gate delays. On the other hand, a carry-lookahead adder requires more hardware for generating carry signals of each adder cell independent of the previous carry outputs.

Using a 4-bit example, the following shows the hardware of a carry-lookahead adder and illustrates how it works. In what follows, a_i and b_i are data inputs of bit i of an adder cell, and c_i is its carry input. c_{i+1} is the carry output and s_i is the sum.

The sum and carry outputs of a full-adder are written as shown below:

```
c_{i+1} = a_i · b_i + a_i · c_i + b_i · c_i
s_i     = a_i ⊕ b_i ⊕ c_i
```

The carry output can be rewritten as:

```
c_{i+1} = (a_i + b_i) · c_i + a_i · b_i
        = p_i · c_i + g_i
```

Where,

```
p_i   = a_i + b_i
```

and

```
g_i   = a_i · b_i
```

The p_i term is called *propagate* since if it is '1' it causes c_i to propagate to c_{i+1}. The g_i term is called *generate*, because it generates c_{i+1} if it is '1'. Using p_i, g_i, a_i, b_i and c_o, c_4 of a 4-bit carry-lookahead adder is calculated as shown below:

```
c_1   = p_0 · c_0 + g_0
c_2   = p_1 · c_1 + g_1
      = p_1 · p_0 · c_0 + p_1 · g_0 + g_1
c_3   = p_2 · c_2 + g_2
      = p_2 · p_1 · p_0 · c_0 + p_2 · p_1 · g_0 + p_2 · g_1 + g_2
```

$$c_4 = p_3 \cdot c_3 + g_3$$
$$= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot g_2 + g_2$$

Using the above carry outputs, the sum outputs are calculated as shown:

$$s_0 = a_0 \oplus b_0 \oplus c_0$$
$$s_1 = a_1 \oplus b_1 \oplus c_1$$
$$s_2 = a_2 \oplus b_2 \oplus c_2$$
$$s_3 = a_3 \oplus b_3 \oplus c_3$$

Since the carry expressions use p_i and g_i in two-level logic expressions, and p_i and g_i are directly calculated using adder inputs (a_i and b_i), gate delays do not accumulate in carry signals of a carry-lookahead adder. Thus such an adder is faster than a ripple-carry adder.

Figure 8.39 shows VHDL code of an unconstrained carry-lookahead adder. A for-loop statement generates all the carry outputs that are needed for the sum expressions. Carry of each adder uses *p(i)*, *g(i)* and carry bits of its previous adder. In this description, carry expressions are not expanded when used for next upper carry calculations. Interested readers are encouraged to rewrite this code to make all carry outputs in terms of p_i, g_i and adder inputs.

```
ENTITY c_l_adder IS
   PORT (a, b : IN std_logic_vector;
         cin : IN std_logic;
         s : OUT std_logic_vector;
         cout : OUT std_logic);
END ENTITY c_l_adder;
--
ARCHITECTURE behavioral OF c_l_adder IS
   SIGNAL p, g : std_logic_vector(a'RANGE);
   SIGNAL carry : std_logic_vector(a'LENGTH DOWNTO 0);
BEGIN
   g <= a AND b; p <= a XOR b;
   PROCESS (p, g, carry, cin) BEGIN
      carry(0) <= cin;
      FOR i IN a'REVERSE_RANGE LOOP
         carry(i+1) <= g(i) OR (p(i) AND carry(i));
      END LOOP;
      cout <= carry(a'LENGTH);
   END PROCESS;
   s <= p XOR carry(a'RANGE);
END ARCHITECTURE behavioral;
```

Figure 8.39 A Behavioral Unconstrained Carry-lookahead Adder

8.3.3   Synthesizable Booth Multiplier

This section shows an unconstrained synthesizable, signed, Booth multiplier. We first discuss how the Booth algorithm works and then show the VHDL description of the multiplier.

8.3.3.1   Booth Algorithm.   The Booth algorithm is for signed number 2's complement multiplication. When multiplying M by Q, starting from the right hand side of Q and assuming a '0' to the right of it, a two-bit window is formed. The algorithm considers the pair of bits in this window as two-bit 2's complement numbers. If this number is negative '1', i.e., "10" it subtracts M from adds $-M$ to the accumulated partial result, A, and then it shifts A. If this two-bit number is positive '1', i.e., "01", it adds M to A, and then it shifts A. If the two-bit number in the window is "00" or "11", it just shifts A without adding or subtracting.

When the above is done for a two-bit window of Q, the window is moved one bit position to the left, and the new two bits will be considered. For an n-bit multiplication, this process repeats n times. When done, the bits in A and those shifted to the right of A form the $2n$ bits of the result.

The above procedure is illustrated by use of the following 4-bit multiplication example.
We start with

```
M: 0110          -M: 1010
Q: 1101
A: 0000
```

Appending a '0' to the right of Q, and forming a 2-bit window results in:

```
M: 0110
Q: 110 1, 0
A: 0000
```

Since the above window contains "10", add –M to A, and A is shifted to the right. We also move Q window to the left to prepare it for the next cycle.

```
M: 0110
Q: 11 01 , 0
A: 1010       After add
A: 1101, 0    After shift
```

Since the *Q* window above contains "01", add *M* to *A* and shift. Also move the window:

```
M: 0110        -M: 1010
Q: 1|10|1, 0
A: 0011, 0     After add
A: 0001, 10    After shift
```

Since the *Q* window above contains "10", add –*M* to *A* and shift. Also move the window:

```
M: 0110        -M: 1010
Q: |11|01, 0
A: 1011, 10    After add
A: 1101, 110   After shift
```

Since the above *Q* window contains "11", no adding is to be done, and *A* is shifted to the right.

```
M: 0110        -M: 1010
Q: 1101, 0
A: 1110, 1110
```

After four shifts, the *A* register and its right shifted bits form the multiplication result.

8.3.3.2 Booth Multiplier VHDL Implementation. Figure 8.40 shows the synthesizable code of a Booth algorithm based on the above discussion. Except for a few minor differences that we discuss here, the VHDL code of this figure implements the procedure discussed above.

In the VHDL code of Figure 8.40, an n+1 bit register is used for *Q*. The extra bit is added to the right of *Q* in order to form the first 2-bit window. Upon start, *mp* concatenated with a '0' to its left is loaded into *Q*.

In our VHDL implementation, after each add-and-shift, instead of moving the 2-bit *Q* window to the left, the *Q* register is moved to the right. Furthermore, as left bits of *Q* are emptied as the result of this shifting, bits of *A* are moved into *Q* from the left. This way, when the multiplication process is completed, the result in is *A* and in the upper bits of *Q*.

```
ENTITY booth_mult IS
   PORT (mc, mp : IN std_logic_vector (7 downto 0);
         clk, start : IN std_logic;
         prod : OUT std_logic_vector (15 downto 0);
         busy : OUT boolean);
END ENTITY booth_mult;
--
ARCHITECTURE behavioral OF booth_mult IS
   SIGNAL A, M : std_logic_vector (mc'RANGE);
   SIGNAL Q : std_logic_vector (mc'LENGTH DOWNTO 0);
   SIGNAL sum, dif : std_logic_vector(mc'RANGE);
   SUBTYPE cnt IS INTEGER RANGE 0 TO mc'LENGTH;
   SIGNAL count : cnt := 0;
BEGIN
   sum <= A + M;
   dif <= A - M;
   prod <= A & Q (mc'LENGTH DOWNTO 1);
   busy <= (count < mc'LENGTH);
   Counter: PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF (start = '1') THEN count <= 0;
         ELSIF (count < mc'LENGTH) THEN count <= count + 1;
         END IF;
      END IF;
   END PROCESS;
   RegClocking: PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF (start = '1') THEN
            A <= (OTHERS => '0');
            M <= mc;
            Q <= mp & '0';
         ELSIF (count < mc'LENGTH) THEN
            CASE Q(1 DOWNTO 0) IS
               WHEN "01" =>          --ADD AND SHIFT
                  Q <= sum(0) & Q(Q'LEFT DOWNTO 1);
                  A <= sum(sum'LEFT) &
                       sum(sum'LEFT DOWNTO 1);
               WHEN "10" =>          --SUBTRACT AND SHIFT
                  Q <= dif(0) & Q(Q'LEFT DOWNTO 1);
                  A <= dif(dif'LEFT) &
                       dif(dif'LEFT DOWNTO 1);
               WHEN OTHERS =>        --SHIFT ONLY
                  Q <= A(0) & Q(Q'LEFT DOWNTO 1);
                  A <= A(A'LEFT) & A(A'LEFT DOWNTO 1);
            END CASE;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE behavioral;
```

Figure 8.40 Booth Algorithm VHDL Code

The VHDL code of Figure 8.40 has a clocked process for implementing a counter to keep track of the number of add-and-shift or shift operations. A subtype of INTEGER, i.e., *cnt*, that ranges between 0 and the number of multiplier bits is used for this counter.

The main task of multiplication is done in a clocked process that is responsible for loading proper data into *A*, *M* and *Q* registers. *M* is only loaded at the beginning of a multiplication session. In addition to being initialized when *star* is '1', *A* and *Q* registers are clocked with data that depends on the *Q(1 DOWNTO 0)* window. The *Q* register is shifted with each clock, and *A* is loaded with *sum*, *dif* or it is shifted to the right.

An 8-bit Booth multiplier synthesized with Altera's Quartus II uses 41 logic elements, 29 flip-flops, and 35 pins of a Cyclone II FPGA.

8.4 Components with Separate Control and Data Parts

The previous sections discussed various RT level components and discussed synthesizability and styles for better synthesis. In particular, the Booth algorithm of the previous section presented a good example for an RT level design that is complex enough that a good knowledge of RT level hardware is required for its proper design. In this design, the designer has to decide what registers are, what control states are, and how various combinational and sequential parts of a design are to be connected. The complete design of the Booth multiplier was done in one module without any design hierarchy.

For more complex designs, or for beginners designing moderately complex, RT level circuits, the method we used for the Booth multiplier may not result in an efficient hardware. A complex RT level design requires design partitioning, hierarchy, and top-down design methodology. In this section we elaborate on these topics. We use two designs to illustrate a systematic method for core design at the RT level. The First example is a shift-and-add multiplier, and the second is a small processor model. Both designs are synthesizable.
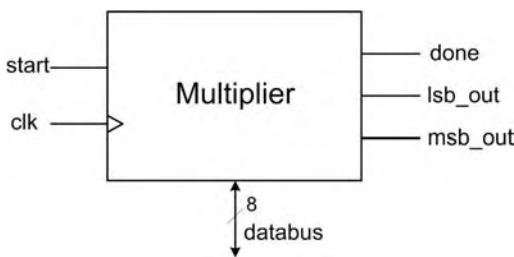
In these designs we will perform control-data partitioning, decide on data components and their control signals, specify interconnections of data components and their corresponding control signals, and finally design our controller.

8.4.1 Sequential Multiplier

Our first example of RT level system design is a shift-and-add sequential multiplier, with an 8-bit *A* and *B* inputs and a 16-bit result.

The block diagram of this circuit is shown in Figure 8.41. This multiplier has an 8-bit bi-directional I/O for inputting its *A* and *B* operands, and outputting its 16-bit output one byte at a time.

Multiplication begins with the *start* pulse. On the clock edge that *start* is '1', operand *A* is on the *databus* and in the next clock, this bus will contain operand *B*. The two operands appear on the bus in two consecutive clock pulses. After accepting these data inputs, the multiplier begins its multiplication process and when it is completed, it starts sending the result out on the *databus*. When the least-significant byte is placed on *databus*, the *lsb_out* output is issued, and for the most-significant byte, *msb_out* is issued. When both bytes are outputted, *done* becomes '1', and the multiplier is ready for another set of data. The multiplexed bi-directional *databus* is used to reduce the total number of pins of the multiplier.

Figure 8.41 Multiplier Block Diagram

Shift-and-add is a simple multiplication method that is slow, but efficient in use of hardware. In this method, depending on bit *i* of operand *A*, either operand *B* is added to the collected partial result and then shifted to the right (when bit *i* is **1**), or (when bit *i* is **0**) the collected partial result is shifted one place to the right without being added to *B*.

As in the Booth multiplier of the previous section, we use a 4-bit example to clarify the above procedure. As shown in Figure 8.42, *A*=**1001** and *B*=**1101** are to be multiplied. Initially at time 0, *A* is in a shift-register with a register for partial results (*P*) on its left.

At the time 0, because *A[0]* is **1**, the partial sum of *B+P* is calculated. This value is **01101** (shown in the upper part of time 1) and has 5 bits to consider carry. The right most bit of this partial sum is shifted into the *A* register, and the other bits replace the old value of *P*. When *A* is shifted, **0** moves into the *A[0]* position. This value is observed at time 1. At this time, because *A[0]* is **0**, **0000** + *P* is calculated (instead of *B+P*). This value is **00110**, the right most bit of which is shifted into *A*, and the rest replace *P*. This process repeats 4 times. At the end of the 4th cycle, the least significant 4 bits of the

multiplication result become available in *A* and the most significant bits in *P*. The example used here performed 9×13 and 117 is obtained as the result of this operation.
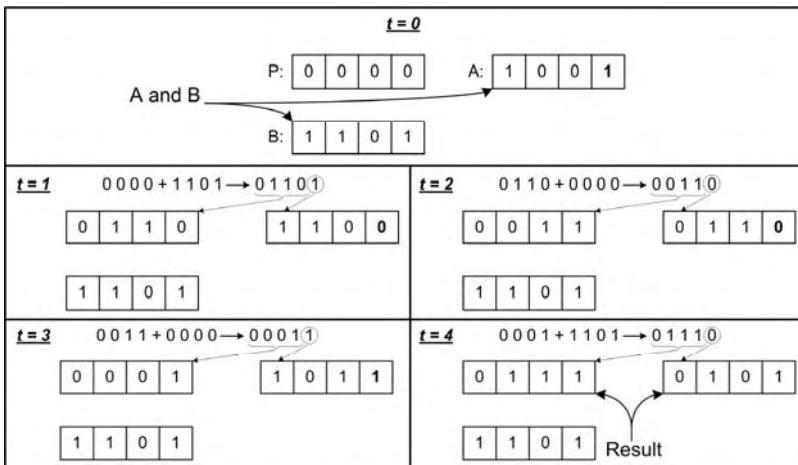
Figure 8.42 Hardware Oriented Multiplication Process

8.4.1.1   Sequential Multiplier Design.   The multiplication process discussed above justifies the hardware implementation that will be discussed here.

The multiplier has a datapath and a controller. The datapath consists of registers, logic units and their interconnecting busses. The controller is a state machine that issues specific signals for controlling data that gets clocked into data registers.
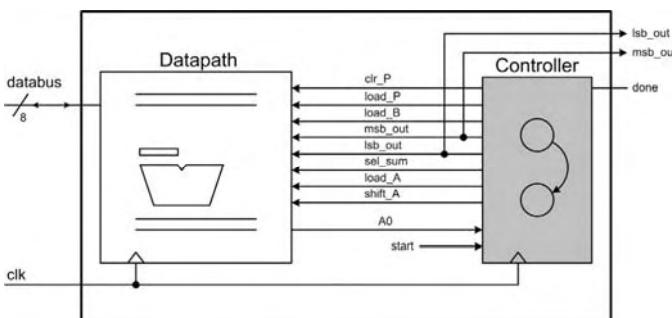
Figure 8.43 Datapath and Controller

As shown in Figure 8.43, the datapath register and the controller are triggered with the same clock signal. On the rising edge of a clock, the controller goes into a new state. In this state, several control sig-

nals are issued, and as a result, the components of the datapath start reacting to these signals. The time given for all activities of the data to stabilize is from one edge of the clock to another. Values that are propagated to the inputs of the datapath registers are clocked into these registers with every positive edge of the clock.

Figure 8.44 shows the datapath of the sequential multiplier. As shown, P and B are outputs of 8-bit registers and A is the output of an 8-bit shift-register. These components are implemented with process statements in the VHDL code of the multiplier. An adder, a multiplexer and two tri-state buffers constitute the other components of this datapath. These components are implemented with concurrent signal assignment statements.
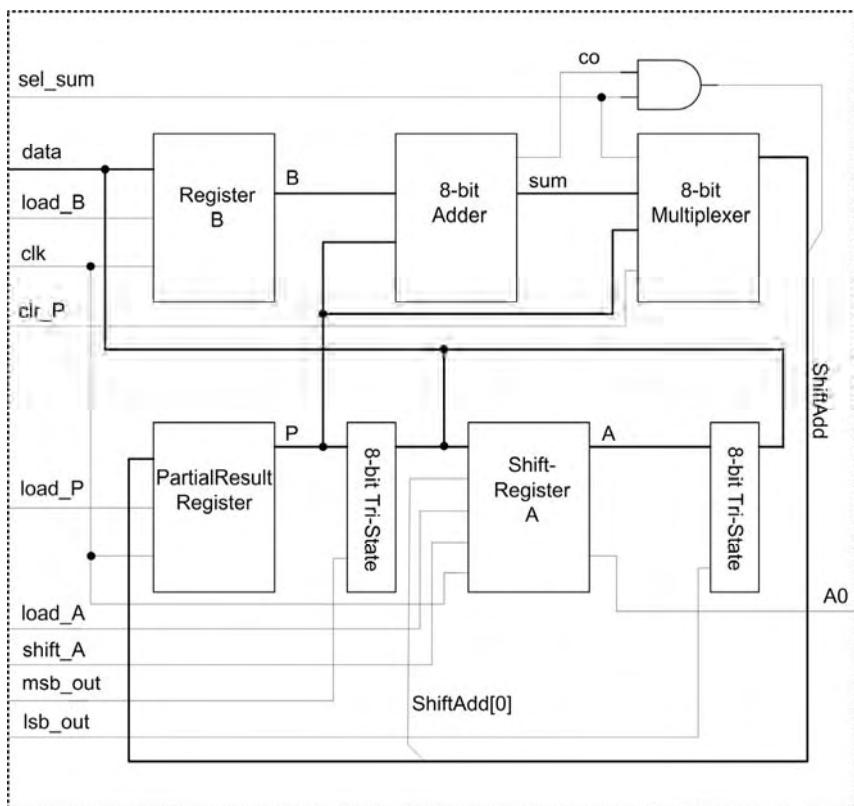
Figure 8.44 Multiplier Block Diagram

Control signals that are outputs of the controller and inputs of the datapath (Figure 8.43), are named according to their functionalities like loading registers, shifting, etc. These signals are shown in the corresponding blocks of Figure 8.44 next to the data component that they control.

The input *databus* connects to the inputs of *A* and *B* to load multiplier and multiplicand into these registers. This bi-directional bus is driven by the outputs of *P* and *A* through tri-state buffers. These tri-states become active when multiplication result is ready.

The output from *B* and *P* are added to form *co* and *sum* to be put in *P* if adding is to take place. Otherwise, *P* is put on *ShiftAdd* to be shifted, while being put back into *P*. *ShiftAdd* is the multiplexer output that selects *sum* or *P*. The *sel_sum* control input determines if *sum* or *P* is to go on the multiplexer output.

The AND function shown in Figure 8.44 selects carry-out from the adder or **0** depending on the value of *sel_sum* control input. This value is concatenated to the left of the multiplexer output to form a 9-bit vector. This vector has *P+B* or *P* with a carry to its left. The rightmost bit of this 9-bit vector is split and goes into the serial input of the shift-register that contains *A*, and the other eight bits go into register *P*. Note that concatenation of the AND outut to the left of the multiplexer output and splitting the right bit from this 9-bit vector, effectively produces a shifted result that is clocked into *P*.

8.4.1.2 Sequential Multiplier Datapath. The complete datapath VHDL description of the multiplier is shown in Figure 8.45. VHDL concurrent signal assignments and process statements are used to describe components of the datapath. As shown here, the first two process statements represent registers *B* and *P*, for operand *B* and the partial result, *P*. The process statement that comes next in this figure represents an 8-bit shift register.

This shift register is used for operand *A* of the multiplier, and it either loads *A* with *data* (controlled by *load_A*) or shifts its contents (controlled by *shift_A*). The *op* 2-bit vector concatenates these control signals.

Signal assignments and conditional signal assignments in the datapath of Figure 8.45 are used for describing combinational parts of this part. The adder for adding *P* and *B* uses the add operator and puts the result on *result*, the eight right-most bits of which form the *sum* vector and its left-most bit is the carry output, *co*.

The conditional signal assignment in Figure 8.45 implements a multiplexer that when enabled (*clr_P* is '0'), selects *P* or *sum* to be clocked into *P*. Other conditional signal assignments in Figure 8.45 are for tri-state buffers at the outputs of *P* and *A* registers (see Figure 8.44).

```
ENTITY datapath IS
   PORT (clk, clr_P, load_P, load_B : IN std_logic;
         msb_out, lsb_out, sel_sum : IN std_logic;
         load_A, shift_A : IN std_logic;
      data : INOUT std_logic_vector (7 DOWNTO 0);
      A0 : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF datapath IS
   SIGNAL sum, ShiftAdd : std_logic_vector (7 DOWNTO 0);
   SIGNAL A, B, P : std_logic_vector (7 DOWNTO 0);
   SIGNAL co : std_logic;
   SIGNAL op : std_logic_vector (1 DOWNTO 0);
   SIGNAL result : std_logic_vector (8 DOWNTO 0);
BEGIN
   PROCESS (clk) BEGIN
      IF(clk = '0' AND clk'EVENT) THEN
         IF (load_B = '1') THEN B <= data;
         END IF;
      END IF;
   END PROCESS;
   --
   PROCESS (clk) BEGIN
      IF(clk = '0' AND clk'EVENT) THEN
         IF (load_P = '1') THEN
            P <= (co AND sel_sum) & ShiftAdd (7 DOWNTO 1);
         END IF;
      END IF;
   END PROCESS;
   --
   PROCESS (clk) BEGIN
      IF(clk = '0' AND clk'EVENT) THEN
         CASE op IS
            WHEN "01" => A <=  ShiftAdd(0) & A(7 DOWNTO 1);
            WHEN "10" => A <= data;
            WHEN OTHERS => A <= A;
         END CASE;
      END IF;
   END PROCESS;

   result <= ('0'&P) + ('0'&B);
   co <= result(8);
   sum <= result(7 DOWNTO 0);

   A0 <= A(0);
   ShiftAdd <= (OTHERS => '0') WHEN clr_P = '1' ELSE
               P WHEN sel_sum = '0' ELSE sum;
   data <= A WHEN lsb_out = '1' ELSE (OTHERS => 'Z');
   data <= P WHEN msb_out = '1' ELSE (OTHERS => 'Z');
   op <= load_A & shift_A;
END ARCHITECTURE procedural;
```

Figure 8.45 Shift-and-add Multiplier Datapath

8.4.1.3 Multiplier Controller. The multiplier controller is a finite state machine that has two starting states, eight multiplication states, and two ending states. Figure 8.46 shows the VHDL code of this circuit in which *state* is declared as an enumeration type of the states of the multiplier controller. The controller uses a combinational process for issuing control signals and a sequential process for clocking the control registers. The current state of the controller is saved in the *current* signal of type *state*.

The VHDL code of the controller consists of a *sequential* process for register clocking and a *combinational* process for activating the control signals.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY controller IS
   PORT (clk, start, A0 : IN std_logic;
         clr_P, load_P, load_B : OUT std_logic;
         msb_out, lsb_out, sel_sum : OUT std_logic;
         load_A, Shift_A, done : OUT std_logic);
END ENTITY;
--
ARCHITECTURE procedural OF controller IS
   TYPE state IS (idle, init,
                  m1, m2, m3, m4, m5, m6, m7, m8,
                  rslt1, rslt2);
   SIGNAL current : state;
BEGIN
   sequential: PROCESS (clk) BEGIN
      IF (clk = '0' AND clk'EVENT) THEN
         CASE current IS
            WHEN idle =>
               IF start = '0' THEN
                  current <= idle;
               ELSE
                  current <= init;
               END IF;
            WHEN init =>
               current <= m1;
            WHEN m1 | m2 | m3 | m4 | m5 | m6 | m7 | m8 =>
               current <= state'SUCC(current);
            WHEN rslt1 =>
               current <= rslt2;
            WHEN rslt2 =>
               current <= idle;
            WHEN OTHERS =>
               current <= idle;
         END CASE;
      END IF;
   END PROCESS; --                                     Continued
```

```
    combinational: PROCESS (current, start, A0) BEGIN
        clr_P <= '0'; load_P <= '0';
        load_B <= '0';
        msb_out <= '0'; lsb_out <= '0';
        sel_sum <= '0'; load_A <= '0';
        Shift_A <= '0'; done <= '0';
        CASE current IS
            WHEN idle =>
                IF start = '0' THEN
                    done <= '1';
                ELSE
                    load_A <= '1';
                    clr_P<= '1';
                    load_P <= '1';
                END IF;
            WHEN init =>
                load_B <= '1';
            WHEN m1 | m2 | m3 | m4 | m5 | m6 | m7 | m8 =>
                Shift_A <= '1';
                load_P <= '1';
                IF (A0 = '1') THEN
                    sel_sum <= '1';
                END IF;
            WHEN rslt1 =>
                lsb_out <= '1';
            WHEN rslt2 =>
                msb_out <= '1';
            WHEN OTHERS =>
                clr_P <= '0'; load_P <= '0';
                load_B <= '0'; msb_out <= '0';
                lsb_out <= '0'; sel_sum <= '0';
                load_A <= '0'; Shift_A <= '0';
                done <= '0';
        END CASE;
    END PROCESS;
END ARCHITECTURE procedural;
```

Figure 8.46 Multiplier Controller

The *sequential* process of Figure 8.46 is sensitive to the positive edge of the clock and is responsible for state transitions. This process has a case-statement that is parallel with that of the combinational process.

The controller declares *datapath* control signals as outputs and issues them in its *combinational* process. Decisions made in this process statement are based on *current*. When *current* is *idle* and *start* is '0', the *done* output remains high. In this state if *start* becomes '1', control signals *load_A*, *clr_P* and *load_P* become active to load *A* with *databus* and clear the *P* register. Clearing *P* requires *clr_P* to put **0**'s on the *ShiftAdd* of the datapath and load the '0's into *P* by asserting

load_P. In *m1* to *m8* states, *A* is shifted, *P* is loaded, and if *A0* is '1', *sel_sum* is asserted. As discussed in relation to *datapath*, *sel_sum* controls shifted *P+B* (or shifted *P*) to go into *P*. In the result states, *lsb_out* and *msb_out* are asserted in two consecutive clocks in order to put *A* and *P* on the *data* bus respectively.

8.4.1.4 Top-level Multiplier. The VHDL code of the multiplier consisting of instantiation of datapath and controller of Figures 8.43 and 8.44 is shown in Figure 8.47. This description is synthesizable.

```
ENTITY Multiplier IS
   PORT (clk, start : IN std_logic;
         databus : INOUT std_logic_vector (7 DOWNTO 0);
      lsb_out, msb_out, done : OUT std_logic);
END ENTITY;
--
ARCHITECTURE structural OF Multiplier IS
   SIGNAL clr_P, load_P, load_B, msb_out_t, A0 : std_logic;
   SIGNAL lsb_out_t, sel_sum, load_A, Shift_A : std_logic;
BEGIN
   dpu : ENTITY WORK.datapath(procedural)
            PORT MAP (clk, clr_P, load_P, load_B,
                      msb_out_t, lsb_out_t, sel_sum,
                      load_A, Shift_A, databus, A0 );
   cu : ENTITY WORK.controller(procedural)
           PORT MAP (clk, start, A0, clr_P, load_P, load_B,
                      msb_out_t, lsb_out_t, sel_sum,
                      load_A, Shift_A, done );
   msb_out <= msb_out_t;
   lsb_out <= lsb_out_t;
END ARCHITECTURE structural;
```

Figure 8.47 Top-level Multiplier Module

8.4.2 von Neumann Computer Model

The previous section was our first step in showing how a complete top-down design could be put together in VHDL. In this section we take our presentation of complete system design one step further, and present design, and implementation for a hardware based on von Neumann computer model.

8.4.2.1 Processor and Memory Model. The von Neumann computer model is based on a processor using instructions and data from a single memory. Using a sequencer (see Figure 8.48), the processor fetches instructions from its memory. An instruction has an opcode indicating the function it is supposed to perform. Using this opcode,

the processor performs its proper operation. Such an operation may involve reading or writing data from or to the memory, for which the same memory as that of the instructions will be used.
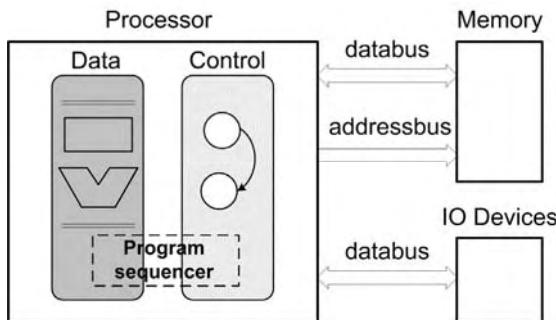
Figure 8.48   von Neumann Process Model

Our design example in this section is a simple von Neumann model with memory accessing mechanism for instructions and data. This design is at the RT Level and has a separate datapath and control unit. In designing the datapath, we partition it into its subcomponents and describe each subcomponent separately. This includes the instruction sequencer part of the datapath that is the program counter component.

8.4.2.2 Processor Model Specification. The example that we use is a simple adding machine, which we refer to as *AddingCPU*. It must be mentioned, that we are using this example to demonstrate our design methodology. This specific example has very little practical value, if it were to be designed for a real application, much simpler coding than what we are presenting here could be used. The techniques presented here will be used for the design and test of the processor of the next section.

Our Adding CPU reads **Load**, **Add**, **Store**, and **Jump** instructions from its memory, and depending on the instruction it reads, loads data, performs addition, stores data into memory, or jumps to another memory location. Figure 8.49 shows the overall structure of this adding machine.

The circuit shown has a 6-bit address bus to address the memory for read and write operations. The 8-bit data bus of this machine is used for data in and out of the machine. Control signals, *reset*, *rd_mem* and *wr_mem* are used for reseting, memory read, and memory write operations.
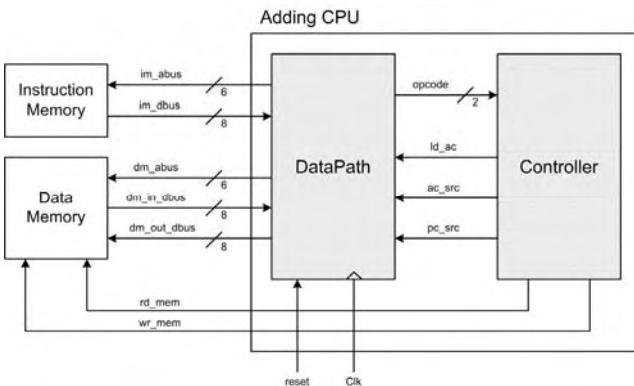
Figure 8.49 Interface of the Adding CPU Example

The machine starts reading its memory from location 0. An 8-bit word fetched from the memory consists of a 2-bit opcode and a 6-bit data or address. This field is either an immediate data or a memory address where the operand of the fetched instruction is. Figure 8.50 shows our machine's opcodes and instruction format.
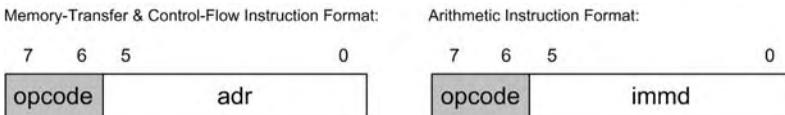
Figure 8.50 Instruction Format

The Adding CPU has a main register called *AC* (accumulator). The **Load** instruction directs the machine to load the addressed data from the memory into *AC*. The **Store** instruction causes contents of *AC* to be written into the addressed location in the memory. The operand of the **Add** instruction is *immd* (immediate). This instruction adds *immd* to the present contents of *AC* and puts the result back into *AC*. The **Jump** instruction loads the 6-bit address into the program counter of our machine, causing the next instruction to be fetched from this address.

8.4.2.3 Designing the Adding CPU. The first step in the design of our adding machine is to decide on its data and control partitioning and decide what goes into its data part and what behavior is expected from its controller.

The datapath of the design has the *AC* register for keeping data to operate on, the *PC* register to keep track of the address being fetched, and an adder unit to perform the addition. In addition, the datapath has an instruction register (*IR*) for storing the most recently

fetched instruction. Data registers are clocked with the same clock as the controller.

The controller part is a state machine that looks at the opcode of the instruction in *IR* and decides on how data is to be routed.

8.4.2.4 Design of Datapath. An indicated above, the main components of the datapath of our design are *AC, PC* (program counter), *IR*, and an *ALU*. Detailed operation of these components will be decided once we decide on the architecture that incorporates them.

Given the Adding CPU description of Section 8.4.2.4, the bussing shown in Figure 8.51 is appropriate for handling the necessary operations mentioned in this section. As shown here, the datapath has an internal *dbus* bus. The external bidirectional *data_bus* drives and is driven by *dbus*. This bus connects to the input of *IR* in order to bring instruction read from the memory into this register.
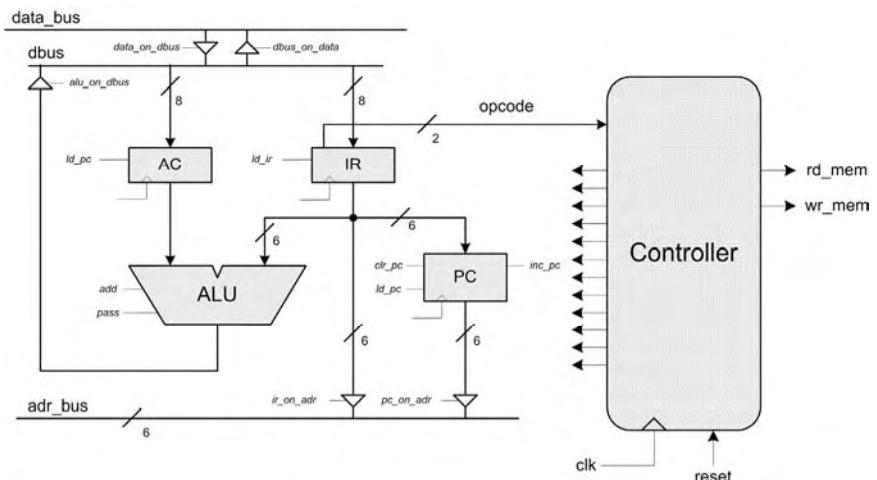
Figure 8.51 Architectural Design of our Adding Machine

IR has a load input (*ld_ir*) that is activated to cause it load from *data_bus*. Similarly, this bus connects to *AC* to bring data read from the memory into this register. The control signal for loading *AC* is *ld_ac*. This control signal is issued when the **Load** instruction is being expected. *PC* has three control signals *ld_pc*, *inc_pc* and *ctr_pc* to load, clear and increment it, respectively. The right most 6-bits of *IR* connect to the input of *PC* for execution of the **Jump** instruction.

For executing the **Store** instruction, *AC* is placed on the left input of *ALU* and from there to *dbus*, which eventually goes on *data_bus*. At the same time, *IR* is placed on *addr_bus* to specify the address in which *AC* data is to be stored. For this purpose, the adder

unit (ALU) has a *pass* control input to make it pass its left input data to its output.

Execution of the **Add** instruction is done by taking one of the add operands from *AC* and the other from *IR*. For this instruction, activating the *add* control input of *ALU* causes the ALU to perform addition.

The simple bussing structure described above facilitates execution of all four instructions of our simple Adding CPU. When a bus has more than one source driving it, e.g., *IR* and *PC* driving *addr_bus*, control signals from the controller select the source.

8.4.2.5 Control Part Design.

After the design of the datapath and figuring control signals and their role in activities in the datapath, the design of the controller becomes a simple matter. The block diagram of this part is shown in Figure 8.52.
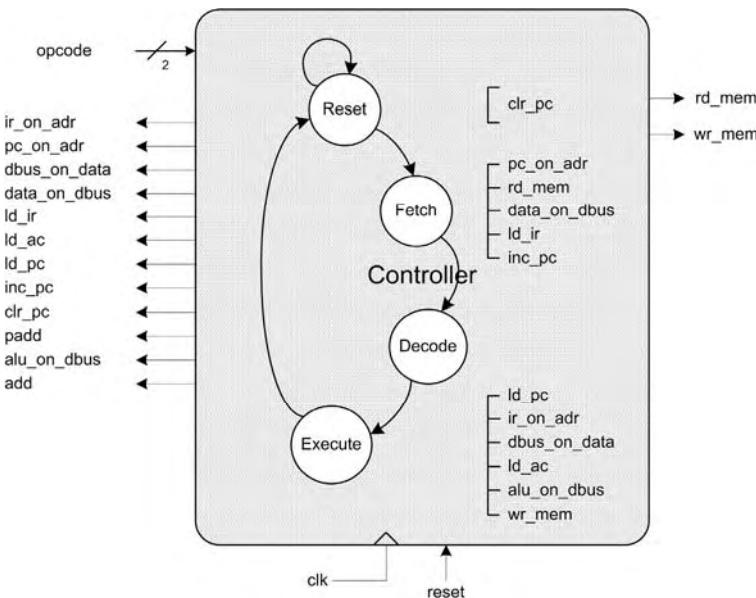
Figure 8.52 Controller of Adding CPU

The controller of our simple von Neumann machine has four states, *Reset*, *Fetch*, *Decode* and *Execute*. As the machine cycles through these states, various control signals are issued. In state *Reset*, for example, the *clr_pc* control signal is issued. State *Fetch* issues *pc_on_adr*, *rd_mem*, *data_on_dbus*, *ld_ir*, and *inc_pc*, to read memory from the present *PC* location, route it to *IR*, load it into *IR*, and increment *PC* for the next memory fetch. Depending on opcode bits, that are the controller inputs, the *Execute* state of the controller is-

sues control signals for execution of **Load**, **Store**, **Add** and **Jump** instructions. The next section discusses details of the controller signals and their role in execution of these instructions.

8.4.2.6 *AddingCPU* **VHDL Description.** We develop the complete VHDL code of our simple adding machine by developing code for the blocks of Figure 8.51. We first describe components of the datapath, and then will form the VHDL code of the datapath by instantiating and wiring these components. The controller will be described next, using a state machine coding style. At the end, the description of our small von Neumann example will be completed by wiring datapath and controller in a top-level VHDL module.

8.4.2.7 Data Components. Datapath components of Adding CPU could be described by process and assignment statements directly in the datapath description of the machine. Recall that this coding technique was used for the multiplier example of the previous section. However, in this example we are taking a more general and extendable approach. We describe our components so that they can be independently simulated and tested. This is necessary for large designs with more complex components. The approach presented here will be used in describing our larger machine in Chapter 10. VHDL code for *PC*, *AC*, *IR*, and *ALU* modules are shown in Figure 8.53.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY AC IS
   PORT (data_in : IN std_logic_vector(7 DOWNTO 0);
         load, clk : IN std_logic;
         data_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE procedural OF AC IS BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF load = '1' THEN
            data_out <= data_in;
         END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;

ENTITY IR IS
   PORT (data_in : IN std_logic_vector(7 DOWNTO 0);
         load, clk : IN std_logic;
         data_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ; --                                        Continued
```

```
ARCHITECTURE procedural OF IR IS BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF load = '1' THEN data_out <= data_in; END IF;
      END IF;
   END PROCESS;
END ARCHITECTURE;

ENTITY PC IS
   PORT (data_in : IN std_logic_vector(5 DOWNTO 0);
         load, inc, clr, clk : IN std_logic;
         data_out : OUT std_logic_vector(5 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE procedural OF PC IS
   SIGNAL pc : std_logic_vector(5 DOWNTO 0);
BEGIN
   PROCESS (clk) BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF clr = '1' THEN
            pc <= (OTHERS => '0');
         ELSIF load = '1' THEN
            pc <= data_in;
         ELSIF inc = '1' THEN
            pc <= pc + 1;
         END IF;
      END IF;
   END PROCESS;
   data_out <= pc;
END ARCHITECTURE;

ENTITY ALU IS
   PORT (a, b : IN std_logic_vector(7 DOWNTO 0);
         pass, add : IN std_logic;
         alu_out : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE functional OF ALU IS
   SIGNAL alu_res : std_logic_vector(7 DOWNTO 0);
BEGIN
   PROCESS (a, b, pass, add) BEGIN
      IF pass = '1' THEN
         alu_res <= a;
      ELSIF add = '1' THEN
         alu_res <= a + b;
      ELSE
         alu_res <= (OTHERS => '0');
      END IF;
   END PROCESS;
   alu_out <= alu_res;
END ARCHITECTURE;
```

Figure 8.53 Datapath Components of the Adding Machine

Accumulator (*AC*) and instruction register (*IR*) are simple registers with *load* enable control inputs. These inputs are driven by control signals coming from the controller through datapath ports. The program counter (*PC*) is a counter with parallel load, increment and clear capabilities. As shown, this component has three control signals to control its functionality. The ALU (the last module in Figure 8.53) is a combinational logic with *pass* and *add* control inputs. If *pass* is **1**, the *a* input goes on the output, and if *add* is **1**, the ALU output becomes the sum of *a* and *b*. Codes presented above are synthesizable and individually testable. These parts are istantiated in the datapath of our machine.

8.4.2.8 *DataPath* **Description.**   Figure 8.54 shows the datapath description of Adding CPU. The module name for this description is *DataPath* and it corresponds to the left-hand side of Figure 8.51. When studying the discussion below and the VHDL code of the datapath, the reader is encouraged to consider Figure 8.51, and make correspondence between VHDL signals and constructs with graphical notations of this figure.

The inputs of the VHDL code of Figure 8.54 are control signals coming from the controller, and the bidirectional *data_bus*. The outputs of this module are the opcode and the address bus. The opcode goes out to the controller and the address bus goes to the memory for operand and instruction fetch.

Following the input and output declarations, in the *DataPath* entity, the structural architecture of this entity declares internal datapath busses and signals. As shown, these declarations are followed by instantiation of data components, *IR*, *PC*, *AC* and *ALU*. Interconnection of these components are done through wires and busses declared by signal declarations of *std_logic_vector* type.

Control signals responsible for loading and incrementing registers and controlling *ALU* function connect to the control inputs of *IR*, *PC*, *AC*, and *ALU*.

In the last part of *datapath*, bus assignments take place. We use bus control signals coming from the controller to drive a left-hand-side bus either with one of its sources or high impedance. For example, *pc_on_adr* control signal either puts *PC* output (*pc_out*) or all Z's on *adr_bus*. The *dbus* bus is declared to connect to the external bidirectional *data_bus*. Two assignments are made to *dbus* using *alu_on_dbus* and *data_on_dbus* control signals. Placement of this intermediate bus on the external data bus of the datapath (*data_bus*) is controlled by *dbus_on_data* control signal. The last statement shown in Figure 8.54 places most significant *IR* bits on the *op_code* output of *DataPath* that goes out to the controller.

