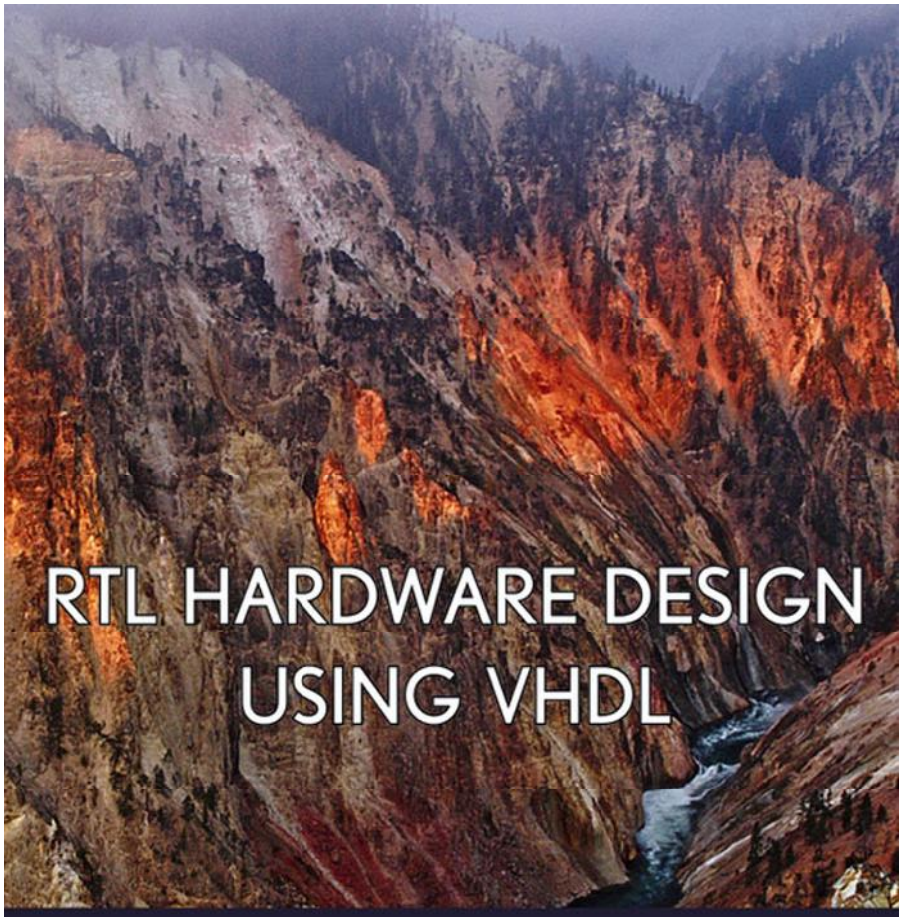
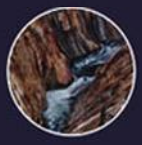


RTL HARDWARE DESIGN USING VHDL

Coding for Efficiency, Portability, and Scalability



PONG P. CHU

RTL HARDWARE DESIGN USING VHDL

Coding for Efficiency, Portability, and Scalability

PONG P. CHU
Cleveland State University

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

This Page Intentionally Left Blank

RTL HARDWARE DESIGN USING VHDL

This Page Intentionally Left Blank

RTL HARDWARE DESIGN USING VHDL

Coding for Efficiency, Portability, and Scalability

PONG P. CHU
Cleveland State University

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Chu, Pong P., 1959–
RTL hardware design using VHDL / by Pong P. Chu.
p. cm.
Includes bibliographical references and index.
"A Wiley-Interscience publication."
ISBN-13: 978-0-471-72092-8 (alk. paper)
ISBN-10: 0-471-72092-5 (alk. paper)
1. Digital electronics—Data processing. 2. VHDL (Computer hardware description language). I. Title.

TK7868.D5C46 2006
621.39'2—dc22 2005054234

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my parents Chia-Chi and Chi-Te, my wife Lee, and my daughter Patricia

This Page Intentionally Left Blank

CONTENTS

PREFACE

With the maturity and availability of hardware description language (HDL) and synthesis software, using them to design custom digital hardware has become a mainstream practice. Because of the resemblance of an HDL code to a traditional program (such as a C program), some users believe incorrectly that designing hardware in HDL involves simply writing syntactically correct software code, and assume that the synthesis software can automatically derive the physical hardware. Unfortunately, synthesis software can only perform transformation and local optimization, and cannot convert a poor description into an efficient implementation. Without an understanding of the hardware architecture, the HDL code frequently leads to unnecessarily complex hardware, or may not even be synthesizable.

This book provides in-depth coverage on the systematical development and synthesis of efficient, portable and scalable register-transfer-level (RT-level) digital circuits using the VHDL hardware description language. RT-level design uses intermediate-sized components, such as adders, comparators, multiplexers and registers, to construct a digital system. It is the level that is most suitable and effective for today's synthesis software.

RT-level design and VHDL are two somewhat independent subjects. VHDL code is simply one of the methods to describe a hardware design. The same design can also be described by a schematic or code in other HDLs. VHDL and synthesis software will not lead automatically to a better or worse design. However, they can shield designers from low-level details and allow them to explore and research better architectures.

The emphasis of the book is on *hardware* rather than *language*. Instead of treating synthesis software as a mysterious black box and listing "recipe-like" codes, we explain the relationship between the VHDL constructs and the underlying hardware structure and illustrate how to explore the design space and develop codes that can be synthesized into efficient cell-level implementation. The discussion is independent of technology and can

be applied to both ASIC and FPGA devices. The VHDL codes listed in the book largely follow the IEEE 1076.6 RTL synthesis standard and can be accepted by most synthesis software. Most codes can be synthesized without modification by the free "demo-version" synthesis software provided by FPGA vendors.

Scope The book focuses primarily on the design and synthesis of RT-level circuits. A subset of VHDL is used to describe the design. The book is not intended to be a comprehensive ASIC or FPGA book. All other issues, such as device architecture, placement and routing, simulation and testing, are discussed exclusively from the context of RT-level design.

Unique features The book is a hardware design text. VHDL and synthesis software are used as tools to realize the intended design. Several unique features distinguish the book:

- Suggest a coding style that shows a clear relationship between VHDL constructs and hardware components.
- Use easy-to-understand conceptual diagrams, rather than cell-level netlists, to explain the realization of VHDL codes.
- Emphasize the reuse aspect of the codes throughout the book.
- Consider RT-level design as an integral part of the overall development process and introduce good design practices and guidelines to ensure that an RT-level description can accommodate future simulation, verification and testing needs.
- Make the design "technology neutral" so that the developed VHDL code can be applied to both ASIC and FPGA devices.
- Follow the IEEE 1076.6 RTL synthesis standard to make the codes independent of synthesis software.
- Provide a set of synthesis guidelines at the end of each chapter.
- Contain a large number of non-trivial, practical examples to illustrate and reinforce the design concepts, procedures and techniques.
- Include two chapters on realizing sequential algorithms in hardware (known as "register transfer methodology") and on designing control path and data path.
- Include two chapters on the scalable and parameterized designs and coding.
- Include a chapter on the synchronization and interface between multiple clock domains.

Book organization The book is basically divided into three major parts. The first part, Chapters 1 to 6, provides a comprehensive overview of VHDL and the synthesis process, and examines the hardware implementation of basic VHDL language constructs. The second part, Chapters 7 to 12, covers the core of the RT-level design, including combinational circuits, "regular" sequential circuits, finite state machine and circuits designed by register transfer methodology. The third part, Chapters 13 to 16, covers the system issues, including the hierarchy, parameterized and scalable design, and interface between clock domains. More detailed descriptions of the chapters follow.

- Chapter 1 presents a "big picture" of digital system design, including an overview on device technologies, system representation, development flow and software tools.
- Chapter 2 provides an overview on the design, usage and capability of a hardware description language. A series of simple codes is used to introduce the basic modeling concepts of VHDL.
- Chapter 3 provides an overview of the basic language constructs of VHDL, including lexical elements, objects, data types and operators. Because VHDL is a strongly typed language, the data types and operators are discussed in more detail.

- Chapter 4 covers the syntax, usage and implementation of concurrent signal assignment statements of VHDL. It shows how to realize these constructs by multiplexing and priority routing networks.
- Chapter 5 examines the syntax, usage and implementation of sequential statements of VHDL. It shows the realization of the sequential statements and discusses the caveats of using these statements.
- Chapter 6 explains the realization of VHDL operators and data types, provides an in-depth overview on the synthesis process and discusses the timing issue involved in synthesis.
- Chapter 7 covers the construction and VHDL description of more sophisticated combinational circuits. Examples show how to transform conceptual ideas into hardware, and illustrate resource-sharing and circuit-shaping techniques to reduce circuit size and increase performance.
- Chapter 8 introduces the synchronous design methodology and the construction and coding of synchronous sequential circuits. Basic "regular" sequential circuits, such as counters and shift registers, in which state transitions exhibit a regular pattern, are examined.
- Chapter 9 explores more sophisticated regular sequential circuits. The design examples show the implementation of a variety of counters, the use of registers as fast, temporary storage, and the construction of pipelined combinational circuits.
- Chapter 10 covers finite state machine (FSM), which is a sequential circuit with "random" transition patterns. The representation, timing and implementation issues of FSMs are studied with an emphasis on its use as the control circuit for a large, complex system.
- Chapter 11 introduces the register transfer methodology, which describes system operation by a sequence of data transfers and manipulations among registers, and demonstrates the construction of the data path (a regular sequential circuit) and the control path (an FSM) used in this methodology.
- Chapter 12 uses a variety of design examples to illustrate how the register transfer methodology can be used in various types of problems and to highlight the design procedure and relevant issues.
- Chapter 13 features the design hierarchy, in which a system is gradually divided into smaller parts. Mechanisms and language constructs of VHDL used to specify and configure a hierarchy are examined.
- Chapter 14 introduces parameterized design, in which the width and functionality of a circuit are specified by explicit parameters. Simple examples illustrate the mechanisms used to pass and infer parameters and the language constructs used to describe the replicated structures.
- Chapter 15 provides more sophisticated parameterized design examples. The main focus is on the derivation of efficient parameterized RT-level modules that can be used as building blocks of larger systems.
- Chapter 16 covers the effect of a non-ideal clock signal and discusses the synchronization of an asynchronous signal and the interface between two independent clock domains.

Audience The intended audience for the book is students in advanced digital system design course and practicing engineers who wish to sharpen their design skills or to learn the effective use of today's synthesis software. Readers need to have basic knowledge of digital systems. The material is normally covered in an introductory digital design course,

which is a standard part in all electrical engineering and computer engineering curricula. No prior experience on HDL or synthesis is needed.

Verilog is another popular HDL. Since the book emphasizes hardware and methodology rather than language constructs, readers with prior Verilog experience can easily follow the discussion and learn VHDL along the way. Most VHDL codes can easily be translated into the Verilog language.

Web site An accompanying web site (`http://academic.csuohio.edu/chu_p/rtl`) provides additional information, including the following materials:

- Errata.
- Summary of coding guidelines.
- Code listing.
- Links to demo-version synthesis software.
- Links to some referenced materials.
- Frequently asked questions (FAQ) on RTL synthesis.
- Lecture slides for instructors.

Errata The book is "self-prepared," which means the author has prepared all materials, including the illustrations, tables, code listing, indexing and formatting, by himself. As the errors are always bound to happen, the accompanying web site provides an updated errata sheet and a place to report errors.

P. P. CHU

Cleveland, Ohio
January 2006

ACKNOWLEDGMENTS

The author would like to express his gratitude to Professor George L. Kramerich for his encouragement and help during the course of this project. The work was partially supported by educational material development grant 0126752 from the National Science Foundation and a Teaching Enhancement grant from Cleveland State University.

P. P. Chu

This Page Intentionally Left Blank

CHAPTER 1

INTRODUCTION TO DIGITAL SYSTEM DESIGN

Developing and producing a digital system is a complicated process and involves many tasks. The design and synthesis of a register transfer level circuit, which is the focus of this book, is only one of the tasks. In this chapter, we present an overview of device technologies, system representation, development flow and software tools. This helps us to better understand the role of the design and synthesis task in the overall development and production process.

1.1 INTRODUCTION

Digital hardware has experienced drastic expansion and improvement in the past 40 years. Since its introduction, the number of transistors in a single chip has grown exponentially, and a silicon chip now routinely contains hundreds of thousands or even hundreds of millions of transistors. In the past, the major applications of digital hardware were computational systems. However, as the chip became smaller, faster, cheaper and more capable, many electronic, control, communication and even mechanical systems have been "digitized" internally, using digital circuits to store, process and transmit information.

As applications become larger and more complex, the task of designing digital circuits becomes more difficult. The best way to handle the complexity is to view the circuit at a more abstract level and utilize software tools to derive the low-level implementation. This approach shields us from the tedious details and allows us to concentrate and explore high-level design alternatives. Although software tools can automate certain tasks, they are capable of performing only limited transformation and optimization. They cannot, and

RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. By Pong P. Chu

will not, do the design or convert a poor design to a good one. The ultimate efficiency still comes from human ingenuity and experience. The goal of this book is to show how to systematically develop an efficient, portable design description that is both abstract, yet detailed enough for effective software synthesis.

Developing and producing a digital circuit is a complicated process, and the design and synthesis are only two of the tasks. We should be aware of the "big picture" so that the design and synthesis can be efficiently integrated into the overall development and production process. The following sections provide an overview of device technologies, system representation, abstraction, development flow, and the use and limitations of software tools.

1.2 DEVICE TECHNOLOGIES

If we want to build a custom digital system, there are varieties of device technologies to choose, from off-the-shelf simple field-programmable components to full-custom devices that tailor the application down to the transistor level. There is no single best technology, and we have to consider the trade-offs among various factors, including chip area, speed, power and cost.

1.2.1 Fabrication of an IC

To better understand the differences between the device technologies, it is helpful to have a basic idea of the fabrication process of an integrated circuit (IC). An IC is made from layers of doped silicon, polysilicon, metal and silicon dioxide, built on top of one another, on a thin silicon wafer. Some of these layers form transistors, and others form planes of connection wires.

The basic step in IC fabrication is to construct a layer with a customized pattern, a process known as *lithography*. The pattern is defined by a *mask*. Today's IC device technology typically consists of 10 to 15 layers, and thus the lithography process has to be repeated 10 to 15 times during the fabrication of an IC, each time with a unique mask.

One important aspect of a device technology is the silicon area used by a circuit. It is expressed by the length of a smallest transistor that can be fabricated, usually measured in *microns* (a millionth of a meter). As the device fabrication process improved, the transistor size continued to shrink and now approaches a tenth of a micron.

1.2.2 Classification of device technologies

There is an array of device technologies that can be used to construct a custom digital circuit. One major characteristic of a technology is how the customization is done. In certain technologies, all the layers of a device are predetermined, and thus the device can be prefabricated and manufactured as a standard off-the-shelf part. The customization of a circuit can be performed "in the field," normally by downloading a connection pattern to the device's internal memory or by "burning the internal silicon fuses." On the other hand, some device technologies need one or more layers to be customized for a particular application. The customization involves the creation of tailored masks and fabrication of the patterned layers. This process is expensive and complex and can only be done in a fabrication plant (known as a *foundry* or a *fab*). Thus, whether a device needed to be fabricated in a fab is the most important characteristic of a technology. In this book, we use

the term *application-specific IC* (*ASIC*) to represent device technologies that require a fab to do customization.

With an understanding of the difference between ASIC and non-ASIC, we can divide the device technologies further into the following types:

- Full-custom ASIC
- Standard-cell ASIC
- Gate array ASIC
- Complex field-programmable logic device
- Simple field-programmable logic device
- Off-the-shelf small- and medium-scaled IC (SSI/MSI) components

Full-custom ASIC In *full-custom ASIC* technology, all aspects of a digital circuit are tailored for one particular application. We have complete control of the circuit and can even craft the layout of a transistor to meet special area or performance needs. The resulting circuit is fully optimized and has the best possible performance. Unfortunately, designing a circuit at the transistor level is extremely complex and involved, and is only feasible for a small circuit. It is not practical to use this approach to design a complete system, which now may contain tens and even hundreds of millions of transistors. The major application of full-custom ASIC technology is to design the basic logic components that can be used as building blocks of a larger system. Another application is to design special-purpose "bit-slice" typed circuits, such as a 1-bit memory or 1-bit adder. These circuits have a regular structure and are constructed through a cascade of identical slices. To obtain optimal performance, full-custom ASIC technology is frequently used to design a single slice. The slice is then replicated a number of times to form a complete circuit.

The layouts of a full-custom ASIC chip are tailored to a particular application. All layers are different and a mask is required for every layer. During fabrication, all layers have to be custom constructed, and nothing can be done in advance.

Standard-cell ASIC In *standard-cell ASIC* (also simply known as *standard-cell*) technology, a circuit is constructed by using a set of predefined logic components, known as *standard cells*. These cells are predesigned and their layouts are validated and tested. Standard-cell ASIC technology allows us to work at the gate level rather than at the transistor level and thus greatly simplifies the design process. The device manufacturer usually provides a library of standard cells as the basic building blocks. The library normally consists of basic logic gates, simple combinational components, such as an and-or-inverter, 2-to-1 multiplexer and 1-bit full adder, and basic memory elements, such as a D-type latch and D-type flip-flop. Some libraries may also contain more sophisticated function blocks, such as an adder, barrel shifter and random access memory (RAM).

In standard-cell technology, a circuit is made of cells. The types of cells and the interconnection depend on the individual application. Whereas the layout of a cell is predetermined, the layout of the complete circuit is unique for a particular application and nothing can be constructed in advance. Thus, fabrication of a standard-cell chip is identical to that of a full-custom ASIC chip, and all layers have to be custom constructed.

Gate array ASIC In *gate array* ASIC (also simply known as *gate array*) technology, a circuit is built from an array of predefined cells. Unlike standard-cell technology, a gate array chip consists of only one type of cell, known as a *base cell*. The base cell is fairly simple, resembling a logic gate. Base cells are prearranged and placed in fixed positions, aligned as a one- or two-dimensional array. Since the location and type are predetermined,

the base cells can be prefabricated. The customization of a circuit is done by specifying the interconnect between these cells. A gate array vendor also provides a library of predesigned components, known as *macro cells*, which are built from base cells. The macro cells have a predefined interconnect and provide the designer with more sophisticated logic blocks.

Compared to standard-cell technology, the fabrication of a gate array device is much simpler, due to its fixed array structure. Since the array is common to all applications, the cell (and transistors) can be fabricated in advance. During construction of a chip, only the masks of metal layers, which specify the interconnect, are unique for an application and therefore must be customized. This reduces the number of custom layers from 10 to 15 layers to 3 to 5 layers and simplifies the fabrication process significantly.

Complex field-programmable device We now examine several non-ASIC technologies. The most versatile non-ASIC technology is the complex field-programmable device. In this technology, a device consists of an array of generic logic cells and general interconnect structure. Although the logic cells and interconnect structure are prefabricated, both are *programmable*. The programmability is obtained by utilizing semiconductor "fuses" or "switches," which can be set as open- or short-circuit. The customization is done by configuring the device with a specific fuse pattern. This process can be accomplished by a simple, inexpensive device programmer, normally constructed as an add-on card or an adaptor cable of a PC. Since the customization is done "in the field" rather than "in a fab," this technology is known as *field programmable*. (In contrast, ASIC technologies are "programmed" via one or more tailored masks and thus are *mask programmable*.)

The basic structures of gate array ASICs and complex field-programmable devices are somewhat similar. However, the interconnect structure of field-programmable devices is predetermined and thus imposes more constraints on signal routing. To reduce the amount of connection, more functionality is built into the logic cells of a field-programmable device, making a logic cell much more complex than a base cell or a standard cell of ASIC. According to the complexity and structure of logic cells, complex field-programmable devices can be divided roughly into two broad categories: *complex programmable logic device* (*CPLD*) and *field programmable gate array* (*FPGA*).

The logic cell of a CPLD device is more sophisticated, normally consisting of a D-type flip-flop and a PAL-like unit with configurable product terms. The interconnect structure of a CPLD device tends to be more centralized, with few groups of concentrated routing lines. On the other hand, the logic cell of an FPGA device is usually smaller, typically including a D-type flip-flop and a small look-up table or a set of multiplexers. The interconnect structure between the cells tends to be distributed and more flexible. Because of its distributive nature, FPGA is better suited for large, high-capacity complex field-programmable devices.

Simple field-programmable device Simple field-programmable logic devices, as the name indicates, are programmable devices with simpler internal structure. Historically, these devices are generically called *programmable logic devices* (*PLDs*). We add the word *simple* to distinguish them from FPGA and CPLD devices. Simple field-programmable devices are normally constructed as a two-level array, with an *and plane* and an *or plane*. The interconnect of one or both planes can be programmed to perform a logic function expressed in sum-of-product format. The devices include *programmable read only memory* (*PROM*), in which the or plane can be programmed; *programmable array logic* (*PAL*), in which the and plane can be programmed; and *programmable logic array* (*PLA*), in which both planes can be programmed.

Unlike FPGA and CPLD devices, simple field-programmable logic devices do not have a general interconnect structure, and thus their functionality is severely limited. They are

gradually being phased out. ROM, PAL and PLA are now used as internal components of an ASIC or CPLD device rather than as an individual chip.

Off-the-shelf SSI/MSI components Before the emergence of field-programmable devices, the only alternative to ASIC was to utilize the prefabricated off-the-shelf SSI/MSI components. These components are small parts with fixed, limited functionality. One example is the 7400 series transistor transistor logic (TTL) family, which contains more than 100 parts, ranging from simple nand gates to a 4-bit arithmetic unit. A custom system can be designed by a bottom-up approach, building the circuit gradually from the small existing parts. A tailored printed circuit board is needed for each application. The major disadvantage of this approach is that the most resources (power, board area and manufacturing cost) are consumed by the "package" but not by the "silicon," which performs the actual computation. Furthermore, none of today's synthesis software can utilize off-the-shelf SSI/MSI components, and thus automation is virtually impossible. As the programmable devices become more capable and less expensive, designing a large custom circuit using SSI/MSI components is no longer a feasible option and should not be considered.

Summary We have reviewed six device technologies used to implement custom digital systems. Among them, off-the-shelf SSI/MSI components and simple programmable devices are gradually being phased out and full-custom ASIC is feasible only for a small, specialized circuit. Thus, for a large digital system, there are only three viable device technologies: standard-cell ASIC, gate array ASIC and CPLD/FPGA. In the following subsection, we examine the trade-offs among these technologies.

1.2.3 Comparison of technologies

Once deciding to develop custom hardware for an application, we need to choose from the three device technologies. The major criteria for selection are *area*, *speed*, *power* and *cost*. The first three involve the technical aspects of a circuit. Cost concerns the expenditure associated with the design and production of the circuit as well as the potential lost profits. Each technology has its strengths and weaknesses, and the "best" technology depends on the needs of a particular application.

Area Chip area (or size) corresponds to the required silicon real estate to implement a particular application. A smaller chip needs fewer resources, simplifies the testing and provides better yield. The chip size depends on the architecture of the circuit and the device technology. The same function can frequently be realized by different architectures, with different areas and speeds. For example, an addition circuit can be realized by a ripple adder (simple but slow), a parallel adder (complex but fast) or a carry-look-ahead adder (somewhere in-between). Once the architecture of a circuit is determined, the area depends on the device technology. In standard-cell technology, the cells and interconnects are customized to this particular application and no silicon is wasted in irrelevant functionality. Thus, the resulting chip is fully optimized and the area is minimal. In gate array technology, the circuit has to be constructed by predefined, prearranged base cells. Since functionality and the placement of the base cells are not tailored to a specific application, silicon use is not optimal. The area of the resulting circuit is normally larger than that of a standard-cell chip. In FPGA technology, a significant portion of the silicon is dedicated to achieving programmability, which introduces a large overhead. Furthermore, the functionalities of logic cells and the interconnect are fixed in advance and it is unlikely that an application

can be an exact match for the predetermined structure. A certain percentage of the capacity will be left unutilized. Because of the overhead and relatively low utilization, the area of the resulting FPGA chip is much larger than that of an ASIC chip.

Due to the drastic difference between the device fabrication process and the diversity of applications, it is difficulty to determine the exact silicon areas in three technologies. However, it is important to recognize that the difference between standard-cell and gate array technologies is much smaller than that of FPGA and ASIC. In general, a gate array chip may need 20% to 100% larger silicon area than that of a standard-cell chip, but an FPGA chip frequently requires two to five times the area of an ASIC chip.

Speed The speed of a digital circuit corresponds to the time required to perform a function, frequently represented by the worst-case propagation delay between input and output signals. A faster circuit is always desirable and is essential for computation-intensive applications. At the architecture level, faster operation can be achieved by using a more sophisticated design, which requires a larger area. However, if the identical architecture is used, a chip with a larger area is normally slower, due to its large parasitic capacitance. Since a standard-cell chip has tailored interconnect and utilizes a minimal amount of silicon area, it has the smallest propagation delay and best speed. On the other hand, an FPGA chip has the worst propagation delay. In addition to its large size, the programmable interconnect has a relatively large resistance and capacitance, which introduces even more delay. As with chip area, the speed difference between standard-cell and gate array technologies is much less significant than that between FPGA and ASIC.

Power Power concerns the energy consumed by a part. In certain applications, such as battery-operated handheld equipment, a low power circuit is of primary importance. At the architecture level, a system can be redesigned to reduce the use of power. If the identical architecture is used, a smaller chip, which consists of fewer transistors, usually consumes less power. Thus, a standard-cell chip consumes the least amount of power and an FPGA chip uses the most power.

Standard-cell technology is clearly the best choice from a technical perspective. A chip constructed using standard-cell ASIC is small and fast, and consumes less power. This should not come as a surprise since the chip is highly optimized and wastes no resources on unnecessary overhead. The price associated with customization is the complexity. Designing and fabricating a standard-cell chip is more involved and time consuming than for the other two technologies.

Cost The design of a custom digital circuit is seldom a goal in itself. It is an economic activity, and the cost is an important, if not the deciding, factor. We consider three major expenses: production cost, development cost, and time-to-market cost.

