# VHDL: HARDWARE DESCRIPTION AND DESIGN

## CHAPTER 1
## INTRODUCTION

The principal objective of the US Government's VHSIC program was to advance the state of the art in the United States in the areas of design, process, and manufacturing technology. The Government's objectives in sponsoring the VHDL effort as part of that program were to ensure that the Very High Speed Integrated Circuits (VHSIC) program would also provide the means to insert these new, advanced, microelectronic components into operational systems more rapidly than had been done in the past, and would provide technology that would allow more cost-effective development of commercial electronics by improving communication within and between companies, and by streamlining the development process. The VHDL effort formally began in June 1981 with a workshop, held in Woods Hole, Massachusetts, to discuss how best to achieve these goals. The workshop, which was sponsored by the Very High Speed Integrated Circuits (VHSIC) program, included members of industry, government, and academia.

The result of this workshop was a clearly stated goal of developing a new, industry standard, hardware description language that was technology independent and met the various needs of the members of the microelectronics community. The Woods Hole workshop also produced detailed set of requirements that formed the technical basis of the Government's formal Request for Proposals.

Throughout the initial development stage the language was submitted to extensive public review. Following these interim reviews, the final version of the language to be developed under the original -contract, known as VHDL Version 7.2, was made available in August of 1985. In July 1986, Intermetrics delivered the initial suite of support software: in February of 1987, a completely revised tool set for version 7.2 was made available.

In March of 1986, the IEEE took on the effort of standardizing VHDL. The VHDL Analysis and Standardization Group was set up to review the language, with a goal of repairing known problems with the language and modifying the language where a broad consensus formed around the modification. The Air Force fully supported this work by awarding a contract to Intermetrics to develop the support software for the new language standard, now known as IEEE-1076. IEEE-1076 was approved in December 1987. This standard is the current definition of VHDL and is the language addressed in this book.

### Very High Speed Description Languages (VHDL)

VHDL was developed to address a number of recurrent problems! in the development, exchange and documentation of digital hardware. For instance, a typical delivery of hardware to the government would include tens of thousands of pages of documentation that needed to be sifted through during acceptance and testing and referred to throughout the maintenance life of the component. When the component needed to be replaced it took a large amount of effort to reconstruct the intended behavior

Scanned with CamScanner

of the component. A good hardware description language solves this problem because the documentation is executable and all elements are tied into a single model.

While there are many hardware description languages, there was, prior to VHDL, no accepted standard in the industry. Many of the existing languages are developed to serve the simulators that run them, and are often proprietary developments of particular companies. Others target a particular technology, design level, or design methodology. VHDL is technology independent, is not tied to a particular simulator or value set, and does not enforce a design methodology on a designer. What it does do is allow the designer the freedom to choose technologies and methodologies while remaining within a single language. No one can foresee the changes that will take place in digital hardware

VHDL thus offers a number of benefits over other hardware description languages.

Covered in PDTGLW

**Public Availability:** VHDL was developed under a Government contract and is now an IEEE standard. The Government has a strong interest in maintaining VHDL as a public standard. The advantages of this status are apparent; without it, many of the other benefits described below ~~Mow~~ would not exist.

**Design Methodology and Design Technology Support:** VHDL was designed to support many different design methodologies (e.g., top-down versus library-based) and design technologies (e.g., synchronous versus asynchronous, or PLA versus random logic). In this way, the language manages to provide design assistance to organizations that operate in very different ways, and that have very different design needs. VHDL is appropriate for use by a CAD/CAE house that sells library-based design tools, and is equally appropriate for an aerospace company that does a large amount of AS1C design.

**Technology and Process Independence:** VHDL was designed to be independent of both technology and process. By this we mean that VHDL does not have embedded within it an understanding of particular technologies or processes. However, such information can be written using VHDL. Thus, a working (i.e., simulatable) description of a system may be developed above the gate level, and then decomposed into gate-level implementations depending on the chosen implementation technology (e.g., CMOS, nMOS, GaAs). The advantages of this capability for second sourcing are apparent.

**Wide Range of Descriptive Capability:** VHDL supports behavioral description of hardware from the digital system (e.g., box) level to the gate level. One of the primary advantages of VHDL lies in the ability to capture the operation of a digital system on a number of these descriptive levels at once, using a coherent syntax and semantics across these levels, and to simulate that system using any mixture of those levels of description. It is therefore possible to simulate designs that mix high-level behavioral descriptions of some subsystems with detailed implementations of other subsystems in the model.

This facilitates the development of a design that correctly reflects the intention of the original specification of the system. During maintenance of the system, redesigns or alterations to the system can be by replacing the VHDL in the description and resimulating with test set.

Many existing hardware description languages operate best at the c and gate level; for this reason, hardware designers may be most familiar with the use of HDL's to

Scanned with CamScanner

support low-level logic design. While purposely chosen to show VHDL in contexts beyond the logic level of digital hardware.

**Design Exchange:** VHDL is a standard and, as a result, VHDL models are guaranteed to run on any system that conforms to that standard. This means that models developed at one location will run at other locations (and produce the same simulation results) whether or not both locations use the same VHDL tool sets. This has many implications for the use of VHDL. As VHDL models of common components become available, designers will simulate their designs with those model descriptions. In this way designers can try a number of f different components and choose those that best fit the constraints and requirements of the system being designed. Another aspect of this capability is that teams can exchange high-level descriptions of the subsystems of a digital system, which then allows each subsystem to be developed independent of progress on the other subsystems.

For example, suppose an organization was developing a digital system and wanted to contract out a subsystem to another organization. The contracting organization could develop a high-level VHDL description of the system with the subsystems broken out into design entities. The high-level description would then be given to the contractor for development. The contractor could develop the subsystem in VHDL while all along having a simulatable description of the rest of the system. The contracting organization would also be able to develop its portion of the system and test without waiting for the subsystem to be developed. This greatly reduces the integration effort and reduces the time of the entire development significantly.

**Large-Scale Design and Design Re-use:** VHDL was modeled on a philosophy similar to that of many modem programming languages — that design decomposition aids are just as important as detailed descriptive capabilities when it comes to supporting the development large designs being executed by multiperson teams. There are several elements of the language specifically targeted at this goal. Packages, configuration declarations, and the concept of multiple bodies exhibiting different implementations of an entity are all present in the language toil support design sharing, experimentation, and design management

These elements of the language also tend to support design it-use, t since they promote the encapsulation of design information and the ability to be flexible in conforming to others' design conventions when using an already-designed part.

**Government Support:** The VHSIC Program Office developed VHDL with a clear vision of how it itself wanted to use VHDL. VHDL1 can be used throughout the life cycle of a digital system, and is likely to become a major component of the procurement process. The Department of Defense is currently requiring VHDL descriptions on all contracts that develop ASIC's. As more VHDL toolsets become available and these toolsets mature the government can be expected to extend its mandates into other kinds of hardware development contracts and to increase the scope of the VHDL requirement to include certain information in the description and to provide simulatable descriptions.

## Terminology and Conventions

Scanned with CamScanner

VHDL is a hardware description language and, therefore, VHDL descriptions are generally used to model hardware components and system; *i.e.* gates, chips, boards, etc. However, VHDL provides an abstract framework for describing hardware which is easily extended into other domains. Therefore, the term *digital device* will be used when referring to a system that is being described in VHDL. A digital device can range from a gate to a microprocessor to a complete system and beyond. The guiding factor is that the underlying system be based on a general stimulus-response model that uses discrete *(i.e.* non-continuous) values.

Throughout this book the person who uses VHDL will be referred to as the *designer*. This is not because VHDL is primarily for those who design discrete systems, but rather because the use of VHDL requires one to model the processes of a system and define the nature of the stimulus that is applied to that system. In this sense, the *designer* may be designing a hardware component, the tests of that hardware component, or the constraints to be placed on the model.

In the examples, VHDL reserved words, such as entity and signal, arc given in bold. References to these reserved words in the text are bolded when the reference is specifically to the term itself, and not a general reference to the concept.