Although we have used tri-state busses, when synthesizing this circuit, we can direct our synthesis tool to use AND-OR or multiplexers to implement these busses.

```
ENTITY datapath IS
   PORT (ir_on_adr, pc_on_adr, dbus_on_data : IN std_logic;
         data_on_dbus, ld_ir, ld_ac, ld_pc : IN std_logic;
         inc_pc, clr_pc,
            pass, add, alu_on_dbus, clk : IN std_logic;
         adr_bus : OUT std_logic_vector(5 DOWNTO 0);
         op_code : OUT std_logic_vector(1 DOWNTO 0);
         data_bus : INOUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE structural OF datapath IS
   SIGNAL dbus, ir_out, a_side :
                  std_logic_vector(7 DOWNTO 0);
   SIGNAL alu_out, b_side : std_logic_vector(7 DOWNTO 0);
   SIGNAL pc_out : std_logic_vector(5 DOWNTO 0);
BEGIN
   IR :  ENTITY WORK.IR(procedural)
         PORT MAP (dbus, ld_ir, clk, ir_out);
   PC :  ENTITY WORK.PC(procedural)
         PORT MAP (ir_out(5 DOWNTO 0), ld_pc, inc_pc,
                     clr_pc, clk, pc_out);
   AC :  ENTITY WORK.AC(procedural)
         PORT MAP (dbus, ld_ac, clk, a_side);
   ALU : ENTITY WORK.ALU(functional)
         PORT MAP (a_side, b_side, pass, add, alu_out );

   b_side <= '0'&'0'&ir_out(5 DOWNTO 0);

   adr_bus <= ir_out(5 DOWNTO 0) WHEN ir_on_adr = '1' ELSE
            (OTHERS => 'Z');
   adr_bus <= pc_out WHEN pc_on_adr = '1' ELSE
            (OTHERS => 'Z');
   dbus <= alu_out WHEN alu_on_dbus = '1' ELSE
          (OTHERS => 'Z');
   data_bus <= dbus WHEN dbus_on_data = '1' ELSE
              (OTHERS => 'Z');
   dbus <= data_bus WHEN data_on_dbus = '1' ELSE
          (OTHERS => 'Z');
   op_code <= ir_out(7 DOWNTO 6);

END ARCHITECTURE;
```

Figure 8.54 Adding CPU Datapath Description

```
ENTITY controller IS
   PORT (rst, clk : IN std_logic;
         op_code : IN std_logic_vector(1 DOWNTO 0);
         rd_mem, wr_mem : OUT std_logic;
         ir_on_adr, pc_on_adr : OUT std_logic;
         dbus_on_data, data_on_dbus, ld_ir : OUT std_logic;
         ld_ac, ld_pc, inc_pc, clr_pc,pass : OUT std_logic;
         add, alu_on_dbus : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF controller IS
   TYPE state IS (Reset, Fetch, Decode, Execute);
   SIGNAL present_state, next_state : state;
BEGIN
   PROCESS (clk)--Sequential
   BEGIN
      IF clk = '1' AND clk'EVENT THEN
         IF rst = '1' THEN
            present_state <= Reset;
         ELSE
            present_state <= next_state;
         END IF;
      END IF;
   END PROCESS;
   --
   PROCESS (present_state, rst)--Combinational
   BEGIN
      rd_mem <= '0'; wr_mem <= '0'; ir_on_adr <= '0';
      pc_on_adr <= '0'; dbus_on_data <= '0';
      data_on_dbus <= '0'; ld_ir <= '0'; pass <= '0';
      ld_ac <= '0'; ld_pc <= '0'; inc_pc <= '0';
      clr_pc <= '0'; add <= '0'; alu_on_dbus <= '0';
      CASE present_state IS
         WHEN Reset  =>
            IF rst = '1' THEN
               next_state <= Reset;
            ELSE
               next_state <= Fetch;
            END IF;
            clr_pc <= '1';
         WHEN Fetch  =>
            next_state <= Decode;
            pc_on_adr <= '1';
            rd_mem <= '1';
            data_on_dbus <= '1';
            ld_ir <= '1';
            inc_pc <= '1';
         WHEN Decode  =>
            next_state <= Execute;

         --                                      Continued
```

```
        WHEN Execute =>
            next_state <= Fetch;
            CASE  op_code IS
                WHEN "00" =>
                    ir_on_adr <= '1'; rd_mem <= '1';
                    data_on_dbus <= '1'; ld_ac <= '1';
                WHEN "01" =>
                    dbus_on_data <= '1'; alu_on_dbus <= '1';
                    pass <= '1'; wr_mem <= '1';
                    ir_on_adr <= '1';
                WHEN "10" =>
                    ld_pc <= '1';
                WHEN "11" =>
                    add <= '1'; alu_on_dbus <= '1';
                    ld_ac <= '1';
                WHEN OTHERS =>
                    rd_mem <= '0'; pc_on_adr <= '0';
                    pass <= '0';
                    ir_on_adr <= '0'; wr_mem <= '0';
                    ld_ac <= '0';
                    dbus_on_data <= '0'; data_on_dbus <= '0';
                    ld_ir <= '0'; alu_on_dbus <= '0';
                    add <= '0';
                    inc_pc <= '0'; clr_pc <= '0';
                    ld_pc <= '0';
            END CASE;
        WHEN OTHERS => next_state <= Reset;
      END CASE;
   END PROCESS;
END ARCHITECTURE;
```

Figure 8.55 Controller VHDL Code

8.4.2.9 Controller Description. The controller code for our adding
machine example is shown in Figure 8.55. This code corresponds to
the right hand side of Figure 8.51 which is shown in more details in
Figure 8.52. In addition to *clk* and *reset*, the controller has the
op_code input that is driven by *IR* and comes to the controller from
the datapath entity (see Figure 8.51).

The sequencing of control states is implemented by a Huffman
style VHDL code. In this style, a process statement handles assign-
ment of values to *present_state*, and another process statement uses
this register output as the input of a combinational logic determining
next_state. This combinational block also sets values to control signals
that are outputs of the controller.

The former process statement synthesizes as a register with ac-
tive high *reset*, and the latter, (i.e., *combinational*) synthesizes to a
combinational block. This statement uses *present_state* and *reset* on
its sensitivity list. For synthesis purposes and to avoid output latches,

all outputs of this block, that are the control signals, are set to their inactive, '0', values. In the body of the *combinational* process a case statement checks *present_state* against the states of the machine (*Reset*, *Fetch*, *Decode*, and *Execute*), and activates the proper control signals.

The *Reset* state activates *clr_pc* to clear *PC* and sets *Fetch* as the next state of the machine. In the *Fetch* state, *pc_on_adr*, *rd_mem*, *data_on_dbus*, *ld_ir*, and *inc_pc* become active, and *Decode* is set to become the next state of the machine. By activating *pc_on_adr* and *rd_mem*, the *PC* output goes on the memory address and a read operation is issued. Assuming the memory responds in the same clock, contents of memory at the *PC* address will be put on *data_bus*. Issuance of *data_on_dbus* puts the contents of this bus on the internal *dbus* of *datapath*. This bus is connected to the input of *IR* and issuance of *ld_ir* loads its contents into this register. The next state of the controller is *Decode*, that makes the new contents of *IR* available for the controller. In the *Execute* state, a newly fetched instruction in *IR* decides on control signals to issue to execute the instruction.

In the *Execute* state, *op_code* is used in a case expression to decide on control signals to issue depending on the opcode of the fetched instruction. The case choices in this statement are four opcode values of **00**, **01**, **10** and **11** that correspond to **Load**, **Store**, **Jump** and **Add** instruction.

For **Load**, *ir_on_adr*, *rd_mem*, *data_on_dbus* and *ld_ac* are issued. These control signals cause the address from *IR* to be placed on the *adr_bus* address bus, memory read to take place, and data from memory to be loaded into *AC*. Data from the memory come through *data_bus* onto *dbus* of *DataPath* by the control signal *data_on_dbus*.

Controller executes the **Store** instruction by issuing *pass*, *ir_on_adr*, *alu_on_dbus*, *dbus_on_data* and *wr_mem*. As shown in Figure 8.51, these signals take contents of *AC* to the input bus of the memory (i.e., *data_bus*), and *wr_mem* causes the writing into the memory to take place. Note that *pass* causes *AC* to pass through *ALU* unchanged.

The **Jump** instruction is executed by enabling *PC* load input, which takes the jump address from *IR* (see Figure 8.51).

The last instruction of this machine is **Add**, for execution of which, *add*, *alu_on_dbus*, and *ld_ac* are issued. This instruction adds data in the upper six bits of *IR* with *AC* and loads the result into *AC*. The *add* control signal instructs *ALU* to add its two inputs; the *alu_on_dbus* puts this output on the internal datapath *dbus*; and the *ld_ac* causes *AC* to be loaded with the result of addition.

8.4.2.10 The Complete Machine. The top-level architecture for our adding machine example is shown in Figure 8.56. In the *structural*

architecture of the *AddingCPU*, *DataPath* and *Controller* modules are instantiated. Port connections of the *Controller* include its output control signals, the opcode input from *DataPath* and the reset external input. Port connections of *DataPath* consist of *adr_bus* and *data_bus* external busses, opcode output, and control signal inputs.

```
ENTITY addingCPU IS
   PORT (reset, clk : IN std_logic;
         adr_bus : OUT std_logic_vector(5 DOWNTO 0);
         rd_mem, wr_mem : OUT std_logic;
         data_bus : INOUT std_logic_vector(7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE structural OF addingCPU IS
   SIGNAL ir_on_adr, pc_on_adr, dbus_on_data : std_logic;
   SIGNAL data_on_dbus, ld_ir, ld_ac, ld_pc : std_logic;
   SIGNAL inc_pc, clr_pc : std_logic;
   SIGNAL pass, add, alu_on_dbus : std_logic;
   SIGNAL op_code : std_logic_vector(1 DOWNTO 0);
BEGIN
   CU: ENTITY WORK.Controller
       PORT MAP (reset, clk, op_code, rd_mem,
                 wr_mem, ir_on_adr, pc_on_adr,
                 dbus_on_data, data_on_dbus,
                 ld_ir, ld_ac, ld_pc, inc_pc,
                 clr_pc, pass, add, alu_on_dbus );
   DP: ENTITY WORK.DataPath
       PORT MAP (ir_on_adr, pc_on_adr, dbus_on_data,
                 data_on_dbus,ld_ir, ld_ac, ld_pc,
                 inc_pc, clr_pc, pass, add,
                 alu_on_dbus, clk, adr_bus, op_code,
                 data_bus );
END ARCHITECTURE;
```

Figure 8.56 AddingCPU Top-level Description

8.5 Summary

This chapter presented VHDL code and descriptions for several hardware components. We emphasized on synthesizable cores, but also considered situations that a core model was to be for evaluation purposes only. In the first sections of this chapter individual stand-alone component descriptions were discussed. In the RT Level section we showed design partitioning and putting sub-components of a system together for formation of complete systems. In the next chapter test techniques will be discussed.

Problems

8.1 Write a generic unconstrained n to 2^n decoder; use *std_logic*. The decoder has an active low enable input an n bit input and a 2^n output. The data inputs and the outputs are active high. When instantiated, this decoder expands to its required size. Show an example of the usage of this decoder.

8.2 Write a push-pop stack model using the VHDL access type. Push is done after a clock pulse, and pop is done first and then clocked. Use the access type so that the stack can be made with no limit. Data on the stack is a record of an 8-bit BIT_VECTOR and a TIME field. The stack has a clock input, but you will only be mimicking the clocking since the access type does not require a clock.

8.3 Write complete VHDL code of a stack that operates on a memory block with 512 word address space and 16-bit word length. The memory has a bidirectional input-output bus, an address bus and appropriate control inputs as described next. The memory is clocked with a *Clk* input and has *Read* and *Write* inputs. When *Read* is 1, the addressed data will be read and when *Write* is 1, data on its data bus will be written to the addressed location. The stack using this memory has *Push*, *Pop*, *Tos* inputs and *Full* and *Empty* outputs, as well as a 16-bit *Datain* input and a 16-bit *Dataout* output. *Push* pushes *Datain* to the top location of the stack; *Pop* removes the top location of the stack and makes it available on *Dataout*, and *Tos* reads the top-of-stack without altering the contents of the stack. *Empty* is asserted when the stack is empty and *Full* is asserted when the stack is full. Show the complete VHDL code including the memory and the stack controller.

8.4 A VHDL structural description of a generic n-bit barrel shifter using the standard *std_logic* types is to be developed. The description uses the Pass gate in partial VHDL code that is shown below. This gate, the graphic notation of which is also shown below, has a source input and a gate input and a drain output. When the gate is 1, the output will be driven by the source; otherwise the output is at 'Z'. The inputs of the barrel-shifter structure are I and L. The data inputs are provided on the I lines, and the number of rotates to left is indicated by bits of the L inputs, e.g., $L=00001000$ rotates I to the left 4 places. Shifted I appears on Z. A 4-bit shift adder shown here clarifies the interconnection that is required. Complete the VHDL code shown here to completely describe an n-bit barrel-shifter. Use generate-statements and make sure your design is parameterized and expandable.
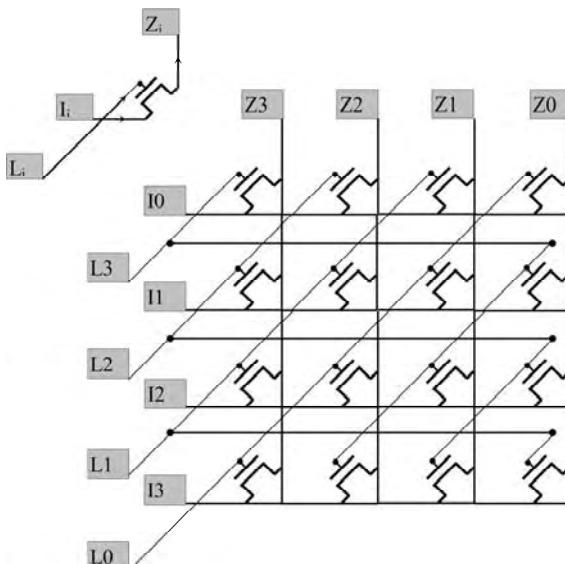
```
ENTITY switch IS PORT
   (source, gate : IN std_logic; drain : OUT std_logic);
END ENTITY;
--
ARCHITECTURE assign OF switch IS BEGIN
  drain <= source WHEN gate = '1' ELSE 'Z';
END ARCHITECTURE;
--
ENTITY barrel_shifter IS PORT
   (ii, ll : IN std_logic_vector;
    zz : OUT std_logic_vector BUS);
END ENTITY;
--
ARCHITECTURE switch_level OF barrel_shifter IS
   . . .
BEGIN
   . . .
END switch_level;

ENTITY barrel_shifter_tester IS END ENTITY;
--
ARCHITECTURE input_output OF barrel_shifter_tester IS
   SIGNAL ii, ll, zz : std_logic_vector (7 DOWNTO 0);
BEGIN
   shifter:ENTITY WORK.barrel_shifter PORT MAP (ii,ll,zz);
   ii <= X"00", X"F5" AFTER 20 US, X"45" AFTER 40 US,
         X"4B" AFTER 60 US;
   ll <= X"00", X"01" AFTER 10 US, X"04" AFTER 20 US,
                X"02" AFTER 30 US, X"80" AFTER 40 US,
                X"00" AFTER 50 US, X"01" AFTER 60 US;
END ARCHITECTURE;
```

8.5   Write a two-dimensional unconstrained $n \times m$ clocked memory. The memory has a $log_2 (n)$ bit address bus and an m bit bidirectional *Databus*. Reading from the memory is done when *CE* is '1' and *RW* is '1'. Writing takes place on the rising edge of the clock when *CE* is '1'. Represent each memory cell with a guarded block statement. Write nested generate statements for the array. Write and wire an unconstrained decoder to the array.

8.6   Write a behavioral description for a 16 bit serial adder circuit. Sixteen bits of *A* and *B* operands are serially fed into the circuit synchronized with a main clock signal. The least significant bits of the operands are received first. The start of data is signaled by a synchronous '1' on the *start* input. The least significant bits of data are received on the *A* and *B* inputs during the clock pulse that follows the start of the add operation. After sixteen clock pulses, the add operation is complete and the result will be available on sixteen parallel data lines. At this time the *done* output of the circuit becomes '1' and stays at this level until the next serial add operation starts again.

   Hardware implementation of this circuit requires a flip-flop for saving intermediate carry values, and a shift register for shifting in the serial results. However, you are to write a behavioral description and not concerned with its hardware implications. Your description should be accurate at the clock level. It is possible to write a single process statement to describe the complete circuit behavior. Input and output ports of your architecture should match those of the actual circuit. Use BIT type for all signals.

8.7   A data processing circuit is to be designed for receiving data sets and outputting the largest data that appears in a data set. The machine has an 8-bit *datain* input, a one-bit *avail* input, a one bit *ready* output and an 8-bit *largest* output. All data are considered to be 8 bit positive numbers. The machine receives data while *avail* is '1' and makes the largest 8-bit data available on *largest* as soon as a data set is completed. Synchronous bytes of a data set appear on *datain* while *avail* is '1'. A data set begins when *avail* becomes 1 and continues while this input is '1'. Termination of a data set is indicated by *avail* becoming '1'. While a new data set is being received, *ready* is 0, and the largest data from the last data set appears on the 8-bit *largest*. When *avail* becomes 0 machine resets, *largest* is updated, *ready* becomes 1, and it becomes ready for the next data set. A) Write VHDL description of the datapath this machine. Use signal assignments, conditional assignments, process statements, and block statements for describing components of the datapath. It is not necessary to generate separate entity-architecture pairs for each of the data components; you can describe gates, registers, comparators, multiplexers,

and other functions in the concurrent body of the datapath. B) Show VHDL description for the controller state machine of this circuit. The controller receives input handshaking signals and issues output *ready* signal as well as signals controlling the flow of data in the registers of the datapath. C) Wire the controller and the datapath to form the complete circuit. Use *std_logic* and arithmetic IEEE packages.

8.8 Given the following state machine description, 1) Show the state diagram this description is implementing. 2) Add an asynchronous reset to this description.

```
ENTITY mooreb IS PORT
   (data, clock : IN BIT; outz, waiting : OUT BIT);
END mooreb;
--
ARCHITECTURE synthesizable OF mooreb IS
   TYPE state IS (aa, bb, cc, dd);
   SIGNAL nxt, present : state;
BEGIN
   reg : PROCESS (clock)
   BEGIN
      IF (clock'EVENT AND clock = '1') THEN
         present <= nxt;
      END IF;
   END PROCESS;
   --
   logic : PROCESS (present, data)
   BEGIN
      CASE present IS
         WHEN aa =>
            IF data = '0' THEN nxt <= aa;
            ELSE nxt <= bb; END IF;
         WHEN bb =>
            IF data = '0' THEN nxt <= cc;
            ELSE nxt <= bb; END IF;
         WHEN cc =>
            IF data = '0' THEN nxt <= aa;
            ELSE nxt <= dd; END IF;
         WHEN dd =>
            IF data = '0' THEN nxt <= cc;
            ELSE nxt <= bb; END IF;
      END CASE;
   END PROCESS;
   outz <= '1' WHEN present = dd ELSE '0';
   waiting <= '1' WHEN present = aa ELSE '0';
END synthesizable;
```
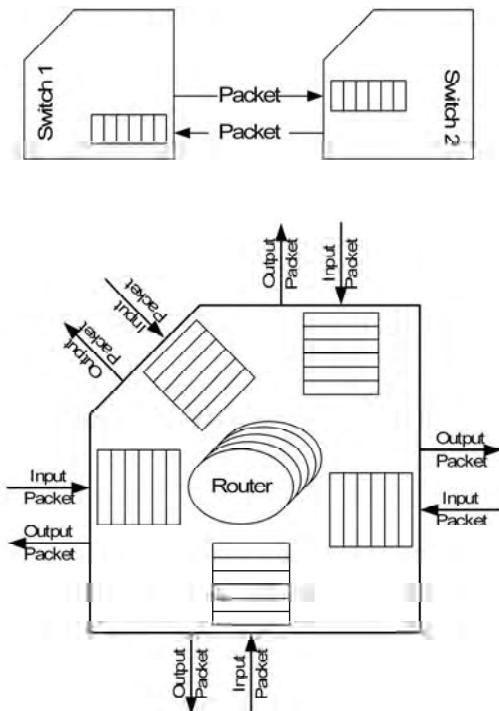
8.9 Design and implement a high level model for NoC switches with five identical ports, routing logic, and a routing table. Each port contains an input buffer for storing the incoming packets. Each packet

must be a record of data and include a header that determines the destination address, and a data payload. When a packet arrives, it will be stored in the input buffer. The router continuously checks the received packets, and according to their destinations routes them to the appropriate output port. The input buffer is a circular FIFO for storing input packets of neighboring switches.

For the routing algorithm, you can use a simple round-robin method. Destination addresses must be stored in the routing table using a static routing methodology. Use an extra field in each packet for indicating how long it has taken the packet to arrive at its destination. A simple measure of this timing is the number of switches a packet has traveled through. In your parametric design you must use high level structures of VHDL.

8.10 A small processor is to be designed. The processor is so unrealistically small that we call it Unrealistic Small Processor or USP. The processor has 4 instructions. An instruction is always only one byte long. Two bits are the opcode and six bits are address to a 64-byte memory. The instructions are *store*, *load*, *jump* and *add*. For these instructions, opcodes are 00, 01, 10 and 11 respectively. The *store* instruction stores contents of AC in memory location addressed by the

6-bit address. The *load* instruction loads AC with contents of the memory. The *jump* instruction causes the next instruction to be fetched from the memory location specified by the 6-bit address of this instruction. The *add* instruction adds memory addressed by the address field to AC and puts the result in AC. Develop code for the individual components of this machine (*ir*, *ac*, *pc*, and *alu*); show the datapath using component instantiations and bus specifications. Show the complete CPU. Use *std_logic* and related arithmetic packages. All types must be of the *std_logic* type. All codes must be synthesizable. Memory read and write are done in one cycle using *rd_mem* and *wr_mem*.

Suggested Reading

Bhasker, Jayaram, *A VHDL Primer, 3rd edition*, 1998, Prentice Hall PTR, ISBN: 978-0130965752.

Chu, Pong, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press, ISBN: 0471720925.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Navabi, Zainalabedin, *Embedded Core Design with FPGAs*, 2006, McGraw Hill-Professional, ISBN: 0071474811.

Perry, Douglas L., *VHDL: Programming by Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

Rushton, Andrew, *VHDL for Logic Synthesis, 2nd edition*, 1998, John Wiley & Sons, ISBN: 047198325X.

9 Core Design Test and Testability

The previous chapters discussed VHDL for describing components that can be used in design of a digital system as cores or subcomponents. Except for a few cases, and in a very limited way, we did not discuss ways of testing these components. Or, at least, we did not elaborate on efficient design test methodologies.

In addition to design test, post manufacturing test is also important for a designer to consider. Furthermore, there are many areas of overlap between design test methodologies, and post manufacturing test. Post-manufacturing test may require test and simulation of a pre-manufacturing model to develop methods and test data for it.

This chapter discusses the "test" issue. In the first part of the chapter we start with simple test cases and develop techniques for data generation and application. We will use an accumulation of several of these techniques to test several of the cores discussed in the previous chapter. In the second part of this chapter, we discuss testability methods incorporated into cores. We show testbenches indicating utilization of such testability methods.

9.1 Issues Related to Design Test

This section introduces some of the vocabularies and terminologies associated with issues related to testing a design. Sections that follow this introductory section, discuss techniques for testing the VHDL description of a design. After completing discussion of design test, a

341

similar section will be devoted to manufacturing test before we discuss testable VHDL designs.

9.1.1 Design Test

Design test refers to the process of testing a design and verifying that the design meets the required specs. In the HDL world, design test usually means HDL simulation and response verification.

In addition to simulation, verification of a design can be performed by formal verification methods. As an alternative to formal verification, dynamic verification methods verify designs by means of simulation and response assertions.

In the context of this chapter, design test means application of test data to a design under test (DUT), simulating the design and verifying the output.

9.1.2 Testbench

VHDL simulation environments provide tools for graphical or textual display of simulation results. Some simulation environments go further, and provide graphical tools for editing input test data to a design module that is being tested. Such tools are referred to as waveform editors, and are usually good for small designs. They become too complex to use for design with many busses and control signals. Another problem with waveform editors is that each simulation environment uses a different procedure for waveform editing, and moving from one simulator to another requires relearning a whole new set of procedures.

This problem can be alleviated by use of VHDL *testbenches*. A VHDL testbench is a VHDL entity-architecture pair that instantiates DUT applies data to it and monitors its output. Because a testbench is in VHDL, it can go from one simulation environment to another. A design module and its corresponding testbench form a simulation model in which DUT is tested for the same input data regardless of what simulation environment is used.

To facilitate development of testbenches, some simulation environments provide testbench tools that automatically generate a template testbench. Such tools also provide ways of inserting templates for generation of test data for applying them to DUT. Using templates is helpful, but a designer must understand testbenches and language constructs that are used for testing a design module.

9.1.3 Coverage

When verifying a design by simulation or by a formal verification method, coverage referrers to the percent of the design that has been exercised. For simulation, a testbench applies data to a DUT, test data generated by the testbench reach certain parts of a circuit and may never create activities in other parts. A good testbench uses a minimum set of data to cover testing of the largest possible portion of the circuit.

Design coverage in the context of VHDL, or other HDLs, is measured by various code coverage methods. Code coverage is the amount of VHDL code covered by a testbench. Code coverage may consist of line coverage (number of lines covered), block coverage (blocks of code between conditions covered), condition coverage (number of conditional statements exercised), and perhaps other code related methods.

Generally, simulation tools provide a coverage measure after simulation is complete. A testbench developer should make adjustment to his or her testbench and/or test data for best coverage in the shortest simulation time.

9.2 Simple Testbenches

Before we start with the presentation of various techniques of test data generation and response monitoring, we present simple testbench examples for circuits that we have discussed in the previous chapters. A combinational and a sequential circuit are being considered here.

9.2.1 Combinational Circuit Testing

Developing a testbench for a combinational circuit is straight forward; however selection of data and how much testing should be done depends on the CUT (circuit under test) and its functionality. Chapter 8 presented a simple ALU (Figure 8.3) that we use here to test. Entity declaration and its port declarations are repeated in Figure 9.1 for reference.

A testbench for *alu* is shown in Figure 9.2. The declarative part of the testbench declares every port of the CUT as signals. The first statement in this testbench is an instantiation statement instantiating our CUT. Association by name is used for associating testbench signals with ports of CUT.

```
ENTITY alu IS
   PORT (a, b : IN std_logic_vector;
         add_sub : IN std_logic;
         func : IN std_logic_vector (1 DOWNTO 0);
         y : OUT std_logic_vector;
         gt, eq, lt, co : OUT std_logic
         );
END ENTITY;
```

Figure 9.1 Ports of *alu* Being Tested

The *procedural* architecture of *alu* of Figure 8.3 implements a five-function *alu* with add, subtract, and logical operators. The ALU has compare outputs that are driven by a comparator and a vector output that is the result of the ALU operation.

Assignment of values to the various inputs of CUT are done differently. For the *ai* input, an initial value is set when *ai* is declared, and it is kept throughout the simulation. For *bi,* timed data at 20 ns time intervals are assigned to it. The add-subtract input (*asi* signal) is held at '0' until it changes to '1' at 80 ns.

As input data are changing, the ALU function input (*fni*) starts at 0 and is incremented every 23 ns. This incrementing guarantees that our ALU is tested for every one of its functions. Incrementing *fni* stops when simulation time reaches 160 ns. The VHDL NOW function returns the current simulation time.

```
ENTITY alu_tester IS END ENTITY;
--
ARCHITECTURE timed OF alu_tester IS
   SIGNAL ai : std_logic_vector (3 DOWNTO 0) := "0110";
   SIGNAL bi : std_logic_vector (3 DOWNTO 0) := "0100";
   SIGNAL asi: std_logic;
   SIGNAL fni : std_logic_vector (1 DOWNTO 0) := "00";
   SIGNAL yo : std_logic_vector (3 DOWNTO 0);
   SIGNAL go, eo, lo, co : std_logic ;
BEGIN
   CUT: ENTITY WORK.alu (procedural) PORT MAP
           (a => ai, b => bi,
              add_sub => asi,
              func => fni,
              y => yo,
              gt => go, eq => eo, lt => lo, co => co );
   bi <= "0011" AFTER 020 NS, "1110" AFTER 040 NS,
         "1000" AFTER 060 NS, "0111" AFTER 080 NS,
         "1001" AFTER 100 NS, "1100" AFTER 120 NS;
   asi <= '0' AFTER 10 NS, '1' AFTER 80 NS;
   fni <= fni + 1 AFTER 23 NS WHEN NOW <= 160 NS ELSE fni;
END ARCHITECTURE timed;
```
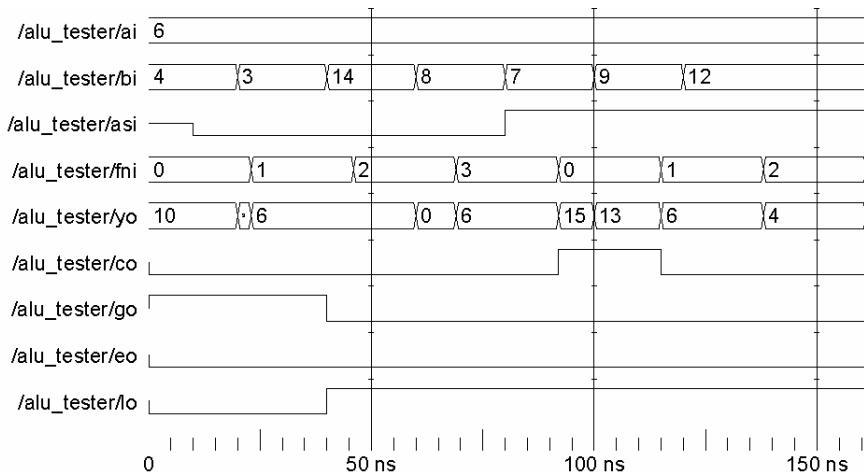
Figure 9.2 Testbench for *alu(procedural)*

Exhaustive testing of a large ALU where all its inputs receive every possible value is too time consuming, and cannot be done. However, testing every ALU function while some random data is being applied to its data inputs is possible and provides a good means of testing the ALU.

The above simulation ends when no more events occur in the simulation model. The *fni* signal has its last event at 161 ns. Figure 9.3 shows the simulation run of the *timed alu_tester* testbench.

Figure 9.3 *alu* **Simulation Results**

9.2.2 Sequential Circuit Testing

Testing sequential circuits involves synchronization of various data inputs with the circuit clock and with each other. We demonstrate a simple template in this section, and synchronization and other more complex issues will be discussed in the next section. The circuit we are testing here is the shift register of Chapter 8 presented in Figure 8.6. The entity declaration of this circuit is repeated in Figure 9.4 for reference. This shift register has INOUT bidirectional port (*dio*) that must either be driven from the inside of the shift register or by the testbench. The testbench must make sure no data is put on this port when the shift register is expected to drive it.

The testbench for *shift_reg* is shown in Figure 9.5. The format of the testbench is basically the same as the one shown in Figure 9.2. Some of the shift register inputs are driven by signal assignments, but the majority is driven by conditional signal assignments using the NOW function.

```
ENTITY shift_reg IS
   PORT (clk, ld, rst, l_r, shen, s_in, oe : IN std_logic;
         dio : INOUT std_logic_vector);
END ENTITY;
```

Figure 9.4 Shift Register Entity Declaration

The circuit clock is a periodic signal with a 10 ns period. The *dio* bidirectional input is incremented by 3 every 19 ns until simulation time reaches 93 ns, at which time it becomes high impedance, allowing the shift register to drive the bus. Periodic signals with different frequencies are put on *ld*, *shen*, *s_in*, and *oe* to cause them to exercise different functions of the shift register.

```
ENTITY shift_reg_tester IS
END ENTITY;
--
ARCHITECTURE timed OF shift_reg_tester IS
   SIGNAL clk, shen, rst, l_r : std_logic := '0';
   SIGNAL ld, s_in, oe : std_logic := '1';
   SIGNAL dio : std_logic_vector (3 DOWNTO 0) := "1001";
BEGIN
   UUT1: ENTITY WORK.shift_reg (synch)
         PORT MAP (clk, ld, rst, l_r, shen, s_in, oe, dio);

   rst <= '0', '1' AFTER 003 NS, '0' AFTER 010 NS,
               '1' AFTER 120 NS, '0' AFTER 127 NS;
   ld <= NOT ld AFTER 37 NS WHEN NOW <= 97 NS ELSE '0';
   shen <= NOT shen AFTER 35 NS WHEN NOW <= 113 NS
                                   ELSE '0';
   l_r <= '0' , '1' AFTER 20 NS, '0' AFTER 40 NS;
   s_in <= NOT s_in AFTER 10 NS WHEN NOW <= 75 NS ELSE '0';
   oe <= '0', '1' AFTER 97 NS, '0' AFTER 113 NS;
   dio <= dio + 3 AFTER 19 NS WHEN NOW <= 93 NS
                                   ELSE (OTHERS => 'Z');
   clk <= NOT clk AFTER 5 NS WHEN NOW <= 133 NS ELSE '0';
END ARCHITECTURE timed;
```

Figure 9.5 Shift Register Testbench

9.3 Testbench Techniques

Various VHDL coding techniques for generation of test data and observing circuit responses are discussed in this section. We use state machines of Chapter 8 as our test modules. The first example is a **101** Moore detector circuit depicted in Figure 9.6.

```
ENTITY moore_detector IS
   PORT (x, rst, clk : IN STD_LOGIC; z : OUT STD_LOGIC);
END ENTITY ;
--
ARCHITECTURE procedural OF moore_detector IS
   TYPE state IS (reset, got1, got10, got101);
   SIGNAL current : state := reset;
BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN
         current <= reset;
         ELSE
            CASE current IS
               WHEN reset =>
                  IF x = '1' THEN current <= got1;
                  ELSE current <= reset; END IF;
               WHEN got1 =>
                  IF x = '0' THEN current <= got10;
                  ELSE current <= got1; END IF;
               WHEN got10 =>
                  IF x = '1' THEN current <= got101;
                  ELSE current <= reset; END IF;
               WHEN got101 =>
                  IF x = '1' THEN current <= got1;
                  ELSE current <= got10; END IF;
               WHEN OTHERS => current <= reset;
            END CASE;
         END IF;
      END IF;
   END PROCESS;
   z <= '1' WHEN current = got101 ELSE '0';
END ARCHITECTURE;
```

Figure 9.6 A 101 Moore Detector For Test

The VHDL code shown in this figure is synthesizable. It has a synchronous reset and produces a '1' on its output when the state of the machine reaches the *got101* state.

9.3.1 Arbitrary Test Data

A simple testbench for *moore_detector* of Figure 9.6 is shown in Figure 9.7. As before, our testbench is an entity with no ports. Within the *timed* architecture of this testbench, three concurrent signal assignments provide data for testing the state machine. The signals on the left hand sides of these assignments are connected to the ports of *moore_detector*.

```
ENTITY moore_detector_tester IS
END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL x, reset, clk, z : std_logic := '0';
BEGIN
   MUT: ENTITY WORK.moore_detector(procedural)
        PORT MAP (x, reset, clk, z);

   reset <='1', '0' AFTER 24 NS;
   x <= NOT x AFTER 7 NS WHEN NOW <= 189 NS;
   clk <= NOT clk AFTER 5 NS WHEN NOW <= 189 NS;
END ARCHITECTURE;
```

Figure 9.7 A Simple Testbench for *moore_detector*

The testbench shown, puts a periodic signal on the *clk* input, and another periodic signal with a different frequency on the *x* input. The *clk* and *x* signals toggle every 5 ns and 7 ns respectively. Both signals cease toggling at time 189 ns. At this time, since no more events occur in our simulation model, the simulation stops. It is a good practice to make sure that a testbench eventually stops itself. This way, we do not have to watch the simulation run for a desired simulation time to arrive. Trying to stop the simulation manually generally results in longer than necessary simulation runs.

With arbitrarily selecting the timing of *clk* and *x*, the waveform generated on *x* may or may not be able to test our machine for a correct **101** sequence. However, periods of *clk* and *x* can be changed to make this happen. With the timing used here, the *moore_detector* output becomes **1** at 55 ns, and every 70 ns from then on until the simulation stops a little after 189 ns.

9.3.2 Random Test Data

This section develops a random number generation function that will be used in this and the next two sections. Instead of assigning a periodic signal to the input *x* of the *moore_detector*, we use our random function generator to assign random data to *x*.

9.3.2.1 Random Procedure. Figure 9.8 shows our *TestPack* package that declares and defines our *random* procedure. In addition to the standard libraries that we have discussed so far, this package also uses the IEEE *math_real* package. This package contains mathematical functions and constants, such as *SIN*, *COS*, *LOG*, and *PI*. The function we use here is *UNIFORM* that returns a pseudo-random number with uniform distribution in the open interval (0.0, 1.0). The declaration of *math_real* package appears in Appendix I.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
USE IEEE.math_real.ALL;

PACKAGE TestPack IS
   PROCEDURE random (VARIABLE seed1, seed2 : INOUT INTEGER;
                  SIGNAL target : INOUT std_logic_vector);
END PACKAGE TestPack;

PACKAGE BODY TestPack IS
   PROCEDURE random (VARIABLE seed1, seed2 : INOUT INTEGER;
                SIGNAL target : INOUT std_logic_vector) IS
      VARIABLE ti : INTEGER;
      VARIABLE size : INTEGER := target'LENGTH;
      VARIABLE tmp : std_logic_vector (size-1 DOWNTO 0);
      VARIABLE rand, within : REAL;
   BEGIN
      UNIFORM (seed1, seed2, rand);
      within := 2.0**size;
      ti := INTEGER (rand * within);
      target <= conv_std_logic_vector (ti, size);
   END PROCEDURE random;
END PACKAGE BODY TestPack;
```

Figure 9.8 Test Package for Random Number Generation

The *random* procedure shown in Figure 9.8 uses *seed1* and *seed2* to pass to *UNIFORM* for making the random number. The seeds are used and modified by *UNIFORM* for the next time it is called. As seen, the mode of these parameters of *random* is INOUT so that they can be modified. Output of *random* is *target* that is a signal of *std_logic_vector* type.

In the body of *random*, *UNIFORM* is called and returns *rand* of REAL type, which is between 0.0 and 1.0. Based on the size of *target*, *within* is calculated that becomes the range of unsigned integer that a binary number of *target* size can take. The value assigned to target becomes *within* multiplied by *rand* converted to *std_logic_vector*.

As mentioned, using *random* requires variables associated with *seed1* and *seed2* to be saved in the calling program. Furthermore, a design using *random* must use the *TestPack* of Figure 9.8.

9.3.2.2 Assigning Random Data to Signals. The VHDL code in Figure 9.9 uses our *random* procedure to assign random values to *x*. This testbench tests the *moore_detector* of Figure 9.6. Since *random* was developed for the general case of a vector target, we have used the one bit std_logic_vector type signal *x* to associate with the *x* scalar input of our *moore_detector*.

```
ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL x : std_logic_vector(0 DOWNTO 0) := "0";
   SIGNAL clk : std_logic := '0';
   SIGNAL reset : std_logic := '1';
   SIGNAL z : std_logic;
BEGIN
   MUT:  ENTITY WORK.moore_detector(procedural)
         PORT MAP (x(0), reset, clk, z);

   reset <= '0' AFTER 24 NS;

   SetX: PROCESS
      VARIABLE seed1, seed2 : POSITIVE := 14;
   BEGIN
      FOR i IN 0 TO 27 LOOP
         WAIT FOR 7 NS;
         random (seed1, seed2, x);
      END LOOP;
      WAIT;
   END PROCESS;

   SetC: PROCESS
   BEGIN
      FOR i IN 0 TO 36 LOOP
         WAIT FOR 5 NS;
         clk <= NOT clk;
      END LOOP;
      WAIT;
   END PROCESS;
END ARCHITECTURE;
```

Figure 9.9 Random Data Testbench

The testbench shown in Figure 9.9 uses a for-loop statement in the *SetX* process to call *random* and assign the resulting vector to *x*. The for loop shown calls *random* every 7 ns for 28 times. After this time, a *wait* statement suspends the *SetX* process forever. This process declares *seed1* and *seed2* for our *random* procedure and initializes them to 14. Any initial value can be used here, and absence of that would assume an initial value of 1. Figure 9.10 shows a simulation run of the VHDL code of Figure 9.9.

The testbench of Figure 9.9 uses a for loop similar to that discussed above for assigning values to the circuit clock. This is a more general format than the simple conditional signal assignment used in Figure 9.7. The format used here (see *SetC* process in Figure 9.9) allows changing the frequency of the clock, or even assigning non periodic values to this signal.
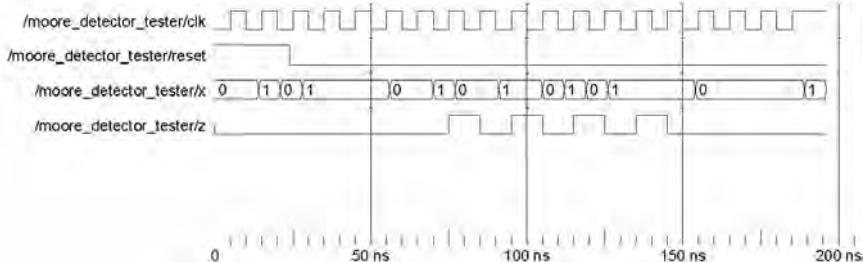
Figure 9.10 Random Data Generated on x

9.3.2.3 Random Timing. A random function can be written to return integers in a given range. This function can then be used for assignment of values to a signal at random time values. For example, suppose *irandom*, the declaration of which is shown below, returns an integer in the range of 0 to *limit* in its *intran* parameter.

```
PROCEDURE irandom(VARIABLE seed1, seed2: INOUT INTEGER;
                  CONSTANT limit: INTEGER;
                  VARIABLE intran: OUT INTEGER);
```

In order to make a wait-statement wait a random amount of time, the following must be done:

```
 irandom (seed1, seed2, 23, int);
WAIT FOR int * 1 ns;
```

The above example makes the range of the wait time between 0 and 23 nanoseconds.

9.3.3 Applying Synchronized Data

The previous examples of testbenches for DUT used independent timing for the clock and data. Where several sets of data are to be applied, synchronization of data with the system clock becomes difficult. Furthermore, changing the clock frequency would require changing the timing of all data inputs of the module being tested.

The testbench of this section (Figure 9.11), that is written for the *moore_detector* of Figure 9.6, uses a process statement to synchronize data applied to *x* with the clock that is generated in the testbench. The *clk* signal is generated in a process statement using a for-loop. As shown in Figure 9.11, the *SetX* process loops forever waiting for the positive edge of *clk*. When found, 3 ns later a new random data is generated for *x*. The stable data after the positive edge of the clock will be used by *moore_detector* on the next leading edge of the clock.

This technique of data application guarantees that changing of data and clock do not coincide.

The 3 ns delay used here makes it possible to use this same testbench for simulating post-synthesis designs as well as behavioral descriptions like that of Figure 9.6. In a post-synthesis simulation, in which component models with actual delay values are used, testbench delays should allow propagation of test signals to complete before application of new test data.

```
ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL clk : std_logic := '0';
   SIGNAL x : std_logic_vector (0 TO 0) := "0";
   SIGNAL reset : std_logic := '1';
   SIGNAL z : std_logic;
BEGIN
   MUT : ENTITY WORK.moore_detector(procedural)
         PORT MAP (x(0), reset, clk, z);

   reset <= '0' AFTER 24 NS;

   Set_X: PROCESS
      VARIABLE seed1, seed2 : POSITIVE := 14;
   BEGIN
      WAIT UNTIL clk = '1';
      WAIT FOR 3 NS;
      random (seed1, seed2, x);
   END PROCESS;

   Set_C: PROCESS
   BEGIN
      FOR i IN 0 TO 13 LOOP
         WAIT FOR 5 NS;
         clk <= NOT clk;
      END LOOP;
      WAIT;
   END PROCESS;
END ARCHITECTURE;
```

Figure 9.11 Clock Synchronized Data Application

9.3.4 Synchronized Display of Results

The technique used in the previous section can be used for synchronized observation of DUT outputs or internal signals.

Figure 9.12 shows another testbench for our *moore_detector*. In this testbench, 1 ns after the positive edge of the clock, that is when

the circuit output is supposed to have its new stable value, the *z* output is displayed using the REPORT statement. Since REPORT takes a string argument, *z* that is of *std_logic* type is turned into string using the 'IMAGE attribute.

```
ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL clk : std_logic := '0';
   SIGNAL x : std_logic_vector (0 TO 0) := "0";
   SIGNAL reset : std_logic := '1';
   SIGNAL z : std_logic;
BEGIN
   MUT : ENTITY WORK.moore_detector(procedural)
         PORT MAP (x(0), reset, clk, z);

   reset <= '0' AFTER 24 NS;

   setX: PROCESS . . .

   setC: PROCESS . . .

   display: PROCESS(clk)
   BEGIN
      IF (clk = '1') AND clk'EVENT THEN
         REPORT "Z:"&std_logic'IMAGE(z);
      END IF;
   END PROCESS;

END ARCHITECTURE;
```

Figure 9.12 Clock Synchronized Output Display

9.3.5 Displaying Interval Objects

In order for a testbench to display signals of a DUT, the signals must be declared in a package instead of being declared in the design. Such signals can be used by the design just like its own signals, and can be read and displayed by the design entity that uses the shared package.

The same can be done for variables used in processes of a design that is being tested, except that the design variables must be declared as shared variables in the package.

We demonstrate the former (visibility of design signals) by use of our *moore_detector* of Figure 9.6. The signal to be displayed by the testbench of this design is *current*. The package shown in Figure 9.13 declares the *current* signal used in Figure 9.14.