Production cost is the expense to produce a single unit. It includes two segments: nonrecurring engineering (NRE) cost and part cost. *NRE cost* (C_{nre}) is the expense that occurs only once (and thus is not recurring) during the production process, regardless of the number of units sold. Thus, it is on a "per design" basis. *Part cost* (C_{per_part}), on the other hand, is on a "per unit" basis, covering the expense required for each individual unit, such as the expense of materials, assembly and manufacturing. Note that the NRE cost is shared by all the units and that the share of each part becomes smaller as the volume increases. The per unit production cost (C_{per_unit}) can be expressed as

$$C_{per\_unit} = C_{per\_part} + \frac{C_{nre}}{\text{units produced}}$$

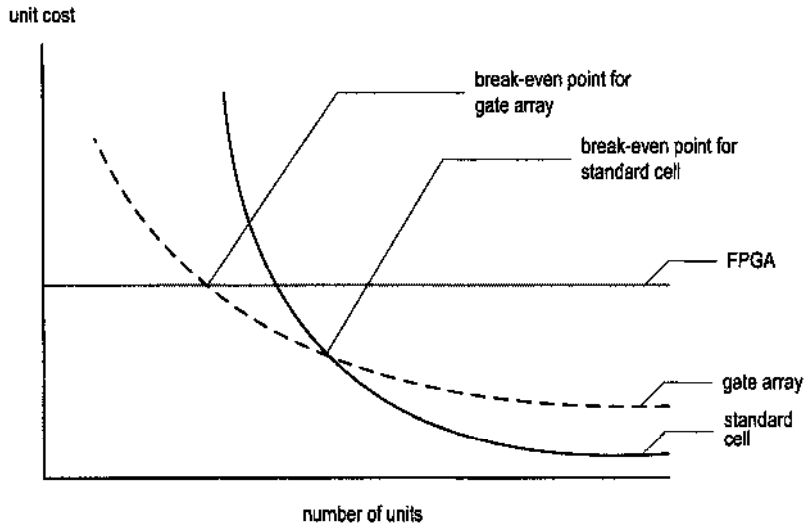


Figure 1.1 Comparison of per unit cost.

The NRE cost of a custom ASIC chip includes the creation of the tailored masks, the development of tests and the fabrication of initial sample chips. The charge is high and can range from several hundred thousand dollars to several million dollars or more. A major factor in the NRE cost is the number of custom masks needed. A standard-cell chip may need 15 or more tailored masks and thus is much more expensive than a gate array chip, which needs only three to five tailored metal layers. On the contrary, an FPGA-based design needs only an inexpensive device programmer to do customization. The NRE cost of creating a mask is negligible and can be considered as zero.

The part cost of an ASIC chip is smaller than that of an FPGA chip since the ASIC chip requires less silicon real estate and has better yield. By the same token, the part cost of a standard-cell chip is smaller than that of a gate array chip since the standard-cell chip is further optimized. If we consider both part cost and NRE cost, the per unit production cost depends on the volume of units, as shown by the previous equation. The volume versus per unit cost plots of three technologies is shown in Figure 1.1. As the volume increases, first the gate array and then the standard-cell technologies, become cost-effective. The intersections of the curves are the break-even points for the FPGA and gate array technologies, and for the gate array and standard-cell technologies.

The second major expense is the *development cost*. The process of transforming an idea to a custom circuit is by no means a simple task. The expense involved in this process is the development cost. It includes the compensation for engineering time as well as the expense of the computing facility and software tools. Although the synthesis procedure is somewhat similar for all device technologies, developing ASIC requires more effort, including physical design, placement and routing, verification and testing. Since the development process is more complex for ASIC, the development cost of an ASIC chip is much higher than that of an FPGA chip. Similarly, due to the high-level optimization, the development cost for a standard-cell chip is much higher than that for a gate array chip.

The third major expense is the *time-to-market cost*. It is actually not a cost, but the lost revenue. In many applications, such as PC peripherals, the life cycle of a product is

Table 1.1 Comparison of device technologies

	FPGA	Gate array	Standard cell
Tailored masks	0	3 to 5	15 or more
Area			best (smallest)
Speed			best (fastest)
Power			best (minimal)
NRE cost	best (smallest)		
Per part cost			best (smallest)
Development cost	best (easiest)		
Time to market	best (shortest)		
Per unit cost		depends on volume	

very short. Eighteen months, the time required to double the chip density, is sometimes considered as the life cycle of the product. Thus, it is very important to introduce the product in a timely manner, and a shipping delay can mean a significant loss in sales. The standard-cell technology requires the most lead time to validate, test and manufacture, ranging from a few months to a year. The gate array technology requires less lead time, from a few weeks to a few months. For FPGA technology, customization involves the programming of a prefabricated chip and can be done in a few minutes.

Summary The major characteristics of the three device technologies are summarized in Table 1.1. In general, the trade-off is between the optimal use of hardware resources (in terms of chip area, speed and power) and the ease of design (in terms of NRE cost, development cost and manufacturing lead time).

The choice of technology is not necessarily mutual exclusive. For example, ASIC and FPGA developments can be done in parallel to get the benefits of both technologies. The FPGA devices are used as prototypes and in initial shipments to cut the manufacturing lead time. When the ASIC devices become available later, they are used for volume production to reduce cost.

1.3 SYSTEM REPRESENTATION

A large digital system is quite complex. During the development and production process, each task may require a specific kind of information about the system, ranging from system specification to physical component layout. The same system is frequently described in different ways and is examined from different perspectives. We call these perspectives the *representations* or *views* of a system. There are three views:

- Behavioral view
- Structural view
- Physical view

A *behavioral view* describes the functionality (i.e., "behavior") of a system. It treats the system as a black box and ignores its internal implementation. The view focuses on the relationship between the input and output signals, defining the output response when a particular set of input values is applied. The description of a behavioral view is seldom unique. Normally, there are a wide variety of ways to specify the same input–output characteristics.

A *structural view* describes the internal implementation (i.e., structure) of a system. The description is done by explicitly specifying what components are used and how these components are connected. It is more or less the schematic or the diagram of a system. In computer software, we use the term *net* to represent a set of wires that are connected to the same node, and use the term *netlist*, which is a collection of nets, to represent the schematic.

A *physical view* describes the physical characteristics of the system and adds additional information to the structural view. It specifies the physical sizes of components, the physical locations of the components on a board or a silicon wafer, and the physical path of each connection line. An example of a physical view is the printed circuit board layout of a system.

Clearly, the physical view of a system provides the most detailed information. It is the final specification for the system fabrication. On the other hand, the behavioral view imposes fewest constraints and is the most abstract form of description.

1.4 LEVELS OF ABSTRACTION

As chip density reaches hundreds of millions of transistors, it is impossible for a human being, or even a computer, to process this amount of data directly. A key method of managing complexity is to describe a system in several levels of abstraction. An *abstraction* is a simplified model of the system, showing only the selected features and ignoring the associated details. The purpose of an abstraction is to reduce the amount of data to a manageable level so that only the critical information is presented. A high-level abstraction is focused and contains only the most vital data. On the other hand, a low-level abstraction is more detailed and takes account of previously ignored information. Although it is more complex, the low-level abstraction model is more accurate and is closer to the real circuit. In the development process, we normally start with a high-level abstraction and concentrate on the most vital characteristics. As the system is better understood, we then include more details and develop a lower-level abstraction.

Four levels of abstraction are considered in digital system development:

- Transistor level
- Gate level
- Register transfer (RT) level
- Processor level

The division of these levels is based primarily on the size of basic building blocks, which are the transistors, logic gates, function modules and processors respectively.

The level of abstraction and the view are two independent dimensions of a system, and each level has its own views. The levels of abstraction and views can be combined in a *Y-chart*, which is shown in Figure 1.2. In this chart, each axis represents a view and the levels of abstraction increase from the center to the outside.

The following subsections discuss the four levels of abstraction. In the discussion, we examine the five main characteristics at each level of abstraction:

- Basic building blocks
- Signal representation
- Time representation
- Behavioral representation
- Physical representation

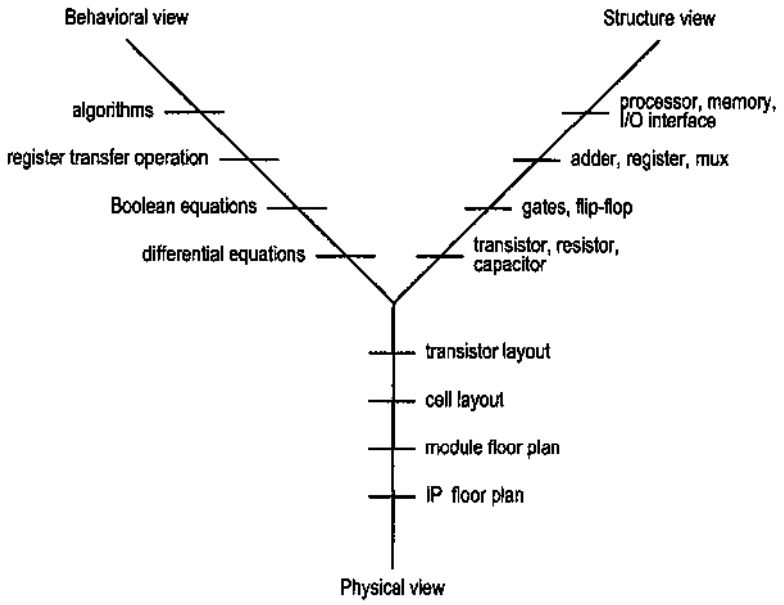


Figure 1.2 Y-chart.

Basic building blocks are the most commonly used parts at the level. These parts are the components used in the structure view. Behavioral and physical representations are the descriptions for the behavioral and physical views.

Signal and timing representations concern how to express a signal's value and how the value changes over time. While the physical signal remains the same, the interpretation of its value and timing is different at each abstraction level. As we expect, more detailed information will be provided at lower levels.

1.4.1 Transistor-level abstraction

The lowest level of abstraction is the transistor level. At this level, the basic building blocks are transistors, resistors, capacitors and so on. The behavior description is usually done by a set of differential equations or even by some type of current–voltage diagram. Analog system simulation software, such as SPICE, can be used to obtain the desired input–output characteristics.

At the transistor level, a digital circuit is treated as an analog system, in which signals are time-varying and can take on any value of a continuous range. For example, the output response of an inverter is plotted at the top of Figure 1.3.

The physical description of the transistor level comprises the detailed layout of components and their interconnections. It essentially defines the masks of various layers and is the final result of the design process.

1.4.2 Gate-level abstraction

The next level of abstraction is the gate level. Typical building blocks include simple logic gates, such as and, or, xor and 1-bit 2-to-1 multiplexer, and basic memory elements, such

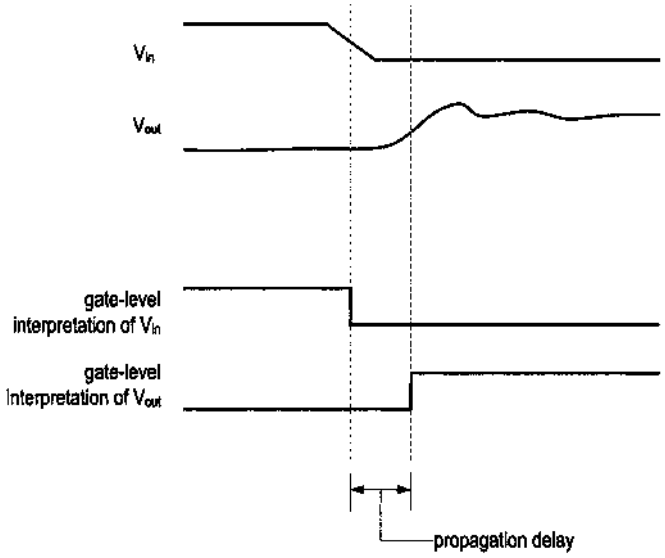


Figure 1.3 Timing characteristic of an inverter.

as latch and flip-flop. Instead of using continuous values, we consider only whether a signal's voltage is above or below a threshold, which is interpreted as logic 1 or logic 0 respectively. Since there are only two values, the input–output behavior is described by Boolean equations. The abstraction essentially converts a continuous system to a discrete system and discards the complex differential equations. Note that logic 0 and logic 1 are only our interpretation, depending on whether a signal's voltage level exceeds a predefined threshold, and the real signal is still the same continuous signal.

The timing information is also simplified at this level. A single discrete number, known as the *propagation delay*, which is defined as the time interval for a system to obtain a stable output response, is used to specify the timing of a gate. The plot at the bottom of Figure 1.3 shows a gate-level interpretation of the corresponding transistor-level signal.

The physical description at this level is the placement of the gates (or cells) and the routing of the interconnection wires.

So far, we use the term *area* or *size* to describe the silicon real estate used to construct a circuit. Alternatively, we can count the number of gates in this circuit (known as *gate count*) and make the measurement independent of the underlying device technology. The area of the two-input nand gate is used as the base unit since it is frequently the simplest physical logic circuit. Instead of using the physical area, we express the size or the complexity of a circuit in terms of the number of equivalent nand gates in that particular device technology.

1.4.3 Register-transfer-level (RT-level) abstraction

At the register-transfer (RT) level, the basic building blocks are modules constructed from simple gates. They include functional units, such as adders and comparators, storage components, such as registers, and data routing components, such as multiplexers. A reasonable name for this level would be *module-level* abstraction. However, the term *register transfer* is normally used in digital design and we follow the general convention.

Register transfer is a somewhat confusing term. It is used in two contexts. Originally, the term was used to describe a design methodology in which the system operation is specified by how the data are manipulated and transferred between storage registers. Since the main components used in the register transfer methodology are the intermediate-size modules, the term has been borrowed to describe module-level abstraction. As the title indicates, the coverage and discussion of this book focus on the RT level. We use the term *RT level* for module-level abstraction and *RT methodology* for the specific design methodology. RT methodology is discussed in Chapters 11 and 12.

The data representation at the RT level becomes more abstract. Signals are frequently grouped together and interpreted as a special kind of data type, such as an unsigned integer or system state. The behavioral description at this level uses general expressions to specify the functional operation and data routing, and uses an extended finite state machine (FSM) to describe a system designed using RT methodology.

A major feature of the RT-level description is the use of a common *clock signal* in the storage components. The clock signal functions as a sampling and synchronizing pulse, putting data into the storage component at a particular point, normally the rising or falling edge of the clock signal. In a properly designed system, the clock period is long enough so that all data signals are stabilized within the clock period. Since the data signals are sampled only at the clock edge, the difference in propagation delays and glitches have no impact on the system operation. This allows us to consider timing in terms of number of clock cycles rather than by keeping track of all the propagation delays.

The physical layout at this level is known as the *floor plan*. It is helpful for us to find the slowest path between the storage components and to determine the clock period.

1.4.4 Processor-level abstraction

Processor-level abstraction is the highest level of abstraction. The basic building blocks at this level, frequently known as *intellectual properties* (*IPs*), include processors, memory modules, bus interfaces and so on. The behavioral description of a system is more like a program coded in a conventional programming language, including computation steps and communication processes. The signals are grouped and interpreted as various data types. Time measurement is expressed in terms of a computation step, which is composed of a set of operations defined between two successive synchronization points. A collection of computations may run concurrently in parallel hardware and exchange data through a predefined communication or bus protocol. The physical layout of a processor-level system is also known as the floor plan. Of course, the components used in a floor plan are much larger than those of an RT-level system.

Table 1.2 summarizes the main characteristics at each level. It lists the typical building blocks, signal representation, time representation, representative behavioral description and representative physical description.

1.5 DEVELOPMENT TASKS AND EDA SOFTWARE

Developing a custom digital circuit is essentially a refining and validating process. A system is gradually transformed from an abstract high-level description to final mask layouts. Along with each refinement, the system's function should be validated to ensure that the final product works correctly and meets the specification and performance goals. The major design tasks of developing a digital system are:

Table 1.2 Characteristics of each abstraction level

	Typical blocks	Signal representation	Time representation	Behavioral description	Physical description
Transistor	transistor, resistor	voltage	continuous function	differential equation	transistor layout
Gate	and, or, xor, flip-flop	logic 0 or 1	propagation delay	Boolean equation	cell layout
RT	adder, mux, register	integer, system state	clock tick	extended FSM	RT-level floor plan
Processor	processor, memory	abstract data type	event sequence	algorithm in C	IP-level floor plan

- Synthesis
- Physical design
- Verification
- Testing

1.5.1 Synthesis

Synthesis is a refinement process that realizes a description with components from the lower abstraction level. The original description can be in either a behavioral view or a structural view, and the resulting description is a structural view (i.e., netlist) in the lower abstraction level. In the Y-chart, the process either moves the system from behavioral view to structural view or moves it from a high-level abstraction to a low-level abstraction. Thus, synthesis either derives a structural implementation from a behavioral description or realizes an upper level description using finer components. As the synthesis process progresses, more details are added. The final result is a gate-level structural representation using the primitive cells from the chosen device technology. To make the process manageable, synthesis is usually divided into several smaller steps, each performing a specific transformation. The major steps are:

- High-level synthesis
- RT-level synthesis
- Gate-level synthesis
- Technology mapping

High-level synthesis transforms an algorithm into an RT-level description, which is specified explicitly in terms of register transfer operations. Due to the complexity of transformation, it can only be applied to relatively simple algorithms in a narrowly defined application domain.

RT-level synthesis analyzes an RT-level behavioral description and derives the structural implementation using RT-level components. It may also perform a limited degree of optimization to reduce the number of components.

Gate-level synthesis is similar to RT-level synthesis except that gate-level components are used in structural implementation. After the initial circuit is derived, two-level or multilevel

optimization is used to minimize the size of the circuit or to meet the timing constraint. In general, generic components are used in gate-level synthesis, and thus the synthesis process is independent of device technology.

Each device technology includes a set of predesigned primitive gate-level components, which can be cells of a standard-cell library or a generic logic cell of an FPGA device. To implement the gate-level circuit in a particular device technology, the generic components have to map into the cells of the chosen technology. The transforming process is known as *technology mapping*. It is the last step in synthesis, and clearly the process is technology dependent.

The synthesis procedure is discussed in detail in Section 6.4.

1.5.2 Physical design

Physical design includes two major parts. The first part is the refinement process between the structural and physical views, which derives a layout for a netlist. The second part involves the analysis and tuning of a circuit's electrical characteristics. The main tasks in physical design include floor planning, placement and routing and circuit extraction.

Floor planning derives layouts at the processor and RT levels. It partitions the system into large function blocks and places these blocks in proper locations to reduce future routing congestion or to achieve certain timing objectives. Furthermore, floor planning may also provide a global plan for the power and clock distribution schemes. *Placement and routing* derives a layout at the gate level. The layout involves the detailed placement of cells and the routing of interconnecting wires.

After the placement and routing are complete, the exact length and location of each interconnect are known, and the associated parasitic capacitance and resistance can be calculated. This process is known as *circuit extraction*. The extracted data are used to construct a resistance and capacitance network, which in turn is used to compute the propagation delays.

In addition to the foregoing tasks, the physical design also includes design rule checking, derivation of the power grid, derivation of the clock distribution network, estimation of power use and assurance of signal integrity.

1.5.3 Verification

Verification is the process of checking whether a design meets the specification and performance goals. It concerns the correctness of the initial design as well as the correctness of refinement processes during synthesis and physical design. Verification has two aspects: *functionality* and *performance*. *Functional verification* checks whether a system generates the desired output response. Performance is represented as certain *timing constraints*. *Timing verification* checks whether the response is generated within the given time constraint. Verification is done in different phases of the design and at different levels of abstraction.

Functional verification The design of a custom system usually begins with a high-level behavioral description. When it is first created, the primary concern is whether the design functions according to the specifications. We need to check its operation and compare its responses to those desired. Once the functionality of the initial design is verified, we can start the refinement process and gradually convert it to a gate-level structural description. In general, if the initial design does not depend on the internal propagation delay (i.e., is not *delay-sensitive*), the functionality should be maintained through the refinement processes.

In the ideal situation, the design should be "correct by construction" and require no further functional verification. In reality, subtle errors may be introduced in a refinement process, and thus functional verification is still performed after each process to ensure that the new, refined description works correctly.

Timing verification Timing verification checks whether a system meets its performance goals, which are normally expressed in terms of maximal propagation delay or minimal clock frequency. At the processor or RT level, the propagation delay of an input–output path can be calculated by identifying the components in the path and summating the individual delays. However, since these components will be further refined and synthesized, the information is just a rough estimation.

At the gate level, the propagation delay of a path is affected by the delays of the components as well as the interconnection wires. The wiring delay depends on the locations and the lengths of wires. Although they can be estimated during synthesis, the exact values can be obtained only after the placement and routing process. As the size of a transistor continues to shrink, the effect of a wiring delay becomes more dominant. This makes timing verification more difficult since accurate delay information is not available during the synthesis process.

Timing issues and propagation delay are discussed in more detail in Section 6.5.1.

Methods of verification The most commonly used verification method is *simulation*, which is the process of constructing a model of a system, executing the model with input test patterns in a computer, and examining and analyzing the output responses. The model can be an actual or a hypothetical circuit that incorporates functionality and timing information. Simulation is a versatile process that be applied at any level of abstraction, and in behavioral as well as structural views. Utilizing simulation allows us to examine a system's operation in a computer and to detect errors without actually constructing the system.

Simulation essentially provides a sequence of snapshots of system operation, defined by a set of input stimuli. However, there is no guarantee that the selected stimuli can exercise every part of the system and verify the correctness of the entire design. Whereas simulation can do spot checks and detect major design mistakes, it cannot guarantee the absence of errors.

Another limitation of simulation comes from its computation complexity. Hardware operation is concurrent and parallel in nature, and it is time consuming to model its operation in a computer, which performs computational steps sequentially. It becomes a serious problem when we want to simulate low-level models, which may consist of hundreds of thousands or even millions of components.

In addition to simulation, several other methods are used for verification, including timing analysis, formal verification and hardware emulation. *Timing analysis* focuses only on the timing aspects of a circuit. It analyzes the structure of a circuit, determines all possible input–output paths, calculates the propagation delays of these paths and determines the relevant timing parameters, such as worst-case propagation delay and maximal clock frequency. Simulation can provide the relevant timing information for the selected test patterns. However, since these test patterns do not always exercise the critical paths, timing analysis is needed to verify that the system meets the timing specifications.

Formal verification applies formal mathematical techniques to analyze a circuit and determine its property. A popular method in formal verification is *equivalence checking*, which compares two representations of a system and determines whether the two representations perform the same function. It is frequently applied in synthesis to verify that the functionality of a synthesized circuit is identical to the original one. Unlike simulation,

formal verification is based on rigorous mathematical reasoning and can ensure that the synthesis is completely error-free.

Hardware emulation physically constructs a prototyping circuit that mimics operation of the system. A common application is to construct an FPGA circuit to emulate a complex ASIC design. Although the FPGA-based system is normally larger and slower than the ASIC system, it is much faster than simulation and it can be physically interfaced with other circuits and studied in detail.

1.5.4 Testing

The meanings of verification and testing are somewhat similar in a dictionary sense. However, they are two very different tasks in digital system development. *Verification* is the process of determining whether a design meets the specification and performance goals. It concerns the correctness of the initial design as well as the refinement processes. On the other hand, *testing* is the process of detecting the physical defects of a die or a package that occurred during manufacturing. When a device is being tested, we already know that the design is correct and the purpose of testing is simply to ensure that this particular part was properly fabricated.

At first glance, testing appears to be easy. All we need to do is simply to apply all possible input combinations and check the output responses. However, because of the large number of input combinations, this approach is not feasible. Instead, we have to utilize special algorithms to obtain a small set of test patterns. This process is known as *test pattern generation*.

For a small circuit, we can develop the testing procedure after completing the initial design and synthesis. However, as a digital circuit becomes larger and more complex, this approach becomes more difficult. Instead of as an afterthought, we have to consider the testing procedure in the initial design and frequently need to add auxiliary circuitry, such as a *scan chain* or *built-in-self-test circuit*, to facilitate the future requirements. This is known as *design-for-test*.

1.5.5 EDA software and its limitations

Developing a large digital circuit is a complicated process that involves complex algorithms and a large amount of data. Computer software is used to automate some tasks. This is known as *electronic design automation* (*EDA*). As computers become more powerful, we may ask if it possible to develop a suite of software and automate the development process completely. The ideal scenario would be that human designers only need to develop a high-level behavioral description, and EDA software will perform the synthesis and placement and routing and derive the optimal circuit implementation automatically. The answer is, unfortunately, negative. This is due to the theoretical limitations that cannot be overcome by faster computers or smart software codes.

The synthesis software should be treated as a tool to perform transformation and local optimization. It cannot alter the original architecture or convert a poor design into a good one. The efficiency of the final circuit depends mainly on the initial description.

The limitations and effective use of the EAD software are elaborated in Section 6.1.

1.6 DEVELOPMENT FLOW

Developing a digital circuit is essentially a refining and validating process, gradually transforming an abstract high-level description into a detailed low-level structural description. While all developments follow the basic refinement–validation process, detailed flows depend on the size of the circuit and the target device technology.

The optimization algorithms used in synthesis software are complex. The needed computation time and memory space increase drastically as the circuit size grows. Thus, size is a limiting factor in many synthesis software tools. The software is most effective for an intermediate-sized circuit, which ranges between 2000 and 50,000 gates. For a larger system, we must first partition the circuit into smaller blocks and then process each block individually.

Another factor is the target device technology. The fabrication processes of FPGA and ASIC are very different. Whereas an FPGA chip is an off-the-shelf part that has been prefabricated and pretested, an ASIC design must go through a lengthy, complex fabrication process. Many extra steps are needed to ensure the correctness of the final physical circuit.

The following subsections show the typical development flow of three different types of designs and explain the extra steps needed as the complexity increases. Three types of designs are:

- Medium-sized design targeting FPGA
- Large design targeting FPGA
- Large design targeting ASIC

1.6.1 Flow of a medium-sized design targeting FPGA

The term *medium-sized* here means a design that requires no partition and does not need predesigned IP cores. It is a circuit with up to about 50,000 gates. Current synthesis software and placement-and-routing software can effectively process a circuit of this complexity. This size is not trivial. It corresponds to that of a moderately complex circuit, such as a simple processor or bus interface. The development flow is depicted in Figure 1.4. It is shown in three columns, representing a synthesis track, physical design track and verification track respectively.

The flow starts with the design file, which is normally an RT-level description of the circuit. It may be accompanied by a set of constraints that specify the timing requirements. A separate file, known as a *testbench*, provides a virtual experiment bench for simulation and verification. It incorporates the code to generate input stimuli and to monitor the output responses. Once these files are created, the circuit can be constructed and verified accordingly. The steps in an ideal flow are detailed below.

1. Develop the design file and testbench.
2. Use the design file as the circuit description, and perform a simulation to verify that the design functions as desired.
3. Perform a synthesis.
4. Use the output netlist file of the synthesizer as the circuit description, and perform a simulation and timing analysis to verify the correctness of the synthesis and to check preliminary timing.
5. Perform placement and routing.