Scanned with CamScanner

design
To model a hardware set using a VHDL language.

technology – look @ the device (built-up)
methodology – design of a device



# CHAPTER 2
# A MODEL OF HARDWARE

– To model a hardware design

VHDL is a language for describing general digital hardware devices. It makes no assumptions about the technology of a device or the methodology used to design a device. Instead, an abstraction of the nature of digital hardware devices is used as the basis for the language. This abstraction includes behavioral, timing, and structural characteristics of digital devices. These characteristics are integrated into a single cohesive language which provides a broad range of options for the description and design of digital devices. It is the ability to create a uniform description of a digital device utilizing multiple levels of abstraction which gives VHDL its power and applicability.

This chapter will present a model of this abstraction of digital devices. This model is appropriate for descriptions of all devices from high-level, abstract models down to gate-level desertions. While theoretical in nature, this model should provide a robust foundation for understanding the concepts presented throughout the rest of the book. After this discussion, it will be easier to understand the nature of VHDL and the VHDL constructs which are used to model discrete systems.

The general model on which VHDL is based consists of three interdependent models: a behavioral model, a timing model, and a structural model. These three models are integrated into a single language.

## A Model of Behavior

Behavior can be defined as the functional interpretation of a particular system. This does not imply the existence of purely structural models or purely behavioral models. All models have a structure and a behavior. In some design automation systems and languages, the behavior is not accessible to the designer except through the use of predefined library items. In VHDL, however, behavior is incorporated directly into the language and the designer has the option of mixing structure and behavior anywhere inside the model.

A digital device is a discrete system; *i.e.* one which transforms discrete-valued input into discrete-valued output. It does this by performing a number of operations or transformations on the input values. The results of these operations are passed into other operations and, finally, into the output values. A graphical representation of this concept is shown in Figure 2.1.

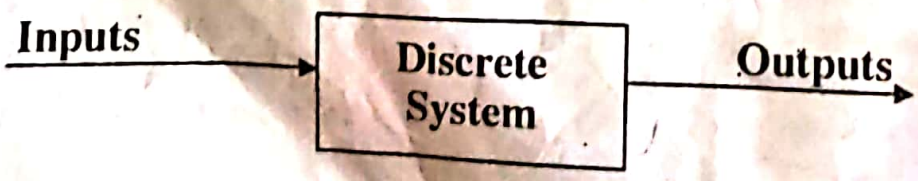



Figure 2.1. Representation of a Discrete System

Scanned with CamScanner

As an example, consider a discrete system P whose behavior is the boolean function XOR. This system has two inputs, In1 and In2, and a single output, Out1. The discrete system is shown in Figure 2.2.

Discrete systems can range from the very simple to the very complex. For example, at the gate level operations are usually boolean functions applied to bit-valued wires and the discrete system is a representation of a logic circuit. On an abstract level, operations can be of any describable complexity. For instance, a microprocessor can be represented as a discrete system and the fetch cycle of that microprocessor could be a single operation.

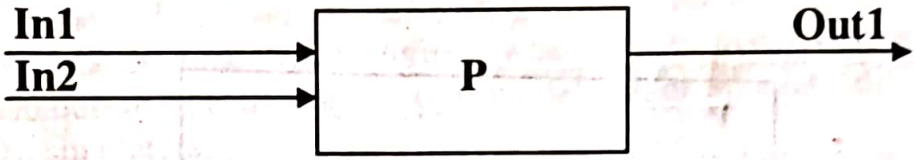



Figure 2.2. Representation of Discrete System P

In VHDL, all operations of a discrete system are described with the same abstract mechanism. Abstraction does not, however, imply complexity. The VHDL modeling mechanisms are essentially straightforward. As stated above, a discrete system is a collection of operations applied to values being passed through the system. In VHDL, we refer to each operation as a *process* and the pathways in which values are passed through the system as *signals*.

A process can be viewed as a program: it is constructed out of procedural statements and can call subprograms much as a program written in a general purpose procedural language like Pascal or C. In VHDL, all processes in a model are said to be executing concurrently. Therefore, a VHDL model is a collection of independent programs running in parallel.

In order to coordinate a set of processes running concurrently, a mechanism is denned to handle the communication between processes. This mechanism is the signal. In particular, signals define a *data pathway* between two processes. This data pathway is *directed;* one side of the pathway generates a value and one side receives that value. Data pathways have some specie characteristics. First of all, each pathway has a type associated with it. The type defines a range of values which may be passed over the pathway. For instance, a data pathway whose type is integer could not contain a value of type bit, or a real number, or any value other than a value of type integer. All communication between processes takes place over these data pathways.

Processes continue to execute until they are suspended. Once suspended, a process can be reactivated. One way a process can be reactivated is by designating a maximum time for the process to remain suspended. When modeling behavior, a designer wants certain actions to take place when given conditions are met or necessary information becomes available. In particular, the designer often wants a process to be reactivated only when some change in the state of the system takes place. Such a change is reflected by a change in the value of a signal, since signals contain the state of the

Sensitivity Channel is a pathway where a process expresses its Sensitivity to a value of data on the pathway

system. VHDL provides a means for a process to express its sensitivity to the value of a data pathway. These pathways are called *sensitivity* channels. When the value on a sensitivity channel changes, the process is reactivated. The *wait statement* is used to designate any timeout conditions and sensitivity channels for a process. When a process executes a wait statement, the system records the reactivation conditions and the process is suspended.

As an example of reactivation conditions, consider the simple discrete system P described above. Figure 2.3 shows a representation of this discrete system which illustrates the data pathways and the internal process which implements the system. The discrete system should react whenever its inputs change, therefore, both inputs are also sensitivity channels for the process. This discrete system has a single operation.

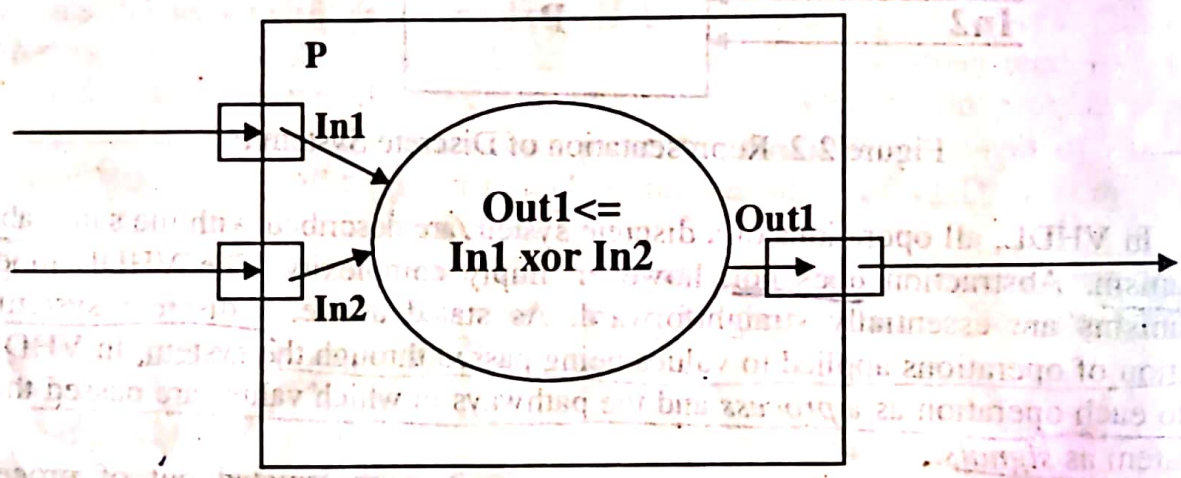
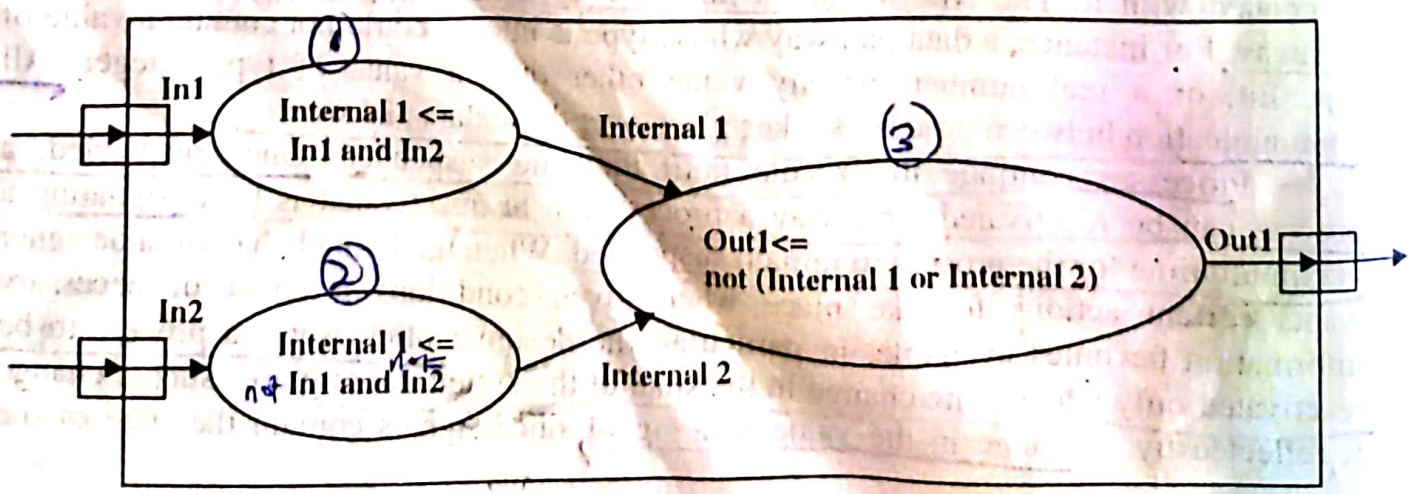