```
PACKAGE moore_observe IS
   TYPE state IS (reset, got1, got10, got101);
   SIGNAL current : state := reset;
END PACKAGE moore_observe;
```

Figure 9.13 Package for Upper-level Observabiliy

The *moore_detector* description must be modified to use the above package, and comment out its own *state* type and *current* signal declarations. With these changes, the testbench shown in Figure 9.14 displays *current*, that is the state of the *moore_detector* circuit, any time a change occurs on this signal.

```
USE WORK.moore_observe.ALL;

ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL clk : std_logic := '0';
   SIGNAL x : std_logic_vector (0 TO 0) := "0";
   SIGNAL reset : std_logic := '1';
   SIGNAL z : std_logic;
BEGIN
   MUT : ENTITY WORK.moore_detector(procedural)
         PORT MAP (x(0), reset, clk, z);

   reset <= '0' AFTER 24 NS;

   SetX: PROCESS . . .

   SetC: PROCESS BEGIN . . .

   m_state: PROCESS (current) BEGIN
      REPORT "current state is: " & state'IMAGE(current);
   END PROCESS;

   display: PROCESS (z) BEGIN
      REPORT "Z:"&std_logic'IMAGE(z);
   END PROCESS;

END ARCHITECTURE;
```

Figure 9.14 Displaying Design Signals From its Testbench

The *m_state* process of this testbench is sensitive to *current*. This signal is made visible to this testbench by the use of the use-statement at the beginning of the testbench. When *current* changes, the *m_state* process wakes up and reports the value of current. This testbench also displays *z* that is a local signal of the testbench.

The method described above becomes difficult to use if a design uses multiple instances of a lower level component, and a signal in these components is to be observed. An alternative solution is to report the value of the signal from within the design itself. For example, the *m_state* process of Figure 9.14 can move into the original description of *moore_detector*. In this case, to distinguish between reports coming from different instances of the same component, the 'INSTANCE-NAME or 'PATH-NAME attributes can be used. For example, the report statement shown below reports *current* and the complete path name of the instance from which the report is being issued.

```
REPORT "current state of" & current 'PATH-NAME &
       "is:" & state 'IMAGE (current);
```

9.3.6 An Interactive Testbench

For the next series of testbenches, we use a different state machine. This is a **1101** Moore detector with *start* and *rst* control inputs. If *start* becomes '0' while searching for **1101**, the machine resets to its initial state. As shown in Figure 9.15 this circuit has 5 states, and its output becomes '1' when it reaches state *got1101*. The extra *start* input of this machine makes it more controllable, and allows a better interaction with its testbench.

Figure 9.16 shows an interactive testbench for testing the state machine of Figure 9.15. The testbench applies a periodic signal to the circuit clock and random data to the *x* input of the *moore_detector*.

As data are being applied to the machine, the *SetS* process looks for two consecutive transitions on the *z* output of the state machine. The two consecutive transitions mean a complete pulse on *z*. When the random data on *x* have been able to generate a pulse on *z*, the testbench restarts the machine by setting *start* to '0' and back to '1'. The for-loop in the process statements repeats searching for *z* and restarting the machine three times. Each time the machine restarts, it starts in state *a*. Figure 9.17 shows the waveform output of the simulation run of our interactive testbench. The waveform also shows the *current* signal that is the state of the Moore detector circuit.

This example shows a simple interactive testbench where the DUT was restarted based on conditions occurring in the circuit. A more intelligent testbench can watch the interval signals of a DUT and adapt its data accordingly. For example, a testbench can change the seeds of its random generation procedure if it finds that the present random data are not triggering enough activities in the circuit.

```
ENTITY moore_detector IS
   PORT (x, start, rst, clk : IN std_logic;
         z : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF moore_detector IS
   TYPE state IS (reset, got1, got11, got110, got1101);
   SIGNAL current : state := reset;
BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN
            current <= reset;
         ELSIF start = '0' THEN
            Current <= reset;
         ELSE
            CASE current IS
               WHEN reset =>
                  IF x = '1' THEN
                     current <= got1;
                  ELSE
                     current <= reset; END IF;
               WHEN got1 =>
                  IF x = '1' THEN
                     current <= got11;
                  ELSE
                     current <= reset; END IF;
               WHEN got11 =>
                  IF x = '1' THEN
                     current <= got11;
                  ELSE
                     current <= got110; END IF;
               WHEN got110 =>
                  IF x = '1' THEN
                     current <= got1101;
                  ELSE
                     current <= reset; END IF;
               WHEN got1101 =>
                  IF x = '1' THEN
                     current <= got11;
                  ELSE
                     current <= reset; END IF;
               WHEN OTHERS => current <= reset;
            END CASE;
         END IF;
      END IF;
   END PROCESS;
   z <= '1' WHEN current = got1101 ELSE '0';
END ARCHITECTURE;
```

Figure 9.15 Moore Sequence Detector Detecting 1101

```
ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL clk : std_logic := '0';
   SIGNAL x : std_logic_vector (0 TO 0) := "0";
   SIGNAL start, reset : std_logic := '1';
   SIGNAL z : std_logic;
BEGIN
   MUT : ENTITY WORK.moore_detector(procedural)
         PORT MAP (x(0), start, reset, clk, z);

   SetS: PROCESS
   BEGIN
      reset <= '0' AFTER 24 NS;
      start <= '1' AFTER 24 NS;
      FOR i IN 0 TO 2 LOOP
         WAIT ON z;
         WAIT ON z;
         start <= '0' AFTER 11 NS,'1' AFTER 24 NS;
      END LOOP;
      WAIT;
   END PROCESS;

   SetX: PROCESS
      VARIABLE seed1, seed2 : POSITIVE := 3;
   BEGIN
      WAIT UNTIL clk = '1';
      WAIT FOR 3 NS;
      random (seed1, seed2, x);
   END PROCESS;

   clk <= NOT clk AFTER 5 NS WHEN NOW <= 350 NS;

 END ARCHITECTURE;
```
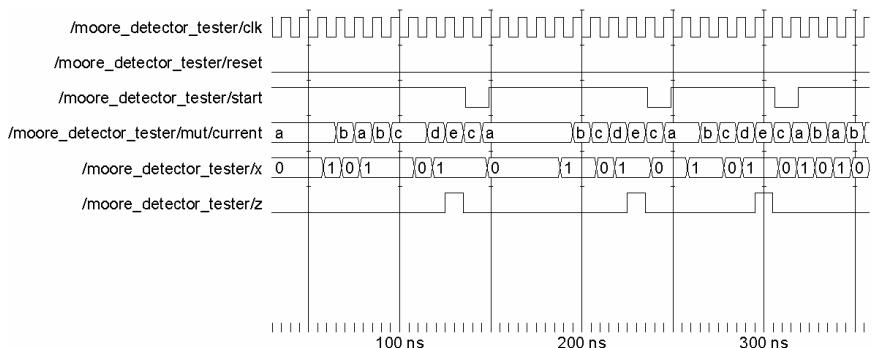
Figure 9.16 An Interactive Testbench

Figure 9.17 Interactive Testbench Simulation Run

9.3.7 Queued Data Application

None of the testbenches discussed so far applied a given set of test data to the circuit input(s). The testbench we are discussing here uses a queue to hold data to be applied to the DUT data input. We take predefined series of bits and assign them to the *x* input of *moore_detector*.

As shown in Figure 9.18, the 19-bit queue in the *SetX* process is initialized with test data. In this process statement, each bit of this buffer is shifted out onto the *x* input of DUT 1 ns after the positive edge of the *clk* clock. As data is shifted, queue is rotated in order for the applied buffered data to be able to repeat. Start and stop control of the state machine are done in the *SetRS* process statement. Figure 9.19 shows the simulation run waveform of this testbench.
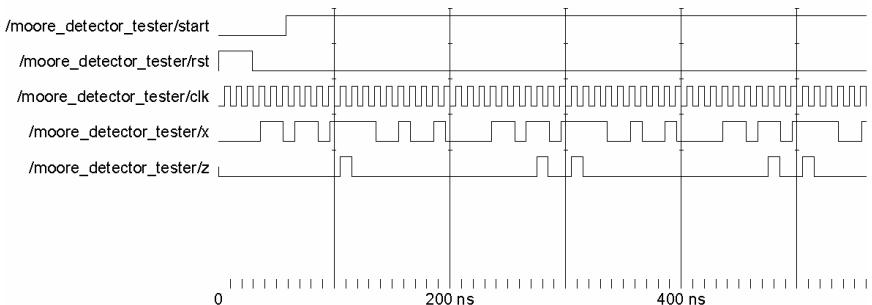
```
ENTITY moore_detector_tester IS END ENTITY;
--
ARCHITECTURE timed OF moore_detector_tester IS
   SIGNAL x, start, rst, clk, z : std_logic := '0';
BEGIN
   MUT: ENTITY WORK.moore_detector(procedural)
        PORT MAP (x, start, rst, clk, z);
   SetRS: PROCESS BEGIN
      rst <= '1'; start <= '0';
      WAIT FOR 29 NS; rst <= '0';
      WAIT FOR 29 NS; start <= '1';
      WAIT;
   END PROCESS SetRS;
   SetX: PROCESS
      VARIABLE queue : std_logic_vector (18 DOWNTO 0)
                          := "0001101101111001001";
   BEGIN
        WAIT UNTIL clk = '1';
        WAIT FOR 1 NS;
        x <= queue (queue'LEFT);
        queue := queue (queue'LEFT-1 DOWNTO 0) & x;
   END PROCESS SetX;
   SetClk: clk <= NOT clk AFTER 5 NS WHEN NOW <= 558 NS;

END ARCHITECTURE;
```

Figure 9.18 Applying Data from a Queue

Figure 9.19 Queued Data Simulation Run

9.3.8   Text File Stimuli and Response

A very flexible way of applying data and displaying responses of a circuit is reading and writing external data files. This is especially useful when dealing with large sets of data. In this section, we discuss the TEXTIO example of Chapter 6 (Section 6.6.2) with some modifications. The complete code of the testbench for an octal 2-to-1 multiplexer is shown in Figure 9.20.

The testbench shown has a *GetData* procedure for reading from a declared file, and *PutData* for writing into one. The *GetData* procedure reads from file *f* and schedules data of file *f* into its target *s* signal. Data in the data file must be preceded with the time that a particular data is to be assigned to the target signal. On the other hand, *PutData* writes the time that is being called and the data of its *w* input on one line of the file *f* that it is writing to.

The testbench in Figure 9.20 has three concurrent invocations of *GetData* for *a*, *b*, and *s* inputs of the multiplexer, respectively. As before, because *GetData* is written for the general case of a vector target, the *s* one bit input of the multiplexer has been declared as a one-bit *std_logic_vector*.

All three invocations of *GetData* are only called at the beginning of the simulation at time 0. At this time *Ain*, *Bin* and *Sin* files are read and appropriate values at appropriate times are scheduled into *a*, *b* and *s*.

The *PutData* invocation is also a concurrent procedure call in the body of the multiplexer testbench. This procedure is called every time an event occurs on *w*. When called, the present time and the value of *w* are written into the file output of this procedure.

```
ENTITY multiplexer8_tester IS
END ENTITY;
--
ARCHITECTURE timed OF multiplexer8_tester IS
   SIGNAL a, b, w1 : std_logic_vector (7 DOWNTO 0);
   SIGNAL s : std_logic_vector (0 TO 0) := "0";

   FILE Ain : TEXT OPEN READ_MODE IS "Ain.dat";
   FILE Bin : TEXT OPEN READ_MODE IS "Bin.dat";
   FILE Sin : TEXT OPEN READ_MODE IS "Sin.dat";
   FILE Wout : TEXT OPEN WRITE_MODE IS "Wout.dat";

   PROCEDURE GetData (SIGNAL s : OUT std_logic_vector;
                      FILE f : TEXT) IS
      VARIABLE lbuf : LINE;
      VARIABLE t : TIME;
      VARIABLE d : std_logic_vector (s'RANGE);
   BEGIN
      WHILE NOT ENDFILE (f) LOOP
         READLINE (f, lbuf);
         READ (lbuf, t);
         READ (lbuf, d);
         s <= TRANSPORT d AFTER t;
      END LOOP;
      FILE_CLOSE (f);
   END PROCEDURE GetData;

   PROCEDURE PutData (w : IN std_logic_vector;
                      FILE f : TEXT) IS
      VARIABLE lbuf : LINE;
   BEGIN
      WRITE (lbuf, NOW, RIGHT, 8, NS);
      WRITE (lbuf, w, RIGHT, 9);
      WRITELINE (Wout, lbuf);
   END PROCEDURE;

BEGIN

   UUT1: ENTITY WORK.multiplexer8 (conditional)
         PORT MAP (a, b, s(0), w1);

   GetData (a, Ain);
   GetData (b, Bin);
   GetData (s, Sin);
   PutData (w1, Wout);

END ARCHITECTURE timed;
```

Figure 9.20 File IO Testbench

   Figure 9.21 shows the three input files and the output file used by the testbench of Figure 9.20.

0 ns 00000000	0 ns 00111000	0 ns 1
7 ns 00001111	10 ns 00101111	5 ns 0
20 ns 10101010	35 ns 10110000	20 ns 1
25 ns 11111111	45 ns 11101010	25 ns 0
35 ns 11110101	50 ns 01100001	35 ns 1
55 ns 11001100	55 ns 00101110	45 ns 0
85 ns 00001010	95 ns 11100011	50 ns 1
115 ns 11001100	110 ns 00011100	55 ns 0
Ain.dat	**Bin.dat**	60 ns 1

	0 ns UUUUUUUU
	0 ns 00111000
	5 ns 00000000
	7 ns 00001111
	20 ns 00101111
	25 ns 11111111
	35 ns 10110000
	45 ns 11110101
	50 ns 01100001
75 ns 0	55 ns 11001100
95 ns 1	60 ns 00101110
100 ns 0	75 ns 11001100
115 ns 1	85 ns 00001010
125 ns 0	95 ns 11100011
150 ns 1	100 ns 00001010
175 ns 0	115 ns 00011100
200 ns 1	125 ns 11001100
Sin.dat	150 ns 00011100
	175 ns 11001100
	200 ns 00011100
	Wout.dat

Figure 9.21 Testbench Input and Output Files

9.4 Complete System Testing

This section shows how the techniques discussed in the previous section can be put together for testing complete systems. We will use the techniques mentioned, and variations of these techniques for testing the sequential multiplier and the adding processor of Chapter 8.

9.4.1 Multiplier Testing

The top-level entity declaration of the sequential multiplier of Chapter 8 is shown in Figure 9.22. The multiplier has a bi-directional multiplexed input-output bus and several input and output control signals. All data in and out of the multiplier go through *databus*.

```
ENTITY Multiplier IS
   PORT (clk, start : IN STD_LOGIC;
         databus : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
      lsb_out, msb_out, done : OUT STD_LOGIC);
END ENTITY;
```

Figure 9.22 Sequential Multiplier Interface

The testbench we are presenting for this circuit is an auto-check interactive one, in which several forms of data applications and result monitorings are demonstrated. The outline of *timed* architecture of *multiplier_tester* testbench is shown in Figure 9.23.

```
ARCHITECTURE timed OF multiplier_tester IS
   SIGNAL clk, start, error : STD_LOGIC := '0';
   SIGNAL databus : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
   SIGNAL lsb_out, msb_out, done : STD_LOGIC;
   SIGNAL im_data : STD_LOGIC_VECTOR (7 DOWNTO 0) :=
                    (OTHERS => 'Z');
   SIGNAL expected_result, multiplier_result :
                    STD_LOGIC_VECTOR (15 DOWNTO 0);
   SIGNAL index : INTEGER := 0;

   FILE d_file1 : TEXT OPEN READ_MODE IS "data1.dat";
   FILE d_file2 : TEXT OPEN READ_MODE IS "data2.dat";
BEGIN
   uut : ENTITY  WORK.Multiplier(structural) PORT MAP
       (clk, start, databus, lsb_out, msb_out, done );

   Apply_Data : PROCESS . . .
   Apply_Start : PROCESS . . .
   Expected_Results : PROCESS . . .
   Compare_Results: PROCESS . . .

   clk <= NOT clk AFTER 50 NS WHEN NOW <= 5000 NS ELSE '0';
   databus <= im_data;
END ARCHITECTURE timed;
```

Figure 9.23 Multiplier Testbench Outline

9.4.1.1 Testbench Declarations. The testbench declares signals to be associated with the ports of the multiplier design. In addition, there are signals for collecting the two bytes of the result and also for calculating the expected multiplier result. The *im_data* intermediate signal is declared for assignment of data to the bi-directional *databus*. As shown in the last part of the testbench of Figure 9.23, *im_data* is assigned to *databus*. When data is to be provided by the testbench to the multiplier, *im_data* will contain that data. When the multiplier is to drive *databus* with its output result, *im_data* must become high-

impedance to allow the multiplier to drive the *databus* INOUT signal. Also recall from Chapter 8 that, the multiplier itself drives its *databus* with all Zs when it is expecting to receive input data.

Our testbench declares *data1.dat* and *data2.dat* data files that contain data for the two operands of the multiplier. These files are opened in the architecture and will be read and assigned to *im_data*.

The testbench shown in Figure 9.23 applies three rounds of test data to the *Multiplier* entity. In each round, data is applied to the design under test and results are read and compared with the expected results. These are the tasks performed by this test bench:

- Read data files *data1.dat* and *data2.dat* and apply data to *databus*
- Apply *start* to start multiplication
- Calculate the expected result
- Wait for multiplication to complete, and collect the calculated result
- Compare expected and calculated results and issue error if they do not match

These tasks are concurrent and are independently timed. Each task is performed by an independent process statement.

9.4.1.2 Reading Data Files. Figure 9.24 shows the *Apply_Data* process that is responsible for reading data and applying them to *im_data*, which in turn goes on *databus*. Binary data from *data1.dat* and *data2.dat* external files are read into *tmp* and then assigned to *im_data*.

```
Apply_data : PROCESS
      VARIABLE tmp : std_logic_vector (7 DOWNTO 0);
      VARIABLE l1, l2 : LINE;
BEGIN
   FOR i IN 0 TO 2 LOOP
      READLINE (d_file1,l1);
      READLINE (d_file2,l2);
      WAIT UNTIL start = '1';
      READ (l1, tmp); im_data <= tmp;
      WAIT UNTIL clk = '1';  WAIT UNTIL clk = '1';
      READ (l2, tmp); im_data <= tmp;
      WAIT UNTIL clk = '1';
      im_data <= (OTHERS => 'Z');
      END LOOP;
   WAIT;
END PROCESS;
```

Figure 9.24 Apply-Data Process, Read Files Apply to DUT

Data read from *data2.dat* is distanced from that of *data1.dat* by two clock periods. This way, the latter is interpreted as data for the A operand and the former for the B multiplication operand. After placing data and allowing the multiplier to register data, *im_data* becomes high impedance. This way, we are allowing *databus* to be driven by the multiplier when its result is ready.

9.4.1.3 Applying Start. Figure 9.25 shows a process statement in which the *start* signal is issued. Using a for loop, three 100 ns pulses distanced by 1400 ns are placed on *start*.

```
Apply_Start : PROCESS BEGIN
   WAIT FOR 180 NS;
   FOR i IN 1 TO 3 LOOP
      WAIT FOR 50 NS; start <= '1';
      WAIT FOR 100 NS; start <= '0';
      WAIT FOR 1350 NS;
   END LOOP;
   WAIT;
END PROCESS;
```

Figure 9.25 Applying *start*

9.4.1.4 Calculating Expected Result. Figure 9.26 shows a process statement that reads from *im_data* the same data that is being placed on *databus*. Using this data that is generated by *Apply_Data*, the *Expected_Results* process calculates the expected multiplication result. After *start*, when *im_data* is updated, the first operand is read into *opnd1*. The next time *im_data* changes, *opnd2* is read. The expected result is calculated using these operands. Note that for detecting new data assignments to *im_data*, the 'TRANSACTION attribute is used. Using this instead 'EVENT, detects new data even if its value is the same as the previous data on *im_data*.

```
Expected_Results : PROCESS
   variable opnd1, opnd2 : STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
   FOR i IN 1 TO 3 LOOP
      WAIT UNTIL start = '1';
      WAIT ON im_data'TRANSACTION;
      opnd1 := im_data;
      WAIT ON im_data'TRANSACTION;
      opnd2 := im_data;
      expected_result <= opnd1 * opnd2;
   END LOOP;
   WAIT;
END PROCESS;
```

Figure 9.26 Calculating Expected Results

9.4.1.5 Reading Multiplier Output. When the multiplier completes its task, it issues *msb_out* and *lsb_out* to signal that it has readied the two bytes of the result. The process statement of Figure 9.27 is triggered by the falling edge of the circuit clock. After a clock edge, if *msb_out* or *lsb_out* is '1', it reads the *databus* and puts in its corresponding position in *multiplier_result*.

```
Actual_Result: PROCESS (clk) BEGIN
   IF(clk = '0' AND clk'EVENT) THEN
      IF (msb_out = '1') THEN
         multiplier_result (15 DOWNTO 8) <= databus;
      END IF;
      IF (lsb_out = '1') THEN
         multiplier_result (7 DOWNTO 0) <= databus;
      END IF;
   END IF;
END PROCESS;
```

Figure 9.27 Reading Actual Multiplier Results

9.4.1.6 Comparing Results. The last process statement in Figure 9.23 is for comparing actual and expected multiplication results. As shown in Figure 9.28, if the result calculated by the testbench (*expected_result*) does not match the result calculated by our multiplier RTL design (*multiplier_result*), an *error* flag is issued. This process makes our multiplier testbench a self-checking one.

```
Compare_Results: PROCESS (clk) BEGIN
   IF(clk = '0' AND clk'EVENT) THEN
      IF (done = '1') THEN
         IF (multiplier_result /= expected_result) THEN
            error <= '1';
         ELSE error <= '0';
         END IF;
      END IF;
   END IF;
END PROCESS;
```

Figure 9.28 Comparing Actual and Expected Results

9.4.2 Processor Testing

This section discusses a testbench for the processor that we developed in Chapter 8. The Adding processor, as discussed in Section 8.4.2, has an 8-bit data bus and a 6-bit address bus.

The testbench in this section uses an external file that contains instructions to be executed. The testbench reads these instruction mnemonics, converts them to binary numbers, applies them to the processor model, and when the program execution is complete it dumps the memory contents into an external file. Details of this testbench are discussed below.

9.4.2.1 Input File Format. An example file containing instructions and data for our processor model is shown in Figure 9.29.

```
00 lda 0f
0f ::: 0F
01 sta 0a
02 add 01
03 sta 0b
04 jmp 00
```
InstructionFile.mem

Figure 9.29 Instruction File Format

The instruction input file is called *InstructionFile.mem*. Each line of this file begins with a hex number representing the memory location of the instruction. The instruction mnemonic and its address follow this location. Initialization of a memory location is done by "::::" code instead of a mnemonic. For example line 2 of Figure 9.29 loads "0F" in memory location "0F".

9.4.2.2 Testbench Architecture. The testbench architecture for our Adding CPU, shown in Figure 9.30, uses the *TEXTIO* and *std_logic_textio* packages. The declarative part of this architecture declares the necessary processor and memory interface signals and the processor memory. The processor memory is a shared variable of type *memory* and is initialized to all 0s. The declarative part of this architecture also declares *InstFile* for the instruction file and *HexFile* for the memory image file.

The last item shown in the declarative part of *file_oriented* architecture of *test_AddingCPU* is the *convert* procedure that converts an instruction file to a memory image and writes in into *mem*.

The body of this architecture instantiates the processor under test, generates a clock signal, initializes the memory image, and performs read and write operations depending on instructions that are executed.

```
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_textio.ALL;
--
ARCHITECTURE file_oriented OF Test_AddingCPU IS
   SIGNAL reset : std_logic := '1';
   SIGNAL clk, control : std_logic := '0';
   SIGNAL adr_bus : std_logic_vector(5 DOWNTO 0);
   SIGNAL rd_mem, wr_mem : std_logic;
   SIGNAL data_bus : std_logic_vector(7 DOWNTO 0);
   SIGNAL mem_data : std_logic_vector(7 DOWNTO 0)
                   := (OTHERS => '0');
   TYPE memory IS
        ARRAY (0 TO 63) OF std_logic_vector(7 DOWNTO 0);
   SHARED VARIABLE mem : memory
                      := (OTHERS => (OTHERS => '0'));
   FILE InstFile, HexFile: TEXT;
   PROCEDURE Convert IS . . . END PROCEDURE;
BEGIN
   UUT: ENTITY WORK.AddingCPU(structural)
        PORT MAP (reset, clk, adr_bus,
                  rd_mem, wr_mem, data_bus);
   clk <= NOT clk AFTER 10 NS WHEN NOW <= 430 NS ELSE '0';

   FileInit: PROCESS . . . END PROCESS;

   MemReadWrite: PROCESS . . . END PROCESS;

   data_bus <= mem_data WHEN control='1'
                        ELSE (OTHERS => 'Z');
END ARCHITECTURE;
```

Figure 9.30 Testbench Architecture Outline

9.4.2.3 Reading an Instruction File. Figure 9.31 shows the *convert* procedure that reads an instruction file (*InstructionFile.mem*) and write an initial memory image into *mem*.

As shown in this figure, a while-loop handles reading instruction lines, converting them to hex and writing them to *mem*. After reading a line of instruction from the logical file *InstFile*, the address part of it is read into *addr* using *HREAD* procedure of the *std_logic_textio* package. The instruction part of this line is read into *opcode* using the *READ* procedure.

As shown in the *convert* procedure, a case statement converts instruction mnemonics into their equivalent opcodes. For each line of the instruction file, *writeData* is formed that is written into *mem* at the *addr* location. Figure 9.32 shows an instruction file converted to *mem* contents by the *convert* procedure.

```
PROCEDURE Convert IS
   VARIABLE l1 : LINE;
   VARIABLE inst : STRING(1 TO 2) := "00";
   VARIABLE opCode : STRING(1 TO 5);
   VARIABLE data, writeData, addr:
                            std_logic_vector(7 DOWNTO 0);
   VARIABLE JustData : std_logic;
BEGIN

   FILE_OPEN (InstFile, "InstructionFile.mem", READ_MODE);
   FILE_OPEN (HexFile, "HexadecimalFile.mem", WRITE_MODE);

   WHILE NOT ENDFILE (InstFile) LOOP
      READLINE (InstFile, l1);
      HREAD (l1, addr);
      READ (l1, opCode);
      JustData := '0';
      CASE opCode IS
         WHEN " lda " =>
            writeData(7 DOWNTO 6) := "00";
         WHEN " sta " =>
            writeData(7 DOWNTO 6) := "01";
         WHEN " jmp " =>
            writeData(7 DOWNTO 6) := "10";
         WHEN " add " =>
            writeData(7 DOWNTO 6) := "11";
         WHEN " ::: " =>
            JustData := '1';
            HREAD (l1, writeData);
         WHEN OTHERS =>
            JustData := '1';
            HREAD(l1, writeData);
      END CASE;

      IF JustData = '0' THEN
         HREAD(l1, data);
         writeData(5 DOWNTO 0) := data(5 DOWNTO 0);
      END IF;

      mem (conv_integer(addr)) := writeData;

   END LOOP;

   FILE_CLOSE(InstFile);
   FILE_CLOSE(HexFile);

END PROCEDURE;
```

Figure 9.31 Converting an Instruction File to memory image

Instruction File	Memory Contents
Line:	Location:
1: 00 lda 0f	00: **0F**
2: 0f ::: 0F	01: **4A**
3: 01 sta 0a	02: **C1**
4: 02 add 01	03: **4B**
5: 03 sta 0b	04: **80**
6: 04 jmp 00	05: **00**
	06: **00**
	07: **00**
	08: **00**
	09: **00**
	10: **0F**
	11: **10**
	12: **00**
	13: **00**
	14: **00**
	15: **0F**

Figure 9.32 Instruction Mnemonics and Hex Memory Data

9.4.2.4 File Initialization and Dump. The *FileInitDump* process of the testbench architecture is shown in Figure 9.33. In this process statement, *convert* is called that fills *mem*. After allowing 405 ns for the simulation run, this process opens the output memory image file and writes contents of *mem* into this file.

```
PROCESS
      VARIABLE l1 : line;
   BEGIN
      Convert;
      WAIT FOR 25 NS;
      reset <= '0';
      WAIT FOR 405 NS;
      FILE_OPEN(HexFile,"HexadecimalFile.mem", WRITE_MODE);
      FOR i IN 0 TO 63 LOOP
         HWRITE(l1,mem(i));
         WRITELINE(HexFile,l1);
      END LOOP;
      FILE_CLOSE(HexFile);
      WAIT;
END PROCESS;
```

Figure 9.33 Hex File Initialization and Dump

9.4.2.5 Memory Read Write. Our testbench uses a very simple memory model and simple read and write procedures. Figure 9.34 shows the *MemReadWrite* process of our testbench.

```
PROCESS BEGIN
   WAIT UNTIL clk = '1' AND clk'EVENT;
   control <= '0';
   WAIT FOR 1 NS;
   IF rd_mem = '1' THEN
      WAIT FOR 1 NS;
      mem_data <= mem(conv_integer(adr_bus));
      control <= '1';
   END IF;
   IF wr_mem = '1' THEN
      WAIT FOR 1 NS;
      mem(conv_integer(adr_bus)) := data_bus;
   END IF;
END PROCESS;
```

Figure 9.34 Memory Read Write Process

Memory read and write operations are synchronized with the testbench clock. In all of the above processes and procedures, *conv_integer* of *std_logic_arith* is used for indexing *mem*. Figure 9.35 shows the simulation run resulting from the execution of program of Figure 9.32.
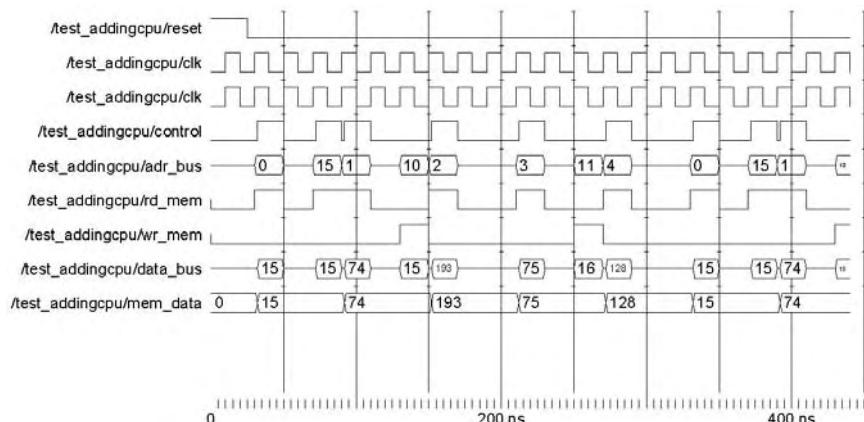
Figure 9.35 Simulation Run of Processor Instructions

The testbench discussed above tests *AddingCPU* for all its instructions. This example shows the power and flexibility of VHDL file handling for testbenches. In modeling larger memories, direct file

read and write becomes inefficient and more elaborate memory and related file handling schemes should be used.

9.5 Issues Related to Manufacturing Test

The rest of this chapter is devoted to design for testability in VHDL. We discuss VHDL models for a configurable LFSR and a MISR that are used in many testability techniques. We will then show a typical scan design and a memory BIST. Testbenches and test application platforms for testable designs that are presented are also discussed.

Before we discuss the VHDL codes of testability components, testable designs, and their corresponding test activation programs, it is necessary to understand why designs are made testable. We will discuss manufacturing test, fault models, and introduce some of the very common test terminologies.

9.5.1 Manufacturing Test

As opposed to the design test that tests a design before it is built, post manufacturing test tests a design after it is built. Post manufacturing test of a circuit under test (CUT) is done by an external test equipment testing our CUT, another circuit testing our CUT, or our CUT testing itself.

Post manufacturing test may require additional testability hardware in the original CUT. When this is done, we say that our design has been made Testable. Testing a testable design still requires test data and testability method to be evaluated before the design is built. Testability of a design is evaluated by application of test data through a testbench, or test platform, to the HDL model of the design.

Post manufacturing test efficiency is measured by test time and manufacturing fault coverage. We will discuss this coverage after we discuss fault models in the next section.

9.5.2 Fault Model

Many kinds of faults can occur after manufacturing a circuit. The transistor structure of a logic gate can have various forms of transistor terminal shorts, opens, as well as various transistor channel faults. Instead having to deal with many such circuit faults, a simple gate level fault model that represents many of these faults is used in digital system testing.

This fault model, that is used almost universally, is *line stuck at fault*. This means that all internal and external faults of a gate are

modeled by *stuck-at-0* and *stuck-at-1* faults on ports of the gate. This means that there are maximum of six faults associated with a two-input logic gate.

Digital system test methodologies, test evaluation and design for test techniques all use the stuck-at fault model.

9.5.3 Test Generation

A test generation program finds input test vectors for activating individual faults and propagating their effects to the circuit output. Usually faults are treated one at time, and test vectors are found for each fault acting independently. Treating faults this way is referred to as *single stuck-at fault* model.

9.5.4 Fault Simulation

Fault simulation is when a circuit is simulated with the presence of a fault. Fault simulation can verify if a certain test, when applied to a faulty circuit, can make the output of the circuit to be different than the good circuit output. When this happens we say that the fault is detected.

Fault simulation is done by first injecting fault *f* into the good circuit model (referred to as *golden model*). A circuit faulted as such is referred to as the faulty model of fault *f*. A testbench using this faulty model applies a given set of test vectors to the primary inputs of the faulty circuit and waits for its output to show a different behavior than that of the good circuit.

9.5.5 Fault Coverage

 A testbench or a set of test vectors (test) is evaluated by its fault coverage. A fault simulator is the primary tool for fault coverage evaluation.

For a given test, fault coverage is the ratio of faults detected by the test set to the total number of circuit faults. Generally fault coverage of test vectors that are specially generated for the given circuit faults (deterministic tests) are higher than randomly generated tests. However, random tests are easier and cheaper to generate. A test set with high coverage is also referred to as an efficient test set.

9.5.6 Testability

In order to detect a fault, a test vector at the primary input of the CUT has to propagate to the site of fault and activate it, and then it has to propagate the fault effect to the primary output of the circuit.

This means that a fault must be controllable and observable. Good controllability and observability in large combinational circuits and in sequential circuits are difficult to get. In order to improve these parameters, additional hardware is added to a circuit to reach test data to hard-to-get places, and take outputs from circuit nodes that are hard to propagate to the primary outputs. The added hardware improves controllability and observability of the circuit and makes it more testable.

9.6 Core Test Support Modules

Core testing involves generation of test, application of test to cores, and collection of test responses. Input test vectors and output response of a circuit can be stored external to the CUT or can be generated by the CUT or other neighboring cores.

Because of large external space (memory and/or hard disk) requirements, often, some or all test vectors are generated by on-chip pseudo-random generators, and expected output responses are analyzed by on-chip signature generators.

This section presents design of a configurable, linear feedback shift-register (LFSR), and a configurable multi-input signature register (MISR). An LFSR is used for pseudo-random input test vector generation, while a MISR is used for output response analysis.

An LFSR is a shift register with feedback and exclusive-or gates in its feedback or shift path. The initial content of the register is referred to as *seed*, and the position of XOR gates is determined by the polynomial (*poly*) of the LFSR. A MISR (Multiple Input Signature Register) is like an LFSR, with input and output parallel data.

9.6.1 LFSR

Figure 9.36 shows an LFSR made of D-type flip-flops and XOR gates in its shift path. The position of XOR gates determine the poly of this circuit, which is *poly*=10101. The *seed*, which is the initial value of the register, affects set and reset inputs of the individual flip-flops of the shift register. The LFSR *seed* and *poly* determine bit values that are generated on the serial output of the circuit (*sout*), as serial input bits (*sin*) are being shifted in.
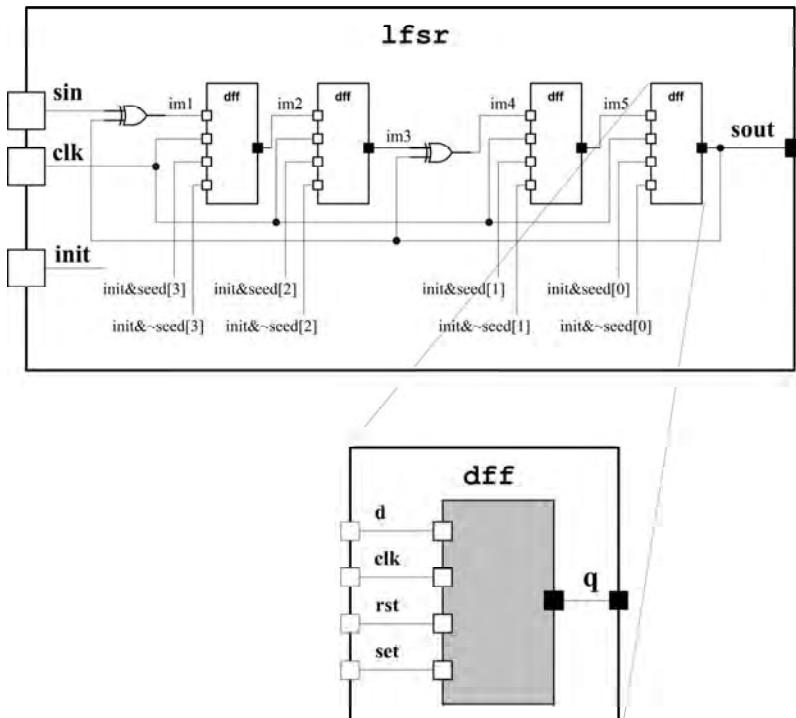
Figure 9.36 An LFSR with 10101 Polynomial

Figure 9.37 shows the LFSR VHDL code. This code describes the structure of LFSR using XOR gates and positive edge D-type flip-flop (not shown here) with asynchronous set and reset inputs. As with most designs in this and the previous chapter, this circuit uses unconstrained array declarations. The *lfsr* module wires n flip-flops, n-2 XOR gates in between flip-flops, and set and reset inputs of the flip-flops according to the *seed* parameter of this module. When *init* becomes '1', the LFSR seed is asynchronously loaded into the register.

We have used generate statements for wiring the flip-flops of this LFSR. The generate statement shown is based on the size of the *seed* generic parameter. The nested generate statements shown, generate *im_d* vector for the inputs of the flip-flops, and then instantiate *dff* flip-flops with bits of this vector.

As shown in Figure 9.36, the left most flip-flop input is different from the reset. For this, the *first* if-generate statement in Figure 9.37 generates the input of this flip-flop. The rest of the LFSR flip-flop inputs are generated using the *rest* generate statement. In each iteration of the outer generate statement of Figure 9.37 (*dffs* labeled statement), a *dff* is instantiated with bits of *im_set*, *im_rst*, *imd*, and *im_6* intermediate vectors.

```
ENTITY lfsr IS
   GENERIC (poly, seed : std_logic_vector);
   PORT (clk, init, sin : IN std_logic;
         sout : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF lfsr IS
   COMPONENT dff IS
      PORT (clk, set, rst, d : IN std_logic;
            q : OUT std_logic);
   END COMPONENT ;
   FOR ALL : dff USE ENTITY WORK.dff(procedural);
   SIGNAL im_set, im_rst, im_d, im_q
                            : std_logic_vector (seed'RANGE);
BEGIN
   dffs: FOR i IN seed'RANGE GENERATE
      first: IF (i=seed'LEFT) GENERATE
         im_d(i) <= sin XOR im_q (seed'RIGHT);
      END GENERATE;
      rest:  IF (i/=seed'LEFT) GENERATE
         im_d(i) <= im_q (i+1) XOR
                    (poly (i) AND im_q (seed'RIGHT));
      END GENERATE;
      im_set(i) <= init AND seed(i);
      im_rst(i) <= init AND NOT seed(i);
      dffi: dff PORT MAP
         (clk, im_set(i), im_rst(i), im_d(i), im_q(i));
   END GENERATE;
   sout <= im_q (seed'RIGHT);
END ARCHITECTURE;
```

Figure 9.37 Structural LFSR VHDL Code

9.6.2 MISR

A *multiple input signature register* (MISR) is used for signature generation and data compression. Over a period of several clocks, parallel data of a MISR are compressed with the existing MISR data. The final data depends on the MISR initial data (*seed*) and its XOR and feedback structure (*poly*).

Figure 9.38 shows a MISR with a configurable polynomial (*poly*). The circuit has a reset input that initializes it to **0000**. This initial value is considered as the seed of this MISR example.

The VHDL code of Figure 9.39 corresponds to the hardware of Figure 9.38. The process block in this code handles resetting and signature generation. The generation of the signature is based on the input *poly* that configures the feedback (*shift_reg(i)*) connections to the XOR gates that are between the flip-flops.
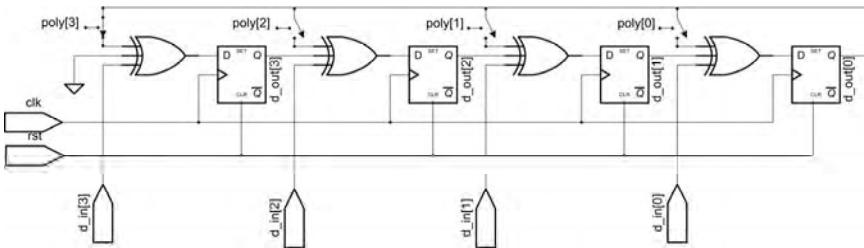
Figure 9.38 MISR Circuit

The process statement in the VHDL code of Figure 9.39 uses a for-loop to construct flip-flop inputs of the MISR. As shown, bit *i* of the shift-register input (*shift_reg(i)*) is set to a Boolean expression that involves the right-most shift-register bit, the corresponding poly bit (*poly(i)*), the shift register bit to its left (*shift_reg(i+1)*), and bit *i* of the *d_in* parallel input. Polynomial bit (*poly(i)*) is ANDed with the right-most bit to select or unselect the feedback path into the ith flip-flop input. The *shift_reg* vector formed by the process statement shown in the VHDL code is clocked into this register. The register output is assigned to *d_out* MISR output outside of the process.

```
ENTITY misr IS
   GENERIC (poly : std_logic_vector);
   PORT (clk, rst : IN std_logic;
         d_in : IN std_logic_vector;
         d_out : OUT std_logic_vector);
END ENTITY;
--
ARCHITECTURE procedural OF misr IS
   SIGNAL shift_reg :
          std_logic_vector (d_in'LENGTH DOWNTO 0);
BEGIN
   PROCESS (clk) BEGIN
      IF (clk='1' AND clk'EVENT) THEN
         IF rst='1' THEN
            shift_reg <= (OTHERS => '0');
         ELSE
            FOR i IN d_out'RANGE LOOP
               shift_reg (i) <=
                   (shift_reg(shift_reg'RIGHT) AND poly(i))
                   XOR shift_reg (i+1) XOR d_in (i);
            END LOOP;
         END IF;
      END IF;
   END PROCESS;
   d_out <= shift_reg (d_in'LENGTH-1 DOWNTO 0);
END ARCHITECTURE procedural;
```

Figure 9.39 MISR VHDL Description

The unconstrained design of MISR discussed above adjusts itself to the size of its input vector when instantiated. It is the responsibility of the instantiating architecture to check for conformity of the sizes of input, poly, and output vectors. A more robust model would use assertion statements to issue warning messages in case of non conformity.

9.7 Scan Design and Test Application

A testability technique that is often used in cores is the *scan design*. In principle, a scan design consists of a shift-register that brings in serial data from accessible circuit pins into inaccessible internal lines of a circuit under test. A scan register provides controllability and observability. This section uses a typical sequential circuit as the circuit to be tested. We show how a circuit is made testable by adding a scan register to its inputs, states, and outputs. We will then develop a testbench for this testable design. the testbench applies data to the testable circuit and collects the circuit responses through the scan register.

9.7.1 Starting Design

The starting design to be made testable is a Moore machine with six states, two inputs (*x1* and *x0*), and two outputs. The *x0* input resets the machine, and *x1* is the data input. While *x0* is 0, if a "110" sequence appears or *x1*, then output bit becomes 1, and if "101" is detected, output bit 1 becomes 1.

We have used the Huffman style of coding for describing this circuit. Instead of using an enumeration type for the states, six combinations of a 3-bit *std_logic_vector* represent present and next states of the machine. This is done so that state flip-flops are available to become part of a scan register. Partial code of this state machine is shown in Figure 9.40. State names and the combinational process are included in this VHDL code.

The rectangular box shown in the VHDL code of Figure 9.40 marks the section of the code that will be affected by the insertion of a scan register. Parts of this extra code are in the declarative part of the *procedural* architecture, and others are in the statement part.