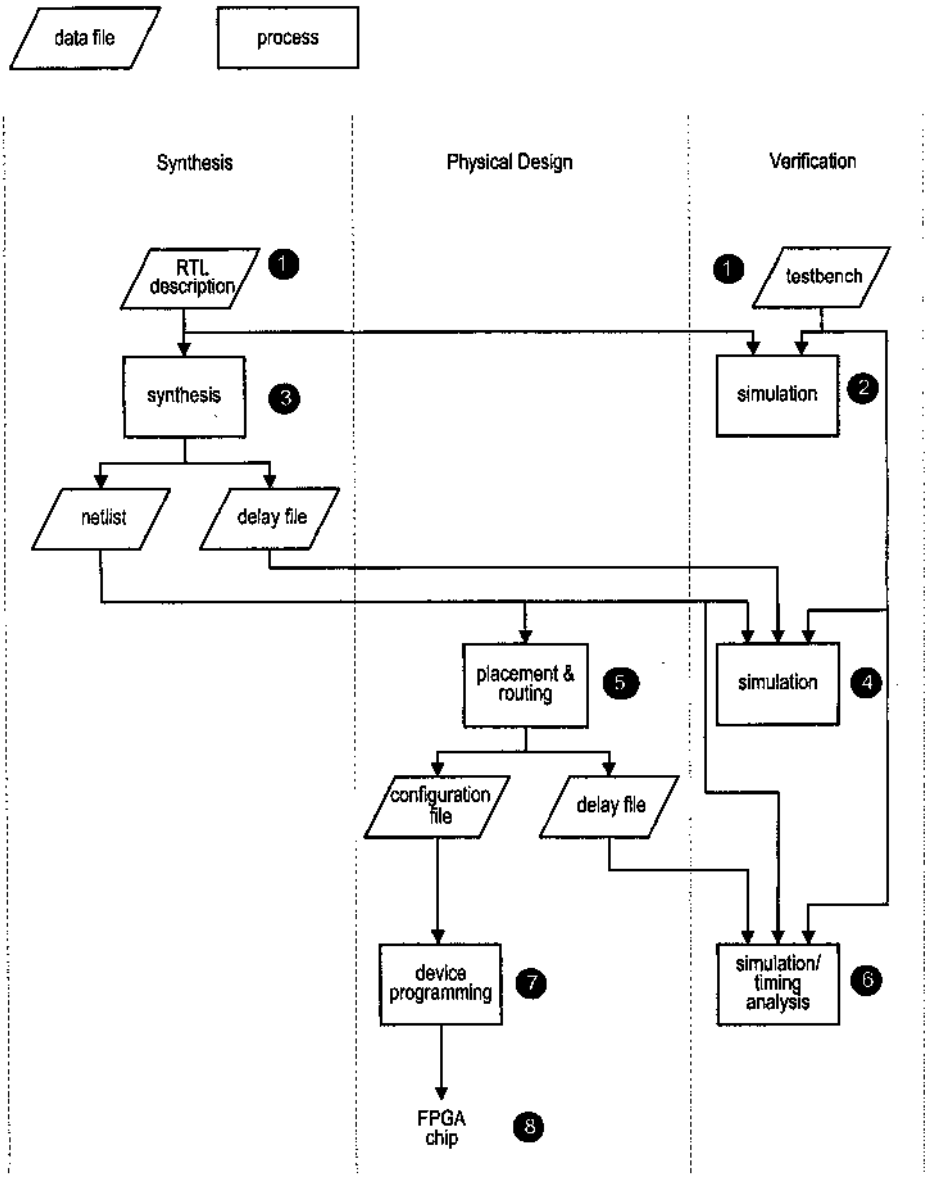


Figure 1.4 Development flow of a medium-sized design targeting FPGA.

6. Annotate the accurate timing information to the netlist, and perform a simulation and timing analysis to verify the correctness of the placement and routing and to check whether the circuit meets the timing constraints.
7. Generate the configuration file and program the device.
8. Verify operation of the physical part.

The flow described above represents an ideal process since it assumes that the initial design description follows the functional specification and meets the timing constraints. In reality, the development flow may consist of several iterations to correct the functional errors or timing problems. We may need to revise the original design file or to fine-tune parameters in synthesis and placement-and-routing software.

1.6.2 Flow of a large design targeting FPGA

A large, complex digital circuit may contain several hundred thousand or even a few million gates. Synthesis tools are not able to perform transformation and optimization effectively in this range. It is necessary to partition the circuit into smaller blocks and to process the blocks individually. The partition process also allows us to use previously designed subsystems or commercial IP cores.

To accommodate a larger design, additional processes must be added to the flow of Figure 1.4. The initial design description tends to be an abstract, high-level behavioral description of the circuit. In the synthesis track, a *partition process* is needed to divide the systems into blocks of adequate size and functionality. The output of the partition process can be considered as a netlist of large blocks. Some blocks may be already designed and verified subsystems, either from a previous project design or from a commercial IP vendor. The other blocks must be designed and synthesized individually as medium-sized circuits, following the development flow of the previous subsection.

In the verification track, an extra step is needed to verify the correctness of the partition results and to check the initial timing. Because of the large number of components, the gate-level netlist becomes very involved, and simulation consumes a significant amount of time. Formal verification techniques and cycle-based simulation are frequently used as an alternative to verify the functionality.

In the physical design track, a floor planning process may be needed. It performs initial placement for the processor-level blocks.

1.6.3 Flow of a large design targeting ASIC

Due to the complexity of ASIC fabrication, the development flow becomes more involved. The additional requirements are the inclusion of a testing track and the expansion of the physical design track.

The purpose of testing is to detect defects in the fabrication process. FPGA devices are tested by vendors before being shipped, and thus we don't need to worry about physical defects of the device. On the other hand, the testing is the integral part of the ASIC design and plays an important role. At the RT level, additional built-in-self-test circuits and special scanning control circuits are frequently added to aid the final testing. These circuits become an integral part of the design and have to be synthesized and verified. At the gate level, scan registers will be strategically inserted around circuit blocks or I/O boundaries. The scan circuit also needs to be synthesized and verified along with the regular design. Finally, test vectors have to be generated for combinational circuit blocks, and simulation has to be performed to ensure that the vectors provide proper fault coverage.

In FPGA-based flow, the physical design track involves only the floor planning and placement and routing, which is accomplished by configuring the FPGA device's programmable interconnect structure. The physical design process of an ASIC device is much more complicated since it involves development and verification of the masks. After placement and routing, several additional steps are needed, including design rule checking, physical verification and circuit extraction.

Due to the high NRE cost of an ASIC-based device, it is important that the circuit is simulated and checked thoroughly before fabrication. Thus, the verification track of ASIC-based design flow has to be more comprehensive and more exhaustive.

1.7 OVERVIEW OF THE BOOK

1.7.1 Scope

This book focuses primarily on the design and synthesis of RT-level circuits. A subset of the VHDL hardware description language is used to describe the design. The book is not intended to be a comprehensive ASIC or FPGA book. All other issues, such as device architecture, placement and routing, simulation and testing, are discussed only from the context of RT-level design.

After completing this book, readers should be able to develop and design efficient RT-level systems or subsystem blocks. A physical chip for a medium-sized FPGA design or a large, manually partitioned FPGA design can be obtained with a general synthesis and placement and routing software package. Additional knowledge and more specialized software tools are needed to cover the other tasks for an ASIC design.

1.7.2 Goal

The goal of the book is to learn how to *systematically develop efficient, portable RT-level designs that can easily be integrated into a larger system.* The goal includes three major parts:

- Design for efficiency
- "Design for large"
- Design for portability

Design for efficiency Availability of HDL and synthesis software relieves us from many tedious, repetitive implementation details and allows us to explore the design at a more abstract level. However, algorithms used in synthesis software can only do transformation and perform local search and optimization. They cannot, and will not, create a good design description or convert a poor design description to a good one. The quality of the circuit lies primarily in the initial description.

The book shows the relationship between VHDL constructs and the hardware components as well as the effective use of synthesis software tools, and introduces a disciplined way to develop the initial description that leads to efficient implementation.

"Design for large" We use the term *design for large* loosely to cover three aspects:

- Design of a large module
- Design to be incorporated in a larger system
- Design to facilitate the overall development process

The main purpose of most digital system books, including this one, is to illustrate basic concepts and procedures. For clarity, the design examples are normally explained by circuits with small input size. However, the design and description for a system with a small number of inputs (e.g., a 2-bit multiplier) and a system with a larger number of inputs (e.g., a 32-bit multiplier) can be very different. Although small-input-size examples are used in this book, the design approach and coding style are aimed at a large input size, and thus the design can easily be expanded to a larger, more practical system.

As the digital system becomes more complex, an RT-level description is likely to be a part of a larger system. Although a large processor-level system is not the focus of this book, the coding and development take this into consideration so that the RT-level design can easily be incorporated into a larger system when needed.

The discussion in Section 1.6 shows that the development of a large digital system involves many tasks. RT-level design and synthesis are not an isolated part. A poorly constructed RT-level circuit makes simulation, verification and testing processes unnecessarily difficult or even impossible, and sometimes may need to be revised at a later stage of the development process. While the book focuses on RT-level design and synthesis, it treats this task as an integral part of the development process and uses methodology that can facilitate and even simplify other tasks.

Design for portability Portability means that the same design description can be used in different applications. We can examine design for portability from three perspectives:

- Device independent
- Software independent
- Design reuse

Device independent means that the same design description can be synthesized to different device technologies. From time to time, the same design may need to migrate to a different technology. It can be from one FPGA vendor to another or from FPGA to ASIC for volume production. The design descriptions of this book carefully avoid any device-dependent feature so that the code can be used for multiple device technologies.

Software independent means that the design description can be accepted by most synthesis software. Since synthesis is a very complex process, software packages from different vendors have different capabilities, support different subsets of hardware description language and may have different interpretations on some subtle language constructs. We try to use the minimal common denominator of the synthesis software so that a design description can be accepted by most software tools and its function will be interpreted in a similar manner.

Design reuse means that the whole or part of the design description can be used again in a different application or project. We interpret the term *reuse* in a broad sense, from the copying of a few lines of code to a complete IP core. While developing an IP core is not the primary goal, we try to make the code modular and scalable when possible so that the same code can be reused in different applications with minimal or no revision.

1.8 BIBLIOGRAPHIC NOTES

This book includes a short bibliographic section at the end of each chapter. The purpose of the section is to provide several of the most relevant references for further exploration. A complete comprehensive bibliography is provided at the end of the book.

Developing a large digital system is a complex process. The text, *Methodology Manual for System-on-a-Chip Designs, 3rd edition* by M. Keating and P. Bricaud, provides an overview and guidelines for the process. The text, *The Design Warrior's Guide to FPGAs* by C. M. Maxfield, introduces relevant issues on FPGAs. Two texts, *FPGA-Based System Design* and *Modern VLSI Design: System-on-Chip Design, 3rd edition*, both by W. Wolf, provide more in-depth reviews of the FPGA and ASIC technologies.

Problems

1.1 An engineer claims the following about the digital format: "In a digital system, logic 0 and logic 1 are represented by two voltage levels. Since there is a significant voltage difference between the two levels, noise will not affect the logic value, and thus digitized information is immune to noise." Is the statement correct? Explain.

1.2 Volume of sale (i.e., the number of parts sold) is a factor when determining which device technology is to be used. Assume that a system can be implemented by FPGA, gate array or standard-cell technology. The per part cost is \$15, \$3 and \$1 for FPGA, gate array and standard cell respectively. Gate array and standard-cell technologies also involve a one-time mask generation cost of \$20,000 and \$100,000 respectively.

(a) Assume that the number of parts sold is N. Derive the equation of per unit cost for the three technologies.

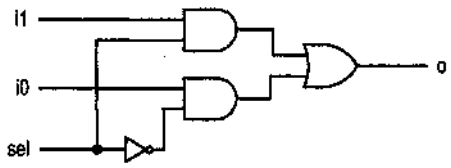
(b) Plot the three equations with N as the x-axis.

(c) Determine the range of N for which FPGA technology has the minimal per unit cost.

(d) Determine the range of N for which gate array technology has the minimal per unit cost.

(e) Determine the range of N for which standard-cell technology has the minimal per unit cost.

1.3 What is the view (behavioral, structural or physical) of the following illustration?



1.4 What is abstraction? Why is it important for digital system design?

1.5 What is the difference between testing and verification?

1.6 In Figure 1.4, the synthesized circuit is simulated in steps 4 and 6. Is the simulation in step 6 necessary? Explain.

CHAPTER 2

OVERVIEW OF HARDWARE DESCRIPTION LANGUAGES

A digital system can be described at different levels of abstractions and from different points of view. As the design process progresses, the level and view are changed, either by human designers or by software tools. It is desirable to have a common framework to exchange information among the designers and various software tools. *Hardware description languages (HDLs)* serve this purpose. In this chapter we provide an overview of the design, use and capability of HDLs. The basic concept and essential modeling features are introduced by a series of codes to show the "big picture" of HDLs. The detailed syntax, language constructs and associated semantics are discussed in subsequent chapters.

2.1 HARDWARE DESCRIPTION LANGUAGES

A digital system can be described at different levels of abstraction and from different points of view. An HDL should faithfully and accurately model and describe a circuit, whether already built or under development, from either the structural or behavioral views, at the desired level of abstraction. Because HDLs are modeled after hardware, their semantics and use are very different from those of traditional programming languages. The following subsections discuss the need, use and design of an HDL.

2.1.1 Limitations of traditional programming languages

There are wide varieties of computer programming languages, from Fortran to C to Java. Unfortunately, they are not adequate to model digital hardware. To understand their limita-

tions, it is beneficial to examine the development of a language. A programming language is characterized by its syntax and semantics. The *syntax* comprises the grammatical rules used to write a program, and the *semantics* is the "meaning" associated with language constructs. When a new computer language is developed, the designers first study the characteristics of the underlying processes and then develop syntactic constructs and their associated semantics to model and express these characteristics.

Most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will. The sequential process model has two major benefits. At the abstract level, it helps the human thinking process to develop an algorithm step by step. At the implementation level, the sequential process resembles the operation of a basic computer model and thus allows efficient translation from an algorithm to machine instructions.

The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When a signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete. After completion, each part updates the value of the corresponding output port. If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations. This description shows several unique characteristics of digital systems, including the connections of parts, concurrent operations, and the concept of propagation delay and timing. The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages (i.e., HDLs) that are designed to model digital hardware.

2.1.2 Use of an HDL program

To better understand HDL, it is helpful to examine the use of an HDL program. In a traditional programming language, a program is normally coded to solve a specific problem. It takes certain input values and generates the output accordingly. The program is first compiled to machine instructions and then run on a host computer. On the other hand, the application of an HDL program is very different. The program plays three major roles:

- *Formal documentation.* A digital system normally starts with a word description. Unfortunately, since human language is not precise, the description is frequently incomplete and ambiguous, and the same description may be subject to different interpretations. Because the semantics and syntax of an HDL are defined rigorously, a system specified in an HDL program is explicit and precise. Thus, an HDL program can be used as a formal system specification and documentation among various designers and users.
- *Input to a simulator.* As we discussed in Chapter 1, simulation is used to study and verify the operation of a circuit without constructing the system physically. An HDL simulator provides a framework to model the concurrent operations in a sequential host computer, and has specific knowledge of the language's syntactic constructs and the associated semantics. An HDL program, combined with test vector generation and a data collection code, forms a testbench, which becomes the input to the

HDL simulator. During execution, the simulator interprets HDL code and generates responses accordingly.

- *Input to a synthesizer.* The modern development flow is based on the refinement process, which gradually converts a high-level behavioral description to a low-level structural description. Some refinement steps can be performed by synthesis software. The synthesis software takes an HDL program as its input and realizes the circuit from the components of a given library. The output of the synthesizer is a new HDL program that represents the structural description of the synthesized circuit.

2.1.3 Design of a modern HDL

The fundamental characteristics of a digital circuit are defined by the concepts of entity, connectivity, concurrency and timing. *Entity* is the basic building block, modeling after a part of a real circuit. It is self-contained and independent, and has no implicit information about other entities. *Connectivity* models the connecting wires among the parts. It is the way that entities interact with one another. Since the connections of a system are seldom formed as a single thread, many entities may be active at the same time and many operations are performed in parallel. *Concurrency* describes this type of behavior. *Timing* is related to concurrency. It specifies the initiation and completion of each operation and implicitly provides a schedule and order of multiple operations.

The goal of an HDL is to describe and model digital systems faithfully and accurately. To achieve this, the cornerstone of the language should be based on the model of hardware operation, and its semantics should be able to capture the fundamental characteristics of the circuits.

As we discussed in Chapter 1, a digital system can be described at four different levels of abstraction and from three different points of view. Although these descriptions have similar fundamental characteristics, their detailed representations and models vary significantly. Ideally, we wish to develop a single HDL to cover all the levels and all the views. However, this is hardly feasible because the vast differences between abstraction levels and views will make the language excessively complex. Modern HDLs normally cover descriptions in structural and behavior views, but not in physical view. They provide constructs to support modeling at the gate and RT levels, and to a limited degree, at processor and transistor levels. The highlights of modern HDLs are as follows:

- The language semantics encapsulate the concepts of entity, connectivity, concurrency, and timing.
- The language can effectively incorporate propagation delay and timing information.
- The language consists of constructs that can explicitly express the structural implementation (i.e., a block diagram) of a circuit.
- The language incorporates constructs that can describe the behavior of a circuit, including constructs that resemble the sequential process of traditional languages, to facilitate abstract behavioral description.
- The language can efficiently describe the operations and structures at the gate and RT levels.
- The language consists of constructs to support a hierarchical design process.

2.1.4 VHDL

VHDL and Verilog are the two most widely used HDLs. Although the syntax and "appearance" of the two languages are very different, their capabilities and scopes are quite similar.

Both are industrial standards and are supported by most software tools. VHDL is used in this book since it has better support for parameterized design.

VHDL stands for VHSIC (very high speed integrated circuit) HDL. The development of VHDL was sponsored initially by the US Department of Defense as a hardware documentation standard in the early 1980s and then was transferred to the IEEE (Institute of Electrical and Electronics Engineers). IEEE ratified it as IEEE standard 1076 in 1987, which is referred to as VHDL-87. Each IEEE standard is reviewed every few years and is revised as needed. IEEE revised the VHDL standard in 1993, which is referred to as VHDL-93, and made minor modifications and bug fixes in 2001, which is referred to as VHDL-2001. Since no new language construct is added in the new version, there is no significant difference between VHDL-93 and VHDL-2001. A suffix is sometimes added to the IEEE standard to indicate the year the standard was released. For example, VHDL-87 and VHDL-2001 are known as IEEE standards 1076-1987 and IEEE 1076-2001 respectively.

After the initial release, various extensions were developed to facilitate various design and modeling requirements. These extensions are documented in several IEEE standards:

- IEEE standard 1076.1-1999, *VHDL Analog and Mixed Signal Extensions (VHDL-AMS)*: defines the extension for analog and mixed-signal modeling.
- IEEE standard 1076.2-1996, *VHDL Mathematical Packages*: defines extra mathematical functions for real and complex numbers.
- IEEE standard 1076.3-1997, *Synthesis Packages*: defines arithmetic operations over a collection of bits.
- IEEE standard 1076.4-1995, *VHDL Initiative Towards ASIC Libraries (VITAL)*: defines a mechanism to add detailed timing information to ASIC cells.
- IEEE standard 1076.6-1999, *VHDL Register Transfer Level (RTL) Synthesis*: defines a subset that is suitable for synthesis.
- IEEE standard 1164-1993 *Multivalue Logic System for VHDL Model Interoperability (std_logic_1164)*: defines new data types to model multivalue logic.
- IEEE standard 1029.1-1998, *VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES)*: defines how to use VHDL to exchange information in a simulation environment.

Standards 1076.3, 1076.6 and 1164 are related to synthesis and are discussed in Chapter 3.

2.2 BASIC VHDL CONCEPT VIA AN EXAMPLE

As its name indicates, HDL describes hardware. Thus, it is essential to read or write HDL code from hardware's perspective. A simple example in this section shows the basic modeling concepts used in HDL and demonstrates the semantic differences between HDLs and traditional programming languages. The example is coded in VHDL and the language constructs are mostly self-explanatory. The purpose of the example is to provide a big picture of HDL and VHDL. The detailed syntax and language constructs are studied in subsequent chapters.

The example is a circuit that detects even parity. There are one output, `even`, and three inputs, `a(2)`, `a(1)` and `a(0)`, which are grouped as a bus. The output is asserted when there are even numbers (i.e., 0 or 2) of 1's from the inputs. The truth table of this circuit is shown in Table 2.1.

The VHDL codes for a general description, pure structural description, pure behavioral description and testbench are discussed in the following subsections.

Table 2.1 Truth table of an even-parity detector circuit

a(2)	a(1)	a(0)	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

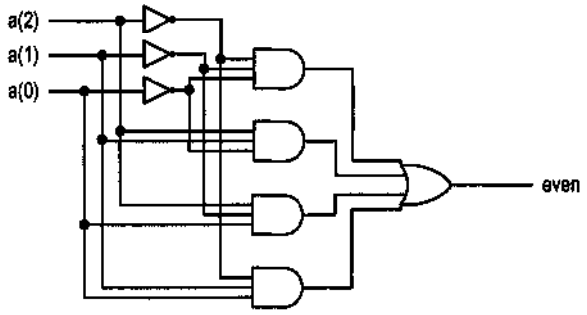


Figure 2.1 Two-level and–or implementation of an even-parity detector circuit.

2.2.1 General description

From Boolean algebra, we know that each row of a truth table represents a product term and the output can be written as the sum-of-products expression

$$even = a(2)' \cdot a(1)' \cdot a(0)' + a(2)' \cdot a(1) \cdot a(0) + a(2) \cdot a(1)' \cdot a(0) + a(2) \cdot a(1) \cdot a(0)'$$

The expression can be realized by a two-level and–or circuit, as shown in Figure 2.1.

The first VHDL description is based on this expression and the code is shown in Listing 2.1. In this book, the reserved words are in boldface font, as in **library**, and comments are in italic font, as in *-- this is a comment.*

Listing 2.1 Even-parity detector based on a sum-of-products expression

```
library ieee;
use ieee.std_logic_1164.all;

-- entity declaration
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;
```

```
-- architecture body
architecture sop_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4) after 20 ns;
   p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
   p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
   p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
   p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
end sop_arch ;
```

The code consists of two major units: entity declaration and architecture body. The *entity declaration* is:

```
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;
```

It specifies the input and output ports of this circuit. There are one output port, even, and one input port, a, which is a three-element array, representing a(2), a(1) and a(0).

The *architecture body* specifies the internal operation or organization of a circuit. The first line of the architecture body shows the name of the body, sop_arch (for sum-of-products architecture), and the corresponding entity, even_detector:

```
architecture sop_arch of even_detector is
```

The next line is the signal declaration:

```
signal p1, p2, p3, p4: std_logic;
```

The p1, p2, p3 and p4 signals here can be interpreted as wires that connect the internal parts. The declaration is visible inside this architecture.

The actual architectural description is encompassed within **begin** and **end** sop_arch:

```
   even <= (p1 or p2) or (p3 or p4) after 20 ns;
   p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
   p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
   p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
   p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
```

The fundamental building block inside the architecture body is a concurrent statement. For example, the first line is a *concurrent statement*:

```
even <= (p1 or p2) or (p3 or p4) after 20 ns;
```

A concurrent statement can be thought of as a circuit part. The left-hand-side signal or port is the output, and all the signals and ports appearing in the right-hand-side expression are the input signals. The right-hand-side expression can be considered as the operation performed by this circuit. The result is available after a specific amount of propagation delay, which is specified by the **after** clause. This particular concurrent statement can be interpreted as a circuit with inputs, p1, p2, p3 and p4, and with an output, even. It performs the or operation among the four inputs, and the operation takes 20 ns. The other four statements can be interpreted in a similar fashion.

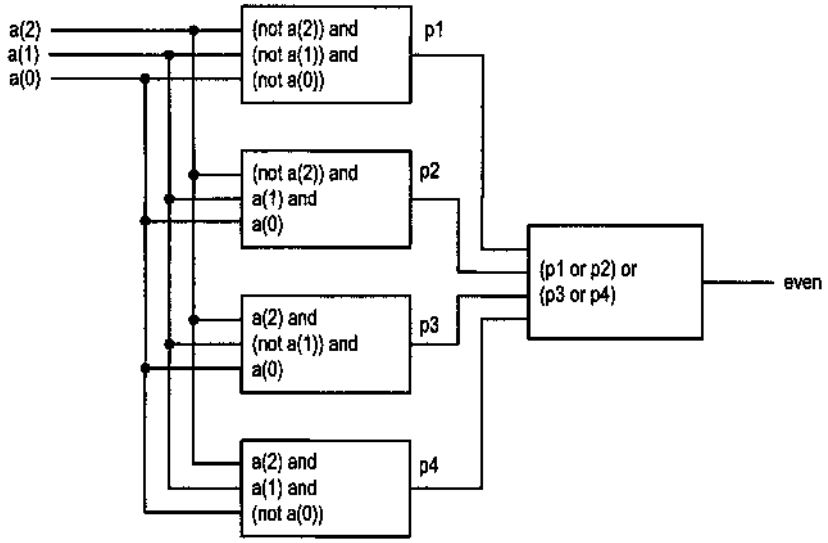


Figure 2.2 Conceptual diagram of `sop_arch` architecture.

This architecture body consists of five concurrent statements, which can be interpreted as a collection of five circuit parts. These concurrent statements are linked through common signals (or *nets*). When a signal appears on both the right- and left-hand sides, it implies that there is a wire connecting the two parts. Thus, a larger circuit is constructed implicitly through these connections. The conceptual diagram described by this code is shown in Figure 2.2.

Note that since each concurrent statement represents a circuit part and its interconnection, the order of these concurrent statements does not matter. For example, we can rearrange the code as

```
p2 <= (not a(2)) and a(1) and a(0) after 12 ns;
p3 <= a(2) and (not a(1)) and a(0) after 12 ns;
even <= (p1 or p2) or (p3 or p4) after 20 ns;
p1 <= (not a(2)) and (not a(1)) and (not a(0)) after 15 ns;
p4 <= a(2) and a(1) and (not a(0)) after 12 ns;
```

Unlike sequential execution of statements in traditional programming language, concurrent statements are independent and can be activated in parallel. When a concurrent statement's input changes, it is "awakened" and evaluates the expression accordingly. The result will be available after the specific propagation delay, and the new value will be assigned to the output signal. The change of output signals, in turn, may activate other statements and invoke a new round of evaluations.

The incorporation of propagation delay with each concurrent statement is the key ingredient to model the operation of hardware and to ensure the proper interpretation of VHDL code. Sometimes the after clause is omitted because the delay information is not available, as in

```
even <= (p1 or p2) or (p3 or p4);
```

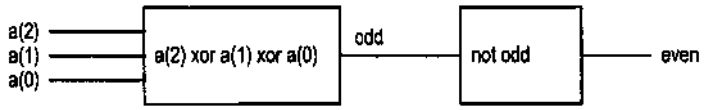


Figure 2.3 Conceptual diagram of the `xor_arch` architecture body.

In this case, VHDL semantics specifies that there is an implicit δ-delay (delta delay) associated with the operation. A *δ-delay* is an infinitesimal delay that is greater than zero but smaller than any physical number. The previous line can be interpreted as

```
even <= (p1 or p2) or (p3 or p4) after δ;
```

Thus, regardless whether there is an after clause, there is always a propagation delay associated with a concurrent statement.

The truth table is just one method to realize the even-parity detector circuit. An alternative is to use an xor ($\oplus$) operation. Recall that the xor operation can be used to detect odd parity since the $a \oplus b$ expression becomes '1' only when there is a single '1' from the inputs. Thus, the even-parity detector circuit can be implemented by an xor network followed by an inverter and the expression can be written as

$$even = (a(2) \oplus a(1) \oplus a(0))'$$

The architecture body based on this description is shown in Listing 2.2.