Figure 2.3. Discrete System P with Data Pathways and Process

The discrete system used to model a digital device is contained inside a *design entity*. A design entity has two parts: an entity declaration and an architectural body. The interface defines the inputs and outputs of the device and the body defines the discrete system used to model the device.

In many discrete systems, it is possible to decompose the behavior into a number of operations. For instance, the device depicted by discrete system P above could be represented by an equivalent discrete system Q which has three operations. The new discrete system is shown in Figure 2.4.



Stimulus - response

designated

The model of a discrete system determines when to response to a stimulus. The time for a stimulus to a model to occur is called simulation time.

The discrete system Q has three operations which are functionally equivalent to the single operation of discrete system P above.

## A Model of Time

The model of digital devices used here is based on a stimulus-response paradigm; when there is stimulus to the model, the model responds and then waits for more stimulus. This stimulus occurs at given times which are designated by the model of the discrete system. The "time" at which something occurs means simulation time and not the time of an implementation platform's internal clock. Since VHDL is concurrent but is also designed to be able to run on non-parallel machines it is necessary to create a definition of simulation time which defines when events occur during the course of simulation. Without such a definition, one model might simulate differently on two different VHDL simulators.

When a process generates a value on a data pathway it may also designate the amount of time before the value is sent over the pathway this is referred to as *scheduling a transaction* after the given time, it is possible to schedule any number of transactions for a data pathway. The collection of transactions for a signal is called the *driver* of the signal. The driver is a set of time/value pairs which hold the value of each transaction and the time at which the transaction should occur.

VHDL has a two-stage model of time. This two-stage model is referred to as the *simulation cycle*. The simulation cycle is the conceptual abstraction under which a hardware model described in VHDL is exercised. This abstraction is based on a generalized model of the stimulus and response behavior of digital hardware; *i.e.* a functional component *reacts* to activity on its input connections and *responds* through its output connections. In VHDL, this model revolves around the concepts of processes and signals. Processes are the functional components which are connected with signals. A process may react to a change in value on a signal to which it is connected by transmitting new data to other processes via signals.

During the first stage of the simulation cycle, values are propagated through the data pathways (signals). This stage is complete when all data pathways which are scheduled to obtain new values at the current simulation time are updated. During the second stage, those active elements (processes) which receive information on their sensitivity channels are exercised until they suspend (via the execution of a wait statement). This stage is completed when all active processes are suspended. At the completion of the simulation cycle, the simulation clock is set to the next simulation time at which a transaction is to occur and the cycle is started again. The simulation cycle is depicted in Figure 2.5.

Scanned with CamScanner

when the new value being assigned to the signal (passed through the data pathway) is different from the previous value of the signal, an event is said to be occurred.

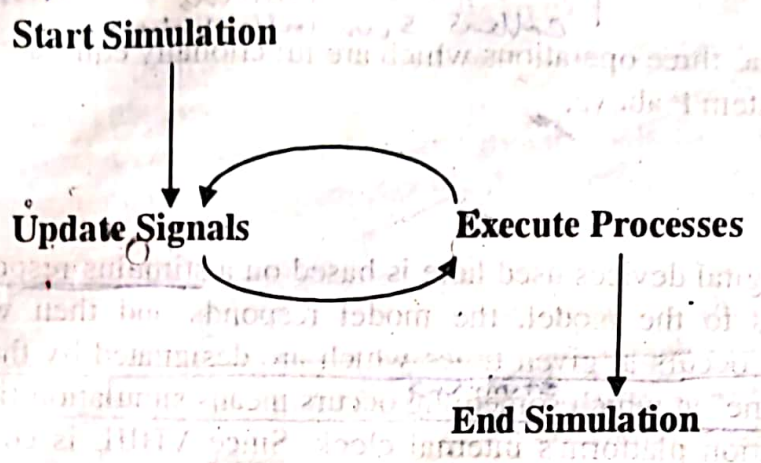



Figure 2.5. The Simulation Cycle

The above model means that there is always some delay between the time a process puts a value on a data pathway and the time at which the data pathway reflects that value. In particular, if no delay is given in the assignment of a value to the data pathway a delta delay is used. This delay does not update the time of the simulation clock but does require the passing of a new simulation cycle. It is important to take note of this fact because the language mechanisms for assigning values into a data pathway look deceptively like variable assignments in this and other languages but the effect is quite different. When a value is assigned in the pathway it is not immediately available to processes which read the value from the pathway. There is a *delay* between assigning and updating a signal value.

If the new value being assigned to a signal (*i.e.* passed through a data pathway) is different from the previous value of the signal, an *event* is said to have occurred. When a process is sensitive to a signal, it is sensitive to events on that signal, not to general transactions. If the process needs to become active when there is a transaction (*i.e.* even if the new value of the signal is the same as the old value) the process should be made sensitive to the signal-valued attribute Transaction which is described later.

## A Model of Structure

When a model of a digital device requires more than a few operations, the representation of the associated discrete system can become difficult to manage. Such models often contain conceptual partitions which can be used to decompose the model into functionally related sections. This decomposition is the called the structure of the model.

Many digital devices are designed by combining a number of subdevices together and tying (wiring) the subdevices together. Each subdevice is a discrete system unto itself. A digital device which uses these subsystems is a new discrete system incorporating each of the subsystems.

The outermost data pathways of a discrete system are defined by the interface to the digital device which is defined by the entity. When a discrete system ties together two subsystems, it is really connecting a data pathway from one subsystem to the data pathway of the other subsystem. In this way, the two subsystems can communicate with