```
ARCHITECTURE procedural OF moore4 IS
   CONSTANT reset  : std_logic_vector(2 downto 0) := "000";
   CONSTANT got1   : std_logic_vector(2 downto 0) := "001";
   CONSTANT got10  : std_logic_vector(2 downto 0) := "010";
   CONSTANT got11  : std_logic_vector(2 downto 0) := "011";
   CONSTANT got101 : std_logic_vector(2 downto 0) := "100";
   CONSTANT got110 : std_logic_vector(2 downto 0) := "101";
   SIGNAL p_state, n_state : std_logic_vector(2 downto 0);
   . . . Scan Code . . .
BEGIN
   . . . Scan Code . . .
   PROCESS (p_state, x1, x0) -- x1=x , x0=rst
   BEGIN
      n_state <= reset;
      outputs <= "00";
      IF x0 ='1' THEN n_state <= reset;
      ELSE
         CASE p_state IS
            WHEN reset =>
               IF x1 = '1' THEN n_state <= got1;
               ELSE n_state <= reset; END IF;
               outputs <= "00";
            WHEN got1 =>
               IF x1 = '0' THEN n_state <= got10;
               ELSE n_state <= got11; END IF;
               outputs <= "00";
            WHEN got10 =>
               IF x1 = '1' THEN n_state <= got101;
               ELSE n_state <= reset; END IF;
               outputs <= "00";
            WHEN got11 =>
               IF x1 = '1' THEN n_state <= got11;
               ELSE n_state <= got110; END IF;
               outputs <= "00";
            WHEN got101 =>
               IF x1 = '1' THEN n_state <= got11;
               ELSE n_state <= got10; END IF;
               outputs <= "10";
            WHEN got110 =>
               IF x1 = '1' THEN n_state <= got101;
               ELSE n_state <= reset; END IF;
               outputs <= "01";
            WHEN OTHERS =>
               n_state <= reset;
               outputs <= "00";
         END CASE;
      END IF;
   END PROCESS;
END ARCHITECTURE;
```

Figure 9.40 Testability Circuit

9.7.2 Scan Insertion

Scan insertion in our sequence detector circuit is done according to the diagram of Figure 9.41. The gray parts of this diagram are the original circuit, and the other parts are the inserted scan hardware. On the input side, the scan has a scan register and a multiplexer that selects between the primary inputs and the scanned input data.
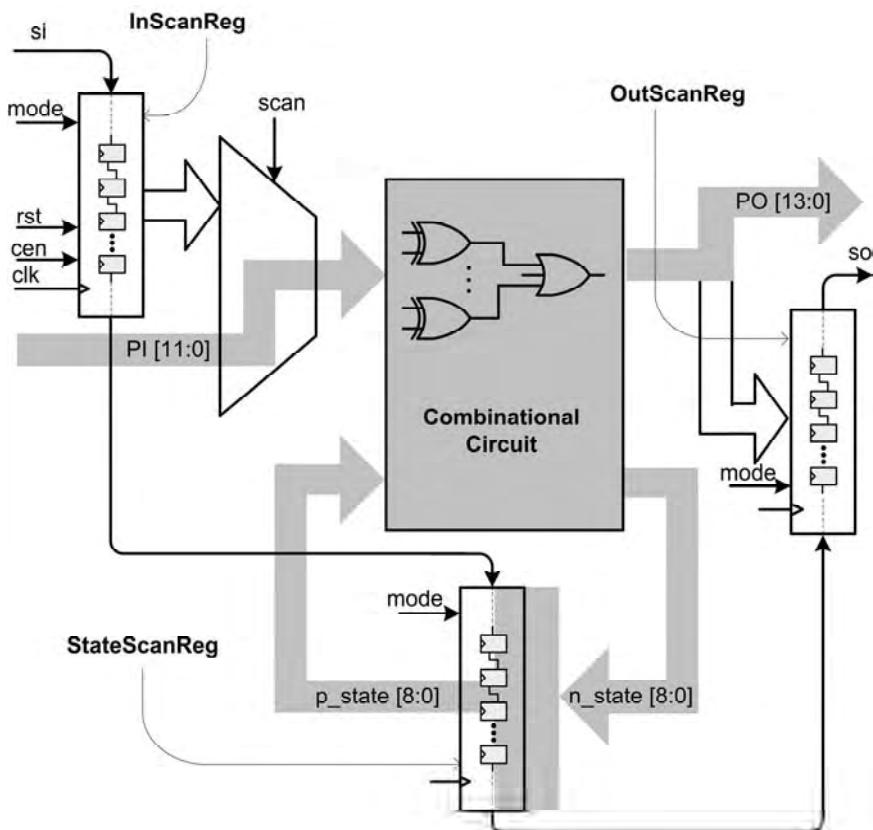
Figure 9.41 Scan Insertion in a Typical Sequential Circuit

On the output side, the scan register is setup to capture output and shift it out on its *so* (scan out) output. The state register is setup so that it either works in the parallel normal mode, or it shifts its serial input into the register. The VHDL code added to the code of Figure 9.40 for the implementation of the scan design of Figure 9.41 is shown in Figure 9.42. The code shown in this figure takes the place of the boxed area in Figure 9.40.

```
. . .
SIGNAL in_scan_reg, out_scan_reg
                            : std_logic_vector(1 downto 0);
    COMPONENT scan_reg IS
      PORT (si, clk, rst, cen, m : IN std_logic;
            so : OUT std_logic;
            parin : IN std_logic_vector;
            parout : OUT std_logic_vector);
    END COMPONENT;
    FOR ALL: scan_reg USE ENTITY WORK.scan_reg(procedural);
BEGIN
   InScanReg : scan_reg PORT MAP
                            (si, clk, rst, in_reg_cen,
                             m, in_reg_so, in_scan_reg,
                             in_scan_reg);
   StateScanReg : scan_reg PORT MAP
                            (in_reg_so, clk, rst, '1', m,
                             st_reg_so, n_state, p_state);
   OutScanReg : scan_reg PORT MAP
                            (st_reg_so, clk, rst, '1', m,
                             so, outputs, out_scan_reg);

  --MUX for selecting appropriate input data in either mode
   (x1, x0) <= in_scan_reg WHEN scan='1' ELSE inputs;
. . .
```

Figure 9.42 Scan Registers Inserted in Moore Machine Code

9.7.3 Scan Testbench

The scan testbench module shifts data into the serial input of our testable circuit and collects data from its serial output.

The testbench reads a line of input and output consisting of 10 bits. The first five bits are for the inputs of the CUT. Two of these bits are for the primary inputs of the CUT and the other three are for the present state of the machine. The second five bits of a data line read by the testbench constitutes the response of the circuit. Two of these five bits are the primary outputs, and the other three are collected from the state register.

The testbench reads a line, takes the input part, serializes it, shifts it into the circuit, clocks the circuit, shifts data out, and when all bits are collected, they are compared with the expected response of the circuit. As response bits are being shifted out, a new input is read and is shifted into the circuit. Because of this, when a line is read, its output part is saved as *expected_result* to be checked with the generated result when it is shifted out. The complete VHDL code of this testbench is shown in Figure 9.43.

```
ENTITY scan_tester IS
   PORT (clk, Nbar_T : IN std_logic;
         si : OUT std_logic; so : IN std_logic;
         m : OUT std_logic;
         in_reg_cen : OUT std_logic;
         scan, error : OUT std_logic;
         error : OUT std_logic);
END ENTITY;
--
ARCHITECTURE file_oriented OF scan_tester IS
   FILE test_file : TEXT OPEN READ_MODE IS "TestData.dat";
   SIGNAL exp_res_sig : std_logic_vector (4 DOWNTO 0);
   SIGNAL gen_res_sig : std_logic_vector (4 DOWNTO 0);
BEGIN
   Read_File : PROCESS
      VARIABLE l1 : LINE;
      VARIABLE test_in : std_logic_vector (4 DOWNTO 0);
      VARIABLE test_out : std_logic_vector (4 DOWNTO 0);
      VARIABLE generated_res:std_logic_vector (4 DOWNTO 0);
     VARIABLE expected_result:std_logic_vector(4 DOWNTO 0);
   BEGIN
      error <= '0';
      scan <= '0';
      WAIT UNTIL Nbar_T = '1';
      scan <= '1';
      test_out := "00000";--expected result for first test

      WHILE NOT ENDFILE(test_file) LOOP
         error <= '0';
         expected_result := test_out;
         exp_res_sig <= test_out;--just for debugging
         READLINE (test_file, l1);
         READ (l1, test_in);
         READ (l1, test_out);
         FOR i IN 4 DOWNTO 0 LOOP -- shift in input data
            WAIT UNTIL clk='1' AND clk'EVENT;--shift on pos
            m <= '1';
            in_reg_cen <= '1';
            si <= test_in(i);
            --collect on neg edge
            WAIT UNTIL clk='0' AND clk'EVENT;
            generated_res(i) := so;
            gen_res_sig(i) <= so; --just for debugging
         END LOOP;

         IF generated_res /= expected_result THEN
            error <= '1';
         END IF;

         WAIT UNTIL clk='1' AND clk'EVENT; --capture res
         m <= '0';
         in_reg_cen <= '0';
      END LOOP; --                                Continued
```

```
      WAIT UNTIL Nbar_T='0'; --normal operation
      scan <= '0';
      WAIT;
   END PROCESS;
END ARCHITECTURE file_oriented;
```

Figure 9.43 Testbench for Scan Testable Circuit

The testbench shown, uses TEXTIO procedures to read line of an input file. A while loop in this description reads a line of stimuli and responses until the end of file is reached. While application of data and capturing and shifting the response continues, if the generated and expected results do not match, an error will be issued. Other details of the mechanism of this test strategy can be found from its VHDL code.

9.7.4 Top Level Tester

At the top-level, the *procedural* architecture of the tester instantiates the testable CUT (Figure 9.40 and Figure 9.42) and its testbench (Figure 9.43), and operates the circuit in its test mode and its normal mode of operation. Figure 9.44 shows the VHDL code of the top-level tester of our example.

```
ENTITY sc_m4 IS END ENTITY ;
--
ARCHITECTURE procedural OF sc_m4 IS
   SIGNAL scan, m, si, so, in_reg_cen : std_logic;
   SIGNAL rst, x, error, clk, Nbar_T : std_logic := '0';
   SIGNAL inputs, outputs :  std_logic_vector(1 downto 0);

BEGIN
   Moore4 : ENTITY WORK.moore4(procedural)
            PORT MAP (rst, clk, scan, m, inputs,
                       outputs, si, so, in_reg_cen);
   Scan_Tester : ENTITY WORK.scan_tester(file_oriented)
                 PORT MAP (clk, Nbar_T, si, so, m,
                           in_reg_cen, scan, error);
   Nbar_T <= '1' AFTER 30 NS, '0' AFTER 600 NS;
   rst <= '1' AFTER 4 NS , '0' AFTER 18 NS;
   x <='1' AFTER 619 NS,'0' AFTER 633 NS,'1' AFTER 647 NS,
       '0' AFTER 675 NS,'1' AFTER 690 NS, '0' AFTER 758 NS;
   clk <= NOT clk AFTER 7 NS WHEN NOW <= 800 NS ELSE '0';

   inputs <= x&rst;

END ARCHITECTURE;
```

Figure 9.44 Top-level Tester

9.8 Memory BIST

Another testability method is to make a design self-testable. This way, input test data generation and output response analysis are all done inside the circuit. This test methodology is called *Built-In Self Test* or BIST. This section examines a memory BIST. The memory to be tested is the unconstrained memory model discussed in Chapter 6 that was also used in Chapter 8 for design of a stack.

9.8.1 Memory BIST Architecture

Figure 9.45 shows the memory BIST architecture. The memory to be tested in shown in gray and solid line blocks show the test circuitry. Test data that is to be applied is generated by this BIST circuitry and applied to the memory. As data is being read from the memory, it is compared with the reproduction of the same data that was written into specific memory locations. After writing and reading all locations, we expect all data read from the memory to be the same as those that were written into it.
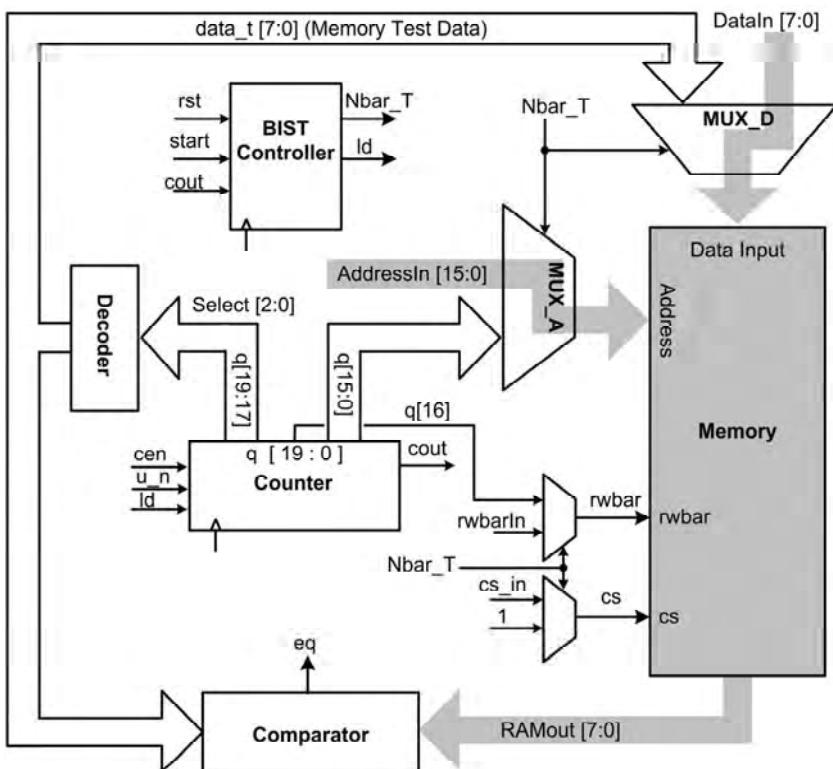
Figure 9.45 Memory BIST Architecture

9.8.1.1 Counter. InputData, address and switching between reading and writing the memory are provided by a counter. The least significant bits of the counter provide addressing for all locations of the memory. The counter bit to the left of the address group of bits toggles between write and read operations. The three most significant bits of the counter are decoded to generate eight test vectors for testing memory words.

Figure 9.46 shows the VHDL code of this counter. The counter carry-out (*cout*) becomes '1' when the count reaches its maximum.

```
ENTITY counter IS
   PORT (d_in : IN std_logic_vector;
         clk, ld, u_d, cen : IN std_logic;
         q : OUT std_logic_vector; cout : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF counter IS
   SIGNAL cnt_reg :
             std_logic_vector(q'LENGTH DOWNTO 0) REGISTER;
BEGIN
   cl: BLOCK (clk = '1' AND NOT clk'STABLE) BEGIN
      en: BLOCK (cen = '1' AND GUARD) BEGIN
         cnt_reg <= GUARDED ('0'&d_in) WHEN ld='1'
                      ELSE cnt_reg + 1 WHEN u_d = '1'
                      ELSE cnt_reg - 1;
      END BLOCK;
   END BLOCK;
   q <= cnt_reg(q'RANGE);
   cout <= cnt_reg(q'LEFT);
END ARCHITECTURE;
```

Figure 9.46 Memory BIST Counter

9.8.1.2 Decoder. The test data decoder uses a 3-bit input vector to lookup memory test patterns shown in Figure 9.47. The VHDL code of the decoder is shown in Figure 9.48.

Test Pattern	Input	Decoder Output
0	000	00000000
1	001	00001111
2	010	00110011
3	011	01010101
4	100	11111111
5	101	11110000
6	110	11001100
7	111	10101010

Figure 9.47 Decoder: Test Pattern Generation

```
ENTITY decoder IS
   PORT (input : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
         output : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END ENTITY ;
--
ARCHITECTURE procedural OF decoder IS
   TYPE std_logic_2d IS
      ARRAY (3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
   CONSTANT table : std_logic_2d :=(
                "01010101",
                "00110011",
                "00001111",
                "00000000");
   SIGNAL out_temp : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
      out_temp <= table(conv_integer(input(1 DOWNTO 0)));
      output <= out_temp WHEN input(2) = '0' ELSE
                NOT out_temp;
END ARCHITECTURE;
```

Figure 9.48 Memory BIST Pattern Generator Decoder

9.8.1.3 Multiplexers. The multiplexers of the BIST architecture of Figure 9.45 select between normal memory inputs and BIST provided inputs. When *Nbar_T* is '1', the memory in working in the normal mode and when this input becomes '0' it operates in test mode.

9.8.1.4 Comparator. Initially, the same test pattern is written into all memory locations, and then this data is read out from all locations. As data are being written and read the decoder input, and thus test patterns, remain unchanged. A comparator checks memory data with decoder output. When memory is being tested and data is being read, the comparator should have same data on both its inputs.

9.8.2 Test Session

A *test session* begins when the counter is all 0s, and ends when the counter reaches all 1s. Starting with all 0s, test pattern 0 is written in location 0. As the counter is incremented, this same pattern is written into all memory locations. When all memory locations are written into, the counter increments, causing the least significant part of it (address bits) to roll over to all 0s, and the *RWbar* bit to become 1. When this happens, the same data will now be read from all memory locations. When this is done, the *RWbar* bit becomes *1*, address starts back at 0, and the next test pattern starts being written into all locations. This process continues for all eight test patterns.

When done, all test patterns have been written into and read from all memory locations. While this is happening, the comparator checks for a mismatch and issues an error if it finds one.

9.8.3 BIST Controller

The BIST controller shown in Figure 9.49 starts the counter when it receives the *start* signal and waits for the carry-out (*cout*) of the counter.

```
ENTITY BIST_controller IS
   PORT (start, rst, clk, cout : IN std_logic;
         Nbar_T, ld : OUT std_logic);
END ENTITY ;
--
ARCHITECTURE procedural OF BIST_controller IS
   TYPE state IS (reset, test);
   SIGNAL current : state := reset;
BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
         IF rst = '1' THEN
            current <= reset;
         ELSE
            CASE current IS
               WHEN reset =>
                  IF start = '1' THEN current <= test;
                  ELSE current <= reset; END IF;
               WHEN test =>
                  IF cout = '1' THEN current <= reset;
                  ELSE current <= test; END IF;
               WHEN OTHERS => current <= reset;
            END CASE;
         END IF;
      END IF;
   END PROCESS;
   Nbar_T <= '1' WHEN current = test ELSE '0';
   ld <= '1' WHEN current = reset ELSE '0';
END ARCHITECTURE;
```

Figure 9.49 BIST Controller

9.8.4 BIST Structure

The VHDL code of Figure 9.50 shows the complete BIST structure including the RAM that is being tested. Components instantiated in this description are according to the diagram of Figure 9.45. In addi-

tion to the components instantiated, this code has a process state-
ment that issues the *fail* flag if the comparator finds a mismatch.

```
ENTITY BIST IS

   PORT (start, rst, clk, csin, rwbarin : IN std_logic;
         opr : IN BOOLEAN;
         address : IN std_logic_vector;
         datain : IN std_logic_vector;
         dataout : OUT std_logic_vector;
         fail : OUT std_logic);

END ENTITY ;
--
ARCHITECTURE structural OF BIST IS

   SIGNAL zero : std_logic_vector (9 DOWNTO 0)
              := (OTHERS => '0');
   SIGNAL cout, ld, Nbar_T, cs : std_logic;
   SIGNAL rwbar, gt, eq, lt : std_logic;
   SIGNAL q : std_logic_vector (9 DOWNTO 0);
   SIGNAL data_t : std_logic_vector (7 DOWNTO 0);
   SIGNAL ramin, ramout : std_logic_vector (7 DOWNTO 0);
   SIGNAL ramaddr : std_logic_vector (5 DOWNTO 0);

BEGIN

   CNTRL : ENTITY WORK.BIST_controller(procedural)
             PORT MAP (start, rst, clk, cout, Nbar_T, ld);

   CNT : ENTITY WORK.counter(procedural)
             PORT MAP (zero, clk, ld, '1', '1', q, cout);
   DEC : ENTITY WORK.decoder(procedural)
             PORT MAP (q(9 DOWNTO 7), data_t);
   MUX_D : ENTITY WORK.multiplexer(procedural)
             PORT MAP (datain, data_t, Nbar_T, ramin);
   MUX_A : ENTITY WORK.multiplexer(procedural)
             PORT MAP (address, q(5 DOWNTO 0),
             Nbar_T, ramaddr);
   rwbar <= rwbarin WHEN Nbar_T = '0' ELSE q(6);

   cs <= csin WHEN Nbar_T = '0' ELSE '1';

   RAM : ENTITY WORK.std_logic_ram (behavioral)
             PORT MAP (ramaddr, ramin, ramout,
             cs, rwbar, opr);

   CMP: ENTITY WORK.compartor (expression)
             PORT MAP (data_t, ramout, gt, eq, lt);


   --                                            Continued
```

```
   PROCESS(clk)
   BEGIN
      IF (clk = '1' AND clk'EVENT) THEN

         IF (Nbar_T = '1') AND (rwbar = '1') AND (opr)
         THEN
            IF eq = '0'
            THEN
               fail <= '1';
            END IF;
         ELSE
            fail <= '0';
         END IF;

      END IF;
   END PROCESS;

   dataout <= ramout;
END ARCHITECTURE;
```

Figure 9.50 Memeory BIST Structure

9.8.5 BIST Tester

The memory and its BIST are tested in the *timed* architecture of the *BIST_Tester* testbench. This testbench is shown in Figure 9.51. The testbench initially loads external file data into the memory at time 5 ns when *operate* becomes TRUE. Then at some arbitrary times data is written into and read from the memory. The BIST test session begins when *start* becomes 1 at time 50 ns. Testing continues until all RAM locations have been tested. While the memory is being tested, external read and write operations are ignored.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY BIST_tester IS END ENTITY BIST_tester;

ARCHITECTURE timed OF BIST_tester IS
   SIGNAL ramin, ramout : std_logic_vector (7 DOWNTO 0);
   SIGNAL addr : std_logic_vector (5 DOWNTO 0);
   SIGNAL cs, rwbar, start : std_logic := '0';
   SIGNAL rst, fail, clk : std_logic := '0';
   SIGNAL operate : BOOLEAN;
BEGIN

--                                              Continued
```

```
   UU1: ENTITY WORK.BIST (structural)
        PORT MAP (start, rst, clk, cs, rwbar, operate,
                  addr, ramin, ramout, fail);
   clk <= NOT clk AFTER 5 NS WHEN NOW <= 800 NS ;
   operate <= TRUE AFTER 5 NS, FALSE AFTER 800 NS;
   cs <= '0', '1' AFTER 15 NS, '0' AFTER 337 NS;
   rwbar <= '1', '1' AFTER 190 NS;
   addr <= "101100" AFTER 020 NS,"101110" AFTER 040 NS…

   ramin<="11110001" AFTER 010 NS,"00101100" AFTER 030 NS…

      rst <= '0';
      start <= '0', '1' AFTER 50 NS;

END ARCHITECTURE timed;
```

Figure 9.51 Memory and BIST Testbench

9.9 Summary

This chapter discussed various test issues in VHDL. We started with testbench techniques and discussed several complete testbench examples that utilized our techniques. This part was followed by modeling test applications and testability hardwares in VHDL. In this part we discussed a typical scan design and a memory BIST. The memory BIST example in this chapter showed the role of a testbench in verifying performance of a BIST that itself is for testing a different module. In the examples of this chapter we used the standard *std_logic_1164* package, and took advantage of TEXTIO procedures whenever possible. The examples were made general and in many cases unconstrained so that they can easily be adapted to other designs.

Problems

9.1   Insert delays in the Boolean expressions of the array multiplier of Section 8.3.1 and recompile your design. Generate a testbench for this new array multiplier. Wait for the worst-case delay of your circuit before reading the outputs from the multiplier outputs. The testbench should check the expected result with the result coming from the multiplier.

9.2   Write an interactive testbench for the Booth multiplier of Section 8.3.3. Instantiate the multiplier, read external data from external files, apply data to the multiplier, and check the results with your expected results.

9.3   In test applications it becomes necessary to inject a fault into a circuit line. The fault overrides the normal value of the line and sets it to '1' or '0' depending on stuck-at-1 or stuck-at-0 faults that are be-ing injected. We are to develop utilities to enable us to inject faults into lines of *std_logic* type. For this purpose, every circuit line be-comes a record with a normal *std_logic* logic value and a field that specifies the fault if there is one. This filed takes *sa1*, *sa0* and *nofault* values for stuck-at-1, stuck-at-0 and none faulty values. This value decides if a line takes its normal value or a faluty '1' or '0'. A) Show *std_ulogic_faultabe* type of record as described above. B) Show utili-ties for generating a resolution function to resolve between multiple drivers for lines of type *std_ulogic_faultabe* C) Write a resolution function that generates a normal *std_logic* value if no fault is injected on a line, or it generates '1' or '0' if a fault is injected on one of the drivers of a line. For this you can use the *resolution_table* and/or the *resolved* function of the *std_logic_1164* package.

9.4   Fault simulation is the process of injecting faults into circuit lines and simulating to observe propagation of faulty values. Fault simulation requires a method of injecting faults into circuit lines. At the gate level, fault injection can be done by forcing faulty values on circuit outputs. For this to happen, a gate can be given a generic pa-rameter for each of its outputs called *fault* that can take either of three values *sa0* (Stuck at 0), *sa1* (Stuck at 1) or *none*. When this ge-neric indicates a faulty value, that value must be placed on circuit outputs, otherwise the gate will produce its normal logic value that is based on its inputs. A) Declare all types necessary for this simple fault injection procedure. B) Write complete faultable description of a two input NAND gate. C) Show instantiation of this gate for insert-ing a *sa0* fault on its output.

9.5   Modify the memory BIST of Section 9.8 to replace its compara-tor with a MISR. The MISR connects to the output of the memory. The MISR signature is presumed to be pre-calculated during simula-tion. With each read from the memory, the MISR is updated. After a complete test session, compare the expected signature with the signa-ture in MISR and report an error is a discrepancy is found.

9.6   Modify the scan design of Section 9.7 in such a way that an LFSR generates the input serial data and a SISR (Serial Input Signa-ture Register) collects the output serial data. An SISR is a MISR with only a serial input. Turn this design into a BIST so that the BIST controller starts the serial data generation and when a test session ends, it compares the signature in SISR with a pre-calculated signa-

ture. Your BIST controller has a fixed number of clocks for a test session. When the count reaches the pre-determined number of clocks, the controller compares the signature with the expected (determined by simulation) signature.

9.7   The following partial code is the sequential block of a controller coded in Huffman style of coding. Modify this code to add to it a Shadow register. This requires a 10-bit register and control signals for shifting serial data, applying test data to the circuit and reading circuit response into the Shadow register. The test hardware also requires a serial-in and a serial-out for serial data being shifted in and serial data being shifted out (*Sin* and *Sout*). The test circuitry (those associated with the Shadow register) uses the main circuit clock, but has a test-enable input, *Ten*, that enables clocking of the shadow register. In the normal mode of operation, if *Ten* is active, data from *Sin* will be shifted in and data of the Shadow register will be shifted out on *Sout*. To apply the contents of the Shadow register to the combinational part of the Circuit-Under-Test (*P_state*), *TestApplyAndCapture* must become '1'. While *TestApplyAndCapture* is '1', the next clock edge will load the data output from the combinational part of the hardware (*N_state*) into the Shadow register. After this, *TestApplyAndCapture* can become '1' and data captured in the Shadow register can start being shifted out. While this is being done, the circuit operates in its normal mode of operation. Show hardware description corresponding to the Shadow register and its application to the CUT.

```
SIGNAL N_state : std_logic_vector (9 DOWNTO 0);
SIGNAL P_state : std_logic_vector (9 DOWNTO 0);
. . .

PROCESS (clk, rst) BEGIN

   IF (rst = '1' AND rst'EVENT) THEN
      P_state <= "000000000";
   ELSIF (clk = '1' AND clk'EVENT) THEN
       P_state <= N_state;
   END IF;

END PROCESS;
```

9.8   Given the following VHDL code of a 1011 detector, modify it so that the control flip-flops can become chained in a scan register when the circuit is in the test mode. The test mode is entered when *TMode* becomes '1'. In this case, data on *TDin* will be shifted into the register (flip-flops of the controller) with each *Clk* (*Clk* is the normal clock

of the circuit) and existing data in the register will be clocked out onto
TDout signal. Shifting data on *TDin* into the register happens simul-
taneous with data being shifted out on *TDout*. Shifted data becomes
the *present_state* of the circuit. To clock the *next_state*, *TMode* must
become '0' and clock applied. When *TMode* is '0', the circuit works in
normal mode.

```
ENTITY asynch_reset_detector IS
  PORT (x, r, clk : IN std_logic; z : OUT std_logic);
END;

ARCHITECTURE behavioral OF asynch_reset_detector IS
   TYPE state IS std_logic_vector (2 DOWNTO 0);
   SIGNAL next_state, present_state : state;
   CONSTANT a : state := "000";
   CONSTANT b : state := "001";
   CONSTANT c : state := "010";
   CONSTANT d : state := "011";
   CONSTANT e : state := "100";
BEGIN

  reg : PROCESS (clk, r) BEGIN
     IF r = '1' THEN present_state <= a;
     ELSIF (clk'EVENT AND clk = '1') THEN
        present_state <= next_state;
      END IF;
  END PROCESS;

  logic : PROCESS (present_state, x)
  BEGIN
     CASE present_state IS
        WHEN a =>
           IF x = '0' THEN next_state <= a;
           ELSE next_state <= b; END IF;
        WHEN b =>
           IF x = '0' THEN next_state <= c;
           ELSE next_state <= b; END IF;
        WHEN c =>
           IF x = '0' THEN next_state <= a;
           ELSE next_state <= d; END IF;
        WHEN d =>
           IF x = '0' THEN next_state <= c;
           ELSE next_state <= e; END IF;
        WHEN e =>
           IF x = '0' THEN next_state <= c;
           ELSE next_state <= b; END IF;
    END CASE;
  END PROCESS;

  z <= '1' WHEN present_state = e;

END behavioral;
```

9.9 The Entity declaration of our CUT (Circuit Under Test) is given. Write a test bench to read test data from file *CUT_test.inp*, apply them to the CUT, read output data from the CUT, and write output of the CUT to *CUT_test.out*. Note that applying data and reading data must be concurrent. The format of the data input is shown below. A line of input data begins with the time of data followed by an 8-bit *std_logic_vector*, a 1-bit *std_logic_vector*, and a 9-bit string. The format of the output is shown below. A line of output begins with the time of data followed by a 2-digit Hexadecimal data, a one-bit *std_logic_vector*, and a 5-bit string. The 9-character input string is applied to the CUT as it is read from the input file. The 5-character output string is read from the CUT and is written into the output file exactly as read from the CUT. Data read and applied to the CUT are as read from or to the corresponding files; i.e., no recoding is needed. Input times are the times at which the particular data is to be applied to the CUT. Output times are times at which an output of the CUT changes value.

```
ENTITY Circuit_Under_Test IS
   PORT (   inbus: IN std_logic_vector (7 DOWNTO 0);
            inflag: IN std_logic;
            instr: IN STRING (1 TO 9);
            outbus: OUTstd_logic_vector (7 DOWNTO 0);
            outflag: OUT std_logic;
            status: OUT STRING (1 TO 5) );
END Circuit_Under_Test;
```

Input Data:
```
0100 NS 110011ZZ 1 LDA R1 R0
0240 NS 10011111 0 JMP 000F5
0740 NS 10011111 1 STA 000F5
```

Output Data:
```
0109 NS F7 1 Load
0257 NS AA 0 Read
0731 NS 05 1 Write
```

Suggested Reading

Abramovici, Miron, Melvin A. Breuer, and Arthur D. Friedman, *Digital System Testing and Testable Design*, Computer Science Press, 1990, ISBN: 0-7167-8179-4.

Accellera, *Open Verification Library: Assertion Monitor Reference Manual,* www.accellera.org, v1.0, 2005.

Bening, Lionel and Harry D. Foster, *Principles of Verifiable RTL Design Second Edition - A Functional Coding Style Supporting Verification Processes in Verilog*, Springer, 2nd edition, 2001, ISBN: 0792373685.

Bushnell, Michael L., and Vishwani D. Agrawal, *Essential of Electronic Testing, for Digital Memory & Mixed-Signal VLSI Circuits*, Klwuer Academic Publishing, 2000, ISBN: 0-7923-7991-8.

Chu, Pong, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press, ISBN: 0471720925.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

Jha, Niraj, and Sandeep Gupta, *Testing of Digital Systems*, Cambridge University Press, 2003, ISBN: 0-521-77356-3.

Lipsett, Roger, and Cary Ussery, *VHDL Hardware Description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Miczo, Alexander, *Digital Logic Testing and Simulation, 2nd Edition*, John Wiley & Sons, Inc., Publishing, 2003, ISBN: 0-471-43995-9.

Navabi, Zainalabedin, *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification*, 2006, McGraw Hill-Professional, ISBN: 0070144564-1.

Navabi, Zainalabedin, *Embedded Core Design with FPGAs*, 2006, McGraw Hill-Professional, ISBN: 0071474811.

Perry, Douglas L., and Harry Foster, *Applied Formal Verification for Digital Circuit Design*, 2005, McGraw-Hill Professional, ISBN: 978-0071443722.

10 Design, Test and Application of a Processor Core

Processors play a major role in the design of embedded systems. An embedded processor may be used as the central processing unit of an embedded system, or it may just be used as a convenient and fast way of implementing a hardware function. With embedded systems, understanding how a processor works, its software, and software utilities, such as compilers and assemblers, are key topics that a hardware designer should be familiar with.

This chapter provides the basic concepts and techniques that are necessary for design and utilization of an embedded processor. The necessary background materials for this chapter are RT level design and test techniques presented in Chapters 8 and 9. This provides the necessary background for understanding the hardware of a processor. In the description of processor hardware we begin with a simple processor example that has the basic properties found in most processing units and then continue with a more realistic processor. The hardware and software of this processor, which we refer to as SAYEH, becomes the main focus of this chapter. In a top-down fashion, we will show control-data partitioning of our example processors and design and implementation of the individual parts of these machines. After the completion of the processor design in VHDL, we will show how this processor can be used as an embedded core for the design of an FIR filter. This examples shows utilization of hardware processors.

395

10.1 Design of SAYEH Processor Core

This section shows design and description of a small processor in VHDL. The CPU is SAYEH (Simple Architecture, Yet Enough Hardware) that has been designed for educational and benchmarking purposes. The design is simple, and follows the design strategy used for the Adding CPU of Chapter 8.

This section heavily relies on the reader knowing the implementation details that we presented in Chapter 8. Unlike the Adding CPU, SAYEH is a processor with a set of instructions and an architecture that can be used for real embedded processor applications. However, the design, implementation, VHDL coding style, and test strategy of SAYEH are very similar to those of the Adding CPU. Because of the size of the VHDL code of SAYEH, we will not show complete codes of all components of SAYEH in this chapter. Instead, In many cases we show code outlines similar to those of the Adding CPU.

10.1.1 Details of Processor Functionality

The simple CPU example discussed here has a register file that is used for data processing instructions. The CPU has a 16-bit data bus and a 16-bit address bus. The processor has 8 and 16-bit instructions. Short instructions contain shadow instructions, which effectively pack two such instructions into a 16-bit word. Figure 10.1 shows SAYEH interface signals.
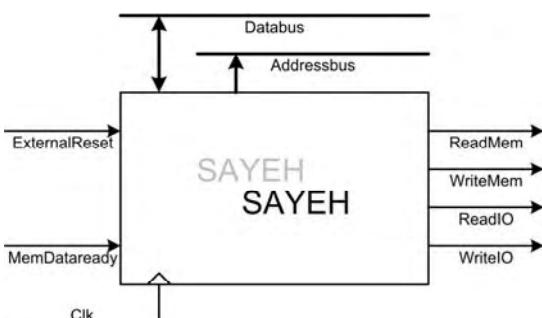

Figure 10.1 SAYEH Interface

10.1.1.1 CPU Components. SAYEH uses its register file for most of its data instructions. Addressing modes of this processor also take advantage of this structure. Because of this, the addressing hardware of SAYEH is simple and the register file output is used in address calculations.

SAYEH components that are used by its instructions include the standard registers such as the Program Counter, Instruction Register, the Arithmetic Logic Unit, and Status Register. In addition, this processor has a register file forming registers *R0*, *R1*, *R2* and *R3* as well as a Window Pointer that defines *R0*, *R1*, *R2* and *R3* within the register file. CPU components and a brief description of each are shown below.

- **PC:** Program Counter, 16 bits
- **R0, R1, R2, and R3:** General purpose registers part of the register file, 16 bits
- **Reg File:** The general purpose registers form a window of 4 in a register file of 64 registers
- **WP:** Window Pointer points to the register file to define *R0*, *R1*, *R2* and *R3*, 6 bits
- **IR:** Instruction Register that is loaded with a 16-bit, an 8-bit, or two 8-bit instructions, 16 bits
- **ALU:** The ALU that can AND, OR, NOT, Shift, Compare, Add, Subtract and Multiply its inputs, 16 bit operands
- **Z flag:** Becomes **1** when the ALU output is **0**
- **C flag:** Becomes **1** when the ALU has a carry output

10.1.1.2 SAYEH Instructions. The general format of 8-bit and 16-bit SAYEH instructions is shown in Figure 10.2. The 16-bit instructions have the *Immediate* field and the 8-bit instructions do not. The *OP-CODE* filed is a 4-bit code that specifies the type of instruction. The *Left* and *Right* fields are two bit codes selecting *R0* through *R3* for source and/or destination of an instruction. Usually, *Left* is used for destination and *Right* for source. The *Immediate* filed is used for immediate data, or if two 8-bit instructions are packed, it is used for the second instruction.

15 12	11 10	09 08	07 00
OPCODE	*Left*	*Right*	*Immediate*

Figure 10.2 SAYEH Instruction Format

Our processor has a total of 29 instructions as shown in Table 10.1. Instructions with *I* immediate field are 16-bit instructions and the rest are 8-bit instructions. Instructions that use the *Destination* and *Source* fields (designated by *D* and *S* in the table of instruction set) have an opcode that is limited to 4 bits. Instructions that do not

require specification of source and destination registers use these fields as opcode extensions. In addition to *nop*, hex code 0F is used as filler for the right most 8-bits of a 16-bit word that only contains an 8-bit instruction in its 8 left-most bits.

Table 10.1 Instruction Set of SAYEH

Instruction Mnemonic and Definition		Bits 15:0	RTL notation: *comments or condition*
nop	No operation	0000-00-00	No operation
hlt	Halt	0000-00-01	Halt, fetching stops
szf	Set zero flag	0000-00-10	Z <= '1'
czf	Clr zero flag	0000-00-11	Z <= '0'
scf	Set carry flag	0000-01-00	C <= '1'
ccf	Clr carry flag	0000-01-01	C <= '0'
cwp	Clr Window pointer	0000-01-10	WP <= "000"
mvr	Move Register	0001-D-S	$R_D <= R_S$
lda	Load Addressed	0010-D-S	$R_D <= (R_S)$
sta	Store Addressed	0011-D-S	$(R_D) <= R_S$
inp	Input from port	0100-D-S	In from R_S write to R_D
oup	Output to port	0101-D-S	Out to port R_D from R_S
and	AND Registers	0110-D-S	$R_D <= R_D \& R_S$
orr	OR Registers	0111-D-S	$R_D <= R_D \mid R_S$
not	NOT Register	1000-D-S	$R_D <= \sim R_S$
shl	Shift Left	1001-D-S	$R_D <= sla\ R_S$
shr	Shift Right	1010-D-S	$R_D <= sra\ R_S$
add	Add Registers	1011-D-S	$R_D <= R_D + R_S + C$
sub	Subtract Registers	1100-D-S	$R_D <= R_D - R_S - C$
mul	Multiply Registers	1101-D-S	$R_D <= R_D * R_S$:*8-bit multiplication*
cmp	Compare	1110-D-S	R_D, R_S *(if equal:Z=1; if $R_D<R_S$: C=1)*
mil	Move Immd Low	1111-D-00-I	$R_{DL} <= \{8\text{'bZ, I}\}$
mih	Move Immd High	1111-D-01-I	$R_{DH} <= \{I, 8\text{'bZ}\}$
spc	Save PC	1111-D-10-I	$R_D <= PC + I$
jpa	Jump Addressed	1111-D-11-I	$PC <= R_D + I$
jpr	Jump Relative	0000-01-11-I	PC <= PC + I
brz	Branch if Zero	0000-10-00-I	PC <= PC + I :*if Z is 1*
brc	Branch if Carry	0000-10-01-I	PC <= PC + I :*if C is 1*
awp	Add Win pntr	0000-10-10-I	WP <= WP + I

In the instruction set, addressed locations in the memory are indicated by enclosing the address in a set of parenthesis. For these instructions, the processor issues *ReadMem* or *WriteMem* signals to the memory. When input and output instructions (*inp, oup*) are executed, SAYEH issues *ReadIO* or *WriteIO* signals to its IO devices.

10.1.2 SAYEH Datapath

The datapath of SAYEH is shown in Figure 10.3. The main components of this machine are: *Addressing Unit* that consists of *PC (Program Counter)* and *Address Logic*, *IR (Instruction Register)*, *WP (Window Pointer)*, *Register File* that consists of *Left Decoder1* and *Right Decoder2*, *ALU (Arithmetic Logic Unit)*, and *Flags*. As shown in Figure 10.3, these components are either hardwired or connected through three-state busses. Component inputs with multiple sources, such as the right hand side input of *ALU*, use three-state busses. Three-state busses in this structure are *Dastabus* and *OpndBus*. In this figure, signals that are in italic are control signals issued by the controller. These signals control register clocking, logic unit operations and placement of data in busses.
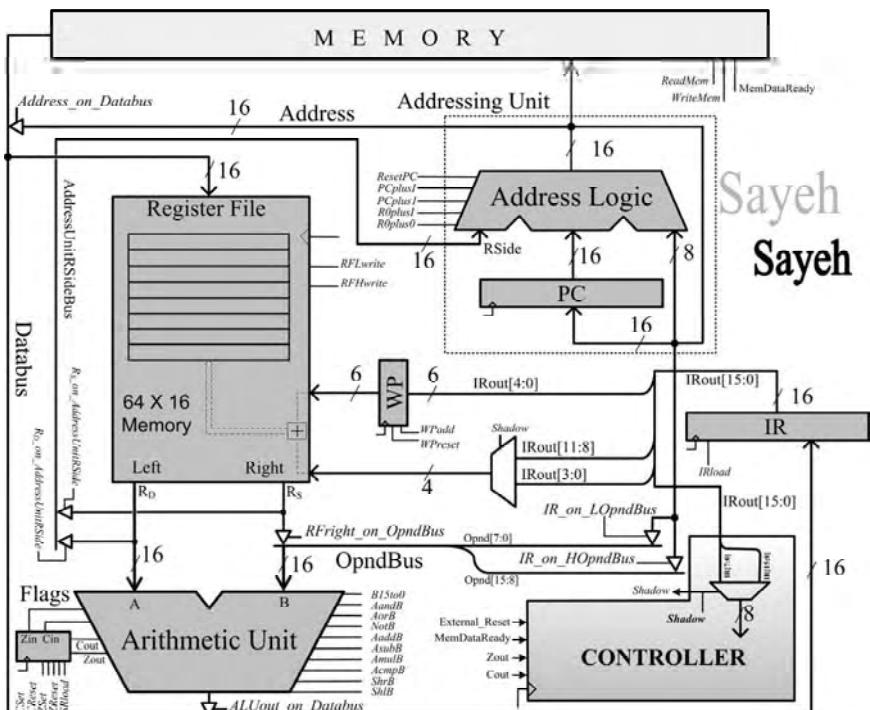

Figure 10.3 SAYEH Datapath

10.1.2.1   Datapath Components.   Figure 10.4 shows the hierarchical structure of SAYEH components. The processor has a *datapath* and a *controller*. *Datapath* components are *Addressing Unit*, *IR*, *WP*, *Register File*, *Arithmetic Unit,* and the *Flags* register. The *Addressing Unit* is further partitioned into the *PC* and *Address Logic*.