Listing 2.2 Even-parity detector based on an xor network

```
architecture xor_arch of even_detector is
   signal odd: std_logic;
begin
   even <= not odd;
   odd <= a(2) xor a(1) xor a(0);
end xor_arch;
```

Again, the two concurrent statements represent two circuit parts, and the conceptual diagram is shown in Figure 2.3. Since no explicit after clause is used, both statements take a δ-delay to operate.

2.2.2 Structural description

In a structural view, a circuit is constructed of smaller parts. The description specifies what types of parts are used and how these parts are connected. The description is essentially a schematic, representing a block diagram or circuit diagram. Although we treat a concurrent statement of the preceding section as a circuit part, it is our interpretation and the code is not considered as a real structural description. Formal VHDL structural description is done by using the concept of *component*. A component can be either an existing or a hypothetical part. It first has to be *declared* (make known) and then can be *instantiated* (actually used) in the architecture body as needed.

Let us consider the even-parity detector circuit again. Assume that there is a library with predesigned parts, `xor2` and `not1`, which perform the xor and inverting functions respectively. The even-parity detector circuit can be realized by the two parts, as shown in the circuit diagram of Figure 2.4. Based on the schematic, a structural description can be derived accordingly. The code of the architecture body is shown in Listing 2.3.

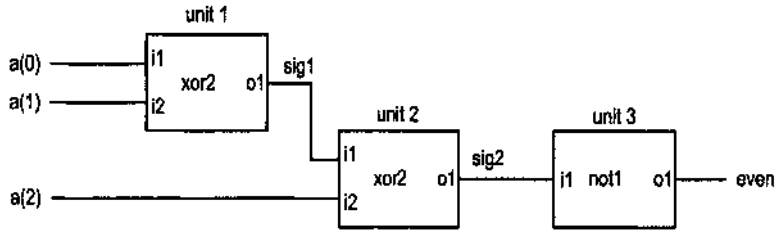


Figure 2.4 Structural diagram of the `str_arch` architecture.

Listing 2.3 Even-parity detector based on a structural description

```
architecture str_arch of even_detector is
   -- declaration for xor gate
   component xor2
      port(
         i1, i2: in std_logic;
         o1: out std_logic
      );
   end component;
   -- declaration for invertor
   component not1
      port(
         i1: in std_logic;
         o1: out std_logic
      );
   end component;
   signal sig1,sig2: std_logic;

begin
   -- instantiation of the 1st xor instance
   unit1: xor2
      port map (i1 => a(0), i2 => a(1), o1 => sig1);
   -- instantiation of the 2nd xor instance
   unit2: xor2
      port map (i1 => a(2), i2 => sig1, o1 => sig2);
   -- instantiation of invertor
   unit3: not1
      port map (i1 => sig2, o1 => even);
end str_arch;
```

Inside the architecture, the components are declared first. For example, the declaration for xor2 is

```
   component xor2
      port(
         i1, i2: in std_logic;
         o1: out std_logic
      );
   end component;
```

The information contained inside the declaration is similar to that of entity declaration, which specifies the input and output ports of a circuit. In addition to component declaration, there is also a declaration for two internal signals, `sig1` and `sig2`.

The architecture body consists of three statements, each representing a *component instantiation*. The first one is

```
unit1: xor2
   port map (i1=>a(0), i2=>a(1), o1=>sig1);
```

There are three elements in this statement. The first is the label, `unit1`, which serves as a unique id for this part. The second is the initiated component, `xor2`. The last is **port map** $\cdots$, which specifies the mapping between the formal signals (the I/O ports used in component declaration) and actual signals (the signals used in the architecture body). The mapping indicates that `i1`, `i2` and `o1` are connected to `a(0)`, `a(1)` and `sig1` respectively. The code is essentially the textual description of the circuit diagram in Figure 2.4. The three component instantiations together describe the complete circuit. The connections between the components are done implicitly by using the same signal names.

Component instantiation is one type of concurrent statement and can be mixed with other types of concurrent statements. When an architecture body consists only of component instantiations, as in this example, it is just a textual description of a schematic. This is a clumsy way for humans to conceptualize and comprehend this kind of representation. However, textual description put everything into a single VHDL framework so that the description can be handled by the same software tools. There is special design entry software that can convert a schematic to structural VHDL code and vice versa.

Component declaration contains only I/O port information, as in entity declaration. The components can be treated as empty sockets, which provide no clues about their internal functions. A component can be an existing, predesigned circuit or a hypothetical system that is still under construction. An architecture body will be bound with the component at the time of simulation or synthesis. In this example, the components may already be coded, compiled and stored in a library earlier. Their VHDL descriptions are shown in Listing 2.4.

Listing 2.4 Predesigned component

```
-- 2-input xor gate
library ieee;
use ieee.std_logic_1164.all;
entity xor2 is
   port(
      i1, i2: in std_logic;
      o1: out std_logic
  );
end xor2;

architecture beh_arch of xor2 is
begin
   o1 <= i1 xor i2;
end beh_arch;

-- invertor
library ieee;
use ieee.std_logic_1164.all;
entity not1 is
   port(
```

```
        i1: in std_logic;
        o1: out std_logic
    );
end not1;
architecture beh_arch of not1 is
begin
    o1 <= not i1;
end beh_arch;
```

Structural description and the use of components help the design in several ways. First, they facilitate hierarchical design. A complex system can be divided into several smaller subsystems, each represented by a component and designed individually. The subsystem, if needed, can be further divided into even smaller modules. Second, they provide a method to use predesigned circuits. These circuits, including complex IP cores and certain specialized library cells, can be instantiated in the description and treated as black boxes. Finally, structural description can be used to represent the result of synthesis: a gate- or cell-level netlist.

2.2.3 Abstract behavioral description

In a large design, the implementation can be very complex and the construction can be a time-consuming process. In the beginning, we frequently just want to study system operation rather than focusing on construction of the actual circuit, and prefer an abstract description. Since human reasoning and algorithms resemble a sequential process, the sequential semantics of traditional language is more adequate. VHDL provides language constructs that resemble the sequential semantics, including the use of variable and sequential execution. These features are considered as exceptions to the regular VHDL semantics, and they are encapsulated in a special construct, known as a *process*. This kind of code is sometimes referred to as *behavioral description*. However, there is no precise definition for the term behavioral description. According to VHDL, all codes, except for pure component instantiation, are considered as behavioral.

The basic skeleton of a process is

```
process(sensitivity_list)
    variable declaration;
begin
    sequential statements;
end process;
```

A process has a *sensitivity list*, which is composed of a set of signals. When a signal in the sensitivity list changes, the process is activated. Inside the process, the semantic is similar to that of a traditional programming language. Variables can be used and execution of the statements is sequential. The use of process is shown in two examples, both describing the even-parity detector circuit. The first example is based on the xor network, as in the `xor_arch` architecture. The architecture body is shown in Listing 2.5.

Listing 2.5 Even-parity detector based on a behavioral description

```
architecture beh1_arch of even_detector is
signal odd: std_logic;
begin
    -- invertor
    even <= not odd;
```

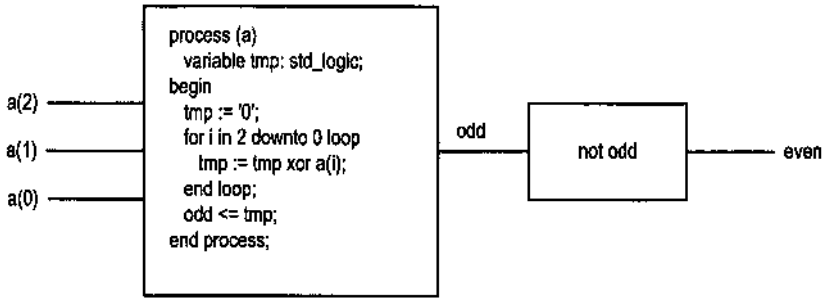


Figure 2.5 Conceptual diagram of the `beh1_arch` architecture.

```
    -- xor network for odd parity
    process(a)
       variable tmp: std_logic;
    begin
       tmp := '0';
       for i in 2 downto 0 loop
          tmp := tmp xor a(i);
       end loop;
       odd <= tmp;
    end process;
end beh1_arch;
```

The xor network is described by a process that utilizes a variable and a for loop statement. Unlike signal and signal assignment in a concurrent statement, the variable and loop do not have direct hardware counterparts. We treat a process as one indivisible part whose behavior is specified by the sequential statements. The graphic interpretation of the `beh1` architecture is shown in Figure 2.5.

The second example uses a single process to describe the desired operation in an algorithm. The algorithm first sums up the number of 1's from input, performs a modulo-2 operation to find the remainder, and then uses an if statement to check the value of the remainder to generate the final result. The VHDL code is shown in Listing 2.6.

Listing 2.6 Even-parity detector based on another behavioral description

```
architecture beh2_arch of even_detector is
begin
   process(a)
      variable sum, r: integer;
   begin
      sum := 0;
      for i in 2 downto 0 loop
         if a(i)='1' then
            sum := sum +1;
         end if;
      end loop ;
      r := sum mod 2;
      if (r=0) then
         even <= '1';
      else
```

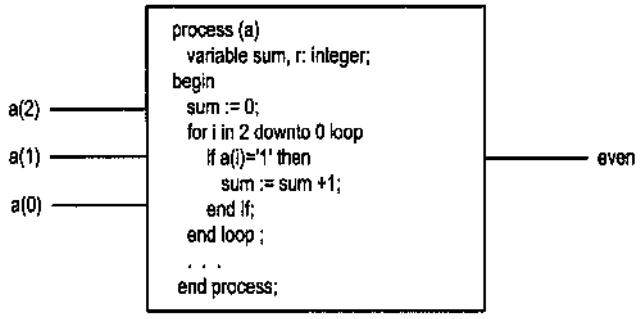


Figure 2.6 Conceptual diagram of the `beh2_arch` architecture.

```
        even <='0';
      end if;
   end process;
end beh2_arch;
```

Since there is only one process, the graphic interpretation has only one part, as in Figure 2.6. While the code is very straightforward and easy to understand, it provides no clues about the underlying structure or how to realize the code in hardware.

2.2.4 Testbench

One major use of a VHDL program is simulation, which is used to study the operation of a circuit or to verify the correctness of a design. Performing simulation is similar to doing an experiment with a physical circuit, in which we connect the circuit's input to a stimulus (e.g., a function generator) and observe the output (e.g., by logic analyzer). Simulating a VHDL description is like doing a virtual experiment, in which the physical circuit is replaced by the corresponding VHDL description. Furthermore, we can develop VHDL utility routines to imitate the stimulus generator (which is known as a *test vector generator*) and to collect and compare the output responses. The framework is known as a *testbench*.

A simple VHDL testbench for the previous even detection circuit is shown in Listing 2.7. The testbench includes a test vector generator that generates a stimulus and a verifier that verifies the correctness of the output response. The testbench consists of an entity declaration and architecture body. Since the testbench is self-contained, no port is specified in the entity declaration. There are three concurrent statements in the architecture body, including one component instantiation and two processes. The component instantiation specifies that the `even_detector` is used and its I/O pins are connected to the internal test generator and verifier. The first process is the stimulus generator. It produces all possible test vector combinations, from "000" to "111". These vectors are generated in sequential order, each lasting for 200 ns. The second process is the verifier. It takes the input test vector, waits for 100 ns to let the output settle down, checks the output value with the known value and reports the results. The two processes are for demonstration purposes only, and we don't need to worry about the syntax detail.

Listing 2.7 Simple testbench

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity even_detector_testbench is
end even_detector_testbench;

architecture tb_arch of even_detector_testbench is
   component even_detector
      port(
         a: in std_logic_vector(2 downto 0);
         even: out std_logic
      );
   end component;
   signal test_in: std_logic_vector(2 downto 0);
   signal test_out: std_logic;

begin
   -- instantiate the circuit under test
   uut: even_detector
      port map( a=>test_in, even=>test_out);
   -- test vector generator
   process
   begin
      test_in <= "000";
      wait for 200 ns;
      test_in <= "001";
      wait for 200 ns;
      test_in <= "010";
      wait for 200 ns;
      test_in <= "011";
      wait for 200 ns;
      test_in <= "100";
      wait for 200 ns;
      test_in <= "101";
      wait for 200 ns;
      test_in <= "110";
      wait for 200 ns;
      test_in <= "111";
      wait for 200 ns;
   end process;
   --verifier
   process
      variable error_status: boolean;
   begin
      wait on test_in;
      wait for 100 ns;
      if ((test_in="000" and test_out = '1') or
          (test_in="001" and test_out = '0') or
          (test_in="010" and test_out = '0') or
          (test_in="011" and test_out = '1') or
          (test_in="100" and test_out = '0') or
          (test_in="101" and test_out = '1') or
          (test_in="110" and test_out = '1') or
          (test_in="111" and test_out = '0'))
      then
```

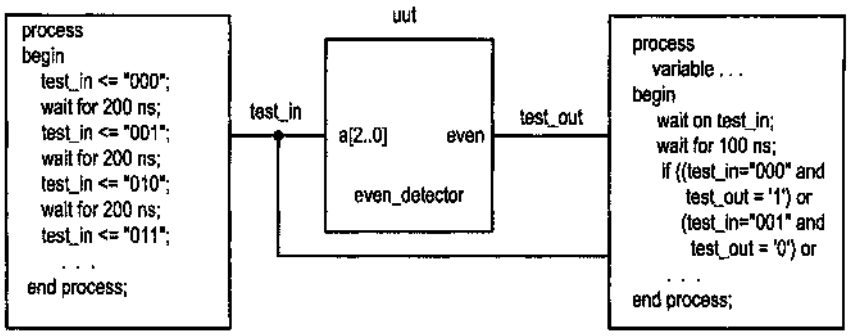


Figure 2.7 Conceptual diagram of an `even_detector` testbench.

```
            error_status := false;
         else
            error_status := true;
         end if;
60       -- error reporting
         assert not error_status
            report "test failed."
            severity note;
      end process;
65 end tb_arch;
```

The graphic interpretation of this VHDL code is shown in Figure 2.7. Most of today's simulation software can keep track of the execution of a VHDL program and display the relevant information in a tabular or graphic format.

2.2.5 Configuration

The VHDL intentionally separates the entity declaration and architecture body into two independent design units. We can associate multiple architecture bodies with a single entity declaration. For example, the `even_detector` entity of this section has about a half dozen architecture bodies. At the time of simulation or synthesis, we can choose a specific architecture body to *bind* with the entity.

An analogy of the entity and architecture is the socket and IC chip. An entity declaration can be thought of as a socket of a printed circuit board, which is empty but has fixed input and output pins. Architecture bodies can be thought of as IC chips with the same outline. While the input and output pins of these chips are identical, their internal circuitry and performances may be very different. We can select a chip and insert it into the socket according to our particular need.

VHDL provides a mechanism, known as *configuration*, to specify the binding information. In the previous example, the `even_detector` entity has five different architecture bodies. The component declaration and component instantiation of `test_bench` does not specify which body is to be used. The `test_bench` is like a printed circuit board with an empty socket, and one of the five possible chips can be inserted into the circuit. A simple configuration declaration unit is shown in Listing 2.8, in which the `sop_arch` architecture is bound with the `even_detector` entity.

Listing 2.8 Simple configuration

```
configuration demo_config of even_detector_testbench is
   for tb_arch
      for uut: even_detector
         use entity work.even_detector(sop_arch);
      end for;
   end for;
end demo_config;
```

In a VHDL program, a configuration unit is not always needed. If there is no configuration unit, the entity is automatically bound with the last compiled architecture body. The configuration is particularly helpful for the development and verification of large systems.

2.3 VHDL IN DEVELOPMENT FLOW

The examples from the previous sections show the basic language constructs and capabilities of VHDL. The choice of these constructs is not accidental. They are carefully selected to facilitate system development. In the following subsections, we discuss the use of VHDL in the development flow and the difference between coding for modeling and coding for synthesis.

2.3.1 Scope of VHDL

The scope and coverage of VHDL in a simplified development flow is illustrated in Figure 2.8. The design of a complex system normally begins with an abstract high-level description, which describes the desired behavior of the system, and a testbench, which includes a set of test vectors to exercise various functions of the system. The description and testbench allow designers to study the system operation in detail, discover any misconception or inconsistency, clarify and finalize the specification, and eventually establish the desired I/O behavior for future verification. The `beh2_arch` architecture (in Listing 2.6) of `even_detector` and the corresponding `test_bench` (in Listing 2.7) resemble these kinds of codes. In a large system, the abstract description is normally not suitable for synthesis. It either leads to unnecessarily complex circuitry or cannot be synthesized at all.

Once the specification and behavior of a system are completely understood, a synthesis-oriented code can be developed. This code is normally an RT-level description and provides a "sketch" of the underlying hardware organization so that synthesis software can derive an efficient implementation. The `xor_arch` and `beh1_arch` architecture bodies in Listings 2.2 and 2.5 resemble this kind of description. A synthesis-oriented description needs to be verified first. By utilizing the VHDL configuration, we can bind the new architecture body to the entity and use the same testbench and the previously established test vectors. After comparing the simulation responses with the known results, we can easily determine whether the new description meets the specification.

Once verified, the synthesis-oriented description can be synthesized. The result is a gate-level netlist, represented by a structural VHDL description. The code will be similar to the `str_arch` architecture body Listing 2.3. In a large design, the description is normally too tedious for humans to comprehend. Instead, it is usually plugged into the testbench via a new configuration unit. The testbench will be simulated to verify the correctness of synthesis and to study the system timing. The netlist description can then be passed to placement and routing software for further processing. The placement and routing tool

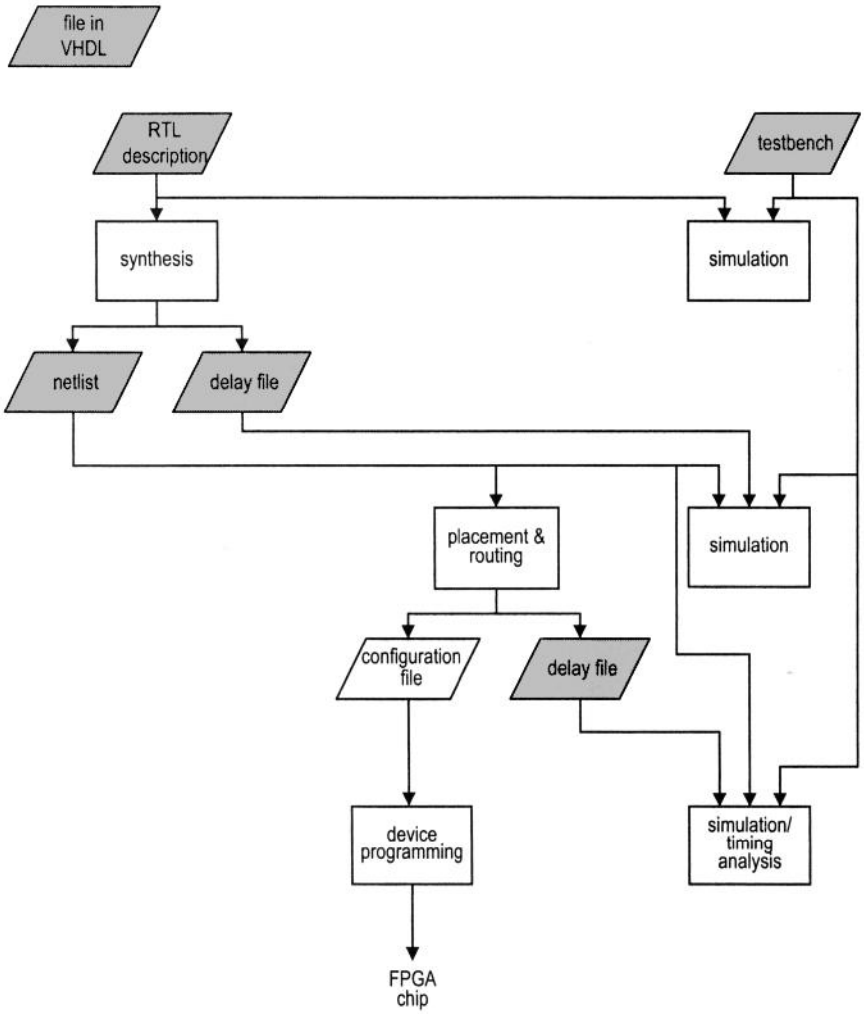


Figure 2.8 Coverage of VHDL in development flow.

will generate the layout or configuration files, which are not in VHDL. However, additional timing information will be augmented to the previous structural description. The new description will again be plugged into the testbench for final timing verification.

In summary, VHDL provides a unified environment for the entire development flow. It not only contains constructs to describe the design at various stages of the design, from the abstraction behavior to the post placement-and-routing cell-level netlist, but also provides a framework for simulation and verification.

2.3.2 Coding for synthesis

VHDL is used to model all aspects of digital hardware and to facilitate the entire design process. After a VHDL code is developed, it can be "executed" in a simulator or synthesizer. The natures of the two executions are quite different.

In simulation, the design is realized only in a virtual environment: the software simulator. The host computer utilizes its instruction set to mimic operation of the circuit. Since the host computer normally contains one processing unit, the circuit simulation is done sequentially, in which all constructs and operators of the VHDL code implicitly shared a single resource in a time-multiplexing fashion. In synthesis, on the other hand, all constructs and operators of the VHDL code are mapped to hardware. Let us consider a task that consists of 10 addition operations. In simulation, the number of addition operators, +, in VHDL code does not play a significant role since only one addition can be simulated at a time. In synthesis, each addition operator is mapped to a hardware adder, which is fairly complex, and thus it is desirable to share the hardware and to reduce the number of addition operators in VHDL description. Similarly, sophisticated control structures, such as loop or conditional branch, can be easily simulated in a sequential host but cannot be efficiently mapped to hardware.

For synthesis, only a subset of VHDL can be used. Many modeling language constructs, such as file operations and assertion statements, are not meaningful for hardware implementation. The others, such as floating-point number or complicated operators, are too complex to be synthesized automatically. IEEE defines a subset of VHDL that is suitable for RT-level synthesis in IEEE standard 1076.6. Even though the scope of the synthesizable subset is restricted, it still contains a rich collection of language constructs and is very flexible. The same circuit can be coded in a wide variety of descriptions, ranging from abstract high-level behavioral-like specification to detailed gate-level structural description. Although all these descriptions can be synthesized, there is no guarantee that the synthesized circuit is an efficient implementation. The synthesis software can perform only local search and local optimization, and the resulting circuit depends heavily on the initial description. An inadequate description consumes a large amount of CPU time during synthesis, introduces excessively complex circuitry and even fails to be synthesized.

This book focuses on RT-level design and synthesis, not VHDL. We are using VHDL as a vehicle to describe our intended hardware implementation. Our emphasis is on coding for synthesis, which means to develop VHDL code that accurately describes the underlying hardware structure and to provide adequate information to guide the synthesis software to generate an efficient implementation.

2.4 BIBLIOGRAPHIC NOTES

HDL is very different from a traditional programming language. The book, *Hardware Description Languages: Concepts and Principles* by S. Ghosh, discusses general issues

in designing HDL. Both VHDL and Verilog are IEEE standards. They are documented by *IEEE Standard VHDL Language Reference Manual* and *IEEE Standard for Verilog Hardware Description Language* respectively. The other relevant VHDL standards are also documented in the IEEE publications. The standards themselves are difficult to read. The text, *The Designer's Guide to VHDL* by P. J. Ashenden, provides a detailed and comprehensive discussion of VHDL. The texts, *Starter's Guide to Verilog 2001* by M. D. Ciletti, and *Verilog HDL, 2nd edition*, by S. Palnitkar, provide good coverage on Verilog.

The verification of a design and the derivation of the testbench are two of the major tasks in the development flow. The text, *Writing Testbenches: Functional Verification of HDL Models, 2nd edition*, by J. Bergeron, discusses this topic in detail.

Problems

2.1 What are the syntax and semantics of a programming language?

2.2 List three major differences between an HDL and a traditional programming language, such as C.

2.3 In a traditional programming language, such as C, we can write the statement `a=!a`, and in VHDL, we can write a concurrent statement as `a <= not a after 10 ns;`.

(a) Draw the circuit diagram for the VHDL statement.

(b) Describe the operation of the circuit in part (a).

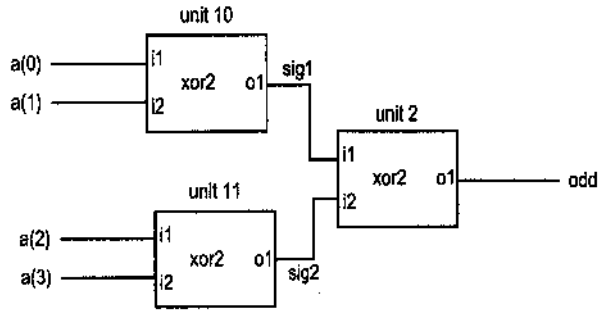
(c) Discuss the differences between the VHDL and C statements.

2.4 For the even-parity detector circuit, rewrite the expression in product-of-sums format. Revise the code of the `sop_arch` architecture body according to the new expression.

2.5 For the VHDL code shown below, treat each concurrent statement as a circuit part and draw the conceptual block diagram accordingly.

```
y <= e1 and e0;
e0 <= (a0 and b0) or ((not a0) and (not b0));
e1 <= (a1 and b1) or ((not a1) and (not b1));
```

2.6 A circuit diagram consisting of the `xor2` component is shown below. Follow the code of the `str_arch` architecture body to derive a structural VHDL description for this circuit.



2.7 The VHDL structural description of a circuit is shown below. Derive the block diagram according to the code.