Functionally related sections used by conceptual partitioning

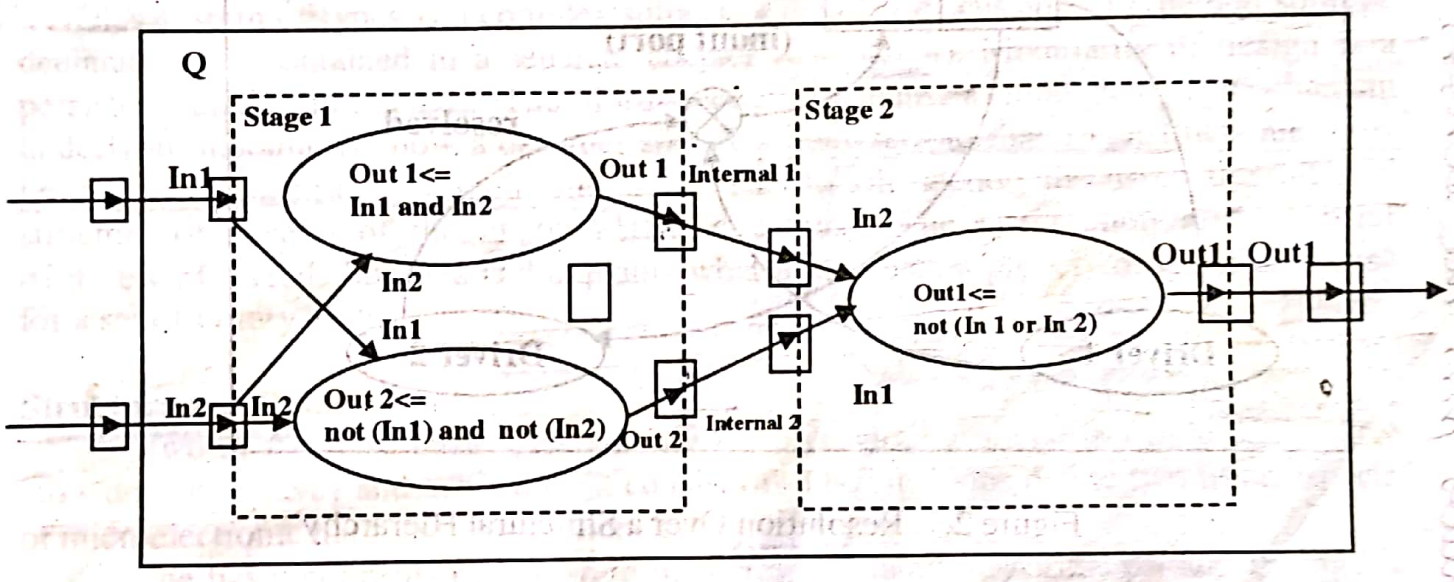
Scanned with CamScanner

ports are D pathway that connect the two subsystem, so at both of them can communicate.



each other. These connections are called *ports* and they have some special characteristics. First, the definition of a port represents a declaration of a signal and, therefore, of a data pathway. Think of data pathways as a Pipe (in the plumbing sense) between two processes. The port represents an open end of the pipe. These open ends can be welded together with signals in a device which uses the subsystem containing the port.

For example, recall the discrete system Q shown earlier in the chapter. This discrete system required three operations. These three operations might be broken into two functional sections: stage 1 and stage 2. The resulting discrete system is shown in Figure 2.6.




The functional sections are "black boxes" for the operations which they contain. By viewing these sections as black boxes it is possible to ignore the actual implementation of the operations.

The first box has two inputs and two outputs and contains the two processes which implement the first phase of the operation. The second box would have two inputs and a single output and would contain the process implementing the second phase of the operation.

The discrete system which wires these two subsystems together defines the data pathways (signals) used to connect the two subsystems and then ties the subsystems together. This has the effect of creating a single black box of the entire system. In a VHDL model of discrete system Q, each of the boxes would be represented by a VHDL design entity.

There are a number of ways in which the model of structure interacts with the model of behavior. First, it is possible to interject a type conversion function onto a data pathway. This is useful when two processes must communicate but the interfaces to the discrete systems in which they are defined do not have the same characteristics. For instance, one system might operate on integer signals and one system might operate on 32 bit encodings of integers. A type conversion could be used to convert between these two representations. Type conversion are discussed in detail later.

Scanned with CamScanner

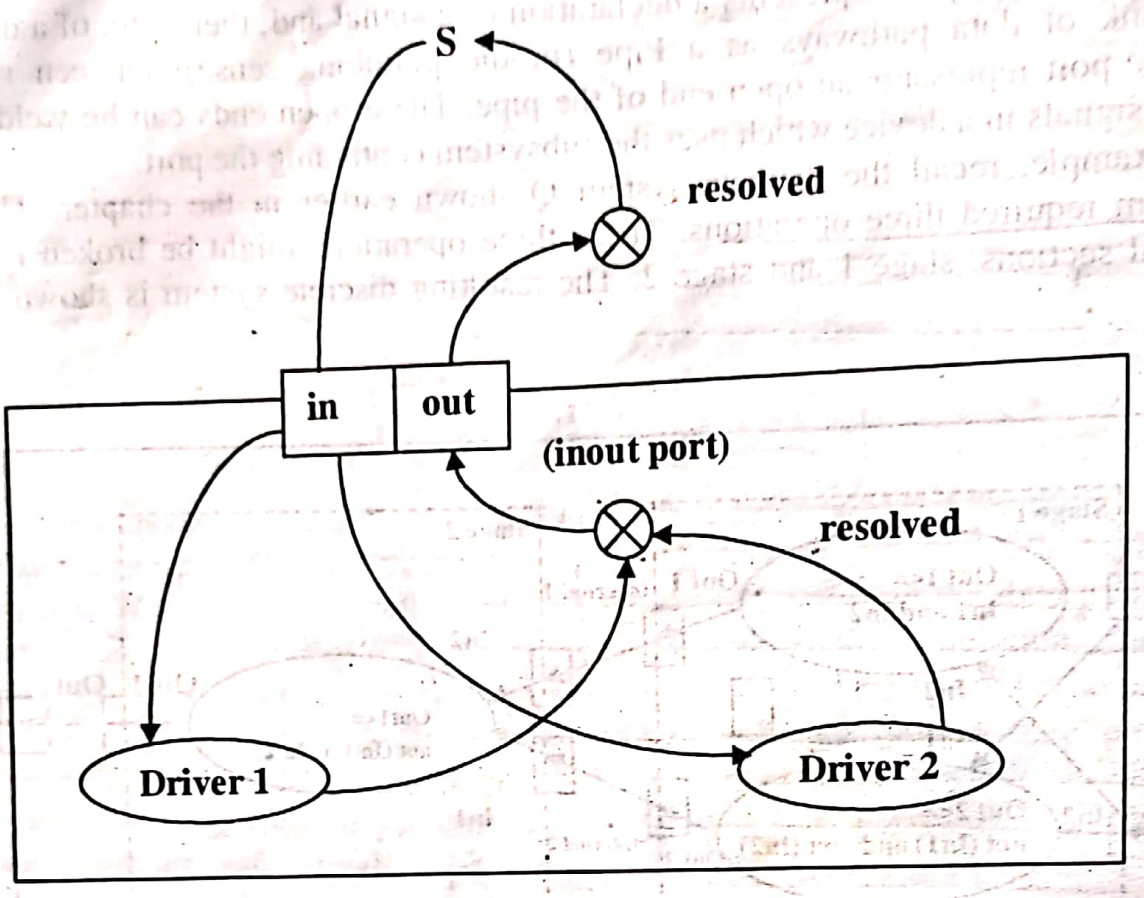



Figure 2.7. Resolution Over a Structural Hierarchy

Another important way in which structure and behavior interact is the resolution of resolved signals. If more than one driver is defined for a signal (*i.e.* more than one process assigns to the signal), the signal is a *resolved signal*. The designer must associate a resolution function with such a signal. The resolution function collects the values sent from each driver and generates a single value. For instance, a tri-state signal might be resolved with a function which performs a wired-or on the input values. In some situations, it is possible to *disconnect* a driver of a resolved signal; *i.e.* to *turn off* the driver. When a driver is turned off, [11] is not passed to the resolution function and, therefore, does not participate in the determination of the resolved value. There are two types of resolved signals: registers and busses. A register signal is *not* resolved if all of its drivers are turned off, while a bus signal is resolved when all of its drivers are turned off. This means that a resolution function for bus signals must handle the resolution of an empty list of values. If a port is resolved then the resolved value is determined before the value is passed out of the subsystem which contains the port. This is depicted in Figure 2.7.

Resolved signals have important implications for inout ports in at model. If a signal is resolved through a hierarchy of inout ports the f value of the out side of each port is resolved all the way up to the top level signal and *then* passed down through each in side of the inout ports. This means that the value coming through the in side of the port is the value resolved at the top level signal and not the value resolved at each port.

The resolution function collects values from the drivers and generates a single value

# CHAPTER 3
# BASICS

This chapter presents a broad view of some of the most important features of VHDL. The first section introduces the notions of structure and behavior and describes in very general terms how these notions relate to various VHDL features, including the VHDL signal, entity declaration, architecture body, process statement and component instantiation statement. VHDL permits designers to declare a virtually unlimited number of data types for characterizing the values held by signals, variables, and constants. The second section of this chapter explains how several basic types and objects are declared. VHDL's system of types is a complex subject; a detailed discussion of type and subtype declarations is contained in a separate chapter (Chapter 4). Modularity of design is a persistent theme in VHDL. The third section explains a fundamental mechanism underlying modularity: how a designer specifies interfaces to major constructs and how he connects constructs at their interfaces. The fourth section discusses the internal structure of several of the major VHDL constructs. The last section gives a brief overview of VHDL libraries and explains what determines a proper "order of analysis" for a set of library units.