The *Addressing Logic* is a combinational circuit that is capable of adding its inputs to generate a 16-bit output that forms the address for the processor memory. *Program Counter* and *Instruction Register* are 16-bit registers. *Register File* is a two-port memory and a file of 64 16-bit registers. The *Window Pointer* is a 6-bit register that is used as the base of the *Register File*. Specific registers for read and write (*R0*, *R1*, *R2* or *R3*) in the *Register File* are selected by its 4-bit input bus coming from the *Instruction Register*. Two bits are used to select a source register and other two bits select the destination register.

When the Window Pointer is enabled, it adds its 6-bit input to its current data. The *Flags* register is a 2-bit register that saves the flag outputs of the *Arithmetic Unit*. The *Arithmetic Unit* is a 16-bit arithmetic and logic unit that has logical, shift, add and compare operations. A 9-bit input selects one of the nine functions of the ALU. This code is provided by the processor controller.
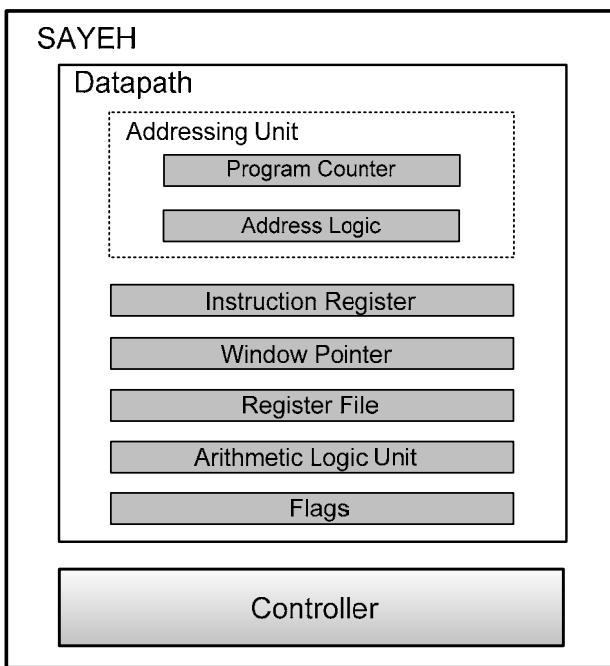

Figure 10.4 SAYEH Hierarchical Structure

Controller of SAYEH has eleven states for various reset, fetch, decode, execute, and halt operations. Signals generated by the controller control logic unit operations and register clocking in the datapath.

SAYEH sequential data components and its controller are triggered on the rising edge of the main system clock. Control signals remain active after one rising edge through the next. This duration allows for propagation of signals through the busses and logic units in the datapath.

10.2 SAYEH VHDL Description

SAYEH is described according to the hierarchical structure of Figure 10.4. Data components are described separately, and then wired to form the datapath. Controller is described in a single VHDL module. In the complete SAYEH description, the datapath and controller are wired together.

The coding style used for the description of this processor is similar to that of the description of *AddingCPU* of Chapter 8. The CD accompanying this book has the complete code of this processor. As previously mentioned, only an outline and key parts of the VHDL code of SAYEH will be included in this chapter.

10.2.1 Data Components

Combinational and sequential SAYEH data components are described here. The combinational ones are like the ALU that performs arithmetic and logical operations. The function of such units is controlled by the controller. The sequential components are clocked with the negative edge of the main CPU clock. These components have functionalities like loading and resetting and are controlled by the controller.

10.2.1.1 Addressing Unit. The Addressing Unit of Figure 10.5 consists of the Program Counter and Address Logic. The Program Counter is a simple register with enabling and resetting mechanisms, while the Address Logic is a small arithmetic unit that performs adding and incrementing for calculating PC or memory addresses.

This unit has a 16-bit input coming from the Register File, an 8-bit input from the Instruction Register, and a 16-bit address output. Control signals of the Addressing Unit are *ResetPC, PCplusI, PCplus1, RplusI, Rplus0,* and *PCenable*. These control signals select what goes on the output of this unit. Shown in Figure 10.6 is the VHDL code of the Program Counter. The Address Logic of Figure 10.7

uses control signal inputs of the Addressing Unit to generate input data to the Program Counter via the *PCout* of Figure 10.5. Constants defined and used here correspond to one-hot control signals from the controller.

```
ENTITY AddressingUnit IS
    PORT (
        Rside : IN std_logic_vector (15 DOWNTO 0);
        Iside : IN std_logic_vector (7 DOWNTO 0);
        Address : OUT std_logic_vector (15 DOWNTO 0);
        clk, ResetPC, PCplusI, PCplus1 : IN std_logic
        RplusI, Rplus0, EnablePC : IN std_logic
    );
END AddressingUnit;

ARCHITECTURE dataflow OF AddressingUnit IS
    COMPONENT pc  . . .  END COMPONENT;
    COMPONENT al  . . .  END COMPONENT;

    SIGNAL pcout : std_logic_vector (15 DOWNTO 0);
    SIGNAL AddressSignal : std_logic_vector (15 DOWNTO 0);
BEGIN
    Address <= AddressSignal;
    l1 : pc PORT MAP (EnablePC, AddressSignal, clk, pcout);
    l2 : al PORT MAP
            (pcout, Rside, Iside, AddressSignal,
             ResetPC, PCplusI, PCplus1, RplusI, Rplus0);
END dataflow;
```

Figure 10.5 *AddressingUnit* **VHDL Code**

```
ENTITY ProgramCounter IS
    PORT (
        EnablePC : IN std_logic;
        input : IN std_logic_vector (15 DOWNTO 0);
        clk : IN std_logic;
        output : OUT std_logic_vector (15 DOWNTO 0)
    );
END ProgramCounter;

ARCHITECTURE dataflow OF ProgramCounter IS BEGIN
    PROCESS (clk) BEGIN
        IF (clk = '1') THEN
            IF (EnablePC = '1') THEN
                output <= input;
            END IF;
        END IF;
    END PROCESS;
END dataflow;
```

Figure 10.6 *ProgramCounter* **VHDL Code**

```
ENTITY AddressLogic IS
   PORT (
      PCside, Rside : IN std_logic_vector (15 DOWNTO 0);
      Iside : IN std_logic_vector (7 DOWNTO 0);
      ALout : OUT std_logic_vector (15 DOWNTO 0);
      ResetPC, PCplusI, PCplus1,RplusI,Rplus0: IN std_logic
   );
END AddressLogic;

ARCHITECTURE dataflow OF AddressLogic IS
   CONSTANT one   : std_logic_vector (4 DOWNTO 0)
                  := "10000";
   CONSTANT two   : std_logic_vector (4 DOWNTO 0)
                  := "01000";
   CONSTANT three : std_logic_vector (4 DOWNTO 0)
                  := "00100";
   CONSTANT four  : std_logic_vector (4 DOWNTO 0)
                  := "00010";
   CONSTANT five  : std_logic_vector (4 DOWNTO 0)
                  := "00001";
BEGIN
   PROCESS (PCside, Rside, Iside, ResetPC,
            PCplusI, PCplus1, RplusI, Rplus0)
      VARIABLE temp : std_logic_vector (4 DOWNTO 0);
   BEGIN
      temp := (ResetPC& PCplusI& PCplus1& RplusI& Rplus0);
      CASE temp IS
         WHEN one   => ALout <= (OTHERS=>'0');
         WHEN two   => ALout <= PCside + Iside;
         WHEN three => ALout <= PCside + 1;
         WHEN four  => ALout <= Rside + Iside;
         WHEN five  => ALout <= Rside;
         WHEN OTHERS => ALout <= PCside;
      END CASE;
   END PROCESS;
END dataflow;
```

Figure 10.7 *AddressLogic* **VHDL Code**

10.2.1.2 Arithmetic Unit. The ALU of SAYEH has nine functions shown in Table 10.2. A Mnemonic and the control code of each function are also shown in this table. The complete code of the ALU is shown in Figure 10.8. For readability, constants corresponding to the mnemonics of Table 10.2 are defined and used in this code.

ALU control codes are one-hot. For example, the select input that causes the ALU to perform the add operation is 0000001000, and it is defined as *AaddBH*. Control inputs of this unit are *B15to0*, *AandB*, *AorB*, *notB*, *shlB*, *shrB*, *AaddB*, *AsubB*, *AmulB* and *AcmpB* that select its various operations.

```
ENTITY ArithmeticUnit IS
    PORT (
      A, B : IN std_logic_vector (15 DOWNTO 0);
      B15to0, AandB, AorB, notB : IN std_logic;
      shlB, shrB, AaddB, AsubB : IN std_logic;
      AmulB, AcmpB : IN std_logic;
      aluout : OUT std_logic_vector (15 DOWNTO 0);
      cin : IN std_logic;
      zout, cout : OUT std_logic
    );
END ArithmeticUnit;

ARCHITECTURE dataflow OF ArithmeticUnit IS

    COMPONENT mult
      PORT ( x, y : IN std_logic_vector (7 DOWNTO 0);
             z : OUT std_logic_vector (15 DOWNTO 0) );
    END COMPONENT;
    FOR ALL : mult USE ENTITY work.mult_8by8 (bitwise);

    CONSTANT B15to0H : std_logic_vector (9 DOWNTO 0)
                    := "1000000000";
    CONSTANT AandBH  : std_logic_vector (9 DOWNTO 0)
                    := "0100000000";
    CONSTANT AorBH   : std_logic_vector (9 DOWNTO 0)
                    := "0010000000";
    CONSTANT notBH   : std_logic_vector (9 DOWNTO 0)
                    := "0001000000";
    CONSTANT shlBH   : std_logic_vector (9 DOWNTO 0)
                    := "0000100000";
    CONSTANT shrBH   : std_logic_vector (9 DOWNTO 0)
                    := "0000010000";
    CONSTANT AaddBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000001000";
    CONSTANT AsubBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000100";
    CONSTANT AmulBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000010";
    CONSTANT AcmpBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000001";

    SIGNAL product : std_logic_vector (15 DOWNTO 0);
    SIGNAL aluoutSignal : std_logic_vector (15 DOWNTO 0);

BEGIN
    PROCESS (A, B, B15to0, AandB, AorB, notB, shlB, shrB,
            AaddB, AsubB, AmulB, cin, aluoutSignal,
            product, AcmpB)
      VARIABLE temp :  std_logic_vector (9 DOWNTO 0);
      VARIABLE sum :  std_logic_vector (16 DOWNTO 0);
      VARIABLE sub :  std_logic_vector (16 DOWNTO 0);
    BEGIN --                                    Continued
```

```
      zout <= '0'; cout <= '0';
      aluoutSignal <= (OTHERS=>'0');
      temp := (B15to0, AandB, AorB, notB,
              shlB, shrB, AaddB, AsubB, AmulB, AcmpB);
      sum := A + B + (16 DOWNTO 1=> '0', 0=> cin) ;
      sub := A - B - (16 DOWNTO 1=> '0', 0=> cin);

      CASE temp IS
         WHEN B15to0H =>
            aluoutSignal <= B;
         WHEN AandBH =>
            aluoutSignal <= A and B;
         WHEN AorBH =>
            aluoutSignal <= A or B;
         WHEN notBH =>
            aluoutSignal <= not (B);
         WHEN shlBH =>
            aluoutSignal <= B (14 DOWNTO 0) & B (0);
         WHEN shrBH =>
            aluoutSignal <= B (15) & B (15 DOWNTO 1);
         WHEN AaddBH =>
            aluoutSignal <= sum (15 DOWNTO 0);
            cout <= sum (16);
         WHEN AsubBH  =>
            aluoutSignal <= sub (15 DOWNTO 0);
            cout <= sub (16);
         WHEN AmulBH  =>
            aluoutSignal <= product;
         WHEN AcmpBH  =>
            aluoutSignal <= (OTHERS=>'1');
            IF (A>B) THEN
               cout <= '1';
            ELSE
               cout <= '0'; END IF;

            IF (A=B) THEN
               zout <= '1';
            ELSE
               zout <= '0'; END IF;
         WHEN OTHERS => aluoutSignal <= (OTHERS=>'0');
      END CASE;
      IF (aluoutSignal = "0000000000000000") THEN
         zout <= '1';
      END IF;
   END PROCESS;

   MULT8x8: mult PORT MAP (A (7 DOWNTO 0),
                          B (7 DOWNTO 0), product);
   aluout <= aluoutSignal;

END dataflow;
```

Figure 10.8 *ArithmeticUnit* **VHDL Code**

In order to insure that no unwanted latches are implied, all ALU outputs are set to their inactive values at the beginning of the process statement of its VHDL code. In a case statement in this code, *aluout* and its flags outputs are set according to the selected control input of the ALU.

The multiplication function (*AmulBH*) of the ALU is handled by instantiating *bitwise* architecture of *mult_8by8*. This is the array multiplier that we presented in Chapter 8. Instantiation of the multiplier appears near the end of the code in Figure 10.8. The output of the multiplier is put on the *product* local signal and is assigned to *aluoutSignal* in the body of the process statement of the ALU.

Instead of instantiating this predefined multiplier, we could use the multiplication operation to leave the implementation of multiplication up to the synthesis tool. Another alternative would be to use a sequential multiplier such as add-and-shift or the Booth multiplier of Chapter 8. In such cases, a separate clock and proper handshaking would be required.

Table 10.2 ALU Operations

Mnemonic	Description	Code
B15to0H	Place B on the output	1000000000
AandBH	Place A and B on the output	0100000000
AorBH	Place A or B on the output	0010000000
notBH	Place not B on the output	0001000000
shlBH	Shift B one bit to the left	0000100000
shrBH	Shift B one bit to the right	0000010000
AaddBH	Place A + B on the output	0000001000
AsubBH	Place A - B on the output	0000000100
AmulBH	Place A * B on the output	0000000010
AcmpBH	Z = 1 if A = B; C = 1 if A < B	0000000001

10.2.1.3 Instruction Register. SAYEH Instruction Register is shown in Figure 10.9. This unit is a 16-bit register with an active high load-enable input. As shown, the only control input of the *InstructionRegister* module is *IRload*.

10.2.1.4 Register File. Figure 10.10 shows the VHDL code of SAYEH Register File. This is a two-port memory with a moving window pointer. Reading the register file is unclocked, while writing is controlled by the rising edge of the clock. At all times, contents of two locations appear on the right and left output ports (*Rout*, *Lout*) of the register file.

```
ENTITY InstrunctionRegister IS
    PORT ( input : IN std_logic_vector (15 DOWNTO 0);
           IRload, clk : IN std_logic;
           output : OUT std_logic_vector (15 DOWNTO 0) );
END InstrunctionRegister;

ARCHITECTURE dataflow OF InstrunctionRegister IS BEGIN
    PROCESS (clk) BEGIN
        IF (clk = '1') THEN
            IF (IRload = '1') THEN
                output <= input;
            END IF;
        END IF;
    END PROCESS;
END dataflow;
```

Figure 10.9 *InstructionRegister* **VHDL Code**

```
ENTITY RegisterFile IS
    PORT (
        input : IN std_logic_vector (15 DOWNTO 0);
        clk : IN std_logic;
        base : IN std_logic_vector (5 DOWNTO 0);
        Laddr, Raddr : IN std_logic_vector (1 DOWNTO 0);
        RFLwrite, RFHwrite : IN std_logic;
        Lout, Rout : OUT std_logic_vector (15 DOWNTO 0) );
END RegisterFile;

ARCHITECTURE dataflow OF RegisterFile IS --MS: please check
    SIGNAL Raddress : std_logic_vector (5 DOWNTO 0);
    SIGNAL Laddress : std_logic_vector (5 DOWNTO 0);
BEGIN
    Laddress <= Base + Laddr;
    Raddress <= Base + Raddr;
    Lout <= MemoryFile (conv_integer(Laddress));
    Rout <= MemoryFile (conv_integer(Raddress));

    PROCESS (clk) BEGIN
        IF (clk = '1') THEN
            IF (RFLwrite = '1') THEN
                MemoryFile (conv_integer(Laddress))(7 DOWNTO 0)
                                         <= input(7 DOWNTO 0);
            END IF;
            IF (RFHwrite = '1') THEN
                MemoryFile(conv_integer(Laddress))(15 DOWNTO 8)
                                         <= input(15 DOWNTO 8);
            END IF;
        END IF;
    END PROCESS;
END dataflow;
```

Figure 10.10 *RegisterFile* **VHDL Code**

For calculation of memory addresses, the base of the window pointer (*Base*) is added to the left and right addresses (*Laddr* and *Raddr*) and absolute addresses are calculated (*Laddress* and *Raddress*). Memory words are read on appropriate left and right outputs (*Lout* and *Rout*) from *Laddress* and *Raddress* absolute locations.

Writing into the memory is done in the location pointed by the left absolute address, *Laddress*. The *RFLwrite* and *RFHwrite* control signals decide whether a write is done to the low order or the high order bits of the Register File. If both these signals are active, writing is done in a 16-bit word addressed by *Laddress*.

10.2.1.5 Window Pointer. The VHDL code of the Window Pointer is shown in Figure 10.11. This unit has two control lines; one is for resetting it and the other is for adding its 6-bit input to its register contents. As with other register structures of SAYEH, this register is positive edge triggered.

```
ENTITY WindowPointer IS
   PORT (
      input : IN std_logic_vector (5 DOWNTO 0);
      clk : IN std_logic;
      WPreset, WPadd : IN std_logic;
      output : OUT std_logic_vector (5 DOWNTO 0)
   );
END WindowPointer;

ARCHITECTURE dataflow OF WindowPointer IS
   SIGNAL outputSignal : std_logic_vector (5 DOWNTO 0);
BEGIN

   PROCESS (clk)
   BEGIN
      IF (clk = '1') THEN
         IF (WPreset = '1') THEN
            outputSignal <= "000000";
         ELSIF (WPadd = '1') THEN
            outputSignal <= outputSignal + input;
         END IF;
      END IF;
   END PROCESS;

   output <= outputSignal;

END dataflow;
```

Figure 10.11 *WindowPointer* VHDL Code

10.2.1.6 Status Register. SAYEH Status Register is a collection of two flags that are set or reset according to their control signals. The VHDL code of this register is shown in Figure 10.12. This unit has five control signals for setting and resetting the two flags and for its synchronous load control. The latter (*SRload*) is used when an ALU operation is to affect the status flags. As with other register structures of SAYEH, this register is positive edge triggered.

```
ENTITY StatusRegister IS
   PORT (
      Cin, Zin, SRload, clk : IN std_logic;
      Cset, Creset, Zset, Zreset : IN std_logic;
      Cout, Zout : OUT std_logic
      );
END StatusRegister;

ARCHITECTURE dataflow OF StatusRegister IS BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF (SRload = '1') THEN
            Cout <= Cin;
            Zout <= Zin;
         ELSIF (Cset='1') THEN
            Cout <= '1';
         ELSIF (Creset='1') THEN
            Cout <= '0';
         ELSIF (Zset='1') THEN
            Zout <= '1';
         ELSIF (Zreset='1') THEN
            Zout <= '0';
         END IF;
      END IF;
   END PROCESS;
END dataflow;
```

Figure 10.12 *StatusRegister* **VHDL Code**

10.2.2 SAYEH Datapath

Figure 10.13 shows the datapath entity and architecture of SAYEH. This unit specifies component instantiations and bussing structure of the CPU according to the diagram of Figure 10.3. Inputs of this module are the processor's data and address busses, as well as control signals that are provided by the controller of the CPU. Control signals shown in the Data Path are routed to the instantiated data components or to the internal buses that are specified in this module.

```
ENTITY DataPath IS
    PORT (
       clk : IN std_logic;
       Databus : inout std_logic_vector (15 DOWNTO 0);
       Addressbus : OUT std_logic_vector (15 DOWNTO 0);
       ResetPC, PCplusI, PCplus1, RplusI, Rplus0,
       Rs_on_AddressUnitRSide, Rd_on_AddressUnitRSide,
       EnablePC, B15to0, AandB, AorB, notB, shlB, shrB,
       AaddB, AsubB, AmulB, AcmpB, RFLwrite, RFHwrite,
       WPreset, WPadd, IRload, SRload,
       Address_on_Databus, ALU_on_Databus,
       IR_on_LOpndBus, IR_on_HOpndBus,
       RFright_on_OpndBus,
       Cset, Creset, Zset, Zreset, shadow : IN std_logic;
       Instruction : OUT std_logic_vector (7 DOWNTO 0);
       Cout, Zout, ShadowEn : OUT std_logic
    );
END DataPath;

ARCHITECTURE dataflow OF DataPath IS
    COMPONENT AU   . . .   END COMPONENT;
    COMPONENT ALU  . . .   END COMPONENT;
    COMPONENT RF   . . .   END COMPONENT;
    COMPONENT IR   . . .   END COMPONENT;
    COMPONENT SR   . . .   END COMPONENT;
    COMPONENT WP   . . .   END COMPONENT;

    SIGNAL Right, Left, OpndBus, ALUout, IRout, Address,
       AddressUnitRSideBus : std_logic_vector (15 DOWNTO 0);
    SIGNAL SRCin, SRZin, SRZout, SRCout : std_logic;
    SIGNAL WPout : std_logic_vector (5 DOWNTO 0);
    SIGNAL Laddr, Raddr : std_logic_vector (1 DOWNTO 0);

BEGIN
    AddressingUnit : AU PORT MAP (
       AddressUnitRSideBus, IRout (7 DOWNTO 0), Address,
       clk, ResetPC, PCplusI, PCplus1, RplusI, Rplus0,
       EnablePC );
    ArithmeticUnit : ALU PORT MAP (
       Left, OpndBus, B15to0, AandB, AorB, notB, shlB, shrB,
       AaddB, AsubB, AmulB, AcmpB, ALUout, SRCout,
       SRZin, SRCin );
    RegisterFile : RF PORT MAP (
       Databus, clk, WPout, Laddr, Raddr, RFLwrite,
       RFHwrite, Left, Right );
    InstrunctionRegister : IR PORT MAP (Databus, IRload,
       clk, IRout);
    StatusRegister : SR PORT MAP (SRCin, SRZin, SRload, clk,
       Cset, Creset, Zset, Zreset, SRCout, SRZout);
    WindowPointer : WP PORT MAP (IRout (5 DOWNTO 0), clk,
       WPreset, WPadd, WPout);

    --                                            Continued
```

```
   AddressUnitRSideBus <=
      Right WHEN Rs_on_AddressUnitRSide='1' ELSE
      Left WHEN Rd_on_AddressUnitRSide='1' ELSE
      (OTHERS=>'Z');

   Addressbus <= Address;

   Databus <=
      Address WHEN Address_on_Databus = '1' ELSE
      ALUout WHEN ALU_on_Databus = '1' ELSE
      (OTHERS=>'Z');

   OpndBus (7 DOWNTO 0) <=
      IRout (7 DOWNTO 0) WHEN IR_on_LOpndBus = '1' ELSE
      (OTHERS=>'Z');

   OpndBus (15 DOWNTO 8) <=
      IRout (7 DOWNTO 0) WHEN IR_on_HOpndBus = '1' ELSE
      (OTHERS=>'Z');

   OpndBus <=
      Right WHEN RFright_on_OpndBus = '1' ELSE
      (OTHERS=>'Z');

   Zout <= SRZout;

   Cout <= SRCout;

   Instruction <=
      IRout(15 DOWNTO 8) WHEN Shadow = '0' ELSE
      IRout(7 DOWNTO 0);

   ShadowEn <=
      '0' WHEN IRout (7 DOWNTO 0) = "00001111" ELSE
      '1';

   Laddr <=
      IRout (11 DOWNTO 10) WHEN Shadow = '0' ELSE
      IRout (3 DOWNTO 2);

   Raddr <=
      IRout (9 DOWNTO 8) WHEN Shadow = '0' ELSE
      IRout (1 DOWNTO 0);

END dataflow;endmodule
```

Figure 10.13 SAYEH *DataPath* Module

Following the declarations, the *dataflow* architecture of the Data Path instantiates Addressing Unit, Arithmetic Unit, Register File, Instruction Register, Status Register, and the Window Pointer. Control signals that are inputs of *DataPath* are passed from this module

to the data components via their ports. For example, *ResetPC* that is an input of *DataPath* and a control signal of *AddressingUnit* appears on the port list of *AddressingUnit* in its instantiation statement.

The part that follows module instantiations makes bus assignments to the internal buses of this module. For example, assignment of the output of *ArithmeticUnit* to *Databus* that is controlled by *ALU_on_Databus* is done by a conditional signal assignment. Note the assignment of (OTHERS=>'Z') to *Databus* when none of the control signals of this bus are active.

In the last part of the *DataPath* module, bits of *IRout* (*IRout* is connected to the output of *IR*) that indicate source and destination registers to the Register File are placed on *Laddr* and *Raddr* inputs of this register. The *Shadow* signal that becomes '1' if a shadow instruction is being executed is used to select appropriate bits of the *IR* as source and destination addresses.

10.2.3 SAYEH Controller

The controller of SAYEH is a state machine with eleven states that issues appropriate control signals to the Data Path. The controller uses the Huffman style of coding, in which the state machine has a large combinational part that is responsible for state transitions and issuing controller outputs. State transitions are done by setting next state values to the *Nstate*. Figure 10.14 shows a general outline of this controller. Various sections of this outline are discussed below.

10.2.3.1 Controller Ports. The instruction register output, ALU flags, and external control signals constitute the inputs of the controller. The outputs of the controller are 38 control signals going to the Data Path and a *Shadow* output that indicates that the controller is handling a shadow instruction. Memory ready and a shadow enable signal are among other inputs of the controller. The *ShadowEN* input comes from a logic circuit in the datapath telling if the shadow space of the present instruction (bits 7 down to 0 of an 8-bit instruction) has a valid instruction.

All the controller outputs are assigned inactive '0' values at the beginning of the *StateSeq* process. These outputs are issued by this process depending on the instruction and the state of the machine.

10.2.3.2 Control States. An enumeration type declaration in the controller description declares the states of the processor. The type used is *state* and is used for declaring *Pstate* and *Nstate* signals for the present and next states of the controller. As mentioned, the coding style of the controller is according to the Huffman style of coding discussed in Chapter 8.

States *reset* and *halt* are for the initial state of the machine and its halt state. In state *fetch* the machine begins fetching a 16-bit instruction that can include an 8-bit instruction and a shadow. State *memread* is entered while our controller is waiting for *memDataReady* signal from the memory indicating that its data is ready. Execution of instructions is performed in the *exec1* state. This state is entered from the *memread* state. The *lda* instruction that is not completed by the *exec1* state requires the additional state of *exec1lda* to complete its memory read. States *exec2* and *exec2lda* are like *exec1* and *exec1lda* except that they handle the shadow part of an instruction. The execute state of most instructions (*exec1* or *exec2*) increment the program counter while the instruction is being executed. However, certain instructions that use the address bus for their execution cannot increment *PC* while they are being executed. For these instructions, the *incpc* state increments the program counter.

10.2.3.3 Opcodes. Referring to Figure 10.14, instruction opcodes are declared as 4-bit parameters in the controller of SAYEH. These parameters are according to the processor's instruction set of Table 10.1.

10.2.3.4 State Machine Flow. The *StateSeq* process shown in Figure 10.14 is entered every time the processor clock clocks a new state in *Pstate*. This process has a case statement with case alternatives for every state of the controller.

Most of the case alternatives are responsible for issuing just a few control signals. For example, the *memread* state issues the memory read signal and conditionally issues *IRload* when *memDataReady* informs the processor that data from the memory is ready.

On the other hand, controller states *exec1* and *exec2* handle most of the tasks for execution of the processor instructions. In the blocks of code corresponding to these case alternatives, there are several case statements that use the instruction opcode for their case expressions. The case alternatives corresponding to these inner case expressions are for the execution of the individual instructions of the processor. Figure 10.14 shows the general outline of these nested case statements.

10.2.3.5 Shadow Instructions. The *ShadowEn* signal that is an input to the controller is set when the hex code 0F (this code indicates that the right-most bits are not used) is not found in the right-most eight bits of a 16-bit instruction. If this input is '1' and execution of an 8-bit instruction is complete, the controller branches to *exec2* to execute the second half of the instruction before the next fetching begins.

```
ENTITY controller IS
   PORT (ExternalReset, clk : IN std_logic;
         ResetPC, PCplusI, PCplus1, RplusI, . . .
         . . . ,       Zset, Zreset, Shadow : OUT std_logic;
         Instruction : IN std_logic_vector (7 DOWNTO 0);
         . . . ,       memDataReady, ShadowEn : IN std_logic)
END controller;
--
ARCHITECTURE dataflow OF controller IS
   TYPE state IS
      (reset, halt, fetch, memread, exec1, exec2,
       exec1lda, exec1sta, exec2lda, exec2sta, incpc);

   CONSTANT b0000: std_logic_vector (3 DOWNTO 0) := "0000";
   CONSTANT b1111: std_logic_vector (3 DOWNTO 0) := "1111";

   CONSTANT nop: std_logic_vector (3 DOWNTO 0) := "0000";
   . . .
   CONSTANT mvr: std_logic_vector (3 DOWNTO 0) := "0001";

   SIGNAL Pstate, Nstate : state;

BEGIN
   StateSeq: PROCESS (Instruction, Pstate, ExternalReset,
                      Cflag, Zflag, memDataReady, ShadowEn)
   BEGIN
      ResetPC <= '0';   PCplusI <= '0';     PCplus1 <= '0';
      RplusI  <= '0';   Rplus0 <= '0';      EnablePC<= '0';
      B15to0 <= '0';    AandB <= '0';       AorB  <= '0';
      . . .
      Rs_on_AddressUnitRSide <= '0';
      Rd_on_AddressUnitRSide <= '0';

      CASE Pstate IS
         WHEN reset =>
         . . .
         WHEN halt =>

         WHEN fetch =>

         WHEN memread =>

         WHEN exec1 =>
            CASE Instruction (7 DOWNTO 4) IS
               WHEN b0000 =>
                  CASE Instruction (3 DOWNTO 0) IS
                     WHEN nop =>
                     . . .
                  END CASE;
               . . .
               WHEN mvr =>
                  -- Reg Operations (Src and Dest)
--                                          Continued
```

```
            WHEN b1111 =>
               CASE Instruction (1 DOWNTO 0) IS
                  WHEN mil =>
                     . . .
               END CASE;
         END CASE;
      WHEN exec1lda =>
      . . .
      WHEN exec1sta =>
      . . .
      WHEN exec2 =>
      . . .
      WHEN exec2lda =>
      . . .
      WHEN exec2sta =>
      . . .
      WHEN incpc =>
      . . .
   END CASE;
END PROCESS;

Clocking: PROCESS (clk) BEGIN
IF (clk = '1') THEN
    Pstate <= Nstate;
END IF;
END PROCESS;
END dataflow;
```

Figure 10.14 SAYEH Controller General Outline

10.2.3.6 Combinational Block. The combinational block of SAYEH controller is the process statement that is labeled as *StateSeq* in the code of Figure 10.14. Transitions from one state to another (i.e., assignments to *Nstate*) and issuing control signals are performed in the case statement. As discussed above, to avoid latches, at the beginning of this process statement all control signals are set to their inactive values.

10.2.3.7 Sequential Block. The last part of the code outline of Figure 10.14 is the process statement that is responsible for clocking the state registers. This process statement is labeled *Clocking* and is sensitive to the positive edge of the clock. With each clock this process clocke *Pstate* into *Nstate*. Control signals issued by the controller remain active through the next rising edge of the system clock.

10.2.3.8 Instruction Execution. Figure 10.15 zooms on the combinational process of the *dataflow* architecture of *controller* and shows the details of execution of *mvr* in the *exec1* state of the controller. Signals issued for the execution of this instruction are shown in this figure.

This instruction reads a word from the right address of the Register File and writes it into its left address. The right and left (source and destination) addresses are provided in the Data Path by connections made from *IR* to the Register File.

The *RFright_on_OpndBus* control signal is issued to read the source register from *RegisterFile* onto *OpndBus*. Since this bus is the input of the ALU, the data on the ALU's right input (*B*) must pass through it to reach its output. For this purpose, the *B15to0* control input of ALU is issued. Once the data reaches the ALU output, it becomes available at the input of the Register File. Issuing *RFLwrite* and *RFHwrite* cause data to be written into the destination into *RegisterFile*.

```
StateSeq: PROCESS (Instruction, Pstate, ExternalReset,
                   Cflag, Zflag, memDataReady, ShadowEn)
   . . .
BEGIN
   . . .
   CASE Pstate IS
      . . .
      WHEN exec1 =>
         CASE Instruction (7 DOWNTO 4) IS
            WHEN b0000 =>
            . . .
            WHEN mvr =>
               RFright_on_OpndBus <= '1';
               B15to0 <= '1';
               ALU_on_Databus <= '1';
               RFLwrite <= '1';
               RFHwrite <= '1';
               SRload <= '1';
               IF (ShadowEn='1') THEN
                  Nstate <= exec2;
               ELSE
                  PCplus1 <= '1';
                  EnablePC <= '1';
                  Nstate <= fetch;
               END IF;
            WHEN lda =>
               Rplus0 <= '1';
               Rs_on_AddressUnitRSide <= '1';
               ReadMem <= '1';
               Nstate <= exec1lda;
               . . .
            WHEN b1111 =>
            . . .
         END CASE;
   END CASE;
END PROCESS;
```

Figure 10.15 Instruction Execution

The partial code of Figure 10.15 also shows assignment of *exec2* to *Nstate* if the instruction we are executing has a shadow. Otherwise, signals for incrementing the Program Counter are issued and the next state is set to *fetch*.

10.2.3.9 Memory Handshaking. The execution discussed above applies to most SAYEH instructions. However, instructions that require memory access, e.g., *lda*, require extra clocks for reading the memory. The first part of the execution of *lda* is shown in Figure 10.15. As shown, for the execution of this instruction, the address is read from Register File and put on the address bus. At the same time, *Read-Mem* is issued to initiate the memory read process.

```
StateSeq: PROCESS (Instruction, Pstate, ExternalReset,
                   Cflag, Zflag, memDataReady, ShadowEn)
   . . .
BEGIN
   CASE Pstate IS
      . . .
      WHEN exec1lda =>
         IF (ExternalReset = '1') THEN
            Nstate <= reset;
         ELSE
            IF (memDataReady = '0') THEN
               Rplus0 <= '1';
               Rs_on_AddressUnitRSide <= '1';
               ReadMem <= '1';
               Nstate <= exec1lda;
            ELSE
               RFLwrite <= '1';
               RFHwrite <= '1';
               IF (ShadowEn='1') THEN
                  Nstate <= exec2;
               ELSE
                  PCplus1 <= '1';
                  EnablePC <= '1';
                  Nstate <= fetch;
               END IF;
            END IF;
         END IF;
      . . .
   END CASE;
END PROCESS;
```

Figure 10.16 Memory Handshaking for *exec1lda*

Also shown in Figure 10.15, the next state for execution of *lda* after *exec1* is *exec1lda*. Details of this state are shown in the partial code of Figure 10.16. In this state, *ReadMem* continues to be issued

and state remains in *exec1lda* until *memDataReady* becomes '1'. In this case, memory data that is available on *Databus* will be clocked into *RegisterFile* by issuing *RFLwrite* and *RFHwrite*.

Executions of other SAYEH instructions are similar to the examples we discussed. The complete VHDL code of SAYEH controller is over 800 lines and is included on the CD that accompanies this book.

10.2.4 Complete SAYEH Processor

Partial code corresponding to the top-level VHDL code of SAYEH is shown in Figure 10.17. This code consists of instantiation of *DataPath* and *controller* architectures. In the *dataflow* architecture of *Sayeh*, control signal outputs of *controller* are wired to the similarly named signals of *DataPath*. The ports of the processor are according to the block diagram of Figure 10.1.

```
ENTITY Sayeh IS
   PORT (
      clk : IN std_logic;
      ReadMem, WriteMem, ReadIO, WriteIO : OUT std_logic;
      . . .
   );
END Sayeh;

ARCHITECTURE dataflow OF Sayeh IS
   COMPONENT dp  . . .  END COMPONENT;
   COMPONENT ctrl  . . .  END COMPONENT;

   SIGNAL Instruction : std_logic_vector (7 DOWNTO 0);
   SIGNAL ResetPC, PCplusI, PCplus1, . . .
BEGIN

   datapath : dp PORT MAP (
      clk, Databus, Addressbus, ResetPC, PCplusI, . . .
      Zreset, shadow, Instruction, Cflag, Zflag, ShadowEn
      );

   controller : ctrl PORT MAP (
      ExternalReset,
      clk, ResetPC, PCplusI, . . .
      );

END dataflow;
```

Figure 10.17 SAYEH Top Level Description

10.3 SAYEH Testbench / Assembler / Memory Model

The complete VHDL description of SAYEH consists of component descriptions like registers, counters, logic units, and a state machine for its controller. Chapter 9 has shown how such components can be tested with testbenches for data application and response monitoring. Obviously testing SAYEH begins with testing its components using such techniques. On the other hand, a complete test of the processor when all its tested components are put together is still necessary. This section discusses top-level testing of SAYEH.

In a testbench, we instantiate SAYEH, and through a memory model, we apply instructions to the CPU and watch its response to these test instructions. For developing such a top-level testbench that is easy for the design engineer to work with, two issues must be considered: Test data format and memory size handling.

Test data format must be at a high level so that the designer testing the CPU can apply large volumes of instructions and data to the CPU. For this purpose, our testbench takes test data in the form of SAYEH instructions and translates them to binary data for the processor to be tested. This scheme was used for our simple *Adding-CPU* and was discussed in the previous section. SAYEH testbench has such a translation program that is, of course, much larger than that of *AddingCPU*.

The other issue that must be considered for a testbench for this design is memory size handling. Recall that our *AddingCPU* example did all its reading and writings directly into an external file representing its complete memory. Having the complete memory image in one file is not practical for the relatively large size of SAYEH memory. Furthermore, moving the complete memory image of the processor being tested into its testbench and declaring it as a two dimensional variable requires too much memory of the computer performing the simulation. In a large design, the actual memory of a design being tested may be larger than the computer it is being tested on.

In developing a testbench for SAYEH, we focus on the issue of memory size handling discussed above. Instead of having all memory image in one file, or all of it declared as a variable, we take an in-between approach. We partition our memory into several pages, and use one file for each page. The file corresponding to a page is named according to the page number it represents. Then, the actual testbench declares a variable of the size of only one such page. This variable is regarded as a buffer. When the CPU model addresses a memory location, the testbench checks to see if that is available in the buffer. If so, data from the buffer will be read or written into according to the CPU request. On the other hand, if a memory location is addressed that is not in the buffer, the testbench writes the contents

of the buffer into its corresponding memory file, and loads the page that has the addressed location into the buffer.

Figure 10.18 shows the overall structure of our testbench. The sections that follow discuss the details of the VHDL code of this testbench. The complete code of this testbench is included in the CD that accompanies this book. Parts of this testbench that have to do with memory swapping can be used as a file-based memory model.
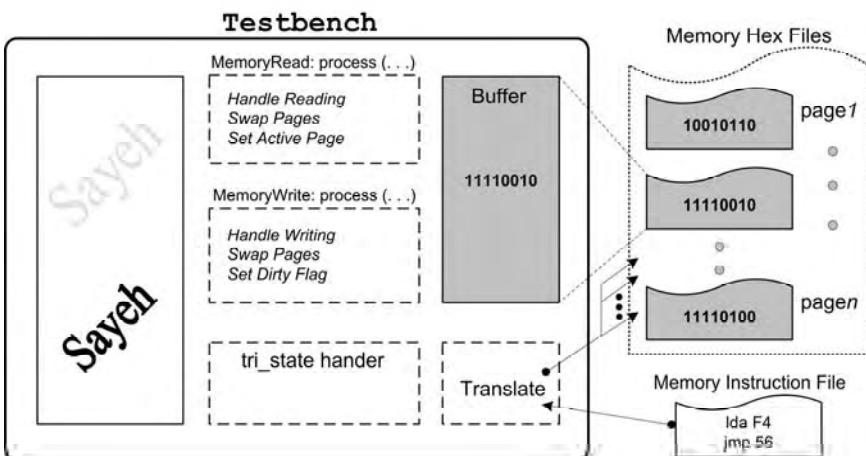

Figure 10.18 Graphical Representation of SAYEH Testbench

10.3.1 Top Level VHDL Testbench

The outline of the VHDL code of SAYEH testbench is shown in Figure 10.19. In this code, the processor and its memory are instantiated and a clock signal is provided. The memory provides proper instructions for the processor to execute. The instructions are synchronized with the edge of the system clock.

Figure 10.18 is the graphical representation of SAYEH testbench. In this figure the box marked "Sayeh" is the processor and everything to its right is included in the functionality of the *Sayeh-Memory(behavioral)* that is instantiated in the testbench.

```
ENTITY TestSayeh IS
END ENTITY TestSayeh;

ARCHITECTURE dataflow OF TestSayeh IS
    SIGNAL clk : std_logic := '0';
    SIGNAL ReadMem, WriteMem, ReadIO, WriteIO : std_logic;
    SIGNAL ExternalReset, memdataready : std_logic;
    SIGNAL Databus : std_logic_vector(15 DOWNTO 0);
    SIGNAL Addressbus : std_logic_vector(15 DOWNTO 0);
BEGIN

    clk <= NOT (clk) AFTER 5 NS WHEN now < 1000000 NS ELSE
           clk;
    ExternalReset <= '1', '0' after 27 ns;

    processor : ENTITY WORK.Sayeh(dataflow) PORT MAP
        (clk, ReadMem, WriteMem, ReadIO, WriteIO,
         Databus, Addressbus, ExternalReset, memdataready
        );

    memory : ENTITY WORK.SayehMemory(behavioral) PORT MAP
        (clk, ReadMem, WriteMem,
         Addressbus, DataBus, memdataready
        );

END dataflow;
```

Figure 10.19 Top-Level SAYEH Testbench

10.3.2 Memory Model

The testbench of Figure 10.20 instantiates *SayehMemory* and wires it to the processor. This memory is responsible for reading a page from the memory external files, translating it to hex, loading a buffer page of internal memory, reading and writing from it, and finally writing memory contents back to the external files. The sections that follow discuss each of these tasks.

At initialization, before the simulation begins, the *assembler* procedure reads a page of instruction from an instruction file and puts its hex equivalent into *buffermem*. The *UpdateFile* procedure, with argument 1, puts this hex image in page 1 of the memory. When the simulation begins, the translated program is available in the memory buffer starting from location 0. Consequent readings of instructions will be done from this memory.