```
library ieee;
use ieee.std_logic_1164.all;
entity hundred_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q_ten, q_one: out std_logic_vector(3 downto 0);
      p_ten: out std_logic
   );
end hundred_counter;

architecture str_arch of hundred_counter is
   component dec_counter
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(3 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one, p_ten: std_logic;
begin
   one_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
end str_arch;
```

2.8 From the description of the VHDL process in Section 2.2.3, discuss the differences between the VHDL process and the traditional programming languages' procedure and function.

2.9 We want to change the input of the even-parity detector circuit from 3 bits to 4 bits, i.e., from `a(2 downto 0)` to `a(3 downto 0)`. Revise the VHDL codes of the five architecture bodies to accommodate the change.

2.10 If we want to change the input of the even-parity detector circuit from 3 bits to 10 bits, discuss the amount of code modifications needed in each architecture body.

2.11 Explain why VHDL treats the entity declaration and architecture body as two separate design units.

2.12 Think of two applications that can use the configuration construct of the VHDL.

CHAPTER 3

BASIC LANGUAGE CONSTRUCTS OF VHDL

To use a programming language, we first have to learn its syntax and language constructs. In this chapter, we illustrate the basic skeleton of a VHDL program and provide an overview of the basic language constructs, including lexical elements, objects, data types and operators. VHDL is a *strongly typed language* and imposes rigorous restriction on data types and operators. We discuss this aspect in more detail.

3.1 INTRODUCTION

VHDL is a complex language. It is designed to describe both the structural and behavioral views of a digital system at various levels of abstraction. Many of the language constructs are intended for modeling and for abstract, behavioral description. Only a small portion of VHDL can be synthesized and realized physically in hardware. The IEEE 1076.6 RTL synthesis standard tries to define a subset that can be accepted by most synthesis tools. The focus of this book is synthesis, and thus the discussion is limited primarily to this subset.

VHDL was revised twice by IEEE and there are three versions: VHDL-87, VHDL-93 and VHDL-2001. Since only simple, primitive language constructs can be synthesized, the revisions do not have a significant impact on synthesis except for some differences in the syntactical appearances. Since IEEE 1076.6 mainly follows the syntax of VHDL-87, we use the syntax of VHDL-87 in the book in general and highlight the difference if any VHDL-93 feature is used.

This chapter discusses only the basic, most commonly used language constructs in VHDL and some extensions defined in IEEE standards 1076.3 and 1164. In subsequent chapters, more specialized features are covered within the context.

3.2 SKELETON OF A BASIC VHDL PROGRAM

3.2.1 Example of a VHDL program

A VHDL program is composed of a collection of *design units*. A synthesizable VHDL program needs at least two design units: an entity declaration and an architecture body associated with the entity. The skeleton of a typical VHDL program can best be explained by an example. Let us consider the `even_detector` circuit of Chapter 2. The VHDL code is shown in Listing 3.1. It uses implicit δ-delays in signal assignment statements. Note that we use the boldface font for the VHDL's reserved words.

Listing 3.1 Even-parity detector

```
library ieee;
use ieee.std_logic_1164.all;
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;

architecture sop_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4);
   p1 <= (not a(0)) and (not a(1)) and (not a(2));
   p2 <= (not a(0)) and a(1) and a(2);
   p3 <= a(0) and (not a(1)) and a(2);
   p4 <= a(0) and a(1) and (not a(2));
end sop_arch ;
```

3.2.2 Entity declaration

The entity declaration describes the external interface, or "outline" of a circuit, including the name of the circuit and the names and basic characteristics of its input and output ports. In the example, the entity declaration indicates that the name of the circuit is `even_detector` and the circuit has a 3-bit input port, a, and a 1-bit output port, even.

The simplified syntax of an entity declaration is

```
entity entity_name is
   port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type
   );
end entity_name;
```

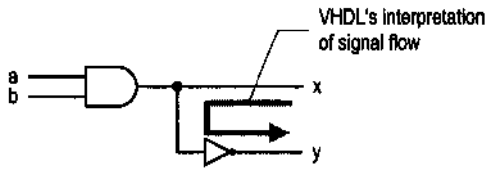


Figure 3.1 Demonstration circuit for mode.

Note that there is no semicolon (;) in the last port declaration.

A *port declaration* is composed of the `port_names`, `mode` and `data_type` terms. The `port_names` and `data_type` terms are self-explanatory. The `mode` term indicates the direction of the signal, which can be **in**, **out** or **inout**. The **in** and **out** keywords indicate that the signal flows "into" and "out of" the circuit respectively. They represent the fact that the corresponding port is an input or an output of the circuit. The **inout** keyword indicates that the signal flows in both directions and that the corresponding port is a bidirectional port. The `mode` term can also be **buffer**. It can cause a subtle compatibility problem and is not used in the book.

In the example, the port declaration shows that there are two ports. The a port is an input signal and its data type is `std_logic_vector(2 downto 0)`, which represents a 3-bit bus, and the even port is an output port and its data type is `std_logic`.

Note that a port with the **out** mode cannot be used as an input signal. For example, consider the simple circuit shown in Figure 3.1. We may be tempted to use the following code to describe the circuit:

```
library ieee;
use ieee.std_logic_1164.all;
entity mode_demo is
   port(
      a, b: in std_logic;
      x, y: out std_logic
   );
end mode_demo;
architecture wrong_arch of mode_demo is
begin
   x <= a and b;
   y <= not x;
end wrong_arch;
```

Since the x signal is used to obtain the y signal, VHDL considers it as an external signal that "flows into" the circuit, as shown in Figure 3.1. This violates the **out** mode and leads to a syntax error. One way to fix the problem is to change the mode of the x port to the **inout** mode. It is a poor solution since the x port is not actually a bidirectional port. A better alternative is to use an internal signal to represent the intermediate result, as shown in the revised code:

```
architecture ok_arch of mode_demo is
   signal ab: std_logic;
begin
   ab <= a and b;
   x <= ab;
   y <= not ab;
end ok_arch;
```

3.2.3 Architecture body

The architecture body specifies the internal operation or organization of a circuit. In VHDL, we can develop multiple architecture bodies for the same entity declaration and later choose one body to bind with the entity for simulation or synthesis. The simplified syntax of an architecture body is

```
architecture arch_name of entity_name is
   declarations;
begin
   concurrent statement;
   concurrent statement;
   concurrent statement;
   . . .
end arch_name;
```

The first line of the architecture body shows the name of the body and the corresponding entity. An architecture body may include an optional declarative section, which consists of the declarations of some objects, such as signals and constants, which are used in the architecture description. The example includes a declaration of internal signals:

```
signal p1, p2, p3, p4: std_logic;
```

The main part of the architecture body consists of the concurrent statements that describe the operation or organization of the circuit. As we discussed in Chapter 2, each concurrent statement describes an individual part and the architecture can be thought of as a collection of interconnected circuit parts. There are a variety of concurrent statements, which are discussed in subsequent chapters.

3.2.4 Design unit and library

Design units are the fundamental building blocks in a VHDL program. When a program is processed, it is broken into individual design units and each unit is analyzed and stored independently. There are five kinds of design units:

- Entity declaration
- Architecture body
- Package declaration
- Package body
- Configuration

We have just studied the entity declaration and architecture body. A *package* of VHDL normally contains a collection of commonly used items, such as data types, subprograms and components, which are needed by many VHDL programs. As the name suggests, a *package declaration* consists of the declaration of these items. A *package body* normally contains the implementation and code of the subprograms.

In VHDL, multiple architecture bodies can be associated with an entity declaration. A *configuration* specifies which architecture body is to be bound with the entity declaration. The package and configuration are discussed in Chapter 13.

A VHDL *library* is a place to store the design units. It is normally mapped into a directory in the computer's hard disk storage. The software defines mapping between the symbolic VHDL library name and the physical directory. By VHDL default, the design units will be stored in a library named `work`.

To facilitate the synthesis, IEEE has developed several VHDL packages, including the `std_logic_1164` package and the `numeric_std` package, which are defined in IEEE standards 1164 and 1076.3. These packages are discussed in Sections 3.5.2 and 3.5.4. To use a predefined package, we must include the *library* and *use statements* before the entity declaration. The first two lines of the example are for this purpose:

```
library ieee;
use ieee.std_logic_1164.all;
```

The first line invokes a library named `ieee`, and the second line makes the `std_logic_1164` package visible to the subsequent design unit. We must invoke this library because we want to use some predefined data types, `std_logic` and `std_logic_vector`, of the `std_logic_1164` package.

3.2.5 Processing of VHDL code

A VHDL program is normally processed in three stages:

1. Analysis
2. Elaboration
3. Execution

During the *analysis stage*, the software checks the syntax and some static semantic errors of the VHDL code. The analysis is performed on a design unit basis. If there is no error, the software translates the code of the design unit into an intermediate form and stores it in the designated library. A VHDL file can contain multiple design units, but a design unit cannot be split into two or more files.

In a complex design, the system is normally described in a hierarchical manner. The top level may include subsystems as instantiated components, as in the example in Section 2.2.2. During the *elaboration stage*, the software starts from the designated top-level entity declaration and binds its architecture body according to the configuration specification. If there are instantiated components, the software replaces each instantiated component with the corresponding architecture body description. The process may repeat recursively until all instantiated components are replaced. The elaboration process essentially selects and combines the needed architectural descriptions, and creates a single "flattened" description.

During the *execution stage*, the analyzed and elaborated description is usually fed to simulation or synthesis software. The former simulates and "runs" the description in a computer, and the latter realizes the description by physical circuits.

3.3 LEXICAL ELEMENTS AND PROGRAM FORMAT

3.3.1 Lexical elements

The lexical elements are the basic syntactical units in a VHDL program. They include comments, identifiers, reserved words, numbers, characters and strings.

Comments A comment starts with two dashes, `--`, followed by the comment text. Anything after the `--` symbol in the line will be ignored. The comment is for documentation purposes only and has no effect on the code. For example, we have added comments to the previous VHDL code:

```
--*****************************************************
-- example to show the caveat of the out mode
--*****************************************************
architecture ok_arch of mode_demo is
   signal ab: std_logic; -- ab is the internal signal
begin
   ab <= a and b;
   x <= ab;                -- ab connected to the x output
   y <= not ab;
end ok_arch;
```

For clarity, we use *italic type* for comments.

Identifiers An identifier is the name of an object in VHDL. The basic rules to form an identifier are:

- The identifier can contain only alphabetic letters, decimal digits and underscores.
- The first character must be a letter.
- The last character cannot be an underscore.
- Two successive underscores are not allowed.

For example, the following identifiers are valid:

```
A10, next_state, NextState, mem_addr_enable
```

On the other hand, the following identifiers violate one of the rules and will cause a syntax error during analysis of the program:

```
sig#3, _X10, 7segment, X10_, hi__there
```

Since VHDL is not case sensitive, the following identifiers are the same:

```
nextstate, NextState, NEXTSTATE, nEXTsTATE
```

It is good practice to be consistent with the use of case. In this book, we use capital letters for symbolic constants and use a special suffix, such as _n, to represent a special characteristics of an identifier. For example, the _n suffix is used to indicate an active-low signal. If we see a signal with a name like oe_n, we know it is an active-low signal.

It is also a good practice to use descriptive identifier for better readability. For example, consider the name for a signal that enables the memory address buffer. The mem_addr_en is good, mae is too short, and memory_address_enable is probably too cumbersome.

Reserved words Some words are reserved in VHDL to form the basic language constructs. These reserved words are:

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

Numbers, characters and strings A *number* in VHDL can be integer, such as `0`, `1234` and `98E7`, or real, such as `0.0`, `1.23456` or `9.87E6`. It can be represented in other number bases. For example, 45 can be represented as `2#101101#` and `16#2D#` in base 2 and base 16 respectively. We can also add an underscore to enhance readability. For example, `12_3456` is the same as `123456`, and `2#0011_1010_1101#` is the same as `2#001110101101#`.

A *character* in VHDL is enclosed in single quotation marks, such as `'A'`, `'Z'` and `'3'`. Note that `1` and `'1'` are different since the former is a number and the latter is a character.

A *string* is a sequence of characters enclosed in double quotation marks, such as `"Hello"` and `"10000111"`. Again, note that that `2#10110010#` and `"10110010"` are different since the former is a number and the latter is a string. Unlike the number, we cannot arbitrarily use an underscore inside a string. The `"10110010"` and `"1011_0010"` strings are different.

3.3.2 VHDL program format

VHDL is a case-insensitive free-format language, which means that the letter case does not matter, and "white space" (space, tab and new-line characters) can be inserted freely between lexical elements. For example, the VHDL program

```
library ieee;
use ieee.std_logic_1164.all;
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;

architecture eg_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4);
   p1 <= (not a(0)) and (not a(1)) and (not a(2));
   p2 <= (not a(0)) and a(1) and a(2);
   p3 <= a(0) and (not a(1)) and a(2);
   p4 <= a(0) and a(1) and (not a(2));
end eg_arch;
```

is the same as

```
library ieee; use ieee.std_logic_1164.all;entity
even_detector is port(a: in std_logic_vector(2
downto 0);even: out std_logic);end even_detector;
architecture eg_arch of even_detector is signal p1,
p2, p3, p4: std_logic; begin even <= (p1 or p2) or
(p3 or p4); p1 <= (not a(0)) and (not a(1)) and
(not a(2)); p2 <= (not a(0)) and a(1) and a(2);
p3 <= a(0) and (not a(1)) and a(2); p4 <= a(0) and
a(1) and (not a(2)); end eg_arch;
```

This extreme example demonstrates the importance of proper formatting. Although the program format does not affect the content or efficiency of a design, it has a significant impact on human users. An adequately documented and formatted program makes the code easier to comprehend and helps us to locate potential design errors. It will save a

tremendous amount of time for future code revision and maintenance. This is perhaps the easiest way to enhance the reusability of the code. Section 3.6.2 lists the basic guidelines for code formatting and documentation.

It is a good idea to include a short "header" comment in the beginning of the file. The header should provide general information about the design and the "design environment." A representative header of the previous VHDL program is shown below.

```
--*************************************************
--
-- Author: p chu
--
-- File: even_det.vhd
--
-- Design units:
--    entity even_detector
--       function: check even # of 1's from input
--       input: a
--       output: even
--    architecture sop_arch:
--       truth-table-based sum-of-products
--       implementation
--
-- Library/package:
--    ieee.std_logic_1164: to use std_logic
--
-- Synthesis and verification:
--    Synthesis software: . . .
--    Options/script: . . .
--    Target technology: . . .
--    Testbench: even_detector_tb
--
-- Revision history
--   Version 1.0:
--   Date: 9/2005
--   Comments: Original
--
--*************************************************
```

The first two parts list the author and file name. The "Design units" part provides a brief description about the design units in the file. The description includes the input and output ports and the function of the entity, and the implementation method of the architecture body. The final "Revision history" part provides general information about the development. The "Library/package" and "Synthesis and verification" parts describe the design environment. The idea here is to provide the necessary information for users to reconstruct or duplicate the implementation. The "Library/package" part lists the packages and libraries that are referred to in the design file, and explains briefly the use of these packages. It is especially essential when a nonstandard or custom package is involved. The "Synthesis and verification" part lists the EDA software and the script or relevant options used in the synthesis, the original targeting device technology, as well as, if available, the testbench used to verify the design. Since synthesis software from different manufacturers supports different subsets of the VHDL and may interpret certain VHDL constructs differently, this information allows future users to duplicate the original implementation.

3.4 OBJECTS

An *object* in VHDL is a named item that holds the value of a specific data type. There are four kinds of objects: **signal, variable, constant** and **file**. A construct known as **alias** is somewhat like an object. We do not discuss the **file** object in this book since it cannot be synthesized.

Signals The signal is the most common object and we already used it in previous examples. A signal has to be declared in the architecture body's declaration section. The simplified syntax of signal declaration is

```
signal signal_name, signal_name, ... : data_type;
```

For example, the following line declares the a, b and c signals with the `std_logic` data type:

```
signal a, b, c: std_logic;
```

According to the VHDL definition, we can specify an optional initial value in the signal declaration. For example, we can assign an initial value of `'0'` to the previous signals:

```
signal a, b, c: std_logic := '0';
```

While this is sometimes handy for simulation purposes, it should not be used in synthesis since not many physical devices can implement the desired effect.

The simplified syntax of signal assignment is

```
signal_name <= projected_waveform;
```

We examined the concept of `projected_waveform` in Section 2.2.1 and discuss it in more detail in Chapter 4. From the synthesis point of view, a signal represents a wire or "a wire with memory" (i.e., a register or latch).

The input and output ports of the entity declaration are also considered as signals.

Variables A variable is a concept found in a traditional programming language. It can be thought of as a "symbolic memory location" where a value can be stored and modified. There is no direct mapping between a variable and a hardware part. A variable can only be declared and used in a process and is local to that process (the exception is a shared variable, which is difficult to use and is not discussed). The main application of a variable is to describe the abstract behavior of a system.

The syntax of variable declaration is similar to that of signal declaration:

```
variable variable_name, variable_name, ... : data_type
```

An optional initial value can be assigned to variables as well.

The simplified syntax of variable assignment is

```
variable_name := value_expression;
```

Note that no timing information is associated with a variable, and thus only a value, not a waveform, can be assigned to a variable. Since there is no delay, the assignment is known as an *immediate assignment* and the notion := is used. We examine variables in detail when the VHDL process is discussed in Chapter 5.

Constants A constant holds a value that cannot be changed. The syntax of constant declaration is

```
constant constant_name: data_type := value_expression;
```

The value_expression term specifies the value of the constant. A simple example is

```
constant BUS_WIDTH: integer := 32;
constant BUS_BYTES: integer := BUS_WIDTH / 8;
```

Note that we use capital letters for constants in this book.

Since an identifier name and data type convey more information than does a literal alone, the proper use of constants can greatly enhance readability of the VHDL code and make the code more descriptive. Consider the behavioral description of even_detector in Section 2.2.3:

```
architecture beh1_arch of even_detector is
   signal odd: std_logic;
begin
   . . .
   tmp := '0';
   for i in 2 downto 0 loop
      tmp := tmp xor a(i);
   end loop;
   . . .
```

The code uses a "hard literal," 2, to specify the upper boundary of the loop's range. It becomes much clearer if we replace it with a symbolic constant:

```
architecture beh1_arch of even_detector is
   signal odd: std_logic;
   constant BUS_WIDTH: integer := 3;
begin
   . . .
   tmp := '0';
   for i in (BUS_WIDTH-1) downto 0 loop
      tmp := tmp xor a(i);
   end loop;
. . .
```

Alias Alias is not a data object. It is the alternative name for an existing object. As a constant, the purpose of an alias is to enhance code clarity and readability. One form of the signal alias is especially helpful for synthesis. Consider a machine instruction of a processor that is 16 bits wide and consists of fields with an operation code and three registers. The instruction is stored in memory as a 16-bit word. After it is read from memory, we can use an alias to identify the individual field:

```
signal word: std_logic_vector(15 downto 0);
alias op: std_logic_vector(6 downto 0) is word(15 downto 9);
alias reg1: std_logic_vector(2 downto 0) is word(8 downto 6);
alias reg2: std_logic_vector(2 downto 0) is word(5 downto 3);
alias reg3: std_logic_vector(2 downto 0) is word(2 downto 0);
```

Clearly, a name like reg1 is more descriptive than word(8 downto 6). Unfortunately, some synthesis software does not support this language construct. We can achieve this in a somewhat cumbersome way by declaring four new signals in the architecture body and assigning them with the proper portions of the word signal.

3.5 DATA TYPES AND OPERATORS

In VHDL, each object has a *data type*. A data type is defined by:

- A set of values that an object can assume.
- A set of operations that can be performed on objects of this data type.

VHDL is a known as *strongly typed language*, which means that an object can only be assigned a value of its type, and only the operations defined with the data type can be performed on the object. If a value of a different type has to be assigned to an object, the value must be converted to the proper data type by a *type conversion function* or *type casting*.

The motivation behind a strongly typed language is to catch errors in the early stage. For example, if a Boolean value is assigned to a signal of integer type or an arithmetic operation is applied to a signal of character type, the software can detect the error during the analysis stage. On the downside, the rigid type requirement may introduce many type-conversion functions and make the code cumbersome and difficult to understand.

To facilitate modeling and simulation, VHDL is rich in data types. In theory, any data type with a finite number of values can be mapped into a set of binary representations and thus can be realized in hardware. However, we refrain from doing this since the mapping introduces another dimension of uncertainty in synthesis and may lead to compatibility problems in larger designs. Our focus is on a small set of predefined data types that are relevant to synthesis. For a signal, we are mainly confined to the `std_logic`, `std_logic_vector`, `signed` and `unsigned` data types. A few user-defined data types will be used for specific applications and they will be discussed as needed.

The following subsections examine the relevant data types, operators and type conversions in VHDL and two synthesis-related IEEE packages.

3.5.1 Predefined data types in VHDL

Commonly used data types There are about a dozen predefined data types in VHDL. Only the following data types are relevant to synthesis:

- `integer`: VHDL does not define the exact range of the `integer` type but specifies that the minimal range is from $-(2^{31}-1)$ to $2^{31}-1$, which corresponds to 32 bits. Two related data types (formally known as *subtypes*) are the `natural` and `positive` data types. The former includes 0 and the positive numbers and the latter includes only the positive numbers.
- `boolean`: defined as `(false, true)`.
- `bit`: defined as `('0', '1')`.
- `bit_vector`: defined as a one-dimensional array with elements of the `bit` data type.

The original intention of the `bit` data type is to represent the two binary values used in Boolean algebra and digital logic. However, in a real design, a signal may assume other values, such as the high impedance of a tri-state buffer's output or a "fighting" value because of a conflict (e.g., two outputs are wired together, forming a short circuit). To solve the problem, a set of more versatile data types, `std_logic` and `std_logic_vector`, are introduced in the IEEE `std_logic_1164` package. To achieve better compatibility, we should avoid using the `bit` and `bit_vector` data types. The `std_logic` and `std_logic_vector` data types are discussed in Section 3.5.2.

In VHDL, data types similar to the `boolean` and `bit` types are known as the *enumeration* data types since their values are enumerated in a list.

Table 3.1 Operators and applicable data types of VHDL-93

Operator	Description	Data type of operand a	Data type of operand b	Data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a mod b	modulo			
a rem b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a sll b	shift-left logical	bit_vector	integer	bit_vector
a srl b	shift-right logical			
a sla b	shift-left arithmetic			
a srl b	shift-right arithmetic			
a rol b	rotate left			
a ror b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a and b	and	boolean, bit, bit_vector	same as a	same as a
a or b	or			
a xor b	xor			
a nand b	nand			
a nor b	nor			
a xnor b	xnor			

Operators About 30 operators are defined in VHDL. In a strongly typed language, the definition of data type includes the operations that can be performed on the object of this data type. It is important to know which data types can be used with a particular operator.

Descriptions of these operators and the applicable data types are summarized in Table 3.1. Only synthesis-related data types are listed. Most operators and data types are self-explanatory. Relational operators and the concatenation operator (&) can be applied to arrays and are discussed in Section 3.5.2.

Note that the operators in the table are defined in VHDL-93. The shift operators and the **xnor** operator are not defined in VHDL-87 and are not supported by IEEE 1076.6 RTL synthesis standard either.

Table 3.2 Precedence of the VHDL operators

Precedence	Operators
Highest	** **abs** **not**
	* / **mod** **rem**
	+ - (identity and negation)
	& + - (addition and subtraction)
	sll **srl** **sla** **sra** **rol** **ror**
	= /= < <= > >=
Lowest	**and** **or** **nand** **nor** **xor** **xnor**

During synthesis, operators in the VHDL code will be realized by physical components. Their hardware complexities vary significantly, and many operators, such as multiplication and division, cannot be synthesized automatically. This issue is discussed in Chapter 6.

The *precedence* of the operators is shown in Table 3.2, which is divided into seven groups. The operators in the same group have the same precedence. The operators in the upper group have higher precedence over the operators in the lower group. For example, consider the expression

```
a + b > c or a < d
```

The + operator will be evaluated first, and then the > and < operators, and then the **or** operator.

If an expression consists of several identical operators, evaluation begins at the leftmost operator and progresses toward the right (known as *left-associative*). For example, consider the expression

```
a + b + c + d
```

The `a + b` expression will be evaluated first, and then `c` is added, and then `d` is added.

Parentheses can be used in an expression. They have the highest preference and thus can alter the order of evaluation. For example, we can use parentheses to make the previous expression be evaluated from right to left:

```
a + (b + (c + d))
```

Unlike the logic expression used in Boolean algebra, the **and** and **or** operators have the same precedence in VHDL, and thus we must use parentheses to specify the desired order, as in

```
(a and b) or (c and d)
```

It is a good practice to use parentheses to make the code clear and readable, even when they are not needed. For example, the expression

```
a + b > c or a < d
```

can be written as

```
((a + b) > c) or (a < d)
```

This is more descriptive and reduces the chance for error or misinterpretation.

Table 3.3 VHDL operators versus conventional Boolean algebra notations

VHDL operator	Boolean algebra notation
not	$'$
and	$\cdot$
or	$+$
xor	$\oplus$
+	$+$

Notation To make an expression compact, we use the conventional symbols $'$, $\cdot$ and $+$ of Boolean algebra for *not*, *and* and *or* operations in our discussion. They are expressed as **not**, **and** and **or** in VHDL code. We also assume that $\cdot$ has precedence over $+$. For example, in our discussion, we may write

$$y = a \cdot b + a' \cdot b'$$

When coded in VHDL, This expression becomes

```
y <= (a and b) or ((not a) and (not b));
```

The notations used in our discussion and VHDL code are summarized in Table 3.3. Note that the + notation is used as both or and addition operations in our discussion. Since they are used in different contexts, it should not introduce confusion.

3.5.2 Data types In the IEEE std_logic_1164 package

To better reflect the electrical property of digital hardware, several new data types were developed by IEEE to serve as an extension to the `bit` and `bit_vector` data types. Theses data types are defined in the `std_logic_1164` package of IEEE standard 1164. In this subsection, we discuss the new data types, the operations defined over these data types and the conversion between these data types and the predefined VHDL data types.

std_logic and std_logic_vector data types The two most useful data types defined in the `std_logic_1164` package are `std_logic` and `std_logic_vector`. Formally speaking, the `std_logic` data type is actually a subtype of the `std_ulogic` data type. Since the `std_ulogic` data type is "unresolved," it has some limitations and will not be used in this book.

To use the new data types, we must include the necessary library and use statements before the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

The `std_logic` data type consists of nine possible values, which are shown in the following list:

```
('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
```

These values are interpreted as follows:

- `'0'` and `'1'`: stand for "forcing logic 0" and "forcing logic 1," which mean that the signal is driven by a circuit with a regular driving current.

- `'Z'`: stands for high impedance, which is usually encountered in a tri-state buffer.
- `'L'` and `'H'`: stand for "weak logic 0" and "weak logic 1," which means that the signal is obtained from wired-logic types of circuits, in which the driving current is weak.
- `'X'` and `'W'`: stand for "unknown" and "weak unknown." The unknown represents that a signal reaches an intermediate voltage value that can be interpreted as neither logic 0 or logic 1. This may happen because of a conflict in output (such as a logic-0 signal and a logic-1 signal being tied together). They are used in simulation for an erroneous condition.
- `'U'`: stands for uninitialized. It is used in simulation to indicate that a signal or variable has not yet been assigned a value.
- `'-'`: stands for don't-care.

Among these values, only `'0'`, `'1'` and `'Z'` are used in synthesis. The `'L'` and `'H'` values are seldom used now since current design practice rarely utilizes a wired-logic circuit. The use of `'Z'` and the potential problem of `'-'` are discussed in Chapter 6.

A VHDL array is defined as a collection of elements with the same data type. Each element in the array is identified by an index. The `std_logic_vector` data type is an array of elements with the `std_logic` data type. It can be thought of as a group of signals or a bus in a logic circuit.

The use of `std_logic_vector` can best be explained by a simple example. Let us consider an 8-bit signal, `a`. Its declaration is

```
signal a: std_logic_vector(7 downto 0);
```

It indicates that the `a` signal has 8 bits, which are indexed from 7 down to 0. The most significant bit (MSB, the leftmost bit) has the index 7, and the least significant bit (LSB, the rightmost bit) has the index 0. We can access a single bit by using an index, such as `a(7)` or `a(2)`, and access a portion of the index by using a range, such as `a(7 downto 3)` or `a(2 downto 0)`.

Another form of `std_logic_vector` is using an ascending range, as in