## Structure and Behavior

Two of the most basic questions we can ask about a microelectronic device are: How does it behave? and What does it consist of? The questions define two basic aspects of microelectronic hardware description: behavior and structure.

The basic concepts of hardware behavior are data transformation and timing. A data value at a *point* on a net may change from high to low (or from true to false, or from '1' to '0', or, at a more abstract level, from 10 to 20, etc.) in response to data transformations elsewhere in the hardware. There is always a delay of some magnitude (however small) between the causing transformation and the caused transformation.

The basic concepts of hardware structure are the *component*, the *port*, and the *signal*. The component is the building block of hardware description; virtually any "size" of building block — gate, chip, or board — can be seen as a component, depending on the level at which the hardware is being described. A port is the component's point of connection to the world, the point *through* which data flows into and out of the component.

The basic concept that ties the behavioral view and the structural view is the *signal*. As a descriptive abstraction of a hardware wire, the signal serves both to hold changing data values and to connect subcomponents. A signal is a path from one component to another component, the path *along* which data flows between components. Signals connect components at their ports.

In VHDL, a component is represented by the *design entity*. This is actually a composite consisting of an *entity declaration* and an" *architecture body*. The entity declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component's ports. The following entity declaration.

Scanned with CamScanner

which declares two in ports and two out ports, is the VHDL equivalent of the "external" view of a half-adder as shown in Figure 3.1:

```
- The entity declaration
entity Half_adder is
    port (
        X : in Bit ;
        Y : in Bit ;
        Sum : out Bit ;
        Carry : out Bit) ;
    end Half_adder;
```

("--" marks a command in VHDL. Anything on a line following "--" is interpreted as comment.)

**Basics**

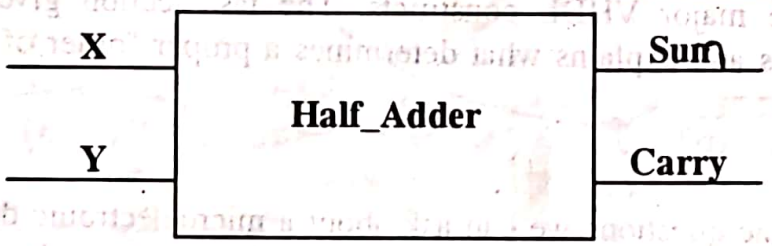



Figure 3.1. Half-adder: Entity Declaration

In an entity declaration, the entity itself and each of its ports is named with an *identifier*. A VHDL identifier is a sequence of characters (not containing any spaces) that obeys the following rules:

- Every character in the sequence is either a letter ('a' to 'z' and 'A' to 'Z'), or a digit ('0' to '9'), or the underscore character ('_').
- The first character in the sequence is a letter.
- The sequence contains no adjacent underscores, nor is the last character in the sequence an underscore.

VHDL identifiers are not case-sensitive; that is, an uppercase letter is considered to be the same letter as its corresponding lowercase letter. Thus, the following sequences of characters are considered to be variant forms of the same identifier.

ABC
abc
aBc

Some identifiers, such as begin, end, and entity, are *reserved words* in VHDL. These identifiers, which are always printed in boldface in this hook, have a fixed meaning in the language and may not be used to designate entities, signals, components, or other user-defined items.

Scanned with CamScanner

The architecture body provides the "internal" view; it describes the behavior or the structure of the component. The following VHDL describes the behavior of the half-adder.

```
-- The architecture body:
architecture Behavioral_description of Half_adder is
begin
  process
      Sum <= X xor Y after 5 Ns ;
      Carry <= X and Y after 5 Ns ;
      wait on X, Y ;
  end process ;
end Behavioral_description ;
```

The architecture body has a *process statement* containing two *signal assignment statements* and a *wait statement.* The first signal assignment statement specifies that the signal connected to the port named Sum will obtain the "exclusive-or" of the value of the signal connected to the port named X and the value of the signal connected to the port named Y after a delay of 5 nanoseconds. Similarly, the second signal assignment statement specifies that the signal connected to the port named Carry will obtain the "and" of the values of the signals connected to the die ports named X and Y. (The left arrow, "<=", in the signal assignment statements is mnemonic for the direction of the data flow.) The wait statement suspends the process until there is an event on either of the signals X or Y, at which time execution resumes at the top of the process statement. The process statement thus describes the data transformation and the timing that together constitute the behavior of the half-adder.

Structure is described in VHDL by declaring signals and connecting the signals to the ports of subcomponents. The following example illustrates building a full-adder out of two half-adders and an OR gate (the corresponding drawings can be seen in Figures 3.2 and. 3.3). The signal Temp_sum connects the sum output of the first half-adder to one input of the second half-adder. The signal Temp_carry_1 connects the carry output from the first half-adder to one input of the OR gate; the signal Temp_carry_2 connects the carry output from the second half-adder to the other input of the OR gate. These connections are specified in three *component instantiation statements.* Each of the-three component instantiation statements names a *component* (Half_adder, Or_gate) that has been declared in a local *component declaration.*

```
entity Full_adder is
  port (
      A : in Bit ; B : in Bit ; Carry_in : in Bit ;
      AB : out Bit ; Carry_out : out Bit) ;
  end Full_adder ;
architecture Structure of Full_adder is
```

```
    -- signal declarations
    signal Temp_sum : Bit ;
    signal Temp_carry_1 : Bit ;
    signal Temp_carry_2 : Bit ;

    -- local component declarations
    component Half_adder
        port (X : in Bit ; Y : in Bit;
            Sum : out Bit ; Carry : out Bit) ;
    end component;
    component Or_gate
        port (In1 : Bit ; In2 : Bit ; Out1 : out Bit) ;
    end component;

begin

    -- component instantiation statements
    U0: Half_adder
        port map (
            X => A, Y => B,
            Sum => Temp_sum, Carry => Temp_carry_1):
    U1: Half_adder
        port map (
            X => Temp_sum, Y => Carry_in,
            Sum => AB, Carry => Temp_carry_2) ;

    U2: Or_gate
        port map (
            In1 => Temp_carry_1, In2 => Temp_carry_2,
            Out1 => Carry_out);
end Structure ;
```

Each component instantiation statement is labeled with an identifier (UO, U1, U2) followed by a colon (:). Besides naming a component (Half_adder, Or_gate) declared in a local component declaration, a component instantiation statement contains an *association list* (this is the parenthesized list following the reserved words port map) that specifies which *actuals* (the signals Temp_sum, Temp_carry_1, and Temp_carry_2, and the ports A, B, Carry_in, AB, and Carry_out) are associated with which *locals* (the ports of the local component declaration). An actual associated with a local port may itself be either a signal or a port. VHDL signals and ports, although they are declared differently, both have projected waveforms (values scheduled to become current at future times), and are both said to belong to the class of signal objects (other classes of objects are *variables* and *constants*). Each element in the example consists of the name of a local port followed by a right arrow (=>) and the name of an actual (signal or port).