The page of the memory that resides in the memory model of Figure 10.20 is *buffermem* that is of type *mem_type*. This type is a 1K array of 16-bit words.

```
ENTITY Sayehmemory IS
   GENERIC (blocksize : integer := 1024;
            segmentsno : integer := 64);
   PORT (
      clk, readmem, writemem : IN  STD_LOGIC;
      addressbus : IN  STD_LOGIC_VECTOR (15 DOWNTO 0);
      databus : INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
      memdataready : OUT std_logic
     );
END Sayehmemory;

ARCHITECTURE behavioral OF Sayehmemory IS
   TYPE mem_type IS ARRAY (0 TO blocksize-1) OF
        STD_LOGIC_VECTOR (15 DOWNTO 0);
        --Data TYPE for a seqment OF memory
BEGIN
   RW: PROCESS (clk)
      VARIABLE buffermem : mem_type
                         := (OTHERS=> (OTHERS=>'0'));
      VARIABLE ad : INTEGER;
      VARIABLE memloadedno : integer := segmentsno+1;
      VARIABLE changemem : BOOLEAN := false;
      VARIABLE init : boolean := true;
   BEGIN
      IF (init = true) THEN
         assembler (buffermem);
         UpdateFILE (buffermem, 1);
         memloadedno := 1;
         init := false;
      END IF;

      IF (clk = '0') THEN
         IF (readmem = '1') THEN
            . . .                      -- Figure 10.22
         ELSIF (writemem = '1') THEN
            . . .                      -- Figure 10.23
         END IF;
      END IF;
   END PROCESS;
END behavioral;
```

Figure 10.20 SAYEH Memory Model

10.3.3 Assembler

A procedure that is called *assembler* handles translation of instructions to hex. The instruction input file is *prog.txt*. This file is read, translated to hex, and is returned via the argument of *assembler*. When translated to hex, instructions and data in *prog.txt* are placed in appropriate locations in the memory buffer.

```
PROCEDURE assembler (VARIABLE mem : OUT mem_type) IS
   FILE code : TEXT OPEN READ_MODE IS "prog.txt";
   VARIABLE instr : LINE;

   VARIABLE addr_std_v : std_logic_vector(15 DOWNTO 0);
   VARIABLE memonic : string (4 DOWNTO 1);
   VARIABLE im_ch : character;
   VARIABLE immediate  : std_logic_vector (7 DOWNTO 0);
   VARIABLE window_ptr : std_logic_vector (5 DOWNTO 0);
   VARIABLE dest_reg   : std_logic_vector (3 DOWNTO 0);
   VARIABLE src_reg    : std_logic_vector (3 DOWNTO 0);

   VARIABLE immi_str2   : string (2 DOWNTO 1);
   VARIABLE adr         : integer := -1;
   VARIABLE shadowEn    : boolean := false;
BEGIN
   WHILE NOT ENDFILE (code) LOOP
      READLINE (code, instr);
      HREAD (instr, addr_std_v);
      READ (instr, memonic);
      CASE memonic IS
         . . .
         WHEN " mvr" =>
            IF (shadowEn=true) THEN
              mem (adr) (7 DOWNTO 4) := "0001";
              READ (instr, immi_str2);
              HREAD (instr, dest_reg);
              mem (adr)(3 DOWNTO 2):=dest_reg (1 DOWNTO 0);
              READ (instr, immi_str2);
              HREAD (instr, src_reg);
              mem (adr) (1 DOWNTO 0):=src_reg (1 DOWNTO 0);
            ELSE
              adr := adr+1;
              mem (adr) (15 DOWNTO 12) := "0001";
              READ (instr, immi_str2);
              HREAD (instr, dest_reg);
              mem(adr)(11 DOWNTO 10):=dest_reg(1 DOWNTO 0);
              READ (instr, immi_str2);
              HREAD (instr, src_reg);
              mem (adr) (9 DOWNTO 8):=src_reg (1 DOWNTO 0);
              mem (adr) (7 DOWNTO 0) := shadowins;
            END IF;
            shadowEn := NOT shadowEn;
         . . .
         WHEN OTHERS =>
            mem (adr) := (OTHERS=>'0');
      END CASE;
   END LOOP;
   FILE_CLOSE (code);
END PROCEDURE assembler;
```

Figure 10.21 SAYEH Assembler in VHDL

The *assembler* procedure of SAYEH testbench is similar to *Convert* of *AddingCPU* testbench, except that *assembler* handles more instructions (all of SAYEH instructions) and writes 16-bit data into its buffer argument. As in *Convert*, if instead of a mnemonic, "`:::`" appears in a line of *prog.txt*, *assembler* treats it as a directive for writing data directly into the specified memory location.

The declaration part of the *assembler* procedure and the part of its code that generates the hex code for the *mvr* instruction are shown in Figure 10.21. Instructions are read and placed in the *mem* output of *assembler* starting from location 0. The address handling *mem* locations is *adr*. This procedure takes advantage of the *std_logic* TEXTIO package for reading the instruction file.

10.3.4 Memory Read

The *RW* process of Figure 10.20 initiates the memory read procedure. The details of this procedure are shown in Figure 10.22. If the addressed memory is greater than the allowed memory size, (OTHERS => 'Z') will be assigned to *databus*.

If the memory address is valid, the memory page number is calculated and if it is different from the existing page, the existing page is written to its corresponding file using the *UpdateFile* procedure. Following this, the required page is read into *beffermem* using *MemLoad*, and the required address is looked up from *buffermem* location.

```
IF (clk = '0') THEN
    IF (readmem = '1') THEN
        memdataready <= '0';
        IF (ad >= (segmentsno*blocksize)) THEN
            databus <= (OTHERS => 'Z');
        ELSE
            IF (memloadedno /= ( (ad/blocksize)+1)) THEN
                IF memloadedno /= (segmentsno+1) THEN
                    IF changemem=true THEN
                        UpdateFILE (buffermem, memloadedno);
                    END IF;
                END IF;
                MemLoad (buffermem, ((ad/blocksize)+1));
                changemem := false;
                memloadedno := (ad/blocksize)+1;
                databus <= buffermem (ad mod blocksize);
            ELSE
            databus <= buffermem (ad mod blocksize);
        END IF;
    END IF;
memdataready <= '1';
```

Figure 10.22 Reading from the Memory

10.3.5 Memory Write

The *RW* process of Figure 10.20 initiates the memory write part of
this process. Figure 10.23 shows the details of memory writing. Tasks
performed here are very similar to those of the read part of the *RW*
process. At all times, *buffermem* holds the current active memory
buffer. When the buffer needs to be cleared for a new addressed page,
UpdateFile writes *buffermem* into the appropriate file. A new re-
quested page is loaded into *buffermem* using the *MemLoad* procedure.
Both these procedures use *memloadno* to build the memory file name
corresponding to the block being updated or loaded.

```
IF (clk = '0') THEN
   ELSIF (writemem = '1') THEN
      memdataready <= '0';
      IF (ad < (segmentsno*blocksize)) THEN
         IF (memloadedno = ((ad/blocksize)+1)) THEN
            IF buffermem (ad mod blocksize) /= databus THEN
               changemem := true;
            END IF;
            buffermem (ad mod blocksize) := databus;
         IF changemem=true THEN
            UpdateFILE (buffermem, memloadedno);
            changemem := false;
         END IF;
      ELSE
      IF memloadedno /= (segmentsno+1) THEN
         IF changemem=true THEN
            UpdateFILE (buffermem, memloadedno); END IF;
         END IF;
         memloadedno := (ad/blocksize)+1;
         MemLoad (buffermem, memloadedno);
         changemem := false;
         IF buffermem (ad mod blocksize)/= databus THEN
            changemem := true;
         END IF;
         buffermem (ad mod blocksize) := databus;
         IF changemem=true THEN
            UpdateFILE (buffermem, memloadedno);
            changemem := false;
         END IF;
      END IF;
   END IF;
   memdataready <= '1';
END IF;
```

Figure 10.23 Writing into the Memory

10.3.6 Memory File Handling

The *MemLoad* procedure, shown in Figure 10.24, reads a page of memory from file *f* and puts it in its *beffermem* output. The *Update-File* that is also shown here writes its *buffermem* argument into file *f*. Both procedures use the *fileno* integer for identifying the file to read or write. Using 'IMAGE attribute and concatenation operations, the *fileno* integer is used to generate a physical file name that corresponds to the page of memory for input or output. These procedures use *std_logic* TEXTIO read and write procedures.

```
--Load a segment from a file
PROCEDURE MemLoad (buffermem : OUT mem_type;
                   fileno : IN integer) IS
   VARIABLE hexcode : String (4 DOWNTO 1);
   VARIABLE memline : LINE;
   VARIABLE offset : integer := 0;
   VARIABLE err_check : file_open_status;
   VARIABLE hexcode_v : std_logic_vector (15 DOWNTO 0);
   FILE f : TEXT;
BEGIN
   buffermem := (OTHERS => (OTHERS =>'0'));
   FILE_OPEN ( err_check, f,
              ("mem" & integer'IMAGE (fileno) & ".hex"),
              READ_MODE);
   IF err_check = open_ok THEN
      WHILE not ENDFILE (f) LOOP
         readline (f, memline);
         HREAD (memline, hexcode_v);
         buffermem (offset) := hexcode_v;
         offset := offset+1;
      END LOOP;
      file_close (f);
   END IF;
END memload;

--Write memory data of a segment to its corresponding file
PROCEDURE updateFILE (buffermem : IN mem_type;
                      fileno : IN integer) IS
   VARIABLE memline : line;
   FILE f : TEXT OPEN WRITE_MODE IS
                ("mem" & integer'IMAGE (fileno) & ".hex");
 BEGIN
    FOR i IN 0 TO blocksize-1 LOOP
       HWRITE (memline, buffermem (i));
       WRITELINE (f, memline);
    END LOOP;
    FILE_CLOSE (f);
 END updatefile;
```

Figure 10.24 Memory File Handling

10.3.7 Sorting Test Program

As an example program for our processor core, Figure 10.25 shows a sorting program for SAYEH. This program reads data starting from the CPU memory and sorts them in descending order. The number of data item to sort is in location 768 and data begins in the next memory location. This sorting program uses two loops for the sorting to be done. When completed, the CPU is put into the halt state.

```
0000  mil  r0 00   :r0=768   starting address in memory
0001  mih  r0 03   :
0002  lda  r1 r0   :r1=      total number of elements
0003  awp  5       :
0004  mil  r0 01   :r5=1     for adding with index each time
0005  mih  r0 00   :
0006  cwp          :
0006  add  r1 r0   :r1=      limit for final r4
0007  mvr  r2 r1   :
0008  awp  2       :
0009  sub  r0 r3   :r2=      limit for index r3
0009  cwp          :
000A  mvr  r3 r0   :r3=      outer index
000A  nop          :
000B  cwp          :
000B  cmp  r3 r2   :  the outer index is equal to its limit
000C  brz  19      :         branch to 0025 if zero
000D  awp  3       :
000E  add  r0 r2   :r3=r3+1 increment outer index
000E  mvr  r1 r0   :r4=r3 set inner index to outer as init
000F  cwp          :
0010  awp  1       :
0011  cmp  r3 r0   : check if inner index reaches its limit
0012  brz  10      :         branch to 0022 if zero
0013  awp  2       :
0014  lda  r3 r0   :r6=(r3)
0015  awp  1       :
0016  add  r0 r1   :r4=r4+r5  increment inner index
0016  lda  r3 r0   :r7=(r4)
0017  cmp  r2 r3   :         check if r6 is greater than r7
0018  brc  07      :         branch to 001F if carry
0019  lda  r1 r0   :r5=(r4)  r5 as an temporary register
0019  sta  r0 r2   :(r4)=r6
001A  cwp          :
001B  awp  3       :
001C  sta  r0 r2   :(r3)=r5
001D  mil  r2 01   :
001E  mih  r2 00   :r5=1     for adding with index each time
001F  cwp          :
0020  awp  5       :
0021  jpa  r0 0E   :         jump to 000F
0022  cwp          :
0023  awp  5       :
0024  jpa  r0 0A   :         jump to 000B
0025  hlt          :
```

Figure 10.25 Sorting Program for SAYEH

The program shown in Figure 10.25 is translated into its hexadecimal equivalent and is put in *buffermem* memory variable. When completed, *memX.hex* files, where *X* is the memory page number, are created. These files represent memory pages affected by the program.
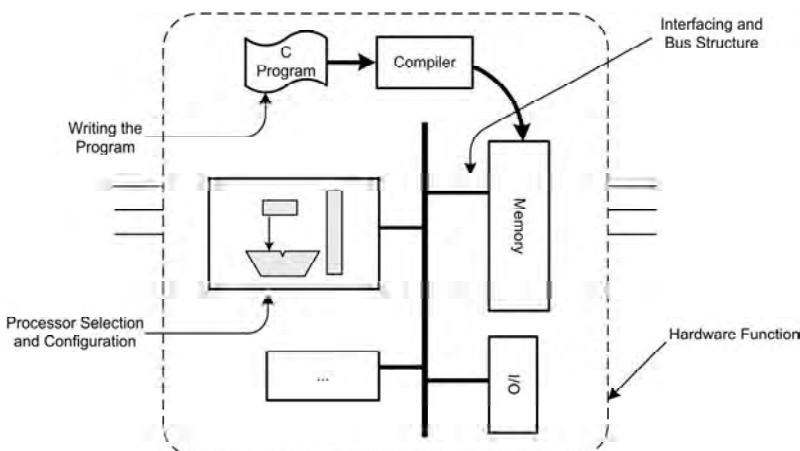
10.4 SAYEH as an Embedded Processor Core

This section shows an FIR filter design using our SAYEH processor as an embedded core. We will discuss the general strategy of a core-based design, filer requirements, our core based architecture, and our filter implementation.

10.4.1 Embedded Core Based Design

An embedded system based on a processor core has a hardware part and a program. The hardware is the architecture of the systems running the software program for the specific application the embedded design is targeted for. The hardware of an embedded system consists of a processor core, interfaces, IO devices, memory, and the necessary bussing. An embedded system development environment usually has a compiler for compiling an application program into the processor core's machine language.

An embedded system designer decides on the hardware architecture for the specific application, and develops the software program for it. The design is completed by putting the hardware and the software together. Figure 10.26 shows an outline of an embedded core based design. We will follow this outline for generation our example system.

Figure 10.26 Hardware Function Implemented by Embedded Processor

10.4.2 Filter Design

The example we will use for implementing with our SAYEH core is an FIR filter. An FIR filter has a set of coefficients that are taken from its impulse response of a filter as shown in Figure 10.27. These coefficients are the main factors in filter design.
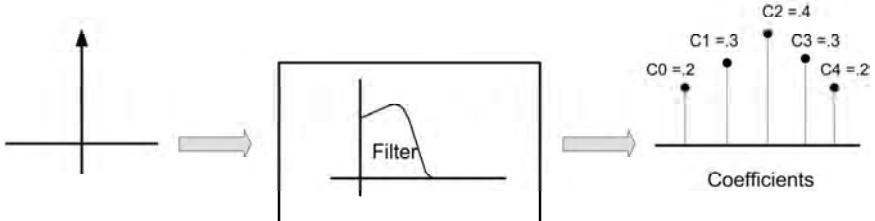
Figure 10.27 Filter Coefficient

Data into the filter are multiplied and accumulated as shown in the RT level implementation of the FIR filter in Figure 10.28. As shown in Figure 10.27, the filter we are designing is a 4th order FIR filter with five coefficients. The RTL implementation of the filter also shows the five coefficients. As shown, a series of registers provide delayed inputs that are multiplied by c_i coefficients and then added together to generate the output. The clock frequency for the registers must the same as the sampling frequency that is twice the largest frequency of the input signal.
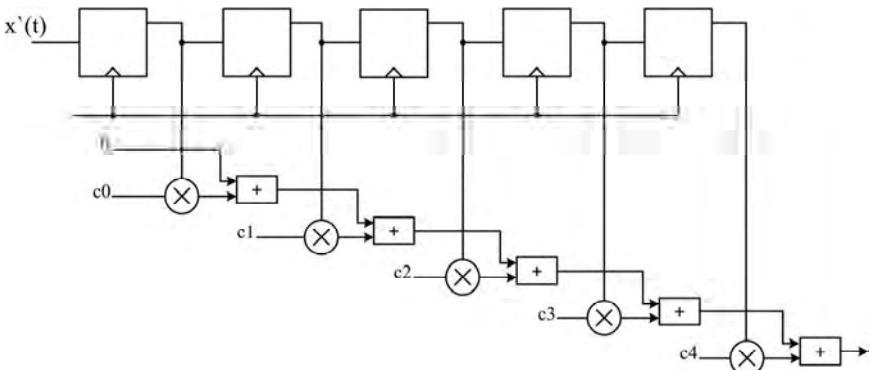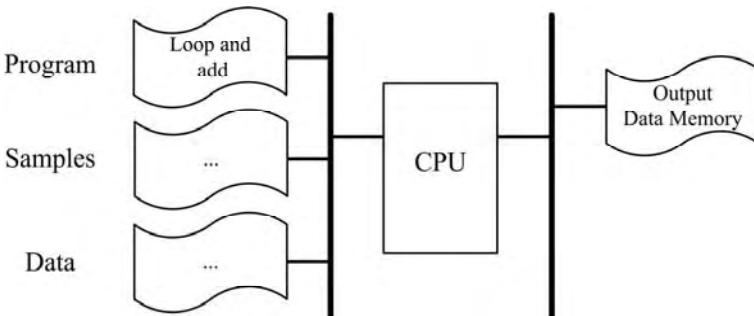
Figure 10.28 RTL FIR Filter

The hardware shown in Figure 10.28 is an iterative hardware and can be described using VHDL generate statements. This hardware can easily be implemented using registers, adders and multipliers. Of all these parts, the multipliers are the most complex and take more hardware than the other components.

10.4.3 Core Based Architecture

The algorithm presented in the RTL implementation of the filter can be implemented with a processor running the iterative add and multiply procedure. The architecture of this hardware is shown in Figure 10.29.

The SAYEH processor we are using for our embedded processor is available in behavioral pre-synthesis VHDL. We will use memory mapped I/O for its interfacing to keep its bussing structure simple. The filter program will be written in C, and we will show a hand translation of it into SAYEH Assembly Language.

Figure 10.29 Processor Core for FIR Implementation

The data memory of the implementation of Figure 10.29 has a section for storing coefficients and another section for storing sampled data. We assume that sampled data have been sampled and collected using a valid sampling rate for the input signal.

The program memory of the processor has a program that reads a new sample data performs shifting and five multiply and add operations, one for each coefficient of the filter. With each data read, the program outputs a new data for the circuit response.

10.4.4 FIR Program

Using the architecture discussed above, the implementation of the FIR filter will be completed by developing a program to load into the program memory of Figure 10.29. This program must be in SAYEH assembly language to use proper IO and memory addresses.

We will start with a C program to show the general operation of this filter, and then hand compile it into SAYEH machine language. This program shown in Figure 10.30 reads data, multiplies them by all coefficients and outputs the result.

```
01: #include <iostream.h>
02: #include <fstream.h>
03: #include <stdlib.h>
04: #include <stdio.h>
05: #include <string.h>
06:
07: int main ()
08: {
09: int history[16] = {0};
10: int i, j, n;
11: int temp;
12: int out[64] = {0};
13: int newInput, inputHigh, inputLow;
14: int newCoeff, coeffHigh, coeffLow;
15: int filterDegree, inputNo;
16: FILE * input;
17: FILE * coeff;
18: FILE * parameter;
19: FILE * output;
20: input = fopen("Input.bin","r");
21: parameter = fopen("Parameter.bin", "r");
22: output = fopen("Output.txt", "w");
23: filterDegree = fgetc(parameter);
24: inputNo = fgetc(parameter);
25: fclose(parameter);
26:     for (n = 0; n<inputNo; n++) {
27:         for (i = filterDegree-1; i>=0; --i) {
28:             history[i+1] = history[i];
29:         }
30:         inputHigh = fgetc(input);
31:         inputLow = fgetc(input);
32:         inputHigh = inputHigh << 8;
33:         newInput  = inputHigh | inputLow;
34:         history[0] = newInput;
35:         temp = 0;
36:         coeff = fopen("Coeff.bin","r");
37:         for (j = 0; j<filterDegree ; ++j) {
38:             coeffHigh = fgetc(coeff);
39:             coeffLow = fgetc(coeff);
40:             coeffHigh = coeffHigh << 8;
41:             newCoeff = coeffHigh | coeffLow;
42:             temp += history[j]*newCoeff;
43:         }
44:         fclose(coeff);
45:         out[n] = temp;
46:         fprintf(output,"%p\n", out[n]);
47:     }
48:     fclose(input);
49:     fclose(output);
50:     return 0;
51: }
```

Figure 10.30 FIR Filter C code

The program shown begins with header files and declarations. Following this part, it opens *input*, *parameter* and *output* files (lines 16 to 19). The *input* file is where sampled data are stored, and *output* is where result will be stored. The *parameter* file is where filter degree, and number of input samples are stored.

The loop that begins on line 26 and ends on line 47 (line numbers are shown in bold) reads data inputs calculates result and outputs data to the *output* file on line 46.

The loop that begins on line 37 and ends on line 43 performs multiplying data by filter coefficients and adding them as many times as there are coefficients. The result is collected in *temp*.

Note in this code that shifting of data that are multiplied (lines 32 and 40) results in using their eight most significant bits. This is done so that we will not require 16-bit multiplications in our implementation of this routine, which makes this C code conforms to the processor that we will be using for the implementation of this design. The processor we will use can only do 8-bit multiplications, which results in low accuracy of our filter design. This is a compromise we had to make to keep our design simple.

10.4.5 FIR Memory and IO Maps

The next step in design of our embedded system is structuring memory and I/O of the processor and design of the CPU external busses. For a large system with many I/O devices and memory hierarchies this step involves design of address logic, I/O handshaking, arbitration, interrupt setting, priority encoding, etc. However, our system is much simpler than this.

Our embedded system needs a data memory for reading filter parameters, data input, filter coefficients, and writing filter outputs. In addition, the system needs an instruction memory for storing the filtering program to be read by the processor.

The bussing structure of our system only consists of processor data bus, address bus, and decoding logic for addressing these memory blocks. Our instruction memory begins at address 0000, and the data memory begins at 0100. We use a clocked memory for the instruction memory, and fast signal cycle asynchronous RAM for the data memory. Figure 10.32 shows the bussing structure of our embedded system.

The decoder and read/write logic shown in Figure 10.31 has AND/OR logic for address decoding and issuing read/write signals. The select logic blocks in this figure are for connection of the bidirectional SAYEH data bus to the data busses of the data memory and instruction memory.
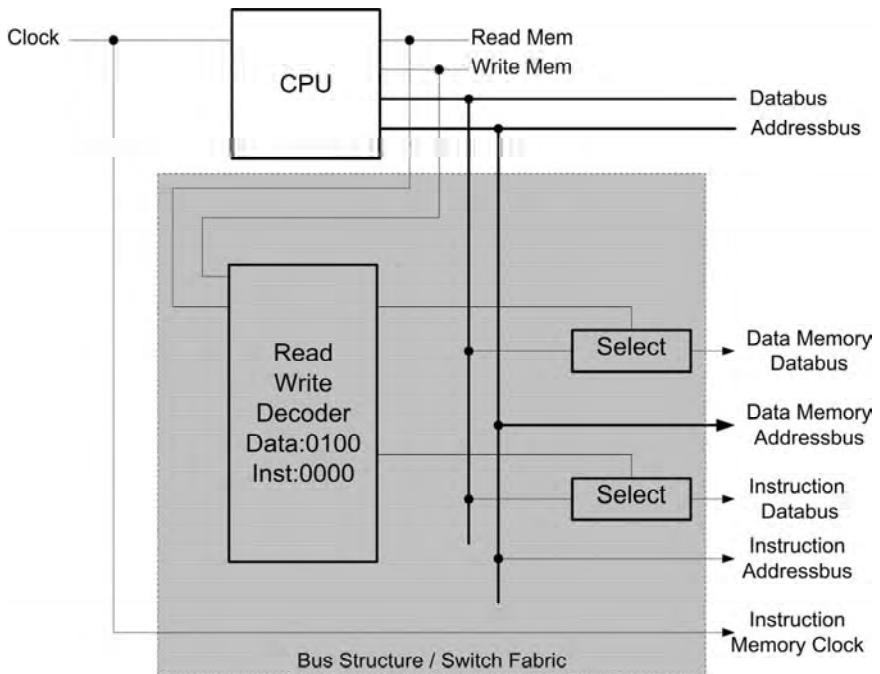
Figure 10.31 Embedded System Bus Structure

10.4.6 Filter Software

The last step in the design of our FIR filter example is the development of its software. The algorithm for this software is that of Figure 10.30, the hardware structure that this software will be implemented in is shown in Figure 10.31, and the processor that the software runs on is SAYEH. In an automated environment, compiling the C program of Figure 10.30 with consideration of memory mappings, would be all that we needed to do for this step of the design. In our case, however, we have to develop our software in SAYEH assembly code.

Considering the memory structure of Figure 10.31 and requirements of our algorithm (Figure 10.30) as to filter parameters, data and coefficients, we decide on the memory mapping shown in Figure 10.32.

The filter program in SAYEH assembly is developed based on the memory map of Figure 10.32. We first read locations 0100 and 0101 for the degree of the filter and the number of input samples. Then each data sample that is read starting in 0140, is multiplied by its corresponding coefficient that being in 0130, stored in locations 0120 to 012f and added to previous data in these same locations. For each

data read, an output is generated that is written starting in location 0180. Filter code in SAYEH assembly is shown in Figure 10.33.
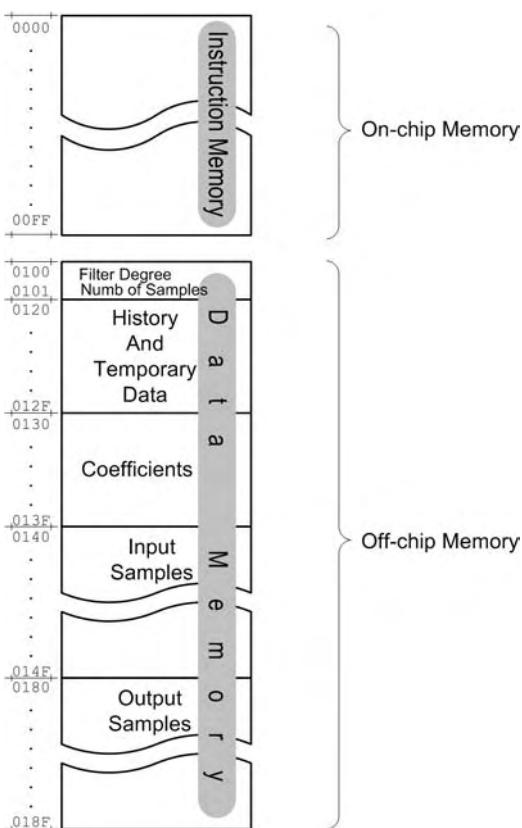
Figure 10.32 Filter Memory Map

The assembly code shown in this figure is translated to SAYEH machine language and becomes available for it to be loaded into the program memory (Figure 10.29). The task of assembly can be done manually, using the VHDL testbench assembler program of Section 10.3, or by writing an assembler for SAYEH.

```
0000 cwp                                   003A lda r1 r3
0001 mil r3 08                             003B mul r0 r1
0002 mih r3 01 //0108: Out Loop Cntr       003C cwp
0003 mil r1 00                             003D add r0 r2 //add MulOut to ACC
0004 mih r1 00                             003E awp 02
0005 sta r3 r1 // init Outer Loop          003F mil r3 00
0006 mil r3 09                             0040 mih r3 01
0007 mih r3 01 //0109: History Cntr        0041 lda r2 r3
0008 sta r3 r1 // init History Cntr        0042 mil r3 0A
0009 mil r3 0A                             0043 mih r3 01
000A mih r3 01 //010A: Mult Loop Cntr      0044 lda r1 r3
000B sta r3 r1 // init Mult Loop Cntr      0045 sub r2 r1
000C awp 04                                0046 brz 11     // quit MulLoop if degree
000D mil r3 00                             0047 mil r2 01
000E mih r3 01                             0048 mih r2 00
000F lda r2 r3 //0100: Filter Degree       0049 add r1 r2 // inc MulLoop
0010 mil r3 20                             004A sta r3 r1
0011 mih r3 01 //0120: 1st Temp Data       004B awp 01
0012 add r2 r3                             004C mil r1 20
0013 mil r3 01                             004D mih r1 01
0014 mih r3 00                             004E add r1 r0
0015 sub r2 r3                             004F mil r2 30
0016 lda r0 r2                             0050 mih r2 01
0017 add r2 r3                             0051 add r2 r0
0018 sta r2 r0 // shift history if n       0052 cwp
0019 mil r2 09                             0053 awp 02
001A mih r2 01                             0054 mil r1 00
001B lda r0 r2                             0055 mih r1 00
001C add r0 r3 // inct History cntr        0056 jpa r1 39
001D sta r2 r0 // save hist cntr at 0109   0057 cwp
001E mil r3 00                             0058 mil r1 80
001F mih r3 01                             0059 mih r1 01 // (0180:01BF) outputs
0020 lda r2 r3                             005A mil r2 08
0021 sub r2 r0                             005B mih r2 01
0022 brz 07     // exit loop if hist=degree 005C lda r3 r2
0023 mil r3 20                             005D add r1 r3
0024 mih r3 01                             005E sta r1 r0
0025 add r2 r3                             005F mil r0 01
0026 mil r3 00                             0060 mih r0 00
0027 mih r3 00                             0061 add r0 r3 // inc outer Loop Counter
0028 jpa r3 13                             0062 sta r2 r0 // add Loop cuntr at 0108
0029 mil r2 40                             0063 mil r3 01
002A mih r2 01 //0104: 1st Input           0064 mih r3 01
002B mil r3 08                             0065 lda r2 r3
002C mih r3 01                             0066 sub r2 r0
002D lda r1 r3 // lda OuterLoopCntr        0067 brz 0A     // quit main Loop if all
002E add r2 r1                             sample
002F lda r0 r2                             0068 mil r0 00
0030 mil r2 20                             0069 mih r0 00
0031 mih r2 01                             006A mil r1 00
0032 sta r2 r0 // new read data to 0120    006B mih r1 00
0033 cwp                                   006C mil r2 00
0034 awp 02                                006D mih r2 00
0035 mil r2 20                             006E mil r3 00
0036 mih r2 01                             006F mih r3 00 // init registers
0037 mil r3 30                             0070 jpa r3 06
0038 mih r3 01 //0130:013F coefficients    0071 hlt
0039 lda r0 r2
```

Figure 10.33 Filter Program Assembly Code

10.5 Summary

This chapter showed design and test of an example processor core that we referred to as SAYEH. We then used this processor core for the implementation of an FIR filter. In the design phase, we used RT level design strategy that we discussed in Chapter 8. Component structures and coding styles were also presented there. Specifically,

the design strategy used for SAYEH was the same as that of the small adding processor of Chapter 8. For testing our SAYEH processor, we used test schemes of Chapter 8. Specifically, the test methodology used for the adding machine was also used for SAYEH. The FIR filter showed an application of SAYEH and it demonstrated how a processor core can be used for implementing a hardware function. This chapter demonstrated design, test and application of processor cores.

Problems

10.1 SAYEH multiplication uses two 16-bit words and ignores the most-significant 8 bits of its two operands. We can correct this by developing a multiplier that uses an array multiplier that multiplies two 16-bit operands and generates its 32-bit output as two 16-bit results. You are to develop such a multiplier and interface it with Sayeh. A) Using an instantiation of a combinational array multiplier (assume this is given) that multiplies two 16-bit operands and generates a 32-bit result, write a multiplier that when it is started it takes its inputs, and generates its output as two 16-bit results. The multiplier is started when a 1 appears on its *start* input. It generates a *ready* signal when the result is ready and when the first half of the result is available on its output. For it to put the second half of the result on its output its *next* input should be asserted. When this happens, *ready* becomes 0 and when the second half is ready it (the multiplier) asserts *ready* again. Write this code completely independent of Sayeh and make sure its handshaking is appropriate for SAYEH to communicate with. You can change its handshaking as long as you follow it in Part B that deals with Sayeh controller. Also, write this unit such that if it is operated by a slower clock, or if it uses a sequential multiplier, it can still be used with Sayeh. B) Modify the controller of Sayeh such that when multiplication is to be done, it starts the multiplier of Part A and waits for the outputs of the multiplier to become ready. The controller should then take one 16-bit output and put it in R_D and take the next one and put it in R_{D+1}.

10.2 Write a testbench for the multiplier of Part A of Problem 1. This testbench reads *data1.dat* and *data2.dat* files, applies 16-bit data read from the two files to the operands of the multiplier and writes the operands and the two results in the *results.dat* file. Furthermore, when a 32-bit result becomes ready, the testbench calculates its own multiplication result by using the * operator, and compares this result with what the multiplier is generating. If a mis-

match happens, the testbench issues an error message using an assert statement. The testbench communicates with the multiplier using its handshaking signals. All data in the data files are in binary, and the testbench should continue reading the input files for as ling as there are new data in the files. It is expected that two input files have equal number of test data in them.

10.3 This chapter used SAYEH for the implementation of an FIR filer. This processor is a general purpose processor and many of its instructions were not used for the filter implementation. In some places, other instructions that SAYEH lacks could make the implementation of the filter easier. In this problem you are to design a processor specifically for filter design. Start the design of your filer processor by taking SAYEH and removing instructions and structures that are not needed (perhaps like the shadowing capability). Then add capabilities (perhaps a better multiplication) that are needed for filers. Complete your design, simulate it and write programs for design of several filters, including the simple FIR filter that we presented.

Suggested Reading

Accellera, *Open Verification Library: Assertion Monitor Reference Manual,* www.accellera.org, v1.0, 2005.

Baker, Louis, *VHDL Programming: With Advanced Topics*, 1992, Wiley Professional Computing, John Wiley & Sons Inc, ISBN: 978-0792390305.

Bening, Lionel and Harry D. Foster, *Principles of Verifiable RTL Design Second Edition - A Functional Coding Style Supporting Verification Processes in Verilog*, Springer, 2nd edition, 2001, ISBN: 0792373685.

Chu, Pong, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press, ISBN: 0471720925.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual, SH94983-TBR (print) SS94983-TBR (electronic), ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.

IEEE Std 1076.6-2004, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, SH95242 (print) SS95242 (electronic), ISBN 0-7381-4064-3 (print) 0-7381-4065-1 (electronic), 2004.

Lipsett, Roger, and Cary Ussery, *VHDL hardware description and Design, 1st edition*, 2001, Springer, ISBN: 978-0792390305.

Navabi, Zainalabedin, *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification*, 2006, McGraw Hill-Professional, ISBN: 0070144564-1.

Patterson, D.A., J.L. Hennessy, P.J. Ashenden, et al., *Computer Organization and Design: The Hardware/Software Interface, Third Edition,* Morgan Kaufmann, San Francisco, 2004, ISBN: 1558606041.

Perry, Douglas L., *VHDL: Programming By Example, 4th edition*, 2002, McGraw-Hill Professional, ISBN: 978-0071400701.

Perry, Douglas L., and Harry Foster, *Applied Formal Verification for Digital Circuit Design*, 2005, McGraw-Hill Professional, ISBN: 978-0071443722.

A   VHDL Keywords

A complete list of VHDL keywords is shown here. This information may be useful for selection of names, and for reading VHDL codes.

ABS	END	NAND
ACCESS	ENTITY	NEW
AFTER	EXIT	NEXT
ALIAS	FILE	NOR
ALL	FOR	NOT
AND	FUNCTION	NULL
ARCHITECTURE	GENERATE	OF
ARRAY	GENERIC	ON
ASSERT	GROUP	OPEN
ATTRIBUTE	GUARDED	OR
BEGIN	IF	OTHERS
BLOCK	IMPURE	OUT
BODY	IN	PACKAGE
BUFFER	INERTIAL	PORT
BUS	INOUT	POSTPONED
CASE	IS	PROCEDURE
COMPONENT	LABEL	PROCESS
CONFIGURATION	LIBRARY	PROTECTED
CONSTANT	LINKAGE	PURE
DISCONNECT	LITERAL	RANGE
DOWNTO	LOOP	RECORD
ELSE	MAP	REGISTER
ELSIF	MOD	REJECT

REM	SLL	UNTIL
REPORT	SRA	USE
RETURN	SRL	VARIABLE
ROL	SUBTYPE	WAIT
ROR	THEN	WHEN
SELECT	TO	WHILE
SEVERITY	TRANSPORT	WITH
SHARED	TYPE	XNOR
SIGNAL	UNAFFECTED	XOR
SLA	UNITS	

B VHDL Language Grammar

This appendix contains the formal Grammar of the standard ANSI/IEEE Std 1076-2002 VHDL language in BNF format. In this format, productions are on the left hand side of an equivalence, two colons and an equal sign are used for equivalence, vertical bars for oring, square brackets for optional parts, and curly brackets for parts that zero or more of them may be used. Language keywords and reserved words are in bold type. Language productions are ordered in alphabetical order.

abstract_literal ::= decimal_literal | based_literal

access_type_definition ::= **access** subtype_indication

actual_designator ::=
       expression
    | *signal*_name
    | *variable*_name
    | *file*_name
    | **open**

actual_parameter_part ::= *parameter*_association_list

actual_part ::=
      actual_designator
    | *function*_name (actual_designator)
    | type_mark (actual_designator)

adding_operator ::= + | – | &

aggregate ::=
  (element_association { , element_association })

alias_declaration ::=
  alias alias_designator [: subtype_indication] **is** name [signature] ;

alias_designator ::= identifier | character_literal | operator_symbol

allocator ::=
   new subtype_indication
  | **new** qualified_expression

architecture_body ::=
  architecture identifier **of** *entity*_name **is**
    architecture_declarative_part
  begin
    architecture_statement_part
  end [**architecture**] [*architecture*_simple_name] ;

architecture_declarative_part ::=
  { block_declarative_item }

architecture_statement_part ::=
  { concurrent_statement }

array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

assertion ::=
  assert condition
    [**report** expression]
    [**severity** expression]

assertion_statement ::= [label :] assertion ;

association_element ::=
  [formal_part =>] actual_part

association_list ::=
  association_element { , association_element }

attribute_declaration ::=
  attribute identifier : type_mark ;

attribute_designator ::= *attribute*_simple_name

attribute_name ::=
  prefix [signature] ' attribute_designator [(expression)]

attribute_specification ::=
        attribute attribute_designator **of** entity_specification **is** expression ;

base ::= integer

base_specifier ::= B | O | X

based_integer ::=
        extended_digit { [underline] extended_digit }

based_literal ::=
        base # based_integer [. based_integer] # [exponent]

basic_character ::=
        basic_graphic_character | format_effector

basic_graphic_character ::=
        upper_case_letter | digit | special_character | space_character

basic_identifier ::= letter { [underline] letter_or_digit }

binding_indication ::=
        [**use** entity_aspect]
        [generic_map_aspect]
        [port_map_aspect]

bit_string_literal ::= base_specifier " [bit_value] "

bit_value ::= extended_digit { [underline] extended_digit }

block_configuration ::=
        for block_specification
                { use_clause }
                { configuration_item }
        end for ;

block_declarative_item ::=
         subprogram_declaration
        | subprogram_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | signal_declaration
        | *shared*_variable_declaration
        | file_declaration
        | alias_declaration
        | component_declaration
        | attribute_declaration
        | attribute_specification
        | configuration_specification

```
          | disconnection_specification
          | use_clause
          | group_template_declaration
          | group_declaration

block_declarative_part ::=
          { block_declarative_item }

block_header ::=
          [ generic_clause
          [ generic_map_aspect ; ] ]
          [ port_clause
          [ port_map_aspect ; ] ]

block_specification ::=
            architecture_name
          | block_statement_label
          | generate_statement_label [ ( index_specification ) ]

block_statement ::=
          block_label :
                    block [ ( guard_expression ) ] [ is ]
                          block_header
                          block_declarative_part
                    begin
                          block_statement_part
                    end block [ block_label ] ;

block_statement_part ::=
          { concurrent_statement }

case_statement ::=
          [ case_label : ]
                    case expression is
                          case_statement_alternative
                          { case_statement_alternative }
                    end case [ case_label ] ;

case_statement_alternative ::=
          when choices =>
                    sequence_of_statements

character_literal ::= ' graphic_character '

choice ::=
            simple_expression
          | discrete_range
          | element_simple_name
          | others
```

choices ::= choice { | choice }

component_configuration ::=
      for component_specification
              [binding_indication ;]
              [block_configuration]
      end for ;

component_declaration ::=
      component identifier [**is**]
              [*local_*generic_clause]
              [*local_*port_clause]
      end component [*component_*simple_name] ;

component_instantiation_statement ::=
      *instantiation_*label :
              instantiated_unit
                    [generic_map_aspect]
                    [port_map_aspect] ;

component_specification ::=
      instantiation_list : *component_*name

composite_type_definition ::=
       array_type_definition
      | record_type_definition

concurrent_assertion_statement ::=
      [label :] [**postponed**] assertion ;

concurrent_procedure_call_statement ::=
      [label :] [**postponed**] procedure_call ;

concurrent_signal_assignment_statement ::=
      [label :] [**postponed**] conditional_signal_assignment
      | [label :] [**postponed**] selected_signal_assignment

concurrent_statement ::=
       block_statement
      | process_statement
      | concurrent_procedure_call_statement
      | concurrent_assertion_statement
      | concurrent_signal_assignment_statement
      | component_instantiation_statement
      | generate_statement

condition ::= *boolean_*expression

condition_clause ::= **until** condition

conditional_signal_assignment ::=
      target <= options conditional_waveforms ;

conditional_waveforms ::=
      { waveform **when** condition **else** }
       waveform [**when** condition]

configuration_declaration ::=
      configuration identifier **of** *entity*_name **is**
            configuration_declarative_part
            block_configuration
      end [**configuration**] [*configuration*_simple_name] ;

configuration_declarative_item ::=
       use_clause
      | attribute_specification
      | group_declaration

configuration_declarative_part ::=
      { configuration_declarative_item }

configuration_item ::=
       block_configuration
      | component_configuration

configuration_specification ::=
      for component_specification binding_indication ;

constant_declaration ::=
      constant identifier_list : subtype_indication [:= expression] ;

constrained_array_definition ::=
      array index_constraint **of** *element*_subtype_indication

constraint ::=
       range_constraint
      | index_constraint

context_clause ::= { context_item }

context_item ::=
       library_clause
      | use_clause

decimal_literal ::= integer [. integer] [exponent]

declaration ::=
       type_declaration
      | subtype_declaration
      | object_declaration

```
            | interface_declaration
            | alias_declaration
            | attribute_declaration
            | component_declaration
            | group_template_declaration
            | group_declaration
            | entity_declaration
            | configuration_declaration
            | subprogram_declaration
            | package_declaration
            | primary_unit
            | architecture_body
```

delay_mechanism ::=
            transport
            | [**reject** *time*_expression] **inertial**