```
signal a: std_logic_vector(0 to 7);
```

Since its MSB is associated with index 0 and may cause some confusion if the array is interpreted as a binary number, we don't use this form in the book.

Overloaded operators Recall that the definition of a data type includes a set of values and a set of operations to be performed on this data type. In VHDL, we can use the same function or operator name for operands of different data types. There may exist multiple functions with the same name, each for a different data type. This is known as *overloading* of a function or operator.

In the `std_logic_1164` package, all logical operators, which include **not**, **and**, **nand**, **or**, **nor**, **xor** and **xnor**, are overloaded with the `std_logic` and `std_logic_vector` data types. In other words, we can perform the logical operations over the objects with the `std_logic` or `std_logic_vector` data types. The overloaded operators are summarized in Table 3.4. Note that the arithmetic operators are not overloaded, and thus these operations cannot be applied.

Type conversion The `std_logic_1164` package also defines several type conversion functions for conversion between the `bit` and `std_logic` data types as well as between the `bit_vector` and `std_logic_vector` data types. The relevant functions are summarized in Table 3.5.

Table 3.4 Overloaded operators in the IEEE std_logic_1164 package

Overloaded operator	Data type of operand a	Data type of operand b	Data type of result
not a	std_logic_vector std_logic		same as a
a **and** b a **or** b a **xor** b a **nand** b a **nor** b a **xnor** b	std_logic_vector std_logic	same as a	same as a

Table 3.5 Functions in the IEEE std_logic_1164 package

Function	Data type of operand a	Data type of result
to_bit(a)	std_logic	bit
to_stdulogic(a)	bit	std_logic
to_bitvector(a)	std_logic_vector	bit_vector
to_stdlogicvector(a)	bit_vector	std_logic_vector

Use of the conversion function is shown below. Assume that the s1, s2, s3, b1 and b2 signals are defined as

```
signal s1, s2, s3: std_logic_vector(7 downto 0);
signal b1, b2: bit_vector(7 downto 0);
```

The following statements are wrong because of data type mismatch:

```
s1 <= b1;          -- bit_vector assigned to std_logic_vector
b2 <= s1 and s2;   -- std_logic_vector assigned to bit_vector
s3 <= b1 or s2;    -- or is undefined between bit_vector
                   -- and std_logic_vector
```

We can use the conversion functions to correct these problems:

```
s1 <= to_stdlogicvector(b1);
b2 <= to_bitvector(s1 and s2);
s3 <= to_stdlogicvector(b1) or s2;
```

The last statement can also be written as

```
s3 <= to_stdlogicvector(b1 or to_bitvector(s2));
```

3.5.3 Operators over an array data type

Several operations are defined over the one-dimensional array data types in VHDL, including the concatenation and relational operators and the array aggregate. In this subsection, we demonstrate the use of these operators with the std_logic_vector data type. Note that these operators can be applied in any array data types, and thus no overloading is needed.

Relational operators for an array In VHDL, the relational operators can be applied to the one-dimensional array data type. The two operands must have the same element type, but their lengths may differ. When an operator is applied, the two arrays are compared element by element. The comparison procedure starts from the leftmost element and continues until a result can be established. If one array reaches the end before another, that array is considered to be "smaller" and the two arrays are considered to be not equal. For example, all following operations return `true`:

```
"011"="011", "011">"010", "011">"00010", "0110">"011"
```

Arrays with unequal lengths can sometimes introduce subtle, unexpected results. For example, assume that the `sig1` and `sig2` signals are with an array data type of different lengths and we accidentally write

```
if (sig1=sig2) then
. . .
else
. . .
```

Because of the different lengths, the comparison expression is always evaluated as false, and thus the then branch will never be taken. This kind of error is difficult to debug since the code is syntactically correct. In this book, we always use operands of identical length.

Concatenation operator The concatenation operator, `&`, is very useful for array manipulation. We can combine segments of elements and smaller arrays to form a larger array. For example, we can shift the elements of the array to the right by two positions and append two 0's to the front:

```
y <= "00" & a(7 downto 2);
```

or append the MSB to the front (known as an arithmetic shift):

```
y <= a(7) & a(7) & a(7 downto 2);
```

or rotate the elements to the right by two positions:

```
y <= a(1 downto 0) & a(7 downto 2);
```

Array aggregate *Array aggregate* is not an operator. It is a VHDL language construct to assign a value to an object of array data type. For the `std_logic_vector` data type, the simplest way to express an aggregate is to use a collection of `std_logic` values inside double quotation marks. For example, if we want to assign a value of `"10100000"` to the a signal, it can be written as

```
a <= "10100000";
```

Another way is to list each value of the element in the corresponding position, which is known as *positional association*. The previous assignment becomes

```
a <= ('1','0','1','0','0','0','0','0');
```

We can also use the form of `index => value` to explicitly specify the value for each index, known as *named association*. The statement can be written as

```
a <= (7=>'1', 6=>'0', 0=>'0', 1=>'0', 5=>'1',
      4=>'0', 3=>'0', 2=>'0');
```

It means that the value associated with index 7 (i.e., a(7)) is '1', the value associated with index 6 is '0', and so on. Note that the order of the index => value pairs does not matter. We can combine the index, as in

```
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
```

or use a reserved word, **others**, to cover all the unused indexes, as in

```
a <= (7|5=>'1', others=>'0');
```

One frequently encountered array aggregate is all 0's, which is used in the initialization of a counter or a memory element. For example, if we want to assign "00000000" to the a signal, we can write

```
a <= (others=>'0');
```

It is more compact than

```
a <= "00000000";
```

The code remains the same even when the width of the a signal is later revised.

3.5.4 Data types in the IEEE numeric_std package

In addition to logical operations, digital hardware frequently involves arithmetic operation as well. If we examine VHDL and the std_logic_1164 package, the arithmetic operations are defined only over the integer data type. To perform addition of the a and b signals, we must use the integer data type, as in

```
signal a, b, sum: integer;
. . .
sum <= a + b;
```

It is difficult to realize this statement in hardware since the code doesn't indicate the range (number of bits) of the a and b signals. Although this does not matter for simulation, it is important for synthesis since there is a huge difference between the hardware complexity of an 8-bit adder and that of a 32-bit adder.

A better alternative is to use an array of 0's and 1's and interpret it as an unsigned or signed number. We can define the width of the input and the size of the adder precisely, and thus have better control over the underlying hardware. The IEEE numeric_std package was developed for this purpose.

Signed and unsigned data types The IEEE numeric_std package is a part of IEEE standard 1176.3. Two new data types, signed and unsigned, are defined in the package. Both data types are an array of elements with the std_logic data type. For the unsigned data type, the array is interpreted as an unsigned binary number, with the leftmost element as the MSB of the binary number. For the signed data type, the array is interpreted as a signed binary number in 2's-complement format. The leftmost element is the MSB of the binary number, which represents the sign of the number.

Note that the std_logic_vector, unsigned and signed data types are all defined as an array of elements with the std_logic data type. Since VHDL is a strongly typed language, they are considered as three independent data types. It is reasonable since the three data types are interpreted differently. For example, consider a 4-bit binary representation "1100". It represents the number 12 if it is interpreted as an unsigned number and represents the number

−4 if it is interpreted as a signed number. It may also just represent four independent bits (e.g., four status signals) if it is interpreted as a collection of bits.

Since the `signed` and `unsigned` data types are arrays, their declarations are similar to that of the `std_logic_vector` data type, as in

```
signal x, y: signed(15 downto 0);
```

To use the `signed` and `unsigned` data types, we must include the library statement before the entity declaration. Furthermore, we must include the `std_logic_1164` package since the `std_logic` data type is used in the `numeric_std` package. These statements are

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Overloaded operators Since the goal of the `numeric_std` package is to support the arithmetic operations, the relevant arithmetic operators, which include **abs**, *, /, **mod**, **rem**, + and -, are overloaded. These operators can now take two operands, with data types `unsigned` and `unsigned`, `unsigned` and `natural`, `signed` and `signed` as well as `signed` and `integer`. For example, the following are valid assignment statements:

```
signal a, b, c, d: unsigned(7 downto 0);
. . .
a <= b + c;
d <= b + 1;
e <= (5 + a + b) - c;
```

The overloading definition of addition and subtraction follows the model of a physical adder. The sum automatically "wraps around" when overflow occurs.

The relational operators, which include =, /=, <, >, <= and >=, are also overloaded. The overloading serves two purposes. First, it makes the operator take two operands with data types `unsigned` and `natural` as well as `signed` and `integer`. Second, for two operands with the `unsigned` or `signed` data types, the overloading overrides the original left-to-right element-by-element comparison procedure and treats the two arrays as two binary numbers. For example, consider the expression `"011" > "1000"`. If the data type of the two operands is `std_logic_vector`, the expression returns `false` because the first element of `"011"` is smaller than the first element of `"1000"`. If the data type of the two operands is `unsigned`, the > operator is overloaded and the two operands are interpreted as 3 and 8 respectively. The expression returns `false` again. However, if the data type is `signed`, they are interpreted as 3 and −8, and thus the expression returns `true`.

A summary of the overloaded operators is given in Table 3.6.

Functions The `numeric_std` package defines several new functions. The new functions include:

- `shift_left`, `shift_right`, `rotate_left`, `rotate_right`: used for shifting and rotating operations. Note that these are new functions, not the overloaded VHDL operators.
- `resize`: used to convert an array to different sizes.
- `std_match`: used to compare objects with the `'-'` value.
- `to_unsigned`, `to_signed`, `to_integer`: used to do type conversion between the two new data types and the `integer` data type.

Table 3.6 Overloaded operators in the IEEE `numeric_std` package

Overloaded operator	Description	Data type of operand `a`	Data type of operand `b`	Data type of result
abs `a` `- a`	absolute value negation	`signed`		`signed`

`a * b`
`a / b`
`a` **mod** `b`
`a` **rem** `b`
`a + b`
`a - b`
	arithmetic operation	`unsigned`
		`unsigned, natural`
		`signed`
		`signed, integer`
			`unsigned, natural`
			`unsigned`
			`signed, integer`
			`signed`
				`unsigned`
				`unsigned`
				`signed`
				`signed`

`a = b`
`a /= b`
`a < b`
`a <= b`
`a > b`
`a >= b`
	relational operation	`unsigned`
		`unsigned, natural`
		`signed`
		`signed, integer`
			`unsigned, natural`
			`unsigned`
			`signed, integer`
			`signed`
				`boolean`
				`boolean`
				`boolean`
				`boolean`

Table 3.7 Functions in the IEEE `numeric_std` package

Function	Description	Data type of operand `a`	Data type of operand `b`	Data type of result

`shift_left(a,b)`
`shift_right(a,b)`
`rotate_left(a,b)`
`rotate_right(a,b)`
	shift left
	shift right
	rotate left
	rotate right
		`unsigned, signed`	`natural`	same as `a`

`resize(a,b)`	resize array	`unsigned, signed`	`natural`	same as `a`

`std_match(a,b)`	compare '-'	`unsigned, signed`
		`std_logic_vector,`
		`std_logic`
			same as `a`	`boolean`

`to_integer(a)`	data type conversion	`unsigned, signed`		`integer`
`to_unsigned(a,b)`		`natural`	`natural`	`unsigned`
`to_signed(a,b)`		`integer`	`natural`	`signed`

The functions are summarized in Table 3.7. The shift functions are similar to the VHDL shift operators but with different data types. Note that the IEEE 1076.6 RTL synthesis standard supports the shift functions of the `numeric_std` package but not the shift operators of VHDL. The synthesis issues of the shift functions and the use of the `std_match` function are discussed in Chapter 6.

Type conversion Conversion between two different data types can be done by a *type conversion function* or *type casting*. There are three type conversion functions in the `numeric_std` package: `to_unsigned`, `to_signed`, and `to_integer`. The `to_intger` function takes an object with an `unsigned` or `signed` data type and converts it to the `integer` data type. The `to_unsigned` and `to_signed` functions convert an integer into an object with the `unsigned` or `signed` data type of a specific number of bits. It takes

Table 3.8 Type conversions of numeric data types

Data type of a	To data type	Conversion function/type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

two parameters. The first is the integer number to be converted, and the other specifies the desired number of bits (or size) in the new unsigned or signed data type.

The std_logic_vector, unsigned and signed data types are all defined as an array with elements of the std_logic data type. They are known as *closely related data types* in VHDL. Conversion between these types is done by a procedure known as *type casting*. To do type casting, we simply put the original object inside parentheses prefixed by the new data type. This can best be explained by an example:

```
signal u1, u2: unsigned(7 downto 0);
signal v1, v2: std_logic_vector(7 downto 0);
. . .
u1 <= unsigned(v1)
v2 <= std_logic_vector(u2);
```

Table 3.8 summarizes all the type conversions in the numeric_std package. Note that the std_logic_vector data type is not interpreted as a number and thus cannot be directly converted to an integer and vice versa.

Type conversion between various numeric data types is frequently confusing to new VHDL users. The following examples of signal assignment statements demonstrate and clarify the use of these data types and data conversions. Assume that some signals are declared as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
. . .
```

The following assignments to the signals u3 and u4 are valid since the + operator is overloaded with the unsigned and natural types:

```
u3 <= u2 + u1;  -- ok, both operands unsigned
u4 <= u2 + 1;   -- ok, operands unsigned and natural
```

On the other hand, the following two assignments are invalid due to type mismatch:

```
u5 <= sg;  -- not ok, type mismatch
u6 <= 5;   -- not ok, type mismatch
```

We must use type casting and the conversion function to covert the expressions to the proper type:

```
u5 <= unsigned(sg);       -- ok, type casting
u6 <= to_unsigned(5,4); -- ok, conversion function
```

The arithmetic operators are not overloaded with the mixed data types `signed` and `unsigned`, and thus the following statement is invalid:

```
u7 <= sg + u1;    -- not ok, + undefined over the types
```

We must convert the data type of the operand as follows:

```
u7 <= unsigned(sg) + u1; -- ok, but be careful
```

We need to be aware of the different interpretations of the `signed` and `unsigned` types. For example, `"1111"` is −1 for the `signed` type but is 15 for the `unsigned` type. This kind conversion should proceed with care.

Two assignments for signals with `std_logic_vector` data type are

```
s3 <= u3;   -- not ok, type mismatch
s4 <= 5;    -- not ok, type mismatch
```

Both of them are invalid because of type mismatch. We must use type casting and a conversion function to correct the problem:

```
s3 <= std_logic_vector(u3); -- ok, type casting
s4 <= std_logic_vector(to_unsigned(5,4)); -- ok
```

Note that two type conversions are needed for the second statement.

Arithmetic operations cannot be applied to the `std_logic_vector` data type since no overloading is defined for this type. Thus, the following statements are invalid:

```
s5 <= s2 + s1; -- not ok, + undefined over the types
s6 <= s2 + 1;  -- not ok, + undefined over the types
```

To fix the problem, we must convert the operands to the `unsigned` (or `signed`) data type, perform addition, and then convert the result back to the `std_logic_vector` data type. The revised code becomes

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ok
s6 <= std_logic_vector(unsigned(s2) + 1);            -- ok
```

3.5.5 The std_logic_arith and related packages

For historical reasons, several packages similar to the IEEE `numeric_std` package are used in some EDA software and existing VHDL codes. The packages are:

- `std_logic_arith`
- `std_logic_unsigned`
- `std_logic_signed`

They are not a part of the IEEE standards, but many software vendors store these packages in the `ieee` library. They can be invoked by

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
. . .
```

Because of the use of the ieee term, these packages sometimes cause confusion. They are not used them in this book. For reference, we explain briefly the use of these packages in this subsection.

The purpose of the std_logic_arith package is similar to that of the numeric_std package. It defines two new data types, unsigned and signed, and overloads the +, - and * operators with these data types. The package also includes similar shifting, sizing and type conversion functions although the names of these functions are different.

Instead of defining new data types, the std_logic_unsigned and std_logic_signed packages define overloaded arithmetic operators for the std_logic_vector data type. In other words, the std_logic_vector data type is interpreted as unsigned and signed binary numbers in the std_logic_unsigned and std_logic_signed packages respectively. The two packages clearly cannot be used at the same time.

With one of the packages, the previous code segments becomes valid and no type conversion is needed:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
. . .
s5 <= s2 + s1; -- ok, + overloaded with std_logic_vector
s6 <= s2 + 1;  -- ok, + overloaded with std_logic_vector
```

The overloading means that we can treat the std_logic_vector data type as "a collection of bits" as well as an unsigned binary number. This package actually beats the motivation behind a strongly typed language. The IEEE 1076.6 RTL synthesis standard states explicitly that the unsigned and signed data types defined in IEEE 1076.3 are the only array types that can be used to represent unsigned and signed numbers.

3.6 SYNTHESIS GUIDELINES

In this and subsequent chapters, we summarize the good design and coding practices mentioned in the chapter and present them as a set of guidelines at the end of the chapter. Since the book focuses on synthesis, these guidelines are applied only to synthesis, not to general modeling or simulation. These suggested guidelines help us to avoid some common mistakes and to increase the compatibility, portability and efficiency of VHDL codes.

3.6.1 Guidelines for general VHDL

- Use the std_logic_vector and std_logic data types instead of the bit_vector or bit data types.
- Use the numeric_std package and the unsigned and signed data types for synthesizing arithmetic operations.
- Use only the descending range (i.e., **downto**) in the array specification of the unsigned, signed and std_logic_vector data types.
- Use parentheses to clarify the intended order of evaluation.

- Don't use user-defined data types unless there is a compelling reason.
- Don't use immediate assignment (i.e., :=) to assign an initial value to a signal.
- Use operands with identical lengths for the relational operators.

3.6.2 Guidelines for VHDL formatting

- Include an information header for each file.
- Be consistent with the use of case.
- Use proper spaces, blank lines and indentations to make the code clear.
- Add necessary comments.
- Use symbolic constant names to replace hard literals in VHDL code.
- Use meaningful names for the identifiers.
- Use a suffix to indicate a signal's special property, such as _n for the active-low signal.
- Keep the line width within 72 characters so that the code can be displayed and printed properly by various editors and printers without wrapping.

3.7 BIBLIOGRAPHIC NOTES

VHDL is a complex language. It is formally specified by IEEE standard 1076. The most recent version, VHDL-2001, is specified by IEEE standard 1076-2001, and VHDL-87 is specified by IEEE standard 1076-1987. The standard is documented in *IEEE Standard VHDL Language Reference Manual*, which sometimes known simply as *LRM*. Since *LRM* gives the formal definition of VHDL, it is difficult to read. The book, *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden, provides a detailed and comprehensive discussion of the VHDL language. It has several chapters on basic VHDL concepts, data types and alias. The book, *VHDL for Logic Synthesis* by A. Rushton, has a chapter on `numeric_std` package and provides a detailed discussion on functions.

After synthesis software is installed, we can normally find the files that contain the source codes of IEEE `std_logic_1164` and `numeric_std` packages as well as `std_logic_arith`, `std_logic_unsigned` and `std_logic_signed` packages. These packages provide detailed information about operator overloading and function definitions.

Although formatting is not real design, good coding style and documentation are essential for a project, especially for a large project that involves many design teams. Many organizations set and enforce their own coding and documentation standards. An example is *VHDL Modeling Guideline* from the European Space Agency.

The text, *Reuse Methodology Manual* by M. Keating and P. Bricaud, also provides some rules and guidelines for the use and formatting of VHDL.

Problems

3.1 Write an entity declaration for a memory circuit whose input and output ports are shown below. Use only the `std_logic` or `std_logic_vector` data types.

- addr: 12-bit address input
- wr_n: 1-bit write-enable control signal
- oe_n: 1-bit output-enable control signal
- data: 8-bit bidirectional data bus

3.2 What is the difference between a variable and a signal?

3.3 What is a strongly typed language?

3.4 What is the limitation of using the `bit` data type to represent a physical signal?

3.5 Assume that a is a 10-bit signal with the `std_logic_vector(9 downto 0)` data type. List the 10 bits assigned to the a signal.

(a)
```
a <= (others=>'1');
```
(b)
```
a <= (1|3|5|7|9=>'1', others=>'0');
```
(c)
```
a <= (9|7|2=>'1', 6=>'0', 0=>'1', 1|5|8=>'0', 3|4=>'0');
```

3.6 Assume that a and y are 8-bit signals with the `std_logic_vector(7 downto 0)` data type. If the signals are interpreted as unsigned numbers, the following assignment statement performs a / 8. Explain.

```
y <= "000" & a(7 downto 3);
```

3.7 Assume the same a and y signals in Problem 3.6. We want to perform a **mod** 8 and assign the result to y. Rewrite the previous signal assignment statement using only the & operator.

3.8 Assume that the following double-quoted strings are with the `std_logic_vector` data type. Determine whether the relational operation is syntactically correct. If yes, what is the result (i.e., `true` or `false`)?

(a) `"0110" > "1001"`

(b) `"0110" > "0001001"`

(c) `2#1010# > "1010"`

(d) `1010 > "1010"`

3.9 Repeat Problem 3.8, but assume that the data type is `unsigned`.

3.10 Repeat Problem 3.8, but assume that the data type is `signed`.

3.11 Determine whether the following signal assignment is syntactically correct. If not, use the proper conversion function and type casting to correct the problem.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6, s7: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
. . .
u1 <= 2#0001#;
u2 <= u3 and u4;
u5 <= s1 + 1;
u6 <= u3 + u4 + 3;
u7 <= (others=>'1');
```

```
s2 <= s3 + s4 -1;
s5 <= (others=>'1');
s6 <= u3 and u4;
sg <= u3 - 1;
s7 <= not sg;
```

3.12 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= shift_left(src, amount);
```

3.13 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= src sll amount;
```

3.14 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= src sll amount;
```

CHAPTER 4

CONCURRENT SIGNAL ASSIGNMENT STATEMENTS OF VHDL

Concurrent signal assignment statements are simple, yet powerful VHDL statements. Since there is a clear mapping between the language constructs of an assignment statement and hardware components, we can easily visualize the conceptual diagram of the VHDL description. This helps us to develop a more efficient design. According to the VHDL definition, concurrent signal assignment statement has two basic forms: the *conditional signal assignment statement* and the *selected signal assignment statement*. For discussion purposes, we add an additional one, the *simple signal assignment statement*, which is a conditional assignment statement without any condition expression.

4.1 COMBINATIONAL VERSUS SEQUENTIAL CIRCUITS

A digital circuit can be broadly classified as combinational or sequential. A *combinational circuit* has no internal memory or state and its output is *a function of inputs only*. Thus, the same input values will always produce an identical output value. In a real circuit, the output may experience a short transient period after an input signal changes. However, the identical output value will be obtained when the signal is stabilized. In term of implementation, a combinational circuit is a circuit without memory elements (latches or flip-flops) or a closed feedback loop. A *sequential circuit*, on the other hand, has an internal state, and its output is *a function of inputs as well as the internal state*.

Although concurrent signal assignment statements can be used to describe sequential circuits, this is not the preferred method. We limit the discussion to combinational circuits

in this chapter. We use the VHDL *process* to specify sequential circuits and study them in Chapter 8.

4.2 SIMPLE SIGNAL ASSIGNMENT STATEMENT

4.2.1 Syntax and examples

A simple signal assignment statement is a conditional signal assignment statement without the condition expression and thus is a special case of a conditional signal assignment statement. In VHDL definition, the simplified syntax of the simple signal assignment statement can be written as

```
signal_name <= projected_waveform;
```

The `projected_waveform` clause consists of two kinds of specifications: the expression of a new value for the signal and the time when the new value takes place. For example, consider the statement

```
y <= a + b + 1 after 10 ns;
```

which indicates that whenever the `a` or `b` signal changes, the expression `a+b+1` will be evaluated, and its result will be assigned to the `y` signal after 10 ns.

The time aspect of `projected_waveform` normally corresponds to the internal propagation delay to complete the computation of the expression. However, since the propagation delay depends on the components, device technology, routing, fabrication process and operation environment, it is impossible to synthesize a circuit with an exact amount of delay. Therefore, for synthesis, explicit timing information is not specified in VHDL code. The default δ-delay is used in the projected waveform. The syntax becomes

```
signal_name <= value_expression;
```

The `value_expression` clause can be a constant value, logical operation, arithmetic operation and so on. Following are a few examples:

```
status <= '1';
even <= (p1 and p2) or (p3 and p4);
arith_out <= a + b + c - 1;
```

Note that the timing aspect is not dropped. It is just specified implicitly as a δ-delay. The previous statements implicitly imply

```
status <= '1' after δ;
even <= (p1 and p2) or (p3 and p4) after δ;
arith_out <= a + b + c - 1 after δ;
```

4.2.2 Conceptual implementation

Deriving the conceptual hardware block diagram for a simple signal assignment statement is straightforward. The entire statement can be thought of as a circuit block. The output of the circuit is the signal in the left-hand side of the statement, and the inputs are all the signals that appear in the right-hand-side value expression. We then map each operator of the value expression into a smaller circuit block and connect their inputs and outputs accordingly. The conceptual diagrams of three previous statements are shown in Figure 4.1.

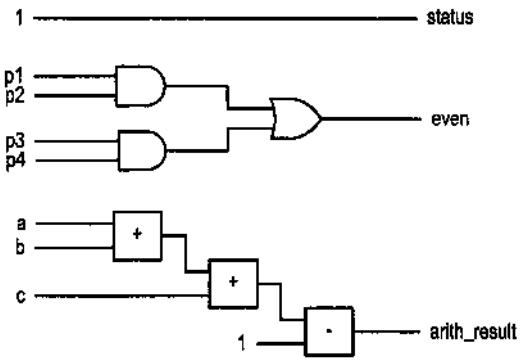


Figure 4.1 Conceptual diagrams of three simple signal assignment statements.

Note that these diagrams are only conceptual sketches. They will be transformed and simplified during synthesis. The circuit sizes of different VHDL operators vary significantly, and some of them, like the division operator, cannot be synthesized automatically. We examine this issue in detail in Chapter 6.

4.2.3 Signal assignment statement with a closed feedback loop

According to VHDL definition, it is syntactically correct for a signal to appear on both sides of a concurrent signal assignment statement. When an output signal is used as an input in the value expression, a closed feedback loop is formed. This may lead to the creation of an internal state or even oscillation. Consider the following VHDL statement:

```
q <= (q and (not en)) or (d and en);
```

In this example, the q signal is the output but also appears in the right-hand-side expression. The q output takes the value of the d signal if the en signal is '1' and it keeps its previous value if the en signal is '0'. Note that the output (i.e., q) now depends on input (i.e., en and d) as well as internal state (the previous value of q), and thus the circuit is no longer a combinational circuit. If we modify the previous statement by inverting q:

```
q <= ((not q) and (not en)) or (d and en);
```

the q output oscillates between '0' and '1' when the en signal is '0'.

When a signal assignment statement contains a closed feedback loop, it becomes sensitive to internal propagation delay and may exhibit race or oscillation. This kind of circuit confuses synthesis software and complicates verification and testing processes. It is a really bad coding practice and should be avoided completely in VHDL synthesis. The shortfall of delay-sensitive design and the disciplined derivation of sequential circuits are discussed in detail in Chapters 8 and 9.

Table 4.1 Function table of a 4-to-1 multiplexer

Input s	Output x
0 0	a
0 1	b
1 0	c
1 1	d

4.3 CONDITIONAL SIGNAL ASSIGNMENT STATEMENT

4.3.1 Syntax and examples

The simplified syntax of conditional signal assignment statement is shown below. As in Section 4.2.2, we assume that a timing specification is embedded implicitly in δ-delay and use `value_expression` to substitute the `projected_waveform` clause:

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
                  . . .
               value_expr_n;
```

The `boolean_expr_i` (i= 1, 2, 3, . . ., n) term is a Boolean expression that returns `true` or `false`. These Boolean expressions are evaluated successively in turn until one is found to be `true`, and the corresponding value expression is assigned to the output signal. In other words, the first Boolean expression, `boolean_expr_1`, is checked first. If it is `true`, the first value expression, `value_expr_1`, will be assigned to the output signal. If it is `false`, the second Boolean expression, `boolean_expr_2`, will be checked next. This process continues until all Boolean expressions are checked. The last value expression, `value_expr_n`, will be assigned to the signal if none of the Boolean expressions is `true`.