Scanned with CamScanner

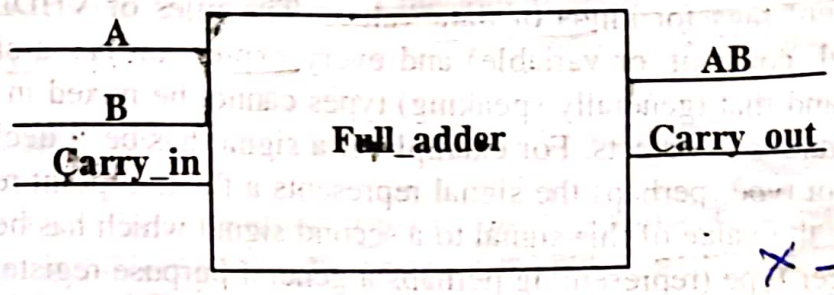



Figure 3.2. Full-adder: Entity Declaration

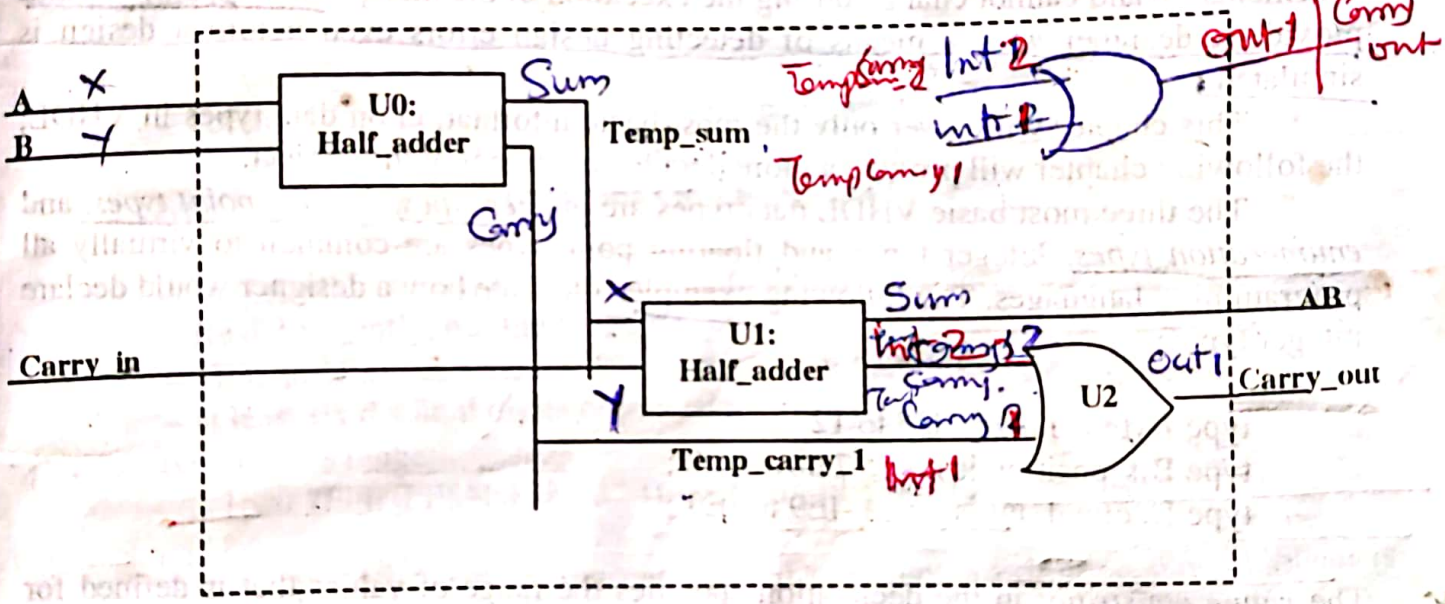



Figure 3.3. Full-adder Architecture Body

## Data Types and Objects

Something that can hold a value (a signal, for example) is an object. In VHDL, every object has a type, the type determining the kind of value the object can hold. In the preceding examples, all signals and ports have the type "Bit"; however, VHDL provides a designer with a rich variety of data types.

## Data Types

If designers always described hardware at the lowest level of abstraction a single data type would suffice. One candidate for this all-purpose data type would be a three-level logic type with the values '0', '1', and 'Z'. However, there are many descriptive goals for which the three-level logic type would simply not be adequate. For example, a hardware description language should make it possible to specify the requirements for a floating point processor in terms of transformations on real numbers. Similarly, a hardware description language should make it possible to specify the top-level design of a microprocessor in terms more abstract than '0', '1' and 'Z'. VHDL was intended to support design at multiple levels and across multiple technologies; consequently, VHDL must provide a means for designers to define their own data types. But VHDL types are

more than convenient tags for kinds of data values. The rules of VHDL stipulate that every object (signal, constant, or variable) and every expression has a single, uniquely determinable type and that (generally speaking) types cannot be mixed in expressions or in assignments of values to objects. For example, if a signal has been declared to have a certain floating point type (perhaps the signal represents a floating point register), then it is an error to assign the value of this signal to a second signal which has been declared to have a certain integer type (representing perhaps a general purpose register). The type of every object and expression can be determined *statically*; that is, the types can be determined prior to simulation — simply by analyzing the VHDL declarations and statements — and cannot change during the execution of the model. Thus types in VHDL provide a designer with a means of detecting design errors even before a design is simulated.

This chapter will cover only the most basic information on data types in VHDL; the following chapter will provide a more detailed treatment of the subject.

The three most basic VHDL data types are *integer types*, *floating point types*, and *enumeration types*. Integer types and floating point types are common to virtually all programming languages. The following examples illustrate how a designer would declare integer types:

```
type Byte is range - I28 to 127 ;
type Bit_position is range 7 downto 0 ;
type Decimal_int is range -IE9 to IE9 ;
```

The *range constraint* in the declaration specifies the range of values that is defined for the type; values outside this range are not legal values for an object of this type. Floating point type declarations are similar in appearance but specify ranges with floating point numbers rather than with integers:

```
type Fraction is range -1 + 0.1E-10 to 1 - 0.1E10 ;
```

Enumeration types have as their values *enumeration literals*. An enumeration literal is either an identifier or a *character literal*. Identifiers were discussed above in the section on entity declarations. A character literal is a single printable ASCII character enclosed in single quotes. The following are examples of character literals:

```
'a'
'A'
' '
''
'%'
'"'
'6'
```

While identifiers are not case-sensitive, character literals are case- sensitive; thus the character literal 'a' is different from the character literal 'A'. An enumeration type declaration lists (between parentheses) the identifiers or character literals that are legal values of the type:

```
type Three_Level_logic is ('0', '1', 'Z') ;
```

VHDL has a small number of *predefined* types, that is, types which are available for use by the designer without the designer having to explicitly declare them. Among these predefined types are the integer type "Integer", the floating point type "Real", and the enumeration types "Boolean", "Bit", "Severity level", and "Character". Here are the definitions of these types:

```
-- VHDL allows each implementation to define the range of
-- Integer differently, but the range must be at least
-- (2**31-1) to (2**31-1)
type Integer is range -2147483648 to 2147483647 ;

-- VHDL allows each implementation to define the range of
-- Real differently, but the range must be at least
-- 1E38 to 1E38, and the implementation must support
-- at least six decimal digits of precision.
type Real is range
    - 16#0.7FFF_FF8#E+32  to  16#0.7FFF_FF8#E+32 ;

-- Type Boolean is really an enumerated type, not a primitive type Boolean is
(False, True) ;

-- Both Boolean and Bit are two-valued enumeration types, but
-- their literals are distinct.
type Bit is ('0', '1') ;

- Type Severity_level is used in assertion statements.
type Severity_level is (Note, Warning, Error, Failure);

-- Type Character has one literal for each of the
-- 128 ASCII characters; the nonprintable characters
-- are represented by identifiers.
type Character is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
    DLE, DC1, DC2, DCS, DC4, NAK, SYN, ETB,
    CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
```

Scanned with CamScanner

‘ ‘, ‘!’, ‘”’, ‘#’, ‘$’, ‘%’, ‘&’, ‘’’,
‘(’, ‘)’, ’*’, ’+’, ‘,’, ‘-’, ‘.’, ‘/’,
‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’,
‘8’, ‘9’, ‘ :’, ‘ ;’, ‘<’, ‘=’, ‘>’, ‘ ?’,

’@’, ’A’, ’B’, ’C’, ’D’, ’E’, ’F’, ’G’,
‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’,
‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’
’X’, ’Y’, ’Z’, ’[’, ‘\’, ‘]’, ‘^’, ‘_’,

‘’’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’,
‘h’, ‘I’, ‘j’, ‘k’, ‘l’, ‘m’, ‘n’, ‘o’,
‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’,
‘x’, ‘y’, ‘z’, ‘{’, ‘|’, ‘}’, ‘~’, DEL ) ;

In addition to these predefined types, the predefined types "Time" deserves mention among the basic VHDL types. The declaration of type "Time", which is discussed in Chapter 4, allows a user to specify time expressions in units such as picosecond, nanosecond, microsecond, etc.