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= **to** | **downto**

disconnection_specification ::=
            disconnect guarded_signal_specification **after** *time*_expression ;

discrete_range ::= *discrete*_subtype_indication | range

element_association ::=
            [choices =>] expression

element_declaration ::=
            identifier_list : element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::=
            entity *entity*_name [(*architecture*_identifier)]
            | **configuration** *configuration*_name
            | **open**

entity_class ::=
            entity | architecture | configuration
            | procedure | function | package
            | type | subtype | constant
            | signal | variable | component
            | label | literal | units

entity_class_entry ::= entity_class [<>]

entity_class_entry_list ::=
  entity_class_entry { , entity_class_entry }

entity_declaration ::=
  entity identifier **is**
    entity_header
    entity_declarative_part
  [**begin**
    entity_statement_part]
  end [**entity**] [*entity*_simple_name] ;

entity_declarative_item ::=
   subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | *shared*_variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | disconnection_specification
  | use_clause
  | group_template_declaration
  | group_declaration

entity_declarative_part ::=
  { entity_declarative_item }

entity_designator ::= entity_tag [signature]

entity_header ::=
  [*formal*_generic_clause]
  [*formal*_port_clause]

entity_name_list ::=
   entity_designator { , entity_designator }
  | **others**
  | **all**

entity_specification ::=
  entity_name_list : entity_class

entity_statement ::=
   concurrent_assertion_statement
  | *passive*_concurrent_procedure_call
  | *passive*_process_statement

entity_statement_part ::=
        { entity_statement }

entity_tag ::= simple_name | character_literal | operator_symbol

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::=
        (enumeration_literal { , enumeration_literal })

exit_statement ::=
        [label :] **exit** [*loop*_label] [**when** condition] ;

exponent ::= E [+] integer | E – integer

expression ::=
          relation { **and** relation }
        | relation { **or** relation }
        | relation { **xor** relation }
        | relation [**nand** relation]
        | relation [**nor** relation]
        | relation { **xnor** relation }

extended_digit ::= digit | letter

extended_identifier ::= \ graphic_character { graphic_character } \

factor ::=
          primary [** primary]
        | **abs** primary
        | **not** primary

file_declaration ::=
        file identifier_list : subtype_indication [file_open_information] ;

file_logical_name ::= *string*_expression

file_open_information ::=
        [**open** *file_open_kind*_expression] **is** file_logical_name

file_type_definition ::=
        file of type_mark

floating_type_definition ::= range_constraint

formal_designator ::=
          *generic*_name
        | *port*_name
        | *parameter*_name

formal_parameter_list ::= *parameter*_interface_list

formal_part ::=
        formal_designator
      | *function*_name (formal_designator)
      | type_mark (formal_designator)

full_type_declaration ::=
     type identifier **is** type_definition ;

function_call ::=
     *function*_name [(actual_parameter_part)]

generate_statement ::=
     *generate*_label :
         generation_scheme **generate**
             [{ block_declarative_item }
        begin]
            { concurrent_statement }
        end generate [*generate*_label] ;

generation_scheme ::=
      for *generate*_parameter_specification
     | **if** condition

generic_clause ::=
     generic (generic_list) ;

generic_list ::= *generic*_interface_list

generic_map_aspect ::=
     generic map (*generic*_association_list)

graphic_character ::=
     basic_graphic_character | lower_case_letter |
     other_special_character

group_constituent ::= name | character_literal

group_constituent_list ::= group_constituent { , group_constituent }

group_declaration ::=
     group identifier : *group_template*_name (group_constituent_list) ;

group_template_declaration ::=
     group identifier **is** (entity_class_entry_list) ;

guarded_signal_specification ::=
     *guarded*_signal_list : type_mark

identifier ::= basic_identifier | extended_identifier

identifier_list ::= identifier { , identifier }

if_statement ::=
        [*if*_label :]
                if condition **then**
                        sequence_of_statements
                { **elsif** condition **then**
                        sequence_of_statements }
                [**else**
                            sequence_of_statements]
                end if [*if*_label] ;

incomplete_type_declaration ::= **type** identifier ;

index_constraint ::= (discrete_range { , discrete_range })

index_specification ::=
          discrete_range
        | *static*_expression

index_subtype_definition ::= type_mark **range** <>

indexed_name ::= prefix (expression { , expression })

instantiated_unit ::=
        [**component**] *component*_name
        | **entity** *entity*_name [(*architecture*_identifier)]
        | **configuration** *configuration*_name

instantiation_list ::=
        *instantiation*_label { , *instantiation*_label }
        | **others**
        | **all**

integer ::= digit { [underline] digit }

integer_type_definition ::= range_constraint

interface_constant_declaration ::=
        [**constant**] identifier_list : [**in**] subtype_indication
        [:= *static*_expression]

interface_declaration ::=
          interface_constant_declaration
        | interface_signal_declaration
        | interface_variable_declaration
        | interface_file_declaration

interface_element ::= interface_declaration

interface_file_declaration ::=
       file identifier_list : subtype_indication

interface_list ::=
       interface_element { ; interface_element }

interface_signal_declaration ::=
       [**signal**] identifier_list : [mode] subtype_indication [**bus**]
       [:= *static*_expression]

interface_variable_declaration ::=
       [**variable**] identifier_list : [mode] subtype_indication
       [:= *static*_expression]

iteration_scheme ::=
        while condition
      | **for** *loop*_parameter_specification

label ::= identifier

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit

library_clause ::= **library** logical_name_list ;

library_unit ::=
        primary_unit
      | secondary_unit

literal ::=
        numeric_literal
      | enumeration_literal
      | string_literal
      | bit_string_literal
      | **null**

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor** | **xnor**

loop_statement ::=
      [*loop*_label :]
           [iteration_scheme] **loop**
                sequence_of_statements
           end loop [*loop*_label] ;

miscellaneous_operator ::= ** | **abs** | **not**

mode ::= **in** | **out** | **inout** | **buffer** | **linkage**

multiplying_operator ::= * | / | **mod** | **rem**

name ::=
      simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

next_statement ::=
    [label :] **next** [*loop*_label] [**when** condition] ;

null_statement ::= [label :] **null** ;

numeric_literal ::=
      abstract_literal
    | physical_literal

object_declaration ::=
      constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration

operator_symbol ::= string_literal

options ::= [**guarded**] [delay_mechanism]

package_body ::=
    package body *package*_simple_name **is**
        package_body_declarative_part
    end [**package body**] [*package*_simple_name] ;

package_body_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | *shared*_variable_declaration
    | file_declaration
    | alias_declaration
    | use_clause
    | group_template_declaration
    | group_declaration

package_body_declarative_part ::=
    { package_body_declarative_item }

package_declaration ::=
    package identifier **is**
        package_declarative_part
    end [**package**] [*package_*simple_name] ;

package_declarative_item ::=
     subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | *shared_*variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration

package_declarative_part ::=
    { package_declarative_item }

parameter_specification ::=
    identifier **in** discrete_range

physical_literal ::= [abstract_literal] *unit_*name

physical_type_definition ::=
    range_constraint
        units
            primary_unit_declaration
            { secondary_unit_declaration }
        end units [*physical_type_*simple_name]

port_clause ::=
    port (port_list) ;

port_list ::= *port_*interface_list

port_map_aspect ::=
    port map (*port_*association_list)

prefix ::=
          name
        | function_call

primary ::=
          name
        | literal
        | aggregate
        | function_call
        | qualified_expression
        | type_conversion
        | allocator
        | (expression)

primary_unit ::=
          entity_declaration
        | configuration_declaration
        | package_declaration

primary_unit_declaration ::= identifier ;

procedure_call ::= *procedure*_name [(actual_parameter_part)]

procedure_call_statement ::= [label :] procedure_call ;

process_declarative_item ::=
          subprogram_declaration
        | subprogram_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | variable_declaration
        | file_declaration
        | alias_declaration
        | attribute_declaration
        | attribute_specification
        | use_clause
        | group_template_declaration
        | group_declaration

process_declarative_part ::=
        { process_declarative_item }

process_statement ::=
        [*process*_label :]
                [**postponed**] **process** [(sensitivity_list)] [**is**]
                        process_declarative_part
                begin
                        process_statement_part
                end [**postponed**] **process** [*process*_label] ;

process_statement_part ::=
      { sequential_statement }

protected_type_body ::=
      protected body
            protected_type_body_declarative_part
      end protected body [*protected_type_*simple name]

protected_type_body_declarative_item ::=
       subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | variable_declaration
      | file_declaration
      | alias_declaration
      | attribute_declaration
      | attribute_specification
      | use_clause
      | group_template_declaration
      | group_declaration

protected_type_body_declarative_part ::=
      { protected_type_body_declarative_item }

protected_type_declaration ::=
      protected
            protected_type_declarative_part
      end protected [*protected_type_*simple_name]

protected_type_declarative_item ::=
       subprogram_specification
      | attribute_specification
      | use_clause

protected_type_declarative_part ::=
      { protected_type_declarative_item }

protected_type_definition ::=
       protected_type_declaration
      | protected_type_body

qualified_expression ::=
       type_mark ' (expression)
      | type_mark ' aggregate

range ::=
       *range_*attribute_name
      | simple_expression direction simple_expression

range_constraint ::= **range** range

record_type_definition ::=
      record
           element_declaration
        { element_declaration }
      end record [*record_type*_simple_name]

relation ::=
      shift_expression [relational_operator shift_expression]

relational_operator ::= = | /= | < | <= | > | >=

report_statement ::=
      [label :]
             report expression
                [**severity** expression] ;

return_statement ::=
      [label :] **return** [expression] ;

scalar_type_definition ::=
       enumeration_type_definition | integer_type_definition
      | floating_type_definition | physical_type_definition

secondary_unit ::=
       architecture_body
      | package_body

secondary_unit_declaration ::= identifier = physical_literal ;

selected_name ::= prefix . suffix

selected_signal_assignment ::=
      with expression **select**
           target <= options selected_waveforms ;

selected_waveforms ::=
      { waveform **when** choices , }
       waveform **when** choices

sensitivity_clause ::= **on** sensitivity_list

sensitivity_list ::= *signal*_name { , *signal*_name }

sequence_of_statements ::=
      { sequential_statement }

sequential_statement ::=
            wait_statement
          | assertion_statement
          | report_statement
          | signal_assignment_statement
          | variable_assignment_statement
          | procedure_call_statement
          | if_statement
          | case_statement
          | loop_statement
          | next_statement
          | exit_statement
          | return_statement
          | null_statement

shift_expression ::=
          simple_expression [shift_operator simple_expression]

shift_operator ::= **sll** | **srl** | **sla** | **sra** | **rol** | **ror**

sign ::= + | −

signal_assignment_statement ::=
          [label :] target <= [delay_mechanism] waveform ;

signal_declaration ::=
          signal identifier_list : subtype_indication [signal_kind]
          [:= expression] ;

signal_kind ::= **register** | **bus**

signal_list ::=
            *signal*_name { , *signal*_name }
          | **others**
          | **all**

signature ::= [[type_mark { , type_mark }] [**return** type_mark]]

simple_expression ::=
          [sign] term { adding_operator term }

simple_name ::= identifier

slice_name ::= prefix (discrete_range)

string_literal ::= "{ graphic_character } " "

subprogram_body ::=
          subprogram_specification **is**
                    subprogram_declarative_part

      begin
           subprogram_statement_part
      end [subprogram_kind] [designator] ;

subprogram_declaration ::=
      subprogram_specification ;

subprogram_declarative_item ::=
       subprogram_declaration
     | subprogram_body
     | type_declaration
     | subtype_declaration
     | constant_declaration
     | variable_declaration
     | file_declaration
     | alias_declaration
     | attribute_declaration
     | attribute_specification
     | use_clause
     | group_template_declaration
     | group_declaration

subprogram_declarative_part ::=
      { subprogram_declarative_item }

subprogram_kind ::= **procedure** | **function**

subprogram_specification ::=
      procedure designator [(formal_parameter_list)]
     | [**pure** | **impure**] **function** designator [(formal_parameter_list)
     ]
                return type_mark
subprogram_statement_part ::=
      { sequential_statement }

subtype_declaration ::=
      subtype identifier **is** subtype_indication ;

subtype_indication ::=
      [*resolution_function_*name] type_mark [constraint]

suffix ::=
       simple_name
     | character_literal
     | operator_symbol
     | **all**

target ::=
      name
     | aggregate

term ::=
       factor { multiplying_operator factor }

timeout_clause ::= **for** *time*_expression

type_conversion ::= type_mark (expression)

type_declaration ::=
        full_type_declaration
      | incomplete_type_declaration

type_definition ::=
        scalar_type_definition
      | composite_type_definition
      | access_type_definition
      | file_type_definition
      | protected_type_definition

type_mark ::=
       *type*_name
      | *subtype*_name

unconstrained_array_definition ::=
      array (index_subtype_definition { , index_subtype_definition })
          of *element*_subtype_indication

use_clause ::=
      use selected_name { , selected_name } ;

variable_assignment_statement ::=
      [label :] target := expression ;

variable_declaration ::=
      [**shared**] **variable** identifier_list : subtype_indication
      [:= expression] ;

wait_statement ::=
      [label :] **wait** [sensitivity_clause] [condition_clause]
      [timeout_clause] ;

waveform ::=
       waveform_element { , waveform_element }
      | **unaffected**

waveform_element ::=
       *value*_expression [**after** *time*_expression]
      | **null** [**after** *time*_expression]

C VHDL Standard Packages

This appendix presents the standard VHDL packages. Section C.1 presents the STANDARD package, and Section C.2 presents the TEXTIO package. In all descriptions of this book, we assumed that all types and functions of the STANDARD package are visible, but the TEXTIO package had to be explicitly declared when needed.

C.1 STANDARD Package

The STANDARD package defines primitive types, subtypes and functions. This package resides in the STD library.

```
-- This is Package STANDARD as defined in the VHDL 1992
--    Language Reference Manual.
--
--    NOTE: VCOM and VSIM will not work properly if these
--          declarations are modified.

-- Version information: @(#)standard.vhd

package standard is
    type boolean is (false,true);
    type bit is ('0', '1');
    type character is (
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em,  sub, esc, fsp, gsp, rsp, usp,
```

```
        ' ', '!', '"', '#', '$', '%', '&', ''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',

        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',

        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', del,

        c128, c129, c130, c131, c132, c133, c134, c135,
        c136, c137, c138, c139, c140, c141, c142, c143,
        c144, c145, c146, c147, c148, c149, c150, c151,
        c152, c153, c154, c155, c156, c157, c158, c159,

        -- the character code for 160 is there (NBSP),
        -- but prints as no char

        ' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
        '¨', '©', 'ª', '«', '¬', '', '®', '¯',
        '°', '±', '²', '³', '´', 'µ', '¶', '·',
        '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',

        'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
        'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
        'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
        'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',

        'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
        'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
        'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
        'ø', 'ù', 'ú', 'û', 'ü', 'Ý', 'þ', 'ÿ' );

    type severity_level is (note, warning, error, failure);
    type integer is range -2147483648 to 2147483647;
    type real is range -1.0E308 to 1.0E308;

    type time is range -2147483647 to 2147483647
        units
            fs;
            ps = 1000 fs;
            ns = 1000 ps;
            us = 1000 ns;
            ms = 1000 us;
            sec = 1000 ms;
            min = 60 sec;
            hr = 60 min;
        end units;
```

```
    subtype delay_length is time range 0 fs to time'high;
    impure function now return delay_length;
    subtype natural is integer range 0 to integer'high;
    subtype positive is integer range 1 to integer'high;
    type string is array (positive range <>) of character;
    type bit_vector is array (natural range <>) of bit;
    type file_open_kind is (
        read_mode,
        write_mode,
        append_mode);
    type file_open_status is (
        open_ok,
        status_error,
        name_error,
        mode_error);
    attribute foreign : string;
end standard;
```

C.2 TEXTIO Package

The TEXTIO package defines types, procedures, and functions for standard text I/O from ASCII files. This package resides in the STD library.

```
---------------------------------------------------------
-- Package TEXTIO as defined in Chapter 14 of the IEEE
--   Standard VHDL Language Reference Manual (IEEE Std.
--   1076-1987), as modified by the Issues Screening and
--   Analysis Committee (ISAC), a subcommittee of the VHDL
--   Analysis and Standardization Group (VASG) on  10
--   November, 1988.  See "The Sense of the VASG", October,
--   1989.
---------------------------------------------------------
-- Version information: %W% %G%
---------------------------------------------------------

package TEXTIO is

    type LINE is access string;
    type TEXT is file of string;
    type SIDE is (right, left);
    subtype WIDTH is natural;

    -- changed for vhdl92 syntax:
    file input : TEXT open read_mode is "STD_INPUT";
    file output : TEXT open write_mode is "STD_OUTPUT";

    -- changed for vhdl92 syntax (and now a built-in):
```

```
procedure READLINE(file f: TEXT; L: out LINE);


procedure READ(L:inout LINE; VALUE: out bit; GOOD : out
               BOOLEAN);

procedure READ(L:inout LINE; VALUE: out bit);


procedure READ(L:inout LINE; VALUE: out bit_vector;
               GOOD : out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out bit_vector);


procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD :
               out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out BOOLEAN);


procedure READ(L:inout LINE; VALUE: out character; GOOD
               : out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out character);


procedure READ(L:inout LINE; VALUE: out integer; GOOD :
               out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out integer);


procedure READ(L:inout LINE; VALUE: out real; GOOD :
               out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out real);


procedure READ(L:inout LINE; VALUE: out string; GOOD :
               out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out string);


procedure READ(L:inout LINE; VALUE: out time; GOOD :
               out BOOLEAN);

procedure READ(L:inout LINE; VALUE: out time);


-- changed for vhdl92 syntax (and now a built-in):
```

```
procedure WRITELINE(file f : TEXT; L : inout LINE);


procedure WRITE(L : inout LINE; VALUE : in bit;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in bit_vector;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in character;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in integer;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in real;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0;
      DIGITS: in NATURAL := 0);


procedure WRITE(L : inout LINE; VALUE : in string;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0);


procedure WRITE(L : inout LINE; VALUE : in time;
      JUSTIFIED: in SIDE := right;
      FIELD: in WIDTH := 0;
      UNIT: in TIME := ns);


-- is implicit built-in:
-- function ENDFILE(file F : TEXT) return boolean;

-- function ENDLINE(variable L : in LINE) return
--      BOOLEAN;
--
-- Function ENDLINE as declared cannot be legal VHDL,
```

```
    --    and the entire function was deleted from the
    --    definition by the Issues Screening and Analysis
    --    Committee (ISAC), a subcommittee of the VHDL
    --    Analysis and Standardization Group (VASG) on 10
    --    November, 1988.  See "The Sense of the VASG",
    --    October, 1989, VHDL Issue Number 0032.
end;
```

D STD_LOGIC_1164 Package

This appendix shows the *Std_Logic* IEEE standard 1164 nine-value logic package. All designs using this package must use the *LIBRARY IEEE;* and *USE IEEE.std_logic_1164.ALL;* statements for making this package and its contents accessible.

```
-- --------------------------------------------------------
--
--
-- Copyright©2004 by the Institute of Electrical and
-- Electronics Engineers, Inc.
-- Three Park Avenue
-- New York, NY 10016-5997, USA
-- All rights reserved.
--
--
-- This document is an unapproved draft of a proposed IEEE
-- Standard. As such, this document is subject to change.
-- USE AT YOUR OWN RISK! Because this is an unapproved
-- draft, this document must not be utilized for any
-- conformance/compliance purposes. Permission is hereby
-- granted for IEEE Standards Committee participants to
-- reproduce this document for purposes of IEEE
-- standardization activities only. Prior to submitting
-- this document to another standards development
-- organization for standardization activities, permission
-- must first be obtained from the Manager, Standards
-- Licensing and Contracts, IEEE Standards Activities
-- Department. Other entities seeking permission to
```

```
-- reproduce this document, in whole or in part, must
-- obtain permission from the Manager, Standards Licensing
-- and Contracts, IEEE Standard Activities Department.
--
-- IEEE Standards Activities Department
-- Standards Licensing and Contracts
-- 445 Hoes Lane, P.O. Box 1331
-- Piscataway, NJ 08855-1331, USA
-- ---------------------------------------------------------
--
--   Title    : std_logic_1164 multi-value logic system
--   Library  : This package shall be compiled into a
--            : library symbolically named IEEE.
--            :
--   Developers: IEEE model standards group (par 1164)
--   Purpose  : This packages defines a standard for
--              designers to use in describing the
--              interconnection data types used in vhdl
--              modeling.
--            :
--   Limitation: The logic system defined in this package
--            : may be insufficient for modeling switched
--            : transistors, since such a requirement is
--            : out of the scope of this effort.
--            : Furthermore, mathematics, primitives,
--            : timing standards, etc. are considered
--            : orthogonal issues as it relates to this
--            : package and are therefore beyond the
--            : scope of this effort.
--            :
--   Note     : No declarations or definitions shall be
--            : included in, or excluded from this
--            : package. The "package declaration"
--            : defines the types, subtypes and
--            : declarations of std_logic_1164. The
--            : std_logic_1164 package body shall be
--            : considered the formal definition of the
--            : semantics of this package. Tool
--            : developers may choose to implement the
--            : package body in the most efficient manner
--            : available to them.
--            :
-- ---------------------------------------------------------
--   modification history :
-- ---------------------------------------------------------
-- version | mod. date:|
--  v4.200 | 01/02/92  |
-- ---------------------------------------------------------
-- version | mod. date:| Copied from original, and began
-- VHDL-200X
-- v5.000 | 06/22/04  | modifications. David Bishop
-- dbishop@vhdl.org
-- ---------------------------------------------------------
```

```
-- rtl_synthesis off
use std.textio.all;
-- rtl_synthesis on
PACKAGE std_logic_1164 IS


   ----------------------------------------------------------
   -- logic state system  (unresolved)
   ----------------------------------------------------------
   TYPE std_ulogic IS ( 'U',                -- Uninitialized
                        'X',                -- Forcing  Unknown
                        '0',                -- Forcing  0
                        '1',                -- Forcing  1
                        'Z',                -- High Impedance
                        'W',                -- Weak     Unknown
                        'L',                -- Weak     0
                        'H',                -- Weak     1
                        '-'                 -- Don't care
                        );
   ----------------------------------------------------------
   -- unconstrained array of std_ulogic for use with the
   -- resolution function
   ----------------------------------------------------------
   TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF
        std_ulogic;

   ----------------------------------------------------------
   -- resolution function
   ----------------------------------------------------------
   FUNCTION resolved ( s : std_ulogic_vector ) RETURN
           std_ulogic;


   ----------------------------------------------------------
   -- *** industry standard logic type ***
   ----------------------------------------------------------
   SUBTYPE std_logic IS resolved std_ulogic;

   ----------------------------------------------------------
   -- unconstrained array of std_logic for use in declaring
   -- signal arrays
   ----------------------------------------------------------
   TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF
        std_logic;

   ----------------------------------------------------------
   -- common subtypes
   ----------------------------------------------------------
   SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';
        -- ('X','0','1')
   SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z';
        -- ('X','0','1','Z')
```

```
SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1';
      -- ('U','X','0','1')
SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z';
      -- ('U','X','0','1','Z')


-----------------------------------------------------------
-- overloaded logical operators
-----------------------------------------------------------
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN
      UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN
      UX01;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN
      UX01;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN
      UX01;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN
      UX01;
function "xnor" ( l : std_ulogic; r : std_ulogic ) return
      ux01;
FUNCTION "not"  ( l : std_ulogic                 ) RETURN
      UX01;


-----------------------------------------------------------
-- vectorized overloaded logical operators
-----------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;

FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;

FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;

FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN
      std_ulogic_vector;
```

```
function "xnor" ( l, r : std_logic_vector  ) return
      std_logic_vector;
function "xnor" ( l, r : std_ulogic_vector ) return
      std_ulogic_vector;

FUNCTION "not"  ( l : std_logic_vector  ) RETURN
      std_logic_vector;
FUNCTION "not"  ( l : std_ulogic_vector ) RETURN
      std_ulogic_vector;


----------------------------------------------------------
-- conversion functions
----------------------------------------------------------
FUNCTION To_bit ( s : std_ulogic; xmap : BIT := '0' )
      RETURN BIT;
FUNCTION To_bitvector ( s    : std_logic_vector;
      xmap : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_bitvector ( s : std_ulogic_vector;
      xmap : BIT := '0') RETURN BIT_VECTOR;

FUNCTION To_StdULogic ( b      : BIT ) RETURN
      std_ulogic;
FUNCTION To_StdLogicVector ( b  : BIT_VECTOR ) RETURN
      std_logic_vector;
FUNCTION To_StdLogicVector ( s  : std_ulogic_vector )
      RETURN std_logic_vector;
alias to_slv is To_StdLogicVector [BIT_VECTOR return
      std_logic_vector] ;
alias to_slv is To_StdLogicVector [std_ulogic_vector
      return std_logic_vector] ;
FUNCTION To_StdULogicVector ( b : BIT_VECTOR ) RETURN
      std_ulogic_vector;
FUNCTION To_StdULogicVector ( s : std_logic_vector )
      RETURN std_ulogic_vector;
alias to_sulv is To_StdULogicVector [BIT_VECTOR return
      std_ulogic_vector] ;
alias to_sulv is To_StdULogicVector [std_logic_vector
      return std_ulogic_vector] ;
----------------------------------------------------------
-- strength strippers and type convertors
----------------------------------------------------------
FUNCTION To_X01 ( s : std_logic_vector ) RETURN
      std_logic_vector;
FUNCTION To_X01 ( s : std_ulogic_vector ) RETURN
      std_ulogic_vector;
FUNCTION To_X01 ( s : std_ulogic ) RETURN X01;
FUNCTION To_X01 ( b : BIT_VECTOR ) RETURN
      std_logic_vector;
FUNCTION To_X01 ( b : BIT_VECTOR ) RETURN
      std_ulogic_vector;
FUNCTION To_X01 ( b : BIT ) RETURN X01;
```

```
FUNCTION To_X01Z ( s : std_logic_vector ) RETURN
      std_logic_vector;
FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN
      std_ulogic_vector;
FUNCTION To_X01Z ( s : std_ulogic ) RETURN X01Z;
FUNCTION To_X01Z ( b : BIT_VECTOR ) RETURN
      std_logic_vector;
FUNCTION To_X01Z ( b : BIT_VECTOR ) RETURN
      std_ulogic_vector;
FUNCTION To_X01Z ( b : BIT ) RETURN X01Z;

FUNCTION To_UX01 ( s : std_logic_vector ) RETURN
      std_logic_vector;
FUNCTION To_UX01 ( s : std_ulogic_vector ) RETURN
      std_ulogic_vector;
FUNCTION To_UX01 ( s : std_ulogic ) RETURN UX01;
FUNCTION To_UX01 ( b : BIT_VECTOR ) RETURN
      std_logic_vector;
FUNCTION To_UX01 ( b : BIT_VECTOR ) RETURN
      std_ulogic_vector;
FUNCTION To_UX01 ( b : BIT ) RETURN UX01;


-----------------------------------------------------------
-- edge detection
-----------------------------------------------------------
FUNCTION rising_edge (SIGNAL s  : std_ulogic) RETURN
      BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN
      BOOLEAN;


-----------------------------------------------------------
-- object contains an unknown
-----------------------------------------------------------
FUNCTION Is_X ( s : std_ulogic_vector ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_logic_vector ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_ulogic ) RETURN BOOLEAN;
-----------------------------------------------------------
-- New/updated funcitons for VHDL-200X fast track
-----------------------------------------------------------
-----------------------------------------------------------
-- overloaded shift operators
-----------------------------------------------------------

function "sll" ( l : std_logic_vector;  r : integer )
      RETURN std_logic_vector;
function "sll" ( l : std_ulogic_vector; r : integer )
      RETURN std_ulogic_vector;

function "srl" ( l : std_logic_vector;  r : integer )
      RETURN std_logic_vector;
function "srl" ( l : std_ulogic_vector; r : integer )
      RETURN std_ulogic_vector;
```

```
function "sla" ( l : std_logic_vector;  r : integer )
     RETURN std_logic_vector;
function "sla" ( l : std_ulogic_vector; r : integer )
     RETURN std_ulogic_vector;

function "sra" ( l : std_logic_vector;  r : integer )
     RETURN std_logic_vector;
function "sra" ( l : std_ulogic_vector; r : integer )
     RETURN std_ulogic_vector;

function "rol" ( l : std_logic_vector;  r : integer )
     RETURN std_logic_vector;
function "rol" ( l : std_ulogic_vector; r : integer )
     RETURN std_ulogic_vector;

function "ror" ( l  : std_logic_vector;  r : integer )
     RETURN std_logic_vector;
function "ror" ( l  : std_ulogic_vector; r : integer )
     RETURN std_ulogic_vector;


------------------------------------------------------------
-- vector/scalar overloaded logical operators
------------------------------------------------------------
FUNCTION "and"  ( l : std_logic_vector;  r : std_ulogic )
     RETURN std_logic_vector;
FUNCTION "and"  ( l : std_ulogic_vector; r : std_ulogic )
     RETURN std_ulogic_vector;
FUNCTION "and"  ( l : std_ulogic; r : std_logic_vector  )
     RETURN std_logic_vector;
FUNCTION "and"  ( l : std_ulogic; r : std_ulogic_vector )
     RETURN std_ulogic_vector;
FUNCTION "nand" ( l : std_logic_vector;  r : std_ulogic )
     RETURN std_logic_vector;
FUNCTION "nand" ( l : std_ulogic_vector; r : std_ulogic )
     RETURN std_ulogic_vector;
FUNCTION "nand" ( l : std_ulogic; r : std_logic_vector  )
     RETURN std_logic_vector;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic_vector )
     RETURN std_ulogic_vector;
FUNCTION "or"   ( l : std_logic_vector;  r : std_ulogic )
     RETURN std_logic_vector;
FUNCTION "or"   ( l : std_ulogic_vector; r : std_ulogic )
     RETURN std_ulogic_vector;
FUNCTION "or"   ( l : std_ulogic; r : std_logic_vector  )
     RETURN std_logic_vector;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic_vector )
     RETURN std_ulogic_vector;
FUNCTION "nor"  ( l : std_logic_vector;  r : std_ulogic )
     RETURN std_logic_vector;
FUNCTION "nor"  ( l : std_ulogic_vector; r : std_ulogic )
     RETURN std_ulogic_vector;
```

```
FUNCTION "nor"  ( l : std_ulogic; r : std_logic_vector  )
      RETURN std_logic_vector;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic_vector )
      RETURN std_ulogic_vector;
FUNCTION "xor"  ( l : std_logic_vector;  r : std_ulogic )
      RETURN std_logic_vector;
FUNCTION "xor"  ( l : std_ulogic_vector; r : std_ulogic )
      RETURN std_ulogic_vector;
FUNCTION "xor"  ( l : std_ulogic; r : std_logic_vector  )
      RETURN std_logic_vector;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic_vector )
      RETURN std_ulogic_vector;
FUNCTION "xnor" ( l : std_logic_vector;  r : std_ulogic )
      RETURN std_logic_vector;
FUNCTION "xnor" ( l : std_ulogic_vector; r : std_ulogic )
      RETURN std_ulogic_vector;
FUNCTION "xnor" ( l : std_ulogic; r : std_logic_vector  )
      RETURN std_logic_vector;
FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic_vector )
      RETURN std_ulogic_vector;


    -----------------------------------------------------------
    -- vector-reduction functions
    -----------------------------------------------------------
    -- function "and" ( arg  : std_logic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION and_reduce ( arg  : std_logic_vector ) RETURN
          std_ulogic;
    -- function "and" ( arg  : std_ulogic_vector ) RETURN
    --     std_ulogic;  -- %%% New syntax
    FUNCTION and_reduce ( arg  : std_ulogic_vector ) RETURN
          std_ulogic;
    -- function "nand" ( arg  : std_logic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION nand_reduce ( arg : std_logic_vector ) RETURN
          std_ulogic;
    -- function "nand" ( arg  : std_ulogic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION nand_reduce ( arg : std_ulogic_vector ) RETURN
          std_ulogic;
    -- function "or" ( arg  : std_logic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION or_reduce ( arg   : std_logic_vector ) RETURN
          std_ulogic;
    -- function "or" ( arg  : std_ulogic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION or_reduce ( arg   : std_ulogic_vector ) RETURN
          std_ulogic;
    -- function "nor" ( arg  : std_logic_vector ) RETURN
    --     std_ulogic; -- %%% New syntax
    FUNCTION nor_reduce ( arg  : std_logic_vector ) RETURN
          std_ulogic;
```

```
-- function "nor" ( arg  : std_ulogic_vector ) RETURN
--    std_ulogic; -- %%% New syntax
FUNCTION nor_reduce ( arg  : std_ulogic_vector ) RETURN
      std_ulogic;
-- function "xor" ( arg  : std_logic_vector ) RETURN
--    std_ulogic; -- %%% New syntax
FUNCTION xor_reduce ( arg  : std_logic_vector ) RETURN
      std_ulogic;
-- function "xor" ( arg  : std_ulogic_vector ) RETURN
--    std_ulogic; -- %%% New syntax
FUNCTION xor_reduce ( arg  : std_ulogic_vector ) RETURN
      std_ulogic;
-- function "xnor" ( arg  : std_logic_vector ) RETURN
--    std_ulogic; -- %%% New syntax
FUNCTION xnor_reduce ( arg : std_logic_vector ) RETURN
      std_ulogic;
-- function "xnor" ( arg  : std_ulogic_vector ) RETURN
--    std_ulogic; -- %%% New syntax
FUNCTION xnor_reduce ( arg : std_ulogic_vector ) RETURN
      std_ulogic;


---------------------------------------------------------
-- match functions
---------------------------------------------------------
FUNCTION match ( l, r : std_ulogic ) RETURN BOOLEAN;
FUNCTION match ( l, r : std_logic_vector ) RETURN
      BOOLEAN;
FUNCTION match ( l, r : std_ulogic_vector ) RETURN
      BOOLEAN;


-- rtl_synthesis off
---------------------------------------------------------
-- Read and Write functions copied from
-- "std_logic_textio"
---------------------------------------------------------
-- Read and Write procedures for STD_ULOGIC and
-- STD_ULOGIC_VECTOR

procedure READ (L : inout LINE; VALUE : out STD_ULOGIC;
      GOOD : out BOOLEAN);
procedure READ (L : inout LINE; VALUE : out STD_ULOGIC);

procedure READ (L : inout LINE; VALUE : out
      STD_ULOGIC_VECTOR; GOOD : out BOOLEAN);
procedure READ (L : inout LINE; VALUE : out
      STD_ULOGIC_VECTOR);

procedure WRITE (L : inout LINE; VALUE : in STD_ULOGIC;
      JUSTIFIED : in SIDE := RIGHT; FIELD : in
      WIDTH := 0);

procedure WRITE (L : inout LINE; VALUE : in
```

```
        STD_ULOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);

  -- Read and Write procedures for STD_LOGIC_VECTOR

  procedure READ (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR; GOOD : out BOOLEAN);
  procedure READ (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR);

  procedure WRITE (L : inout LINE; VALUE : in
        STD_LOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);

  alias bread is read [line, STD_ULOGIC, BOOLEAN] ;
  alias bread is read [line, STD_ULOGIC] ;
  alias bread is read [line, STD_ULOGIC_VECTOR, BOOLEAN] ;
  alias bread is read [line, STD_ULOGIC_VECTOR] ;
  alias bread is read [line, STD_LOGIC_VECTOR, BOOLEAN] ;
  alias bread is read [line, STD_LOGIC_VECTOR] ;
  alias bwrite is write [line, STD_ULOGIC, side, width] ;
  alias bwrite is write [line, STD_ULOGIC_VECTOR, side,
        width] ;
  alias bwrite is write [line, STD_LOGIC_VECTOR, side,
        width] ;


  -- Read and Write procedures for Hex values

  procedure HREAD (L : inout LINE; VALUE : out
        STD_ULOGIC_VECTOR; GOOD : out BOOLEAN);
  procedure HREAD (L : inout LINE; VALUE : out
        STD_ULOGIC_VECTOR);

  procedure HWRITE (L : inout LINE; VALUE : in
        STD_ULOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);

  procedure HREAD (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR; GOOD : out BOOLEAN);
  procedure HREAD (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR);

  procedure HWRITE (L : inout LINE; VALUE : in
        STD_LOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);


-- Read and Write procedures for Octal values

  procedure OREAD (L : inout LINE; VALUE : out
        STD_ULOGIC_VECTOR; GOOD : out BOOLEAN);
  procedure OREAD (L : inout LINE; VALUE : out
```

```
      STD_ULOGIC_VECTOR);

  procedure OWRITE (L : inout LINE; VALUE : in
        STD_ULOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);

  procedure OREAD (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR; GOOD : out BOOLEAN);
  procedure OREAD (L : inout LINE; VALUE : out
        STD_LOGIC_VECTOR);

  procedure OWRITE (L : inout LINE; VALUE : in
        STD_LOGIC_VECTOR; JUSTIFIED : in SIDE := RIGHT;
        FIELD : in WIDTH := 0);

-- rtl_synthesis off

  function to_string (
    value     : std_ulogic_vector;
    justified : side  := RIGHT;
    field     : width := 0
    ) return string;

  alias to_bstring is to_string [std_ulogic_vector, side,
        width return string];

  function to_hstring (
    value     : std_ulogic_vector;
    justified : side  := RIGHT;
    field     : width := 0
    ) return string;

  function to_ostring (
    value     : std_ulogic_vector;
    justified : side  := RIGHT;
    field     : width := 0
    ) return string ;

  function to_string (
    value     : std_logic_vector;
    justified : side  := RIGHT;
    field     : width := 0
    ) return string;

  alias to_bstring is to_string [std_logic_vector, side,
        width return string];

  function to_hstring (
    value     : std_logic_vector;
    justified : side  := RIGHT;
    field     : width := 0
    ) return string;
```

```
function to_ostring (
  value     : std_logic_vector;
  justified : side  := RIGHT;
  field     : width := 0
  ) return string ;

END package std_logic_1164;
```

E STD_LOGIC TEXTIO Package

This appendix shows the *Std_Logic* TEXTIO package. All designs using this package must use the *LIBRARY IEEE;* and *USE STD_LOGIC TEXTIO;*

```
------------------------------------------------------------
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.  All
-- rights reserved.
--
-- This source file may be used and distributed without
-- restriction  provided that this copyright statement is
-- not removed from the file and that any derivative work
-- contains this copyright notice.
--
--  Package name: STD_LOGIC_TEXTIO
--
--  Purpose: This package overloads the standard TEXTIO
--           procedures READ and WRITE.
--
--  Author: CRC, TS
------------------------------------------------------------
use STD.textio.all;
library IEEE;
use IEEE.std_logic_1164.all;

package STD_LOGIC_TEXTIO is
--synopsys synthesis_off
    -- Read and Write procedures for STD_ULOGIC and
    -- STD_ULOGIC_VECTOR
    procedure READ(L:inout LINE; VALUE:out STD_ULOGIC);
    procedure READ(L:inout LINE; VALUE:out STD_ULOGIC;
          GOOD: out BOOLEAN);
    procedure READ(L:inout LINE; VALUE:out
```

479

```
            STD_ULOGIC_VECTOR);
    procedure READ(L:inout LINE; VALUE:out
            STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC;
            JUSTIFIED:in SIDE := RIGHT; FIELD:in
            WIDTH := 0);
    procedure WRITE(L:inout LINE; VALUE:in
            STD_ULOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
            FIELD:in WIDTH := 0);

    -- Read and Write procedures for STD_LOGIC_VECTOR
    procedure READ(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR);
    procedure READ(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure WRITE(L:inout LINE; VALUE:in
        STD_LOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
        FIELD:in WIDTH := 0);
    -- Read and Write procedures for Hex and Octal values.
    -- The values appear in the file as a series of
    -- characters between 0-F (Hex), or 0-7 (Octal)
    -- respectively.
    procedure HREAD(L:inout LINE; VALUE:out
            STD_ULOGIC_VECTOR);
    procedure HREAD(L:inout LINE; VALUE:out
            STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure HWRITE(L:inout LINE; VALUE:in
            STD_ULOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
            FIELD:in WIDTH := 0);
    procedure HREAD(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR);
    procedure HREAD(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure HWRITE(L:inout LINE; VALUE:in
            STD_LOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
            FIELD:in WIDTH := 0);
    -- Octal
    procedure OREAD(L:inout LINE; VALUE:out
            STD_ULOGIC_VECTOR);
    procedure OREAD(L:inout LINE; VALUE:out
            STD_ULOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure OWRITE(L:inout LINE; VALUE:in
            STD_ULOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
            FIELD:in WIDTH := 0);
    procedure OREAD(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR);
    procedure OREAD(L:inout LINE; VALUE:out
            STD_LOGIC_VECTOR; GOOD: out BOOLEAN);
    procedure OWRITE(L:inout LINE; VALUE:in
            STD_LOGIC_VECTOR; JUSTIFIED:in SIDE := RIGHT;
            FIELD:in WIDTH := 0);
    --synopsys synthesis_on
end STD_LOGIC_TEXTIO;
```

F STD_LOGIC_ARITH Package

This appendix shows the *Std_Logic* Arithmetic package. All designs using this package must use the *LIBRARY IEEE;* and *USE IEEE.std_logic_arith ALL;* statements for making this package and its contents accessible.

```
-----------------------------------------------------------
--                                                       --
--                                                       --
-- Copyright (c) 1990,1991,1992 by Synopsys, Inc.  All   --
-- rights reserved.                                      --
--                                                       --
-- This source file may be used and distributed without --
-- restriction provided that this copyright statement   --
-- is not removed from the file and that any derivative  --
-- work contains this copyright notice.                  --
--                                                       --
--  Package name: STD_LOGIC_ARITH                        --
--                                                       --
--  Purpose:                                             --
--   A set of arithemtic, conversion, and comparison     --
--   functions for SIGNED, UNSIGNED, SMALL_INT,          --
--   INTEGER, STD_ULOGIC, STD_LOGIC, and                 --
--   STD_LOGIC_VECTOR.                                   --
--                                                       --
-----------------------------------------------------------

--------------------------------------------------
-- Attributes added to invoke MTI builtin functions
--------------------------------------------------
```

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

package std_logic_arith is

    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;

    attribute builtin_subprogram: string;

    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    attribute builtin_subprogram of
        "+"[UNSIGNED, UNSIGNED return UNSIGNED]: function
        is "stdarith_plus_uuu";

    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    attribute builtin_subprogram of
        "+"[SIGNED, SIGNED return SIGNED]: function is
        "stdarith_plus_sss";

    function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
    attribute builtin_subprogram of
        "+"[UNSIGNED, SIGNED return SIGNED]: function is
        "stdarith_plus_uss";

    function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
    attribute builtin_subprogram of
        "+"[SIGNED, UNSIGNED return SIGNED]: function is
        "stdarith_plus_sus";

    function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    attribute builtin_subprogram of
        "+"[UNSIGNED, INTEGER return UNSIGNED]: function is
        "stdarith_plus_uiu";

    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    attribute builtin_subprogram of
        "+"[INTEGER, UNSIGNED return UNSIGNED]: function is
        "stdarith_plus_iuu";

    function "+"(L: SIGNED; R: INTEGER) return SIGNED;
    attribute builtin_subprogram of
        "+"[SIGNED, INTEGER return SIGNED]: function is
        "stdarith_plus_sis";

    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    attribute builtin_subprogram of
        "+"[INTEGER, SIGNED return SIGNED]: function is
        "stdarith_plus_iss";

    function "+"(L: UNSIGNED; R: STD_ULOGIC) return
        UNSIGNED;
```

```
attribute builtin_subprogram of
    "+"[UNSIGNED, STD_ULOGIC return UNSIGNED]: function
    is "stdarith_plus_uxu";

function "+"(L: STD_ULOGIC; R: UNSIGNED) return
    UNSIGNED;
attribute builtin_subprogram of
    "+"[STD_ULOGIC, UNSIGNED return UNSIGNED]: function
    is "stdarith_plus_xuu";

function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
attribute builtin_subprogram of
    "+"[SIGNED, STD_ULOGIC return SIGNED]: function is
    "stdarith_plus_sxs";

function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "+"[STD_ULOGIC, SIGNED return SIGNED]: function is
    "stdarith_plus_xss";

function "+"(L: UNSIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[UNSIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_uuu";

function "+"(L: SIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[SIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_sss";

function "+"(L: UNSIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[UNSIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_uss";

function "+"(L: SIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[SIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_sus";

function "+"(L: UNSIGNED; R: INTEGER) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[UNSIGNED, INTEGER return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_uiu";

function "+"(L: INTEGER; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
```

```
    "+"[INTEGER, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_iuu";

function "+"(L: SIGNED; R: INTEGER) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[SIGNED, INTEGER return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_sis";

function "+"(L: INTEGER; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[INTEGER, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_iss";

function "+"(L: UNSIGNED; R: STD_ULOGIC) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[UNSIGNED, STD_ULOGIC return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_uxu";

function "+"(L: STD_ULOGIC; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[STD_ULOGIC, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_xuu";

function "+"(L: SIGNED; R: STD_ULOGIC) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[SIGNED, STD_ULOGIC return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_sxs";

function "+"(L: STD_ULOGIC; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[STD_ULOGIC, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_plus_xss";


function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
attribute builtin_subprogram of
    "-"[UNSIGNED, UNSIGNED return UNSIGNED]: function
    is "stdarith_minus_uuu";

function "-"(L: SIGNED; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[SIGNED, SIGNED return SIGNED]: function is
    "stdarith_minus_sss";

function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[UNSIGNED, SIGNED return SIGNED]: function is
```

```
    "stdarith_minus_uss";

function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[SIGNED, UNSIGNED return SIGNED]: function is
    "stdarith_minus_sus";

function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
attribute builtin_subprogram of
    "-"[UNSIGNED, INTEGER return UNSIGNED]: function is
    "stdarith_minus_uiu";

function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
attribute builtin_subprogram of
    "-"[INTEGER, UNSIGNED return UNSIGNED]: function is
    "stdarith_minus_iuu";

function "-"(L: SIGNED; R: INTEGER) return SIGNED;
attribute builtin_subprogram of
    "-"[SIGNED, INTEGER return SIGNED]: function is
    "stdarith_minus_sis";

function "-"(L: INTEGER; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[INTEGER, SIGNED return SIGNED]: function is
    "stdarith_minus_iss";

function "-"(L: UNSIGNED; R: STD_ULOGIC) return
    UNSIGNED;
attribute builtin_subprogram of
    "-"[UNSIGNED, STD_ULOGIC return UNSIGNED]: function
    is "stdarith_minus_uxu";

function "-"(L: STD_ULOGIC; R: UNSIGNED) return
    UNSIGNED;
attribute builtin_subprogram of
    "-"[STD_ULOGIC, UNSIGNED return UNSIGNED]: function
    is "stdarith_minus_xuu";

function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
attribute builtin_subprogram of
    "-"[SIGNED, STD_ULOGIC return SIGNED]: function is
    "stdarith_minus_sxs";

function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[STD_ULOGIC, SIGNED return SIGNED]: function is
    "stdarith_minus_xss";

function "-"(L: UNSIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[UNSIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
```