In the remaining subsection, we use several simple examples to illustrate the use of conditional signal assignment statements. The circuits include a multiplexer, a decoder, a priority encoder and a simple arithmetic logic unit (ALU).

Multiplexer A multiplexer is essentially a virtual switch that routes a selected input signal to the output. The function table of an 8-bit 4-to-1 multiplexer is show in Table 4.1. In this circuit, the `a`, `b`, `c` and `d` signals can be considered as input data, and the `s` signal is a 2-bit selection signal that specifies which input data will be routed to the output. The VHDL code for this circuit is shown in Listing 4.1.

Listing 4.1 4-to-1 multiplexer based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
   port(
      a,b,c,d: in std_logic_vector(7 downto 0);
      s: in std_logic_vector(1 downto 0);
      x: out std_logic_vector(7 downto 0)
   );
```

```
  end mux4;
10
  architecture cond_arch of mux4 is
  begin
     x <= a when (s="00") else
          b when (s="01") else
15        c when (s="10") else
          d;
  end cond_arch;
```

The first two lines are used to invoke the IEEE `std_logic_1164` package so that the `std_logic` data type can be used. The next part is the entity declaration, which specifies the input and output ports of this circuit. The input ports include `a`, `b`, `c` and `d`, which are four 8-bit input data, and `s`, which is the 2-bit control signal. The output port is the 8-bit `x` signal. The architecture part uses one conditional signal assignment statement. The Boolean condition `s="00"` is evaluated first. If it is `true`, the first value expression, `a`, is assigned to `x`. If it is `false`, the next Boolean condition, `s="01"`, will be evaluated. If it is `true`, `b` is assigned to `x` or the next Boolean expression, `s="10"`, will be evaluated. If all three Boolean expressions are `false`, the last value expression, `d`, is assigned to `x`.

There is an issue about the use of the `std_logic` data type. At first glance, it seems that `s` is implied to be `"11"` when the first three Boolean expressions are `false`, and thus `d` is assigned to `x`. However, there are nine possible values in `std_logic` data type and, for the 2-bit `s` signal, there are 81 (i.e., 9*9) possible combinations, including the expected `"00"`, `"01"`, `"10"` and `"11"` as well as the metavalue combinations, such as `"0Z"`, `"UX"`, `"0-"` and so on. Therefore, `d` is assigned to `x` for the `"11"` condition, as well as other 77 metavalue combinations. However, these 77 combinations can exist only in simulation. In a real circuit, comparison of metavalues, as in `s="0Z"`, cannot be implemented, and sometimes is meaningless, as in `s="UX"`. In general, except for the limited use of `'Z'`, the metavalues of the `std_logic` data type will be ignored by synthesis software, and thus the final circuit will be synthesized as we originally expected. Some synthesis software also accepts VHDL code using `'X'` for the unused metavalue combinations:

```
x <= a when (s="00") else
     b when (s="01") else
     c when (s="10") else
     d when (s="11") else
     'X';
```

The code leads to the same physical implementation.

Binary decoder A binary decoder is an n-to-2^n decoder, which has an n-bit input and a 2^n-bit output. Each bit of the output represents an input combination. Based on the value of the input, the circuit activates the corresponding output bit. The function table of a simple 2-to-2^2 decoder is shown in Table 4.2. The VHDL code for this circuit is shown in Listing 4.2.

Listing 4.2 2-to-2^2 binary decoder based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder4 is
   port(
5     s: in  std_logic_vector(1 downto 0);
```

Table 4.2 Function table of a 2-to-2^2 binary decoder

Input s	Output x
0 0	0001
0 1	0010
1 0	0100
1 1	1000

Table 4.3 Function table of a 4-to-2 priority encoder

Input r	Output code	active
1 – – –	11	1
0 1 – –	10	1
0 0 1 –	01	1
0 0 0 1	00	1
0 0 0 0	00	0

```
        x: out std_logic_vector(3 downto 0)
    );
end decoder4;

architecture cond_arch of decoder4 is
begin
    x <= "0001" when (s="00") else
         "0010" when (s="01") else
         "0100" when (s="10") else
         "1000";
end cond_arch;
```

Again, the first two lines are used to invoke the IEEE `std_logic_1164` package. The entity declaration shows the circuit with a 2-bit input, a, and a 4-bit output, x. The architecture body uses one conditional signal assignment statement, which evaluates the Boolean conditions `s="00"`, `s="01"` and `s="10"` one after another. The value expressions are constants that reflect the desired output patterns.

Priority encoder A priority encoder checks the input requests and generates the code of the request with highest priority. The function table of a 4-to-2 priority encoder is shown in Table 4.3. There are four input requests, `r(3)`, `r(2)`, `r(1)` and `r(0)`. The outputs include a 2-bit signal, `code`, which is the binary code of the highest-priority request, and a 1-bit signal, `active`, which indicates whether there is an active request. The `r(3)` request has the highest priority. When it is asserted, the other three requests are ignored and the `code` signal becomes `"11"`. If `r(3)` is not asserted, the second highest request, `r(2)`, is examined. If it is asserted, the `code` signal becomes `"10"`. The process repeats until all the requests are checked. The `code` signal returns `"00"` when only `r(0)` is asserted or no request is asserted. The `active` signal can be used to distinguish the two conditions. The VHDL code for this circuit is shown in Listing 4.3. The requests are grouped together and

Table 4.4 Function table of a simple ALU

Input ctrl	Output result
0--	src0 + 1
100	src0 + src1
101	src0 - src1
110	src0 **and** src1
111	src0 **or** src1

represented by a 4-bit signal, r. Individual bits of the r signal are checked in descending order, starting with r(3). Since operation of the priority encoder is similar to the definition of the conditional signal assignment statement, it is a good way to code this type of circuit (note the simple Boolean expressions in the code). A separate simple signal assignment statement is used to describe the active output.

Listing 4.3 4-to-2 priority encoder based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity prio_encoder42 is
   port(
      r: in std_logic_vector(3 downto 0);
      code: out std_logic_vector(1 downto 0);
      active: out std_logic
   );
end prio_encoder42;

architecture cond_arch of prio_encoder42 is
begin
   code <= "11" when (r(3)='1') else
           "10" when (r(2)='1') else
           "01" when (r(1)='1') else
           "00";
   active <= r(3) or r(2) or r(1) or r(0);
end cond_arch;
```

Simple ALU An ALU performs a set of arithmetic and logical operations. The function table of a simple ALU is shown in Table 4.4. The inputs include two 8-bit data sources, scr0 and src1, and a control signal, ctrl, which specifies the function to be performed. The output is the 8-bit result signal, which is the computed result. There are five functions, including three arithmetic operations, which are incrementing, addition and subtraction, and two logical operations, which are bitwise and and or operations. Furthermore, we assume that the input and output are interpreted as signed integers when an arithmetic function is selected.

For this circuit, the input data are interpreted as a collection of bits for the logical operation and as a signed number for the arithmetic operation. To achieve better portability, we normally use the std_logic_vector data type in the port declaration and then convert it to the desired data type in architecture body. The VHDL code is shown in Listing 4.4. The

IEEE numeric_std package and its signed data type are used to facilitate the arithmetic operation. When an addition, subtraction or incrementing operation is specified, we first convert the input to the signed data type, perform the operation and then convert the result back to the std_logic_vector data type. To make the code clear, we introduce three separate simple signal assignment statements and the sum, diff, and inc signals for the intermediate results of arithmetic operations.

Listing 4.4 Simple ALU based on a conditional signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity simple_alu is
   port(
      ctrl: in  std_logic_vector(2 downto 0);
      src0, src1: in std_logic_vector(7 downto 0);
      result: out std_logic_vector(7 downto 0)
   );
end simple_alu;

architecture cond_arch of simple_alu is
   signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
   inc <= std_logic_vector(signed(src0)+1);
   sum <= std_logic_vector(signed(src0)+signed(src1));
   diff <= std_logic_vector(signed(src0)-signed(src1));
   result <= inc  when ctrl(2)='0' else
             sum  when ctrl(1 downto 0)="00" else
             diff when ctrl(1 downto 0)="01" else
             src0 and src1 when ctrl(1 downto 0)="10" else
             src0 or src1;
end cond_arch;
```

4.3.2 Conceptual implementation

Recall that the syntax of the simplified conditional signal assignment statement is

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
               value_expr_3 when boolean_expr_3 else
                  . . .
               value_expr_n;
```

Its semantics specifies that the Boolean expressions are evaluated in descending order until a condition is true, and then the corresponding value expression is assigned to the output signal. The key to implementing this construct is to achieve the desired descending order of evaluations. In a traditional programming language, descending order is implicitly observed because of the sequential execution of a single, shared CPU. In synthesis, we must use hardware to achieve this task.

The structure of conditional signal assignment statement implies a priority routing network since the Boolean expressions are evaluated in an orderly manner and the one evaluated earlier assumes a higher priority. Once the evaluation of a Boolean expression is true, the

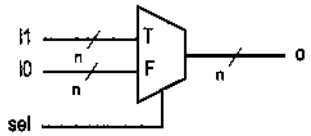


Figure 4.2 Conceptual diagram of an abstract multiplexer.

result of the corresponding value expression is routed to output. Unlike the *temporal* execution of the traditional programming language, the priority routing network is done on a *spatial* basis. Furthermore, since we cannot create hardware dynamically, dedicated hardware is needed for each Boolean expression and each value expression.

In summary, constructing the conditional signal assignment statement requires three groups of hardware:

- Value expression circuits
- Boolean expression circuits
- Priority routing network

Value expression circuits realize the value expressions, `value_expr_1`, $\cdots$, `value_expr_n`, and one of the results is routed to the output. *Boolean expression circuits* realize the Boolean expressions, `boolean_expr_1`, $\cdots$, `boolean_expr_n`, and their values are used to control the priority routing network. The *priority routing network* is the structure that routes and controls the desired value to the output signal.

A priority network can be implemented by a sequence of 2-to-1 multiplexers. To better illustrate the conceptual implementation, we utilize an "abstract multiplexer." Recall that a multiplexer is like a switch and uses a selection signal to select an input port and connect it to the output port. Any signal appearing in that input port will be routed to the output port. In an abstract multiplexer, the selection and input port designation are specified around the data type of the selection signal. Each input port is designated to a value of the data type of the selection signal, and one input port is selected according to the current value of the selection signal. For example, if the selection signal has a data type of `boolean`, there will be two input ports, designated as `T` (for `true`) and `F` (for `false`) respectively. If the selection signal has a value of `true`, the data from the `T` port will be routed to output. On the other hand, if the selection signal has a value of `false`, the data from the `F` port will be selected. The block diagram of this multiplexer is shown in Figure 4.2. The number of bits of the inputs and output may vary, and the symbol, `n`, is used to designate the width of the buses. During synthesis, the symbolic values can easily be mapped into binary representations of a physical multiplexer.

With the 2-to-1 abstract multiplexer, we can start to construct a priority network. Let us first consider a simple conditional signal assignment statement that has only one when clause:

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2;
```

The conceptual realization of this statement is shown in Figure 4.3. The three "clouds" represent the implementations of `value_expr_1`, `value_expr_2` and `boolean_expr_1` respectively. The result of `boolean_expr_1` is connected to the selection signal of the multiplexer. If it is `true`, the result from `value_expr_1` will be routed to the output port of the multiplexer. Otherwise, the result from `value_expr_2` will be routed to the output port.

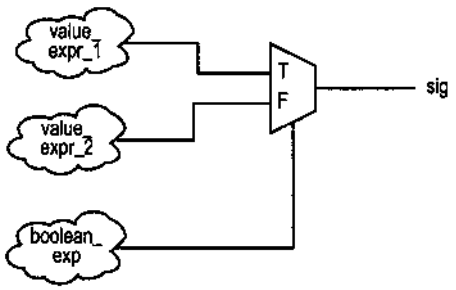


Figure 4.3 Conceptual diagram of a simple conditional signal assignment statement.

When there are more when clauses, we can perform the previous process repetitively and build the routing network in stages. Consider a statement with three when clauses:

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2 when boolean_expr_2 else
       value_expr_3 when boolean_expr_3 else
       value_expr_4;
```

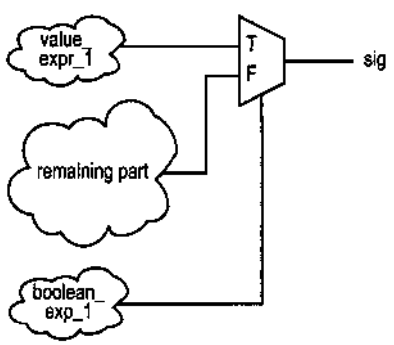
The construction sequence is shown in Figure 4.4. First we construct the first when clause, which corresponds to the highest-priority condition. If the result of `boolean_expr_1` is `true`, the result of the corresponding value expression, `value_expr_1`, is routed to output, as shown in Figure 4.4(a). On the other hand, if the result of `boolean_expr_1` is `false`, the result from the remaining part of the statement, which is shown as a single cloud, will be used. This cloud can be constructed using a multiplexer similar to the first when clause, with its output connected to the F port of the rightmost multiplexer, as shown in Figure 4.4(b). After repeating this process one more time, we construct the third when clause and complete the conceptual implementation, as shown in Figure 4.4(c).

The construction process can be applied repeatedly to any number of when clauses. Since each clause will introduce one extra stage of multiplexer network, the depth of the network grows as the number of clauses increases. Although the conceptual construction is straightforward, it is difficult for synthesis software to transform an extremely deep multiplexer network to an efficient implementation. Thus, we should be aware of the impact on the number of when clauses. Discussion in Chapter 6 provides more insight on this issue.

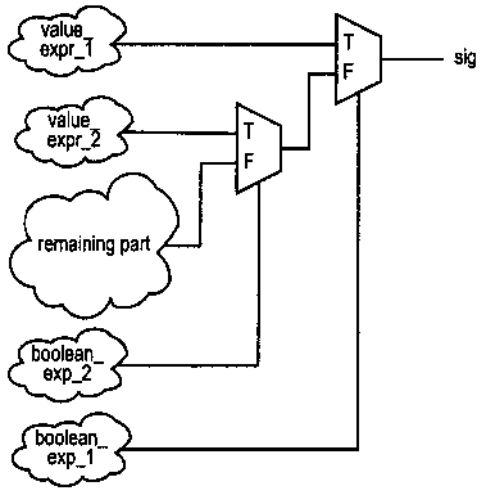
4.3.3 Detailed implementation examples

Obtaining the conceptual diagram is only the first step in synthesis. We must derive the more detailed implementation for the multiplexers and "clouds" and eventually construct everything by using cells of the given technology library. Many of these tasks can be done in synthesis software, which is discussed in Chapter 6. In this section, we manually derive some simple circuits from VHDL segments to illustrate the basic synthesis process.

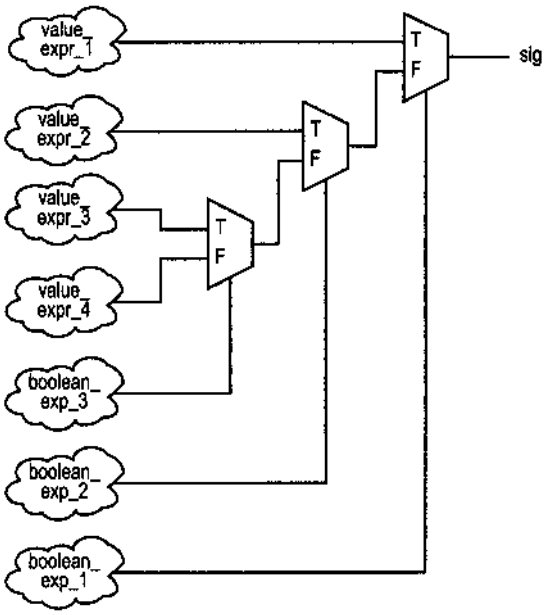
Implementation of a 2-to-1 multiplexer An abstract 2-to-1 multiplexer has two symbolic ports, T and F. We can map it directly to a regular 2-to-1 multiplexer. The schematic of a 1-bit 2-to-1 multiplexer is shown in Figure 4.5(a). The two abstract ports, T and F, are mapped to the $i1$ and $i0$ ports respectively. In this circuit, the and cells can be interpreted as "passing gates," controlled by separate enable signals. When the enable signal is '1',



(a) First when clause

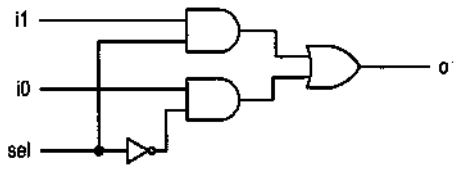


(b) Second when clause

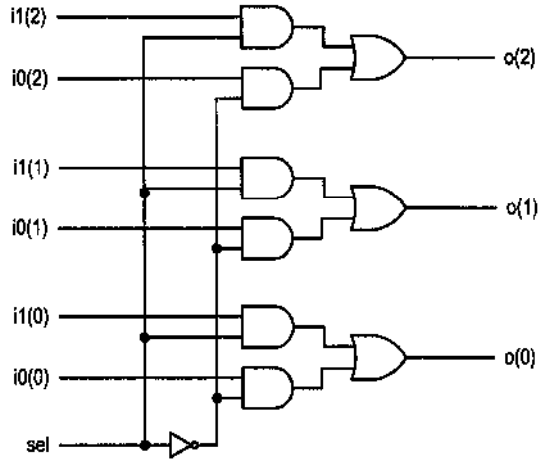


(c) Third when clause

Figure 4.4 Construction of a multi-condition conditional signal assignment statement.



(a) 1-bit 2-to-1 multiplexer



(b) 3-bit 2-to-1 multiplexer

Figure 4.5 Gate-level implementation of a multiplexer.

the gate is open and the input signal is passed to output. When it is '0', the gate is closed and the output is set to '0'. The two enable signals are *sel* and *sel'* respectively, and thus one of the inputs will be passed to output. In terms of a logic expression, the output can be expressed as

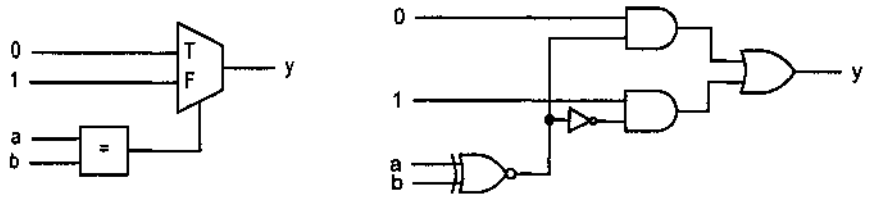
$$o = sel' \cdot i0 + sel \cdot i1$$

For an n-bit 2-to-1 multiplexer, the control signals remain the same, but the gating structure will be duplicated n times. The schematic of a 3-bit 2-to-1 multiplexer is shown in Figure 4.5(b).

Example 1 Consider the following VHDL segment:

```
. . .
signal a,b,y: std_logic;
. . .
y <= '0' when a=b else
     '1';
. . .
```

This is a simple conditional signal assignment statement that contains one when clause. The conceptual diagram is shown in Figure 4.6(a). Let us consider the implementation of `a=b`, which is a 1-bit comparison circuit. According to VHDL definition, the input data type is `std_logic`, which has nine values, and the output data type is `boolean`, whose value can be `true` or `false`. During synthesis, we only consider the `'0'` and `'1'` of the `std_logic`



(a) Conceptual diagram (b) Gate-level diagram

Figure 4.6 Synthesis of example 1.

Table 4.5 Truth table of a 1-bit comparator.

input a b	output a=b
0 0	1
0 1	0
1 0	0
1 1	1

data type since the other seven values are meaningless for a physical circuit. We also map the `true` and `false` to logic 1 and logic 0 of the physical circuit. Now the operation a=b can be represented in a traditional truth table, as shown in Table 4.5. The function can be expressed as $a' \cdot b' + a \cdot b$, or simply $(a \oplus b)'$, which is an xnor gate. We can now refine the conceptual diagram into the gate-level implementation, and the new diagram is shown in Figure 4.6(b). We can derive the logic expression of this circuit. Based on the expression of the multiplexer, the output can be expressed as

$$y = sel' \cdot i0 + sel \cdot i1$$

The sel, $i0$ and $i1$ are connected to $(a \oplus b)'$, '1' and '0' respectively, and thus the expression becomes

$$y = sel' \cdot i0 + sel \cdot i1 = (a \oplus b)'' \cdot 1 + (a \oplus b)' \cdot 0$$

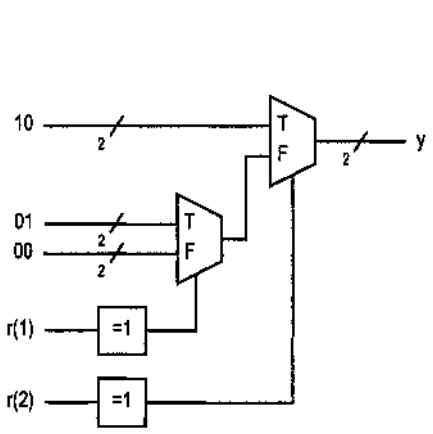
which can be simplified to

$$y = a \oplus b$$

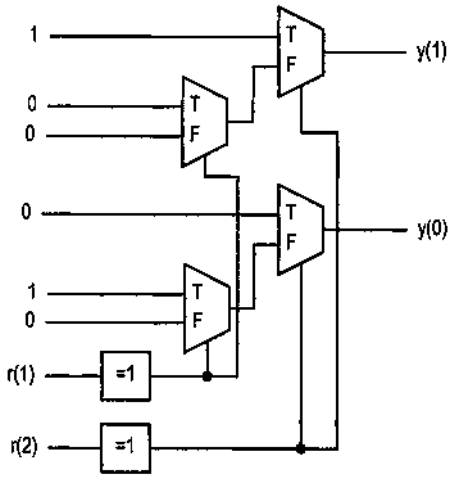
Thus, the final simplified circuit is a single xor gate.

Example 2 Consider the following VHDL segment:

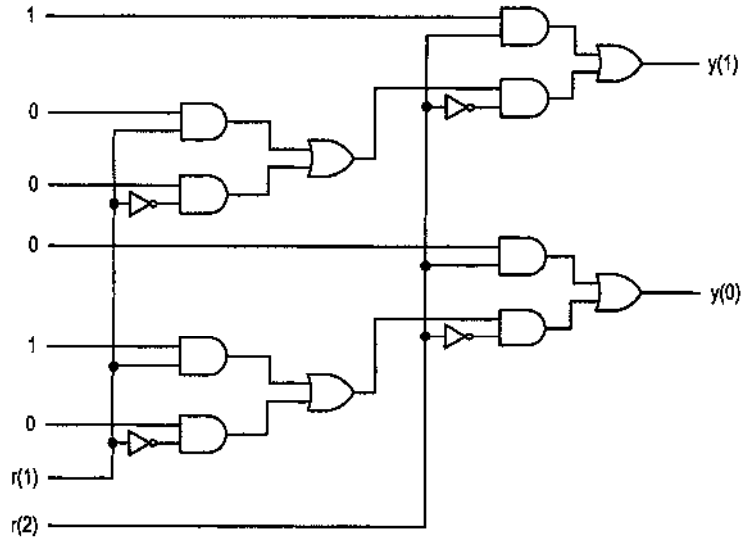
```
. . .
signal r: std_logic_vector(2 downto 1);
signal y: std_logic_vector(1 downto 0);
. . .
y <= "10" when r(2)='1' else
     "01" when r(1)='1' else
     "00";
. . .
```



(a) "Bus" conceptual diagram



(b) Bit-by-bit conceptual diagram



(c) Gate-level diagram

Figure 4.7 Synthesis of example 2.

The conceptual diagram of this segment is shown in Figure 4.7(a). The next step is to derive gate-level implementation. Since the output has two bits, we have to split the conceptual diagram into two single-bit diagrams, as in Figure 4.7(b). Note that the implementation of the Boolean expressions, `r(2)='1'` and `r(1)='1'`, consists simply of the `r(2)` and `r(1)` signals themselves, and no additional logic is needed. After we substitute the multiplexer with its gate-level implementation, the resulting circuits are shown in Figure 4.7(c). We can derive the logic expressions for `y(0)` and `y(1)` using a procedure similar to that in example 1. After simplification, these logic expressions become

$$y(1) = r(2)$$

$$y(0) = r(2)' \cdot r(1)$$

Example 3 Consider the following VHDL segment:

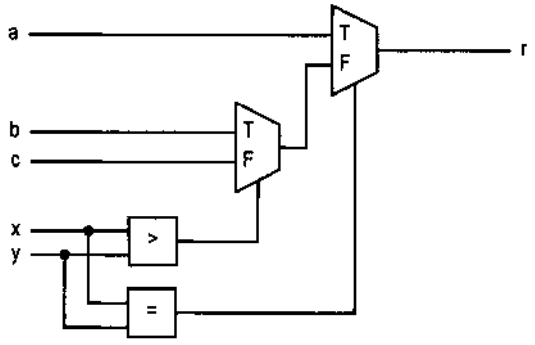
```
. . .
signal a,b,c,x,y,r: std_logic;
. . .
r <= a when x=y else
     b when x>y else
     c;
. . .
```

The conceptual diagram of this segment is shown in Figure 4.8(a). By using the procedure to realize the `a=b` expression of example 1, we can derive the implementation of the `x>y` expression, which is $x \cdot y'$. The corresponding gate level circuit is shown in Figure 4.8(b). We can also derive the logic expression for the output and perform simplification to reduce the circuit size. The logic expression for this circuit is more involved and manually simplifying this circuit becomes a tedious task. This task is better left for software, which is good for a mechanical and repetitive procedure.

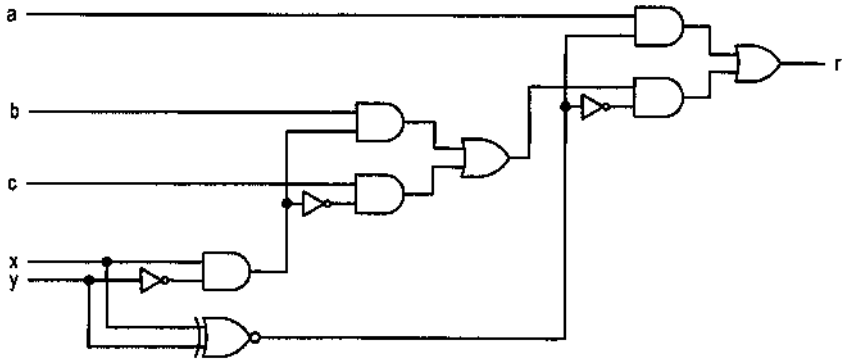
Example 4 Consider the following VHDL segment:

```
. . .
signal a,b,r: unsigned(7 downto 0);
signal x,y: unsigned(3 downto 0);
. . .
r <= a+b when x+y>1 else
     a-b-1 when x>y and y!=0 else
     a+1;
. . .
```

The initial block diagram of this segment is shown in Figure 4.9(a). While the initial block diagram is similar to the previous examples, the value expressions and Boolean expressions are more involved. More complex components, such as an adder and comparator, are needed for implementation. After we implement the clouds, the block diagram is shown in Figure 4.9(b). We can continue to refine the circuit by replacing these components with their gate-level implementations and eventually derive the logic expressions. With these components, performing gate-level simplification becomes much more difficult and good coding practice at the RT level can improve the circuit efficiency significantly. These issues are discussed in Chapter 7.

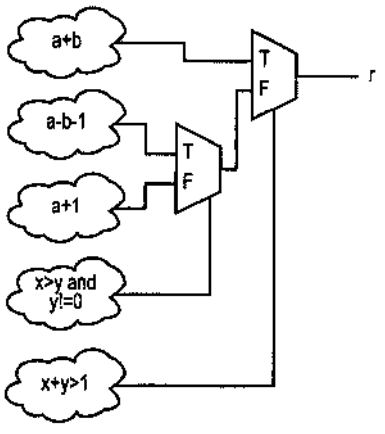


(a) Conceptual diagram

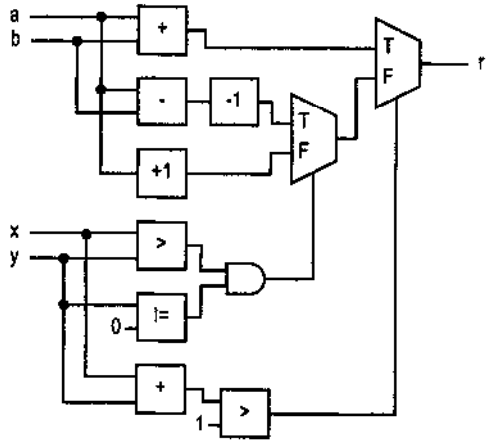


(b) Gate-level diagram

Figure 4.8 Synthesis of example 3.



(a) Initial diagram



(b) Detailed conceptual diagram

Figure 4.9 Refinement of example 4.

4.4 SELECTED SIGNAL ASSIGNMENT STATEMENT

4.4.1 Syntax and examples

The simplified syntax of the selected signal assignment statement is shown below. As in conditional signal assignment statement, we assume that the timing specification is embedded in δ-delay and substitute `value_expression` for the `projected_waveform` clause.

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

The selected signal assignment statement assigns an expression to a signal according to the value of `select_expression`. It is somewhat like a case statement in a traditional programming language. The `select_expression` term is used as the key for selection and it must result in a value of a discrete type or one-dimensional array. In other words, the evaluated result of `select_expression` can have only a finite number of possibilities. For example, a signal of the `bit_vector(1 downto 0)` data type can be used as `select_expression` since it contains only four possible values: `"00"`, `"01"`, `"10"` or `"11"`. A choice (i.e., `choice_i`) must be a valid value or a set of valid values of `select_expression`. The values of choices have to be *mutually exclusive* (i.e., no value can be used more than once) and *all inclusive* (i.e., all values have to be used). In other words, all possible values of `select_expression` must be covered by one and only one choice. The reserved word, **others**, can be used in the last choice (i.e., `choice_n`) to represent all the previously unused values.

We use the same multiplexer, binary decoder, priority encoder and ALU circuit of Section 4.3.1 to illustrate use of the selected signal assignment statement. Since this statement is a natural match to implement a truth table, an additional example is included for this purpose.

Multiplexer Let us consider the 8-bit 4-to-1 multiplexer of Section 4.3.1. The VHDL code for this circuit is shown in Listing 4.5. The entity declaration is identical and thus is omitted.

Listing 4.5 4-to-1 multiplexer based on a selected signal assignment statement

```
architecture sel_arch of mux4 is
begin
   with s select
     x <= a when "00",
          b when "01",
          c when "10",
          d when others;
end sel_arch;
```

We need to be cautious about the metavalues of the `std_logic` and `std_logic_vector` data types. There is an issue about the use of these data types for `select_expression`. Recall that there are nine possible values in `std_logic` data type and there are 81 (i.e., 9*9) possible combinations for the 2-bit `s` signal, including the expected `"00"`, `"01"`, `"10"` and `"11"` as well as 77 other metavalue combinations, such as `"ZZ"`, `"UX"` and `"0-"`, which are

not meaningful in synthesis and will be ignored accordingly. In the code, the **when others** clause covers the "11" choice as well as the metavalue combinations. We cannot simply list the last choice as "11":

```
with s select
  x <= a when "00",
       b when "01",
       c when "10",
       d when "11";
```

This causes a syntax error since only 4 of 81 values are covered, and thus the choices are not all-inclusive. Some synthesis software may accept the following form:

```
with s select
  x <= a when "00",
       b when "01",
       c when "10",
       d when "11",
      'X' when others; -- may also use '-'
```

The last line will be ignored during synthesis and the same physical circuit will be derived.

Binary decoder The VHDL code for the 2-to-2^2 binary decoder of Section 4.3.1 is shown in Listing 4.6. Again, it is necessary to use **others** as the last choice to cover all metavalue combinations.

Listing 4.6 2-to-2^2 binary decoder based on a selected signal assignment statement

```
architecture sel_arch of decoder4 is
begin
   with s select
     x <= "0001" when "00",
          "0010" when "01",
          "0100" when "10",
          "1000" when others;
end sel_arch;
```

Priority encoder The VHDL code for the 4-to-2 priority encoder is shown in Listing 4.7. Recall that "11" will be assigned to code if r(3) is '1'. This consists of eight possible input combinations of the r signal, which are "1000", "1001", "1010", ..., "1111". All of them are listed in the first choice. Note that the symbol | is used for specifying multiple values.

Listing 4.7 4-to-2 priority encoder based on a selected signal assignment statement

```
architecture sel_arch of prio_encoder42 is
begin
   with r select
      code <= "11" when "1000"|"1001"|"1010"|"1011"|
                        "1100"|"1101"|"1110"|"1111",
              "10" when "0100"|"0101"|"0110"|"0111",
              "01" when "0010"|"0011",
              "00" when others;
   active <= r(3) or r(2) or r(1) or r(0);
end sel_arch;
```

Intuitively, we may wish to use the '-' (don't-care) value of the std_logic data type to make the code compact:

```
with r select
  code <= "11" when "1---",
          "10" when "01--",
          "01" when "001-",
          "00" when others;
```

While this is syntactically correct, the code does not describe the intended circuit. In VHDL, the '-' value is treated just as an ordinary value of std_logic. Since the '-' value will never occur in the physical circuit, the "1---", "01--" and "001-" choices will never be met and the code is the same as

```
code <= "00";
```

This, of course, is not the intended priority encoding circuit. We discuss this issue in more detail in Chapter 6.

A simple ALU The VHDL code of the simple ALU specified in Table 4.4 is shown in Listing 4.8. Note that all four possible combinations of the ctrl signal, "000", "001", "010" and "011", are listed in the first choice.

Listing 4.8 Simple ALU based on a selected signal assignment statement

```
architecture sel_arch of simple_alu is
   signal sum, diff, inc: std_logic_vector(7 downto 0);
begin
   inc <= std_logic_vector(signed(src0)+1);
   sum <= std_logic_vector(signed(src0)+signed(src1));
   diff <= std_logic_vector(signed(src0)-signed(src1));
   with ctrl select
      result <= inc           when "000"|"001"|"010"|"011",
                sum           when "100",
                diff          when "101",
                src0 and src1 when "110",
                src0 or src1  when others; -- "111"
end sel_arch;
```

Truth Table Implementation A truth table can be used to specify any combinational function. It is a simple and useful way to describe a small, random combinational circuit. Because the choices list all the possible combinations, the selected signal assignment statement is a natural match for the truth table description. A simple two-input truth table is shown in Table 4.6.

The corresponding VHDL code is shown in Listing 4.9. The a and b signals are concatenated as tmp, which is then used as the select expression. Each row of the truth table now becomes a choice in the selected signal assignment statement and the truth table is implemented accordingly.

Listing 4.9 Truth table based on selected signal assignment statement

```
library ieee;
use ieee.std_logic_1164.all;
entity truth_table is
```

Table 4.6 Truth table of a two-input function

Input a b	Output y
0 0	0
0 1	1
1 0	1
1 1	1

```
    port(
5       a,b: in  std_logic;
        y: out   std_logic
    );
 end truth_table;

10 architecture a of truth_table is
    signal tmp: std_logic_vector(1 downto 0);
 begin
    tmp <= a & b;
    with tmp select
15      y <= '0' when "00",
             '1' when "01",
             '1' when "10",
             '1' when others; -- "11"
 end a;
```

4.4.2 Conceptual implementation

Recall that the syntax of the selected signal assignment is

```
with select_expression select
   signal_name <= value_expr_1 when choice_1,
                  value_expr_2 when choice_2,
                  value_expr_3 when choice_3,
                     . . .
                  value_expr_n when choice_n;
```

Conceptually, the selected signal assignment statement can be thought as an abstract multiplexing circuit that utilizes a selection signal to route the result of the designated expression to output. In this multiplexing circuit, each possible value of `select_expression` has a designated input port in the multiplexer, and `select_expression` works as the selection signal of this multiplexer. Once its value is determined, the result of the designated value expression is passed to the output port of the multiplexer. In Section 4.3.2, we utilized an abstract 2-to-1 multiplexer with a selection signal of the `boolean` data type. The multiplexer can be generalized for other kinds of selection signals. For example, consider a selection signal with $k + 1$ different possible values, `c0`, `c1`, ..., `ck`. The abstract multiplexer has $k + 1$ ports, each corresponding to a value, as shown in Figure 4.10.

It is possible that the input and output have multiple bits and the symbol n is used to designate the width of the buses. The conceptual implementation of the selected signal

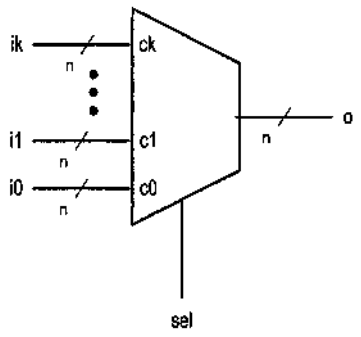


Figure 4.10 Abstract $(k + 1)$-to-1 multiplexer.

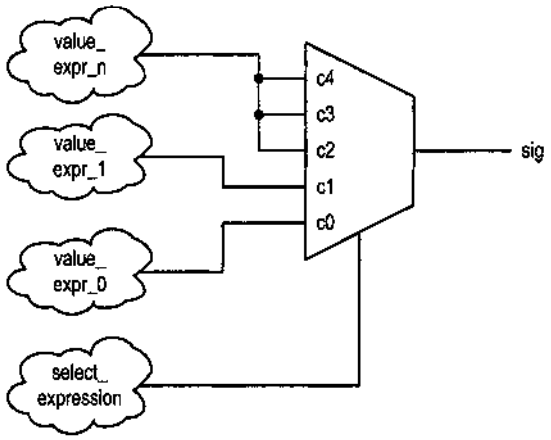


Figure 4.11 Conceptual diagram of a selected signal assignment statement.

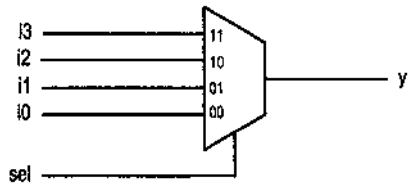
assignment statement involves a single abstract multiplexer and is straightforward. Consider the following statement:

```
with select_expression select
   sig <= value_expr_0 when c0,
          value_expr_1 when c1,
          value_expr_n when others;
```

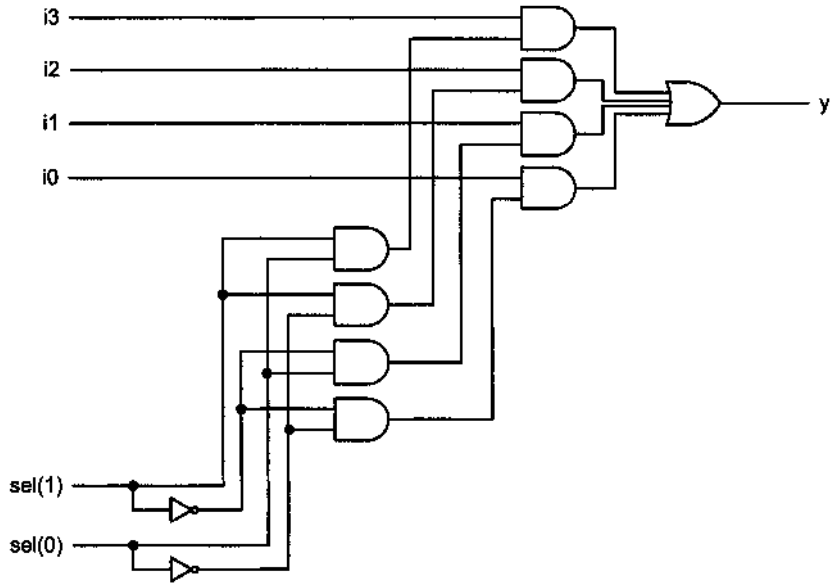
We assume that `select_expression` may result in one of five possible values: `c0`, `c1`, `c2`, `c3` and `c4`. Note that the last choice, **when others**, of this statement implicitly covers `c2`, `c3` and `c4`. The conceptual realization of this statement is shown in Figure 4.11.

The clouds represent the implementation of the three value expressions, `value_expr_0`, `value_expr_1` and `value_expr_n`, and `select_expression` respectively. The evaluated results of the value expressions are fed into the designated input ports of the multiplexer. The result of `select_expression` is connected to the selection port of the multiplexer and its value determines which data will be routed to the output port.

All selected signal assignment statements have a similar conceptual diagram. The main difference is in the number of values that `select_expression` can assume, which in turn determines the size of the multiplexer. Despite the simple conceptual construction, certain



(a) Circuit symbol



(b) Gate-level diagram

Figure 4.12 Circuit symbol and gate-level diagram of a 4-to-1 multiplexer.

device technologies may have difficulty supporting an extremely wide multiplexing circuit. Thus, we should be aware of the number of values in a selection expression.

4.4.3 Detailed Implementation examples

As in the implementation of a conditional signal assignment statement, we continue the refining process and realize the conceptual diagram using gate-level components. Following examples illustrate the derivation.

k-to-1 multiplexer An abstract multiplexer with k symbolic ports can easily be mapped to a physical k-to-1 multiplexer with a $\log_2 k$-bit selection signal. The symbol and gate-level diagram of a 1-bit 4-to-1 multiplexer are shown in Figure 4.12. We use the binary representations, "00", "01", "10" and "11", as the names of the ports. The upper and cells can be thought of as "passing gates," each controlled by an enable signal. The corresponding input will be passed to output when the enable signal is '1'. The bottom part is a 2-to-4 binary decoder that generates the enable signal, in which only one bit is activated. In term of a logic expression, the output can be expressed as

$$y = (sel(1)' \cdot sel(0)') \cdot i0 + (sel(1)' \cdot sel(0)) \cdot i1 + (sel(1) \cdot sel(0)') \cdot i2 + (sel(1) \cdot sel(0)) \cdot i3$$

For a multiple-bit 4-to-1 multiplexer, the enable signals remain the same, but the gating structure will be duplicated multiple times.

In VHDL code, the selection signal frequently has a data type of `std_logic_vector`, which includes many meaningless combinations. During synthesis, only `'0'` and `'1'` of nine values will be used, as we discussed in Section 4.3.1.

Example 1 Consider the following VHDL segments:

```
. . .
signal s: std_logic_vector(1 downto 0);
   . . .
   with s select
      x <= (a and b) when "11",
           (a or b)  when "01"|"10",
           '0'       when others;
. . .
```

This is a simple selected signal assignment statement. The selection expression has a data type of `std_logic_vector(1 downto 0)`. Again, although there are 81 possible values, only `"00"`, `"01"`, `"10"` and `"11"` are meaningful for synthesis. Thus, only a 4-to-1 multiplexer is needed. The conceptual diagram and the refined gate-level diagram are shown in Figure 4.13. The logic expression for this circuit is

$$x = (s(1)' \cdot s(0)') \cdot 0 + (s(1)' \cdot s(0)) \cdot (a+b) + (s(1) \cdot s(0)') \cdot (a+b) + (s(1) \cdot s(0)) \cdot (a \cdot b)$$

Example 2 Consider the truth table in Table 4.6 and the corresponding VHDL segment:

```
tmp <= a & b;
with tmp select
   y <= '0' when "00",
        '1' when "01",
        '1' when "10",
        '1' when others;
```

The conceptual diagram is shown in Figure 4.14. The logic expression for this circuit is

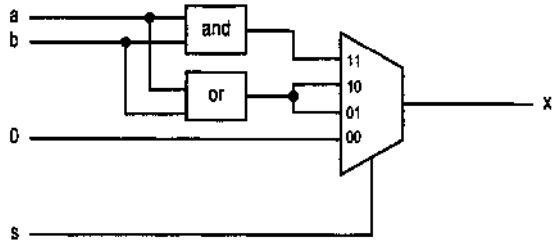
$$y = (a' \cdot b') \cdot 0 + (a' \cdot b) \cdot 1 + (a \cdot b') \cdot 1 + (a \cdot b) \cdot 1$$

The expression can be simplified to $a + b$, which is the or function specified in the truth table.

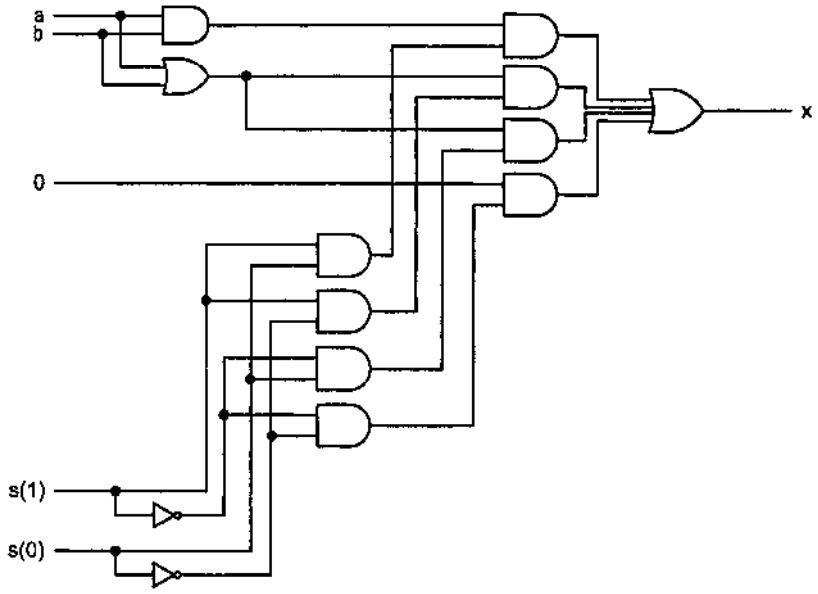
Example 3 Consider the following VHDL segment:

```
. . .
signal a,b,r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);
   . . .
   with s select
      r <= a+1   when "11",
           a-b-1 when "10",
           a+b   when others;
. . .
```

This segment contains more sophisticated expressions. After we realized the value expression clouds, the block diagram is shown in Figure 4.15.



(a) Conceptual diagram



(b) Gate-level diagram

Figure 4.13 Synthesis of example 1.

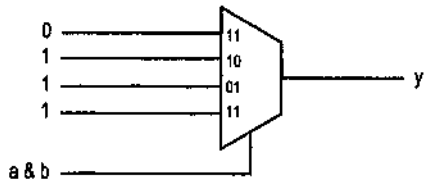


Figure 4.14 Conceptual diagram of truth table–based description.

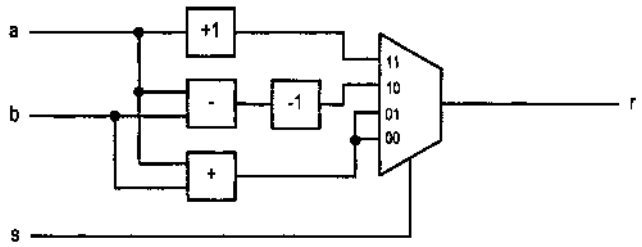


Figure 4.15 Block diagram of example 3.

4.5 CONDITIONAL SIGNAL ASSIGNMENT STATEMENT VERSUS SELECTED SIGNAL ASSIGNMENT STATEMENT

4.5.1 Conversion between conditional signal assignment and selected signal assignment statements

From the synthesis point of view, the conditional signal assignment statement and the selected signal assignment statement imply two different routing structures. The examples presented in the previous sections show that we can describe the same circuit using either a conditional or a selected signal assignment statement. Actually, the conversion between the two forms of assignment statements is always possible.

Converting a selected signal assignment statement to a conditional signal assignment statement is straightforward. Consider a general selected signal assignment statement in which there are eight possible choices: c7, c6, . . ., c1, c0.

```
with sel select
   sig <= value_expr_0 when c0,
          value_expr_1 when c1|c3|c5,
          value_expr_2 when c2|c4,
          value_expr_n when others;
```

We can describe the choices of a when clause as a Boolean expression. For example, **when** c2|c4 can be expressed as (sel=c2) **or** (sel=c4). We can then use these Boolean expressions and convert the selected signal assignment statement to a new format:

```
sig <=
   value_expr_0 when (sel=c0) else
   value_expr_1 when (sel=c1) or (sel=c3) or (sel=c5) else
   value_expr_2 when (sel=c2) or (sel=c4) else
   value_expr_n;
```

Converting a conditional signal assignment statement to a selected statement needs more manipulation. Let us consider a general conditional signal assignment statement with three Boolean expressions:

```
sig <= value_expr_0 when bool_exp_0 else
       value_expr_1 when bool_exp_1 else
       value_expr_2 when bool_exp_2 else
       value_expr_n;
```

We need a 3-bit auxiliary selection signal, sel, in which each bit represents a Boolean expression. By specifying proper choices, we can preserve the desired priority. The converted code is

```
sel(2) <= '1' when bool_exp_0 else '0';
sel(1) <= '1' when bool_exp_1 else '0';
sel(0) <= '1' when bool_exp_2 else '0';
with sel select
   sig <= value_expr_0 when "100"|"101"|"110"|"111",
          value_expr_1 when "010"|"011",
          value_expr_2 when "001",
          value_expr_n when others;
```

Note that the pattern of the selected signal assignment statement is very similar to a priority encoder except that the request signals are replaced by the auxiliary selection signals generated from the Boolean expressions.

4.5.2 Comparison between conditional signal assignment and selected signal assignment statements

In the selected signal assignment statement, each choice can be considered as a row in a table. Thus, this statement is a good match for a circuit described by a truth table or a truth table–like function table, such as the decoder, truth table and multiplexer examples discussed in Section 4.4. On the other hand, it is less effective when certain input conditions are given preferential treatment. For example, if we examine the priority encoder example of Section 4.4, eight of the 16 ports of the multiplexer are connected to an identical expression.

The conditional signal assignment statement implicitly enforces the order of the operation and is a natural match for a circuit that needs to give preferential treatment for certain conditions or to prioritize the operations. The priority encoder is a good example of this kind of circuit. The conditional signal assignment statement can also handle complicated conditions. For example, we can write

```
pc_next <=
   pc_reg + offset when (state=jump and a=b) else
   pc_reg + 1 when (state=skip and flag='1') else
. . .
```

A conditional signal assignment statement is less effective to describe a truth table since it may "overspecify" the circuit and thus add unnecessary constraints. For example, consider the multiplexer of Section 4.3.1. The original VHDL segment is

```
x <= a when (s="00") else
     b when (s="01") else
     c when (s="10") else
     d;
```

The code can also be written as

```
x <= c when (s="10") else
     a when (s="00") else
     b when (s="01") else
     d;
```

or

```
x <= c when (s="10") else
     b when (s="01") else
     a when (s="00") else
     d;
```

or many other possible variations. These codes give priority to the condition in the first when clause, although it is not part of the original specification. While this type of code is not wrong, the extra constraint may introduce additional circuitry and make synthesis and optimization more difficult.

Ideally, the synthesis software should automatically determine the optimal structure and derive identical gate-level implementation, regardless of the language constructs used in VHDL descriptions. In reality, this is possible only for small, trivial designs. For a general design, we have to be aware of the effect of the statements on the routing and the "layout" of the final implementation. These aspects are illustrated by examples in Chapter 7.

4.6 SYNTHESIS GUIDELINES

- Avoid a closed feedback loop in a concurrent signal assignment statement.
- Think of the conditional signal assignment and selected signal assignment statements as routing structures rather than sequential control constructs.
- The conditional signal assignment statement infers a priority routing structure, and a larger number of when clauses leads to a long cascading chain.
- The selected signal assignment statement infers a multiplexing structure, and a large number of choices leads to a wide multiplexer.

4.7 BIBLIOGRAPHIC NOTES

Since the focus of the book is on synthesis, only synthesis-related aspects of the concurrent signal assignment statement are discussed. The complete discussion on these constructs can be found in *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden.

The discussion in this chapter illustrates the general schemes to realize concurrent signal assignment statements in various routing structures. Individual synthesis software may map certain language constructs to specific hardware architectures. The software vendors sometimes include a "style guide" in their documentation. It shows the mapping between hardware architecture and the VHDL language constructs.

Problems

4.1 Add an enable signal, `en`, to a 2-to-4 decoder. When `en` is `'1'`, the decoder functions as usual. When `en` is `'0'`, the decoder is disabled and output becomes `"0000"`. Use the conditional signal assignment statement to derive this circuit. Draw the conceptual diagram.

4.2 Repeat Problem 4.1, but use the selected signal assignment statement instead.

4.3 Consider a 2-by-2 switch. It has two input data ports, `x(0)` and `x(1)`, and a 2-bit control signal, `ctrl`. The input data are routed to output ports `y(0)` and `y(1)` according to the `ctrl` signal. The function table is specified below.