**Objects**

There are three classes of objects in VHDL: signals, variables, and constants. A signal and a variable can both be assigned a succession of values, while a constant is assigned a value one time only (when it is declared). A variable differs from a signal in that the value assigned to a signal does not become the current value until some time in the future while the value assigned to a variable becomes the current value immediately. Unlike signals, which are analogous to wires in hardware, variables have no direct analogs in hardware. Rather, variables are useful in doing the computations required for higher-level modeling of hardware behavior. VHDL is a strongly typed language; every object has a type and can only hold values of that type. The type of an object is fixed in its declaration. The following are examples of object declarations:

```
constant ROM_size : Integer := 16#FFFF# ;
variable In_fetch : Boolean ;
signal Enable : Bit ;
signal CLK, CLEAR : Bit := '0' ;
   -- declares two signals
variable Address : Integer range 0 to ROM_size ;
   -- declares a variable of type Integer with
   -- an added range constraint
```

The general form of an object declaration is

*object-class identifier-list: subtype-indication signal-kind*
*:= expression*

The object class is either constant, variable, or signal. The identifier list is one identifier or multiple identifiers separated by commas (,). The subtype indication specifies the type of the objects declared by the object declaration; the subtype indication may simply name a type or, as in the case of the variable ROM_size, it may name a type with an additional constraint (constraints on types are discussed in Chapter 4). The signal kind is either bus or register; the signal kind is optional and may only appear in a signal declaration. The significance of bus and register will be explained in Chapter 5. The colon-equal (:=) and expression specify an initial default value; this is optional in an object declaration.

VHDL has a large number of rules relating to type compatibility. Among the more important of these rules are the following:

- The type of an expression assigned to an object must be the same as the type of the object.
- The operands of many predefined operators (e.g. arithmetic and relational operators) must be of the same type.
- The type of an actual must be the same as the type of the formal it is connected to.

The declarations of "ROM_size", "In_fetch", and "Enable" in the immediately preceding examples were *explicit object declarations.* But there are other kinds of objects in VHDL, including the following:

- Ports. All ports are of the object class signal. Ports can be declared in entity declarations, component declarations, and block statements. (Entity declarations are discussed below. Component declarations are discussed in Chapter 6. Block statements are discussed in Chapter 7.)
- Generics. Generics are a way of passing environment information-to subcomponents; they can be declared in the same constructs in which ports can be declared. All generics are of the object class constant (Generics are discussed in detail in Chapter 6.)
- Parameters. Parameters are declared on subprograms (functions and procedures). Parameters on functions are of the object class constant; parameters on procedures may be of any object class. (Subprograms are discussed in a later section of this chapter.)
- Indexes on loop statements and generate statements. (Loop statements are discussed in Chapter 5; generate statements are discussed in Chapter 6.)

## Hooking Constructs Together

The example of the full-adder involved an *interface list* that declared local ports in a component declaration and an *association list* in a component instantiation statement that associated actuals with the local ports. Interface lists and association lists are very basic features of VHDL; essentially, they are the means that a designer uses to hook together separate constructs in the language.

### Interface Lists

There are four VHDL constructs that specify *interfaces* | otential points of communication between separate units. These are the entity declaration, the local

component declaration, the block statement, and the subprogram (function or procedure) specification. An *list* is a parenthesized list of *interface elements*, separated by semicolons (;). Each interface element declares one or more interface objects, and each such object is a separate potential point of communication. The objects in a single interface element have three things in common:

- Object class (signal, constant, or variable)
- Mode (direction of data flow: in, out, inout, buffer) 1
- Data type (for example, "Bit"; data types are discussed briefly in this chapter and more fully in the next chapter) |

Each entity declaration, each component declaration, and each block statement has (possibly) two interface lists: one for ports and one for generics. A subprogram specification has one interface list for subprogram parameters. The ports and generics of an entity declaration? or of a block statement are referred to as *formal ports* and *formal generics;* the ports and generics of a local component declaration are referred to as *local ports* and *local generics.* The interface objects of a subprogram specification are referred to as *formal subprogram parameters* or simply *formal parameters.* The following table shows which modes and classes are allowed for each of the three types (port, generic, parameter) of interface object:

| Interface Lists in Entities, Components, Blocks | | |
|---|---|---|
| Interface Object | Class of Object | Allowed Modes |
| port | signal | in out inout buffer |
| generic | constant | in |

| Interface Lists in Subprograms | | |
|---|---|---|
| Interface Object | Class of Object | Allowed Modes |
| parameter parameter | signal | in out inout |
| parameter | constant | in |
| | variable | in out inout |

The general form for a single interface element in an interface list is:

*object-class identifier-list* : *mode subtype-indication* bus
:= *expression*

As in ordinary object declarations (see above), the *object class* is either constant, signal, or variable. The *mode* is either in, out, inout, or buffer (only a port can be of mode buffer; buffer ports will be explained in Chapter 6). The object class and/or the mode of an interface element may be omitted; the defaults that apply are discussed under the various kinds of inteface lists. The reserved word bus is optional but may appear only if the object class is signal; bus signals are discussed in Chapter 5. The colon-equal (:=) and expression together indicate a default initial value to be applied to the objects declared by the interface element; they are optional. Here are some example interface elements:

Scanned with CamScanner

```
-- the minimal amount of information
X : Bit

-- with more than one identifier
X, Y : Bit

-- with class, mode, and default initial value
constant X, Y : in Bit := '0'

-- with a subtype indication consisting of a type and a
-- range constraint
signal X, Y : out Integer range -100 to 100 bus := 0
```

As the last two examples show, several interface elements sharing the same class, mode, type, and default initial value can be grouped into a single interface element containing an *identifier list*. The identifiers in an identifier list are separated by commas (,).

The interface element is almost identical in form to the explicit object declaration. There are two differences. First, a semicolon is actually *a pan of* each object declaration, while a semicolon appears *between* two interface elements. Second, a signal declared in an object declaration can have a signal kind of either bus or register, while an interface signal (port or parameter) can only have the signal kind bus.

## Association Lists

There are four constructs that *specify associations*, or actual communication paths between separate units. These are the component instantiation statement, binding indication, block statement, and subprogram (function or procedure) call. (While a component instantiation statement specifies connections to a local component, a binding indication specifies connections to an entity; this will be explained in Chapter 6.) An *association list* is a parenthesized list of *association elements*, separated by commas (,). Each association element specifies a connection between an actual (an object or expression) in one unit and a formal or local in another unit. Each component instantiation statement, each binding indication, and each block statement has (possibly) two association lists: one for associating actuals with ports and one for associating actuals with generics. A subprogram call has one association list for associating actuals with parameters. Component instantiation statements are said to associate actuals with locals; subprogram calls, binding indications, and block statements are said to associate actuals with formals.

In the example of the full-adder, each association element was in a notation known as *named association*. In *named association*, the formal (or local) as well as the actual is specified: the formal (or local) occurs to the left of the right arrow (=>) and the actual occurs to the right of the right arrow. Since named association specifies unambiguously one formal (or local), it is not required that the order of associations be the same as the order in which the formals (or locals) were declared in the interface list.

Scanned with CamScanner

There is an alternate notation for an association element known as *positional association*. In *positional association*, an association element specifies only an actual; the formal (or local) is inferred by matching the ordinal position of the actual in the association list with the ordinal position of a formal (or local) in the interface list. It is possible for an association list to have some association elements in named notation and some in positional notation; however, no positional associations may follow a named association in an association list Here is the first component instantiation statement from the full-adder example rewritten (with no change in meaning) in positional notation:

Full

U0: Half_adder port map (A, B, Temp_sum, Temp_carry_l) ;

## Major VHDL Constructs

This section discusses a number of the major building blocks available to the VHDL designer: the entity declaration, the architecture body, the subprogram, the package declaration, and the package body.

## Entity Declarations

Examples of entity declarations were given at the beginning of this chapter. The general form of the entity declaration is

```
entity identifier Is
   generic interface-list ;
   port interface-list ;
   declarations
begin
   statements
end identifier ;
```

General form of entity declarations

As mentioned earlier generics are a way for an instantiating component to pass environment information to an instantiated subcomponent formal generic will be discussed in Chapter 6. Formal ports were seen in the entity declarations of the half-adder and the full-adder and will be more fully discussed in Chapter 6.

The entity declaration is one of several major VHDL constructs that may themselves contain declarations. These major constructs are the entity declaration, architecture body, package declaration, package body, and subprogram body - all of which are discussed in ~~this~~ this chapter – as well as the block statement and process statement, which are discussed in later chapters. There is a group of declarations, each of what can occur in each of these major constructs. This group, called the group of *basic declarations*, includes the following:

*type declaration*
*subtype declaration* (explained in Chapter 4)
*constant declaration*
*file declaration* (explained in Chapter 9)

Scanned with CamScanner

*alias declaration* (explained in Chapter 9)
*subprogram declaration* (explained below in this chapter)
*use clause* (explained below in this chapter)

Declarations that can follow the formal ports in an entity declaration include:

*basic declaration*
*signal declaration*
*subprogram body* (explained below in this chapter)
*attribute declaration* (explained in Chapter 9)
*attribute specification* (explained in Chapter 9)
*disconnection specification* (explained in Chapter 9)

An entity declaration can also contain a set of *entity statements* after for reserved word begin. It may seem odd that an entity declaration, which is intended to represent only the interface to a component, should be; allowed to contain statements. However, entity statements are not intended to represent a component's dynamic behavior, instead, entity statements are intended to represent checks or assertions about the component's interface. For example, an entity statement might be used to check that the values of a preset port and a clear port on a flip-flop are not both '0'. Placing this check in the entity declaration rather than in the architecture body (where the rules of VHDL would also allow it to be placed) means that the check will be present in all design entities built from the entity declaration and that the check need not be duplicated in each architecture body representing an alternative implementation of the component. Given this restricted function of entity statements, it is appropriate that the only kinds of statements that are allowed in entity declarations are:

*concurrent assertion statement*
*concurrent procedure call*
*process statement*

(These statements are discussed in Chapter 5.) In keeping with the intention that entity statements do not represent dynamic component behavior, there is a further restriction on any concurrent procedure call or process statement occurring in an entity declaration: they must not assign values to any signal object (signal or port). A concurrent procedure call or process statement that does not assign to any signal object is called a *passive* concurrent procedure call or process statement.

Any of the following elements of the full form may be omitted in an entity declaration: the formal generic declarations; the formal port declarations; the other declarations; the reserved word begin and the concurrent statements; and the closing identifier. Thus, the following minimal entity declaration is legal VHDL:

```
entity E is
end ;
```

Scanned with CamScanner

While this entity declaration might appear to describe a singularly uninteresting piece of hardware with no inputs and no outputs, it is not an entirely useless component. It could, for example, represent a self-contained testbench setup; the architecture of this empty entity might instantiate three subcomponents: a unit under test 2, a test vector generator (possibly one that read in values from a file using VHDL file I/O), and a comparator that checked actual results against predicted results.

**Architecture Bodies**

Examples of architecture bodies were given at the beginning of this chapter. The general form of the declaration of an architecture body is

**architecture** *identifier* of *entity-mark* is
  *declarations*
begin
  *statements*
end *identifier* ;

Declarations that can occur in an architecture body include:

*basic declaration*
*signal declaration*
*subprogram body*
*attribute declaration*
*attribute specification*
*disconnection specification*
*component declaration*
*configuration specification* (discussed in Chapter 6)

Any concurrent statement may occur in an architecture body. Concurrent statements (which are discussed in Chapters 5 and 6) include:

*concurrent signal assignment statement*
*concurrent assertion statement*
*concurrent procedure call*
*process statement*
*component instantiation statement*
*block statement*
*generate statement*

The declarations and/or statements may be omitted in an architecture body, but an architecture body with no statements has no apparent design function.

Scanned with CamScanner

**Subprograms**

VHDL provides a number of predefined operators — such as "+" for addition, "<" for the less-than relation, "xor" for the logical exclusive-or operation — for the construction of expressions that compute values. These and other predefined operators will be discussed in the following chapter. The predefined operators have a fixed number of operands whose types must conform to certain rules. But VHDL also allows a designer to define subprograms, which can be seen as a generalization of operators. Like the operators, they are defined for a fixed number of operands (parameters) whose types must conform to the types in the subprogram declaration. They are more general than operators in two respects. First, a subprogram may be devised for any number of operands (parameters) and any combination of types. Second, while there are subprograms that, like operators, merely compute values without affecting the values of objects or controlling the flow of execution, there are also subprograms (procedures) that assign new values to objects and even suspend execution.

A subprogram is a sequence of declarations and statements that can invoked repeatedly from different locations in a VHDL description. VHDL provides two kinds of subprograms: procedures and functions. Procedures differ from functions in that the invocation of a procedure is a statement while the invocation of a function is an expression. VHDL distinguishes between a subprogram declaration and a corresponding subprogram body. The subprogram declaration contains only interface information, while the subprogram body contains interface information, local declarations, and statements. The distinction between subprogram declaration and subprogram body is analogous to the distinction between entity declaration and architecture body: it is a way of separating interface from algorithm (or, in the case of the architecture body, from structure).

A subprogram declaration is most conveniently described as consisting of a *subprogram specification* followed by a semicolon (;). There are two forms of subprogram specifications, one for a procedure, and one for a function.

**procedure** *identifier interface-list*
**function** *identifier interface-list* **return** *type-mark*

Functions are intended to be used strictly for computing values not for changing the value of any objects associated with the function formal parameters; therefore, all parameters of functions must be of mode in and must be of class signal or constant. Procedures, on the other hand, are permitted to change the values of the objects associated with the procedure formal parameters; therefore, parameters of procedures may be of mode in, out, and inout. If no mode is specified for a subprogram parameter, the parameter is interpreted as having mode in. If no class is specified, parameters of mode in are interpreted as being of class constant and parameters of mode out and inout are interpreted as being of class variable. It is possible to define a procedure or function with no parameters; in this case, the interface list is simply omitted from the subprogram specification. Note that a function, unlike a procedure, must declare the type of the value the function will return. Examples of subprogram declarations follow:

Scanned with CamScanner

```
-- procedure declaration with no parameters
procedure P ;

-- function declaration with no parameters
Function Limit return Real ;

-- object class of parameter is implicit
procedure Mod_256 (X : inout Integer);
-- object class of parameter is explicit
procedure Mod_256 (variable X : inout Integer) ;

-- object class of parameter is implicit
Function Mod_256 (X : Integer) return Byte ;
-- object class of parameter is explicit
function Mod_256 (constant X : in X) return Byte ;

-- parameters of object class signal
Procedure Xor2 (signal In1, In2 : Bit; signal Out1 : out Bit);
```

The full form of a subprogram (procedure or function) body is:

*Subprogram-specification is*
*declarations*
begin
*statements*
end *identifier* ;

The declarations and/or statements may be omitted. However, when a function is called, the function call must terminate by executing a *return statement*, which determines the value returned by the function call (return statements are discussed in Chapter 5). Hence, any function call to a function with an empty body will result in an execution error. The closing identifier (after end) is also optional. Any of the following declarations may occur in a subprogram body:

*basic declaration*
*variable declaration*
*subprogram body*
*attribute declaration*
*attribute specification*

While a subprogram may call other subprograms, it may not instantiate components; therefore, a subprogram should not need to declare signals to wire together subcomponents. Note that a subprogram may itself declare "nested" subprograms.

Here is a simple subprogram body:

Scanned with CamScanner

```
function Min (X, Y : Integer) return Integer is
begin
    if X< Y then
       return X ;
    else
       return.Y ;
    end if;
  end Min ;
```

## Packages and Use Clauses

Data types, constants and subprograms can be declared inside entity declarations and inside architecture bodies, and the types, constants, and subprogram declarations that are declared inside an entity declaration are visible in (can be used by) its associated architecture bodies. However, the data types, constants, and subprograms declared inside entity declarations and architecture bodies are not visible to other entity declarations or their architecture bodies. To make a set of type, constant and subprogram declarations visible to a number of design entities (entity declarations and their associated architecture bodies), VHDL provides the package. The general form of a package declaration is

```
package identifier is
   declarations
end identifier ;
```

Both the declarations and the closing identifier are optional however, an empty package declaration does not serve any apparent design purpose.

The following declarations may occur in the declarations portion of a package declaration;

*basic declaration*
*signal declaration*
*attribute declaration*
*attribute