```
    function is "stdarith_minus_uuu";

function "-"(L: SIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[SIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_sss";

function "-"(L: UNSIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[UNSIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_uss";

function "-"(L: SIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[SIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_sus";

function "-"(L: UNSIGNED; R: INTEGER) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[UNSIGNED, INTEGER return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_uiu";

function "-"(L: INTEGER; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[INTEGER, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_iuu";

function "-"(L: SIGNED; R: INTEGER) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[SIGNED, INTEGER return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_sis";

function "-"(L: INTEGER; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[INTEGER, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_iss";

function "-"(L: UNSIGNED; R: STD_ULOGIC) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[UNSIGNED, STD_ULOGIC return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_uxu";

function "-"(L: STD_ULOGIC; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
```

```
    "-"[STD_ULOGIC, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_xuu";

function "-"(L: SIGNED; R: STD_ULOGIC) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[SIGNED, STD_ULOGIC return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_sxs";

function "-"(L: STD_ULOGIC; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[STD_ULOGIC, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_minus_xss";

function "+"(L: UNSIGNED) return UNSIGNED;
attribute builtin_subprogram of
    "+"[UNSIGNED return UNSIGNED]: function is
    "stdarith_unary_plus_uu";

function "+"(L: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "+"[SIGNED return SIGNED]: function is
    "stdarith_unary_plus_ss";

function "-"(L: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "-"[SIGNED return SIGNED]: function is
    "stdarith_unary_minus_ss";

function "ABS"(L: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "ABS"[SIGNED return SIGNED]: function is
    "stdarith_abs_ss";


function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[UNSIGNED return STD_LOGIC_VECTOR]: function is
    "stdarith_unary_plus_uu";

function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[SIGNED return STD_LOGIC_VECTOR]: function is
    "stdarith_unary_plus_ss";

function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[SIGNED return STD_LOGIC_VECTOR]: function is
    "stdarith_unary_minus_ss";

function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
```

```
    "ABS"[SIGNED return STD_LOGIC_VECTOR]: function is
    "stdarith_abs_ss";

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
attribute builtin_subprogram of
    "*"[UNSIGNED, UNSIGNED return UNSIGNED]: function
    is "stdarith_mult_uuu";

function "*"(L: SIGNED; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "*"[SIGNED, SIGNED return SIGNED]: function is
    "stdarith_mult_sss";

function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
attribute builtin_subprogram of
    "*"[SIGNED, UNSIGNED return SIGNED]: function is
    "stdarith_mult_sus";

function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;
attribute builtin_subprogram of
    "*"[UNSIGNED, SIGNED return SIGNED]: function is
    "stdarith_mult_uss";


function "*"(L: UNSIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "*"[UNSIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_mult_uuu";

function "*"(L: SIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "*"[SIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_mult_sss";

function "*"(L: SIGNED; R: UNSIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "*"[SIGNED, UNSIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_mult_sus";

function "*"(L: UNSIGNED; R: SIGNED) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "*"[UNSIGNED, SIGNED return STD_LOGIC_VECTOR]:
    function is "stdarith_mult_uss";


function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[UNSIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_lt_uu";
```

```
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_lt_ss";

function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_lt_us";

function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_lt_su";

function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "<"[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_lt_ui";

function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_lt_iu";

function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "<"[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_lt_si";

function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<"[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_lt_is";

function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<="[UNSIGNED, UNSIGNED return BOOLEAN]: function
    is "stdarith_lte_uu";

function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<="[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_lte_ss";

function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<="[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_lte_us";

function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
```

```
    "<="[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_lte_su";

function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "<="[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_lte_ui";

function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<="[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_lte_iu";

function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "<="[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_lte_si";

function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "<="[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_lte_is";


function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[UNSIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_gt_uu";

function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_gt_ss";

function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_gt_us";

function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_gt_su";

function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    ">"[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_gt_ui";

function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_gt_iu";
```

```
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    ">"[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_gt_si";

function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">"[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_gt_is";


function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[UNSIGNED, UNSIGNED return BOOLEAN]: function
    is "stdarith_gte_uu";

function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_gte_ss";

function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_gte_us";

function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_gte_su";

function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    ">="[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_gte_ui";

function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_gte_iu";

function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    ">="[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_gte_si";

function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    ">="[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_gte_is";
```

```
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[UNSIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_eq_uu";

function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_eq_ss";

function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_eq_us";

function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_eq_su";

function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "="[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_eq_ui";

function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_eq_iu";

function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "="[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_eq_si";

function "="(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "="[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_eq_is";


function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "/="[UNSIGNED, UNSIGNED return BOOLEAN]: function
    is "stdarith_neq_uu";

function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "/="[SIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_neq_ss";

function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
```

```
    "/="[UNSIGNED, SIGNED return BOOLEAN]: function is
    "stdarith_neq_us";

function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "/="[SIGNED, UNSIGNED return BOOLEAN]: function is
    "stdarith_neq_su";

function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "/="[UNSIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_neq_ui";

function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "/="[INTEGER, UNSIGNED return BOOLEAN]: function is
    "stdarith_neq_iu";

function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
attribute builtin_subprogram of
    "/="[SIGNED, INTEGER return BOOLEAN]: function is
    "stdarith_neq_si";

function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;
attribute builtin_subprogram of
    "/="[INTEGER, SIGNED return BOOLEAN]: function is
    "stdarith_neq_is";


function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return
    UNSIGNED;
attribute builtin_subprogram of
    SHL[UNSIGNED, UNSIGNED return UNSIGNED]: function
    is "stdarith_shl_uuu";

function SHL(ARG: SIGNED; COUNT: UNSIGNED) return
    SIGNED;
attribute builtin_subprogram of
    SHL[SIGNED, UNSIGNED return SIGNED]: function is
    "stdarith_shl_sus";

function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return
    UNSIGNED;
attribute builtin_subprogram of
    SHR[UNSIGNED, UNSIGNED return UNSIGNED]: function
    is "stdarith_shr_uuu";

function SHR(ARG: SIGNED; COUNT: UNSIGNED) return
    SIGNED;
attribute builtin_subprogram of
    SHR[SIGNED, UNSIGNED return SIGNED]: function is
    "stdarith_shr_sus";
```

```
    function CONV_INTEGER(ARG: INTEGER) return INTEGER;
    attribute builtin_subprogram of
        CONV_INTEGER[INTEGER return INTEGER]: function is
        "stdarith_conv_integer_ii";

    function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
    attribute builtin_subprogram of
        CONV_INTEGER[UNSIGNED return INTEGER]: function is
        "stdarith_conv_integer_ui2";
-----------------------------------------------------------
--  If desired, you may select an optional implementation
--  for CONV_INTEGER(UNSIGNED) by changing the value of the
--  attribute:
--      stdarith_conv_integer_ui implements the original
--        CONV_INTEGER(UNSIGNED) VHDL which generates an
--        error if the argument is larger than 31 bits.
--      stdarith_conv_integer_ui2 allows 32 bits. It
--        generates a warning if the argument is 32 bits
--        and the MSB is not zero.
--      The default is stdarith_conv_integer_ui2.
--
--    attribute builtin_subprogram of
--      CONV_INTEGER[UNSIGNED return INTEGER]: function
--      is "stdarith_conv_integer_ui";

    function CONV_INTEGER(ARG: SIGNED) return INTEGER;
    attribute builtin_subprogram of
        CONV_INTEGER[SIGNED return INTEGER]: function is
        "stdarith_conv_integer_si";

    function CONV_INTEGER(ARG: STD_ULOGIC) return
        SMALL_INT;
    attribute builtin_subprogram of
        CONV_INTEGER[STD_ULOGIC return SMALL_INT]: function
        is "stdarith_conv_integer_xz";


    function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER)
        return UNSIGNED;
    attribute builtin_subprogram of
        CONV_UNSIGNED [INTEGER, INTEGER return UNSIGNED]:
        function is "stdarith_conv_unsigned_iu";

    function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER)
        return UNSIGNED;
    attribute builtin_subprogram of
        CONV_UNSIGNED [UNSIGNED, INTEGER return UNSIGNED]:
        function is "stdarith_conv_unsigned_uu";

    function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER)
        return UNSIGNED;
    attribute builtin_subprogram of
```

```
    CONV_UNSIGNED [SIGNED, INTEGER return UNSIGNED]:
    function is "stdarith_conv_unsigned_su";

function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER)
    return UNSIGNED;
attribute builtin_subprogram of
    CONV_UNSIGNED [STD_ULOGIC, INTEGER return
    UNSIGNED]: function is "stdarith_conv_unsigned_xu";


function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER)
    return SIGNED;
attribute builtin_subprogram of
    CONV_SIGNED [INTEGER, INTEGER return SIGNED]:
    function is "stdarith_conv_signed_is";

function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER)
    return SIGNED;
attribute builtin_subprogram of
    CONV_SIGNED [UNSIGNED, INTEGER return SIGNED]:
    function is "stdarith_conv_signed_us";

function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return
    SIGNED;
attribute builtin_subprogram of
    CONV_SIGNED [SIGNED, INTEGER return SIGNED]:
    function is "stdarith_conv_signed_ss";

function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER)
    return SIGNED;
attribute builtin_subprogram of
    CONV_SIGNED [STD_ULOGIC, INTEGER return SIGNED]:
    function is "stdarith_conv_signed_xs";



function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE:
    INTEGER) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    CONV_STD_LOGIC_VECTOR [INTEGER, INTEGER return
    STD_LOGIC_VECTOR]: function is
    "stdarith_conv_slv_iv";

function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE:
    INTEGER) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    CONV_STD_LOGIC_VECTOR [UNSIGNED, INTEGER return
    STD_LOGIC_VECTOR]: function is
    "stdarith_conv_slv_uv";

function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE:
    INTEGER) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
```

```
        CONV_STD_LOGIC_VECTOR [SIGNED, INTEGER return
        STD_LOGIC_VECTOR]: function is
        "stdarith_conv_slv_sv";

    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE:
        INTEGER) return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        CONV_STD_LOGIC_VECTOR [STD_ULOGIC, INTEGER return
        STD_LOGIC_VECTOR]: function is
        "stdarith_conv_slv_xv";

    -- zero extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        EXT [STD_LOGIC_VECTOR, INTEGER return
        STD_LOGIC_VECTOR]: function is
        "stdarith_zeroextend_vv";


    -- sign extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        SXT [STD_LOGIC_VECTOR, INTEGER return
        STD_LOGIC_VECTOR]: function is
        "stdarith_signextend_vv";


end Std_logic_arith;
```

G STD_LOGIC_SIGNED

This appendix shows the *Std_Logic* Signed package. All designs using this package must use the *LIBRARY IEEE;* and *USE IEEE.std_logic_signed.ALL;* statements for making this package and its contents accessible.

```
----------------------------------------------------------
--                                                      --
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.     --
--                              All rights reserved.    --
--                                                      --
-- This source file may be used and distributed without --
-- restriction provided that this copyright statement   --
-- is not removed from the file and that any derivative --
-- work contains this copyright notice.                 --
--                                                      --
--  Package name: STD_LOGIC_SIGNED                      --
--                                                      --
--                                                      --
--      Date:        09/11/91 KN                        --
--                   10/08/92 AMT change std_ulogic to  --
--                   signed std_logic                   --
--                   10/28/92 AMT added signed          --
--                   functions, -, ABS                  --
--                                                      --
-- Purpose:                                             --
--     A set of signed arithemtic, conversion,          --
--       and comparision functions for STD_LOGIC_VECTOR.--
--                                                      --
-- Note:    Comparision of same length std_logic_vector --
--           is defined in the LRM.  The interpretation  --
--           is for unsigned vectors. This package       --
```

497

```
--           will "overload" that definition.          --
--                                                      --
----------------------------------------------------------

----------------------------------------------------
-- Attributes added to invoke MTI builtin functions
----------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_SIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_sss";

    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, INTEGER return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_sis";

    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[INTEGER, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_iss";

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, STD_LOGIC return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_sxs";

    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_xss";


    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "-"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]:function is "stdarith_minus_sss";

    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
```

```
attribute builtin_subprogram of
    "-"[STD_LOGIC_VECTOR, INTEGER return
    STD_LOGIC_VECTOR]:function is "stdarith_minus_sis";

function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[INTEGER, STD_LOGIC_VECTOR return
    STD_LOGIC_VECTOR]:function is "stdarith_minus_iss";

function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[STD_LOGIC_VECTOR, STD_LOGIC return
    STD_LOGIC_VECTOR]:function is "stdarith_minus_sxs";

function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[STD_LOGIC, STD_LOGIC_VECTOR return
    STD_LOGIC_VECTOR]:function is "stdarith_minus_xss";


function "+"(L: STD_LOGIC_VECTOR) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "+"[STD_LOGIC_VECTOR return STD_LOGIC_VECTOR]:
    function is "stdarith_unary_plus_ss";

function "-"(L: STD_LOGIC_VECTOR) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "-"[STD_LOGIC_VECTOR return STD_LOGIC_VECTOR]:
    function is "stdarith_unary_minus_ss";

function "ABS"(L: STD_LOGIC_VECTOR) return
    STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "ABS"[STD_LOGIC_VECTOR return STD_LOGIC_VECTOR]:
    function is "stdarith_abs_ss";


function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    "*"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    STD_LOGIC_VECTOR]: function is "stdarith_mult_sss";


function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
```

```
    "<"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_lt_ss";

function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "<"[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_lt_si";

function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "<"[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_lt_is";


function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "<="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_lte_ss";

function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "<="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_lte_si";

function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "<="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_lte_is";


function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    ">"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_gt_ss";

function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    ">"[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_gt_si";

function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    ">"[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_gt_is";
```

```
function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    ">="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_gte_ss";

function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    ">="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_gte_si";

function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    ">="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_gte_is";


function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_eq_ss";

function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_eq_si";

function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_eq_is";


function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "/="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_neq_ss";

function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "/="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_neq_si";

function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
```

```
    attribute builtin_subprogram of
        "/="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
        function is "stdarith_neq_is";

    function SHL(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        SHL[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_shl_sus";

    function SHR(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        SHR[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_shr_sus";

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return
        INTEGER;
    attribute builtin_subprogram of
        CONV_INTEGER[STD_LOGIC_VECTOR return INTEGER]:
        function is "stdarith_conv_integer_si";


-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE:
--        INTEGER) return STD_LOGIC_VECTOR;

end STD_LOGIC_SIGNED;
```

H STD_LOGIC_UNSIGNED

This appendix shows the *Std_Logic* Unsigned package. All designs using this package must use the *LIBRARY IEEE;* and *USE IEEE.std_logic_unsigned.ALL;* statements for making this package and its contents accessible.

```
-----------------------------------------------------------
--                                                       --
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.      --
--                          All rights reserved.         --
--                                                       --
-- This source file may be used and distributed without  --
-- restriction provided that this copyright statement    --
-- is not removed from the file and that any derivative  --
-- work contains this copyright notice.                  --
--                                                       --
--  Package name: STD_LOGIC_UNSIGNED                     --
--                                                       --
--                                                       --
--      Date:        09/11/92 KN                         --
--                   10/08/92 AMT                        --
--                                                       --
-- Purpose:                                              --
--    A set of unsigned arithmetic, conversion,          --
--      and comparison functions for STD_LOGIC_VECTOR.   --
--                                                       --
-- Note:    Comparision of same length discrete arrays   --
--          is defined in the LRM. This package          --
--          will "overload" that definition.             --
--                                                       --
-----------------------------------------------------------
```

```
-------------------------------------------------
-- Attributes added to invoke MTI builtin functions
-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_UNSIGNED is

    attribute builtin_subprogram: string;

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_uuu";

    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, INTEGER return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_uiu";

    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[INTEGER, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_iuu";

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC_VECTOR, STD_LOGIC return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_uxu";

    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "+"[STD_LOGIC, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]: function is "stdarith_plus_xuu";


    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
        return STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "-"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
        STD_LOGIC_VECTOR]:function is "stdarith_minus_uuu";

    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    attribute builtin_subprogram of
        "-"[STD_LOGIC_VECTOR, INTEGER return
```

```
     STD_LOGIC_VECTOR]:function is "stdarith_minus_uiu";

function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return
     STD_LOGIC_VECTOR;
attribute builtin_subprogram of
     "-"[INTEGER, STD_LOGIC_VECTOR return
     STD_LOGIC_VECTOR]:function is "stdarith_minus_iuu";

function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
     STD_LOGIC_VECTOR;
attribute builtin_subprogram of
     "-"[STD_LOGIC_VECTOR, STD_LOGIC return
     STD_LOGIC_VECTOR]:function is "stdarith_minus_uxu";

function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
     STD_LOGIC_VECTOR;
attribute builtin_subprogram of
     "-"[STD_LOGIC, STD_LOGIC_VECTOR return
     STD_LOGIC_VECTOR]:function is "stdarith_minus_xuu";


function "+"(L: STD_LOGIC_VECTOR) return
     STD_LOGIC_VECTOR;
attribute builtin_subprogram of
     "+"[STD_LOGIC_VECTOR return STD_LOGIC_VECTOR]:
     function is "stdarith_unary_plus_uu";


function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
     return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
     "*"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
     STD_LOGIC_VECTOR]: function is "stdarith_mult_uuu";


function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
     return BOOLEAN;
attribute builtin_subprogram of
     "<"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
     BOOLEAN]: function is "stdarith_lt_uu";

function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return
     BOOLEAN;
attribute builtin_subprogram of
     "<"[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
     function is "stdarith_lt_ui";

function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return
     BOOLEAN;
attribute builtin_subprogram of
     "<"[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
     function is "stdarith_lt_iu";
```

```
function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "<="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_lte_uu";

function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "<="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_lte_ui";

function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "<="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_lte_iu";


function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    ">"[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_gt_uu";

function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    ">"[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_gt_ui";

function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    ">"[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_gt_iu";


function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    ">="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_gte_uu";

function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    ">="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_gte_ui";

function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
```

```
attribute builtin_subprogram of
    ">="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_gte_iu";


function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_eq_uu";

function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_eq_ui";

function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_eq_iu";


function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
    return BOOLEAN;
attribute builtin_subprogram of
    "/="[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    BOOLEAN]: function is "stdarith_neq_uu";

function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return
    BOOLEAN;
attribute builtin_subprogram of
    "/="[STD_LOGIC_VECTOR, INTEGER return BOOLEAN]:
    function is "stdarith_neq_ui";

function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return
    BOOLEAN;
attribute builtin_subprogram of
    "/="[INTEGER, STD_LOGIC_VECTOR return BOOLEAN]:
    function is "stdarith_neq_iu";

function SHL(ARG:STD_LOGIC_VECTOR;COUNT:
    STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    SHL[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    STD_LOGIC_VECTOR]: function is "stdarith_shl_uuu";

function SHR(ARG:STD_LOGIC_VECTOR;COUNT:
    STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
attribute builtin_subprogram of
    SHR[STD_LOGIC_VECTOR, STD_LOGIC_VECTOR return
    STD_LOGIC_VECTOR]: function is "stdarith_shr_uuu";
```

```
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return
        INTEGER;
    attribute builtin_subprogram of
        CONV_INTEGER[STD_LOGIC_VECTOR return INTEGER]:
        function is "stdarith_conv_integer_ui2";
-----------------------------------------------------------
-- If desired, you may select an optional implementation
-- for CONV_INTEGER(STD_LOGIC_VECTOR)
-- by changing the value of the attribute:
--      stdarith_conv_integer_ui implements the original
--      CONV_INTEGER(STD_LOGIC_VECTOR) VHDL which
--          generates an error if the argument is larger
--          than 31 bits.
--      stdarith_conv_integer_ui2 allows 32 bits. It
--      generates a warning if the
--          argument is 32 bits and the MSB is not zero.
--      The default is stdarith_conv_integer_ui2.
--
--    attribute builtin_subprogram of
--        CONV_INTEGER[STD_LOGIC_VECTOR return INTEGER]:
--        function is "stdarith_conv_integer_ui";
-----------------------------------------------------------

-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE:
--    INTEGER) return STD_LOGIC_VECTOR;

end STD_LOGIC_UNSIGNED;
```

▌ math_real Package

This appendix shows the *math_real* package. All designs using this package must use the *LIBRARY IEEE;* and *USE IEEE.math_real.ALL;* statements for making this package and its contents accessible.

```
----------------------------------------------------------
--
-- Copyright 1996 by IEEE. All rights reserved.
--
-- This source file is an essential part of IEEE Std
-- 1076.2-1996, IEEE Standard VHDL Mathematical Packages.
-- This source file may not be copied, sold, or included
-- with software that is sold without written permission
-- from the IEEE Standards Department. This source file may
-- be used to implement this standard and may be
-- distributed in compiled form in any manner so long as
-- the compiled form does not allow direct decompilation of
-- the original source file.
-- This source file may be copied for individual use be
-- tween licensed users. This source file is provided on an
-- AS IS basis. The IEEE disclaims ANY WARRANTY EXPRESS OR
-- IMPLIED INCLUDING ANY WARRANTY OF MERCHANTABILITY AND
-- FITNESS FOR USE FOR A PARTICULAR PURPOSE. The user of
-- the source file shall indemnify and hold IEEE harmless
-- from any damages or liability arising out of the use
-- thereof.
--
-- Title:       Standard VHDL Mathematical Packages (IEEE
--              Std 1076.2-1996, MATH_REAL)
--
-- Library:     This package shall be compiled into a
```

```
--                 library symbolically named IEEE.
--
-- Developers:   IEEE DASC VHDL Mathematical Packages
--               Working Group
--
-- Purpose:      This package defines a standard for
--               designers to use in describing VHDL models
--               that make use of common REAL constants
--               and common REAL elementary mathematical
--               functions.
--
-- Limitation:   The values generated by the functions in
--               this package may vary from platform to
--               platform, and the precision of results
--               is only guaranteed to be the minimum
--               required by IEEE Std 1076-1993.
--
-- Notes:
--                 No declarations or definitions shall be
--                 included in, or excluded from,this package.
--                 The "package declaration" defines the
--                 types, subtypes, and declarations of
--                 MATH_REAL.
--                 The standard mathematical definition and
--                 conventional meaning of the mathematical
--                 functions that are part of this standard
--                 represent the formal semantics of the
--                 implementation of the MATH_REAL package
--                 declaration.  The purpose of the MATH_REAL
--                 package body is to provide a guideline for
--                 implementations to verify their
--                 implementation of MATH_REAL.  Tool
--                 developers may choose to implement the
--                 package body in the most efficient
--                 manner available to them.
--
-- ---------------------------------------------------------
-- Version   : 1.5
-- Date      : 24 July 1996
-- ---------------------------------------------------------
package MATH_REAL is
    constant CopyRightNotice: STRING
      := "Copyright 1996 IEEE. All rights reserved.";

    --
    -- Constant Definitions
    --
    constant  MATH_E : REAL := 2.71828_18284_59045_23536;
                                          -- Value of e
    constant  MATH_1_OVER_E : REAL :=
        0.36787_94411_71442_32160;
                                        -- Value of 1/e
    constant  MATH_PI : REAL := 3.14159_26535_89793_23846;
```

```
                                              -- Value of pi
constant  MATH_2_PI : REAL :=
    6.28318_53071_79586_47693;
                                        -- Value of 2*pi
constant  MATH_1_OVER_PI : REAL :=
    0.31830_98861_83790_67154;
                                        -- Value of 1/pi
constant  MATH_PI_OVER_2 : REAL :=
    1.57079_63267_94896_61923;
                                        -- Value of pi/2
constant  MATH_PI_OVER_3 : REAL :=
    1.04719_75511_96597_74615;
                                        -- Value of pi/3
constant  MATH_PI_OVER_4 : REAL :=
    0.78539_81633_97448_30962;
                                        -- Value of pi/4
constant  MATH_3_PI_OVER_2 : REAL :=
    4.71238_89803_84689_85769;
                                         -- Value 3*pi/2
constant  MATH_LOG_OF_2 : REAL :=
    0.69314_71805_59945_30942;
                                      -- Natural log of 2
constant  MATH_LOG_OF_10 : REAL :=
    2.30258_50929_94045_68402;
                                     -- Natural log of 10
constant  MATH_LOG2_OF_E : REAL :=
    1.44269_50408_88963_4074;
                                       -- Log base 2 of e
constant  MATH_LOG10_OF_E: REAL :=
    0.43429_44819_03251_82765;
                                      -- Log base 10 of e
constant  MATH_SQRT_2: REAL :=
    1.41421_35623_73095_04880;
                                      -- square root of 2
constant  MATH_1_OVER_SQRT_2: REAL :=
    0.70710_67811_86547_52440;
                                    -- square root of 1/2
constant  MATH_SQRT_PI: REAL :=
    1.77245_38509_05516_02730;
                                     -- square root of pi
constant  MATH_DEG_TO_RAD: REAL :=
    0.01745_32925_19943_29577;
              -- Conversion factor from degree to radian
constant  MATH_RAD_TO_DEG: REAL :=
    57.29577_95130_82320_87680;
              -- Conversion factor from radian to degree

--
-- Function Declarations
--
function SIGN (X: in REAL ) return REAL;
    -- Purpose:
    --          Returns 1.0 if X > 0.0; 0.0 if X = 0.0;
```

```
     --          1.0 if X < 0.0
     -- Special values:
     --          None
     -- Domain:
     --          X in REAL
     -- Error conditions:
     --          None
     -- Range:
     --          ABS(SIGN(X)) <= 1.0
     -- Notes:
     --          None

 function CEIL (X : in REAL ) return REAL;
     -- Purpose:
     --          Returns smallest INTEGER value (as REAL)
     --          not less than X
     -- Special values:
     --          None
     -- Domain:
     --          X in REAL
     -- Error conditions:
     --          None
     -- Range:
     --          CEIL(X) is mathematically unbounded
     -- Notes:
     --          a) Implementations have to support at
     --             least the domain
     --                 ABS(X) < REAL(INTEGER'HIGH)

 function FLOOR (X : in REAL ) return REAL;
     -- Purpose:
     --          Returns largest INTEGER value (as REAL)
     --          not greater than X
     -- Special values:
     --          FLOOR(0.0) = 0.0
     -- Domain:
     --          X in REAL
     -- Error conditions:
     --          None
     -- Range:
     --          FLOOR(X) is mathematically unbounded
     -- Notes:
     --          a) Implementations have to support at
     --             least the domain
     --                 ABS(X) < REAL(INTEGER'HIGH)

 function ROUND (X : in REAL ) return REAL;
     -- Purpose:
     --          Rounds X to the nearest integer value
     --          (as real). If X is halfway between two
     --          integers, rounding is away from 0.0
     -- Special values:
     --          ROUND(0.0) = 0.0
```

```
    -- Domain:
    --          X in REAL
    -- Error conditions:
    --          None
    -- Range:
    --          ROUND(X) is mathematically unbounded
    -- Notes:
    --          a) Implementations have to support at
    --             least the domain
    --                 ABS(X) < REAL(INTEGER'HIGH)

function TRUNC (X : in REAL ) return REAL;
    -- Purpose:
    --          Truncates X towards 0.0 and returns
    --          truncated value
    -- Special values:
    --          TRUNC(0.0) = 0.0
    -- Domain:
    --          X in REAL
    -- Error conditions:
    --          None
    -- Range:
    --          TRUNC(X) is mathematically unbounded
    -- Notes:
    --          a) Implementations have to support at
    --             least the domain
    --                 ABS(X) < REAL(INTEGER'HIGH)

function "MOD" (X, Y: in REAL ) return REAL;
    -- Purpose:
    --          Returns floating point modulus of X/Y,
    --          with the same sign as Y, and absolute
    --          value less than the absolute value of Y,
    --          and for some INTEGER value N the result
    --          satisfies the relation
    --          X = Y*N + MOD(X,Y)
    -- Special values:
    --          None
    -- Domain:
    --          X in REAL; Y in REAL and Y /= 0.0
    -- Error conditions:
    --          Error if Y = 0.0
    -- Range:
    --          ABS(MOD(X,Y)) < ABS(Y)
    -- Notes:
    --          None

function REALMAX (X, Y : in REAL ) return REAL;
    -- Purpose:
    --          Returns the algebraically larger of X
    --          and Y
    -- Special values:
    --          REALMAX(X,Y) = X when X = Y
```

```
        -- Domain:
        --          X in REAL; Y in REAL
        -- Error conditions:
        --          None
        -- Range:
        --          REALMAX(X,Y) is mathematically unbounded
        -- Notes:
        --          None

    function REALMIN (X, Y : in REAL ) return REAL;
        -- Purpose:
        --          Returns the algebraically smaller of X
        --          and Y
        -- Special values:
        --          REALMIN(X,Y) = X when X = Y
        -- Domain:
        --          X in REAL; Y in REAL
        -- Error conditions:
        --          None
        -- Range:
        --          REALMIN(X,Y) is mathematically unbounded
        -- Notes:
        --          None

    procedure UNIFORM(variable SEED1,SEED2:inout POSITIVE;
        variable X:out REAL);
        -- Purpose:
        --          Returns, in X, a pseudo-random number
        --          with uniform distribution in the open
        --          interval (0.0, 1.0).
        -- Special values:
        --          None
        -- Domain:
        --          1 <= SEED1 <= 2147483562;
        --          1 <= SEED2 <= 2147483398
        -- Error conditions:
        --           Error if SEED1 or SEED2 outside of valid
domain
        -- Range:
        --          0.0 < X < 1.0
        -- Notes:
        --          a) The semantics for this function are
        --             described by the algorithm published
        --             by Pierre L'Ecuyer in "Communications
        --             of the ACM," vol. 31, no. 6, June
        --             1988, pp. 742-774. The algorithm is
        --             based on the combination of two
        --             multiplicative linear congruential
        --             generators for 32-bit platforms.
        --
        --          b) Before the first call to UNIFORM, the
        --             seed values (SEED1, SEED2) have to be
        --             initialized to values in the range
```

```
     --             [1, 2147483562] and [1, 2147483398]
     --             respectively.  The seed values are
     --             modified after each call to UNIFORM.
     --
     --          c) This random number generator is port
     --             able for 32-bit computers, and it has
     --             a period of ~2.30584*(10**18) for
     --             each set of seed values.
     --
     --          d) For information on spectral tests for
     --             the algorithm, refer to the L'Ecuyer
     --             article.

  function SQRT (X : in REAL ) return REAL;
     -- Purpose:
     --         Returns square root of X
     -- Special values:
     --         SQRT(0.0) = 0.0
     --         SQRT(1.0) = 1.0
     -- Domain:
     --         X >= 0.0
     -- Error conditions:
     --         Error if X < 0.0
     -- Range:
     --         SQRT(X) >= 0.0
     -- Notes:
     --          a) The upper bound of the reachable
     --             range of SQRT is approximately given
     --             by:
     --                 SQRT(X) <= SQRT(REAL'HIGH)

  function CBRT (X : in REAL ) return REAL;
     -- Purpose:
     --         Returns cube root of X
     -- Special values:
     --         CBRT(0.0) = 0.0
     --         CBRT(1.0) = 1.0
     --         CBRT(-1.0) = -1.0
     -- Domain:
     --         X in REAL
     -- Error conditions:
     --         None
     -- Range:
     --         CBRT(X) is mathematically unbounded
     -- Notes:
     --          a) The reachable range of CBRT is
     --             approximately given by:
     --                 ABS(CBRT(X)) <= CBRT(REAL'HIGH)

  function "**" (X : in INTEGER; Y : in REAL) return
     REAL;
     -- Purpose:
     --         Returns Y power of X ==>  X**Y
```

```
      -- Special values:
      --         X**0.0 = 1.0; X /= 0
      --         0**Y = 0.0; Y > 0.0
      --         X**1.0 = REAL(X); X >= 0
      --         1**Y = 1.0
      -- Domain:
      --         X > 0
      --         X = 0 for Y > 0.0
      --         X < 0 for Y = 0.0
      -- Error conditions:
      --         Error if X < 0 and Y /= 0.0
      --         Error if X = 0 and Y <= 0.0
      -- Range:
      --         X**Y >= 0.0
      -- Notes:
      --         a) The upper bound of the reachable
      --            range for "**" is approximately given
      --            by:
      --               X**Y <= REAL'HIGH

function "**" (X : in REAL; Y : in REAL) return REAL;
      -- Purpose:
      --         Returns Y power of X ==>  X**Y
      -- Special values:
      --         X**0.0 = 1.0; X /= 0.0
      --         0.0**Y = 0.0; Y > 0.0
      --         X**1.0 = X; X >= 0.0
      --         1.0**Y = 1.0
      -- Domain:
      --         X > 0.0
      --         X = 0.0 for Y > 0.0
      --         X < 0.0 for Y = 0.0
      -- Error conditions:
      --         Error if X < 0.0 and Y /= 0.0
      --         Error if X = 0.0 and Y <= 0.0
      -- Range:
      --         X**Y >= 0.0
      -- Notes:
      --         a) The upper bound of the reachable
      --            range for "**" is approximately given
      --            by:
      --               X**Y <= REAL'HIGH

function EXP (X : in REAL ) return REAL;
      -- Purpose:
      --         Returns e**X; where e = MATH_E
      -- Special values:
      --         EXP(0.0) = 1.0
      --         EXP(1.0) = MATH_E
      --         EXP(-1.0) = MATH_1_OVER_E
      --         EXP(X) = 0.0 for X <= -LOG(REAL'HIGH)
      -- Domain:
      --         X in REAL such that EXP(X) <= REAL'HIGH
```

```
    -- Error conditions:
    --          Error if X > LOG(REAL'HIGH)
    -- Range:
    --          EXP(X) >= 0.0
    -- Notes:
    --          a) The usable domain of EXP is approxi
    --            mately given by:
    --                X <= LOG(REAL'HIGH)

function LOG (X : in REAL ) return REAL;
    -- Purpose:
    --          Returns natural logarithm of X
    -- Special values:
    --          LOG(1.0) = 0.0
    --          LOG(MATH_E) = 1.0
    -- Domain:
    --          X > 0.0
    -- Error conditions:
    --          Error if X <= 0.0
    -- Range:
    --          LOG(X) is mathematically unbounded
    -- Notes:
    --          a) The reachable range of LOG is ap
    --            proximately given by:
    --                LOG(0+) <= LOG(X) <= LOG(REAL'HIGH)

function LOG2 (X : in REAL ) return REAL;
    -- Purpose:
    --          Returns logarithm base 2 of X
    -- Special values:
    --          LOG2(1.0) = 0.0
    --          LOG2(2.0) = 1.0
    -- Domain:
    --          X > 0.0
    -- Error conditions:
    --          Error if X <= 0.0
    -- Range:
    --          LOG2(X) is mathematically unbounded
    -- Notes:
    --          a)The reachable range of LOG2 is
    --            approximately given by:
    --             LOG2(0+) <= LOG2(X) <= LOG2(REAL'HIGH)

function LOG10 (X : in REAL ) return REAL;
    -- Purpose:
    --          Returns logarithm base 10 of X
    -- Special values:
    --          LOG10(1.0) = 0.0
    --          LOG10(10.0) = 1.0
    -- Domain:
    --          X > 0.0
    -- Error conditions:
    --          Error if X <= 0.0
```

```
    -- Range:
    --           LOG10(X) is mathematically unbounded
    -- Notes:
    --      a) The reachable range of LOG10 is
    --           approximately given by:
    --           LOG10(0+) <= LOG10(X) <= LOG10(REAL'HIGH)

function LOG (X: in REAL; BASE: in REAL) return REAL;
    -- Purpose:
    --           Returns logarithm base BASE of X
    -- Special values:
    --           LOG(1.0, BASE) = 0.0
    --           LOG(BASE, BASE) = 1.0
    -- Domain:
    --           X > 0.0
    --           BASE > 0.0
    --           BASE /= 1.0
    -- Error conditions:
    --           Error if X <= 0.0
    --           Error if BASE <= 0.0
    --           Error if BASE = 1.0
    -- Range:
    --           LOG(X, BASE) is mathematically unbounded
    -- Notes:
    --           a) When BASE > 1.0, the reachable range
    --              of LOG is approximately given by:
    --                  LOG(0+, BASE) <= LOG(X, BASE) <=
    --                  LOG(REAL'HIGH, BASE)
    --           b) When 0.0 < BASE < 1.0, the reachable
    --              range of LOG is approximately given
    --              by:
    --               LOG(REAL'HIGH, BASE) <= LOG(X, BASE)
    --               <= LOG(0+, BASE)

function  SIN (X : in REAL ) return REAL;
    -- Purpose:
    --           Returns sine of X; X in radians
    -- Special values:
    --           SIN(X) = 0.0 for X = k*MATH_PI, where k
    --           is an INTEGER SIN(X) = 1.0 for X =
    --           (4*k+1)*MATH_PI_OVER_2, where k is an
    --           INTEGER
    --           SIN(X) = -1.0 for X =
    --           (4*k+3)*MATH_PI_OVER_2, where k is an
    --           INTEGER
    -- Domain:
    --           X in REAL
    -- Error conditions:
    --           None
    -- Range:
    --           ABS(SIN(X)) <= 1.0
    -- Notes:
    --           a) For larger values of ABS(X), degraded
```

```
    --              accuracy is allowed.

function  COS ( X : in REAL ) return REAL;
    -- Purpose:
    --              Returns cosine of X; X in radians
    -- Special values:
    --              COS(X) = 0.0 for X =
    --              (2*k+1)*MATH_PI_OVER_2, where k is an
    --              INTEGER
    --              COS(X) = 1.0 for X = (2*k)*MATH_PI,
    --              where k is an INTEGER
    --              COS(X) = -1.0 for X = (2*k+1)*MATH_PI,
    --              where k is an INTEGER
    -- Domain:
    --              X in REAL
    -- Error conditions:
    --              None
    -- Range:
    --              ABS(COS(X)) <= 1.0
    -- Notes:
    --              a) For larger values of ABS(X), degraded
    --                 accuracy is allowed.

function  TAN (X : in REAL ) return REAL;
    -- Purpose:
    --              Returns tangent of X; X in radians
    -- Special values:
    --              TAN(X) = 0.0 for X = k*MATH_PI, where k
    --              is an INTEGER
    -- Domain:
    --              X in REAL and
    --              X /= (2*k+1)*MATH_PI_OVER_2, where k is
    --              an INTEGER
    -- Error conditions:
    --              Error if X = ((2*k+1) * MATH_PI_OVER_2),
    --              where k is an INTEGER
    -- Range:
    --              TAN(X) is mathematically unbounded
    -- Notes:
    --              a) For larger values of ABS(X), degraded
    --                 accuracy is allowed.

function  ARCSIN (X : in REAL ) return REAL;
    -- Purpose:
    --              Returns inverse sine of X
    -- Special values:
    --              ARCSIN(0.0) = 0.0
    --              ARCSIN(1.0) = MATH_PI_OVER_2
    --              ARCSIN(-1.0) = -MATH_PI_OVER_2
    -- Domain:
    --              ABS(X) <= 1.0
    -- Error conditions:
    --              Error if ABS(X) > 1.0
```

```
    -- Range:
    --          ABS(ARCSIN(X) <= MATH_PI_OVER_2
    -- Notes:
    --          None

function  ARCCOS (X : in REAL ) return REAL;
    -- Purpose:
    --          Returns inverse cosine of X
    -- Special values:
    --          ARCCOS(1.0) = 0.0
    --          ARCCOS(0.0) = MATH_PI_OVER_2
    --          ARCCOS(-1.0) = MATH_PI
    -- Domain:
    --          ABS(X) <= 1.0
    -- Error conditions:
    --          Error if ABS(X) > 1.0
    -- Range:
    --          0.0 <= ARCCOS(X) <= MATH_PI
    -- Notes:
    --          None

function  ARCTAN (Y : in REAL) return REAL;
    -- Purpose:
    --          Returns the value of the angle in
    --          radians of the point
    --          (1.0, Y), which is in rectangular
    --          coordinates
    -- Special values:
    --          ARCTAN(0.0) = 0.0
    -- Domain:
    --          Y in REAL
    -- Error conditions:
    --          None
    -- Range:
    --          ABS(ARCTAN(Y)) <= MATH_PI_OVER_2
    -- Notes:
    --          None

function  ARCTAN (Y : in REAL; X : in REAL) return
    REAL;
    -- Purpose:
    --          Returns the principal value of the angle
    --          in radians of the point (X, Y), which is
    --          in rectangular coordinates
    -- Special values:
    --          ARCTAN(0.0, X) = 0.0 if X > 0.0
    --          ARCTAN(0.0, X) = MATH_PI if X < 0.0
    --          ARCTAN(Y, 0.0)=MATH_PI_OVER_2 if Y > 0.0
    --          ARCTAN(Y,0.0)=-MATH_PI_OVER_2 if Y < 0.0
    -- Domain:
    --          Y in REAL
    --          X in REAL, X /= 0.0 when Y = 0.0
    -- Error conditions:
```

```
    --            Error if X = 0.0 and Y = 0.0
    -- Range:
    --            -MATH_PI < ARCTAN(Y,X) <= MATH_PI
    -- Notes:
    --            None

function SINH (X : in REAL) return REAL;
    -- Purpose:
    --            Returns hyperbolic sine of X
    -- Special values:
    --            SINH(0.0) = 0.0
    -- Domain:
    --            X in REAL
    -- Error conditions:
    --            None
    -- Range:
    --            SINH(X) is mathematically unbounded
    -- Notes:
    --            a) The usable domain of SINH is approxi
    --               mately given by:
    --                   ABS(X) <= LOG(REAL'HIGH)


function COSH (X : in REAL) return REAL;
    -- Purpose:
    --            Returns hyperbolic cosine of X
    -- Special values:
    --            COSH(0.0) = 1.0
    -- Domain:
    --            X in REAL
    -- Error conditions:
    --            None
    -- Range:
    --            COSH(X) >= 1.0
    -- Notes:
    --            a) The usable domain of COSH is
    --               approximately given by:
    --                   ABS(X) <= LOG(REAL'HIGH)

function TANH (X : in REAL) return REAL;
    -- Purpose:
    --            Returns hyperbolic tangent of X
    -- Special values:
    --            TANH(0.0) = 0.0
    -- Domain:
    --            X in REAL
    -- Error conditions:
    --            None
    -- Range:
    --            ABS(TANH(X)) <= 1.0
    -- Notes:
    --            None
```

```
    function ARCSINH (X : in REAL) return REAL;
        -- Purpose:
        --          Returns inverse hyperbolic sine of X
        -- Special values:
        --          ARCSINH(0.0) = 0.0
        -- Domain:
        --          X in REAL
        -- Error conditions:
        --          None
        -- Range:
        --          ARCSINH(X) is mathematically unbounded
        -- Notes:
        --          a) The reachable range of ARCSINH is
        --             approximately given by:
        --                 ABS(ARCSINH(X)) <= LOG(REAL'HIGH)

    function ARCCOSH (X : in REAL) return REAL;
        -- Purpose:
        --          Returns inverse hyperbolic cosine of X
        -- Special values:
        --          ARCCOSH(1.0) = 0.0
        -- Domain:
        --          X >= 1.0
        -- Error conditions:
        --          Error if X < 1.0
        -- Range:
        --          ARCCOSH(X) >= 0.0
        -- Notes:
        --          a) The upper bound of the reachable
        --             range of ARCCOSH is approximately
        --             given by:
        --                 ARCCOSH(X) <= LOG(REAL'HIGH)

    function ARCTANH (X : in REAL) return REAL;
        -- Purpose:
        --          Returns inverse hyperbolic tangent of X
        -- Special values:
        --          ARCTANH(0.0) = 0.0
        -- Domain:
        --          ABS(X) < 1.0
        -- Error conditions:
        --          Error if ABS(X) >= 1.0
        -- Range:
        --          ARCTANH(X) is mathematically unbounded
        -- Notes:
        --          a) The reachable range of ARCTANH is
        --             approximately given by:
        --                 ABS(ARCTANH(X)) < LOG(REAL'HIGH)

end  MATH_REAL;
```

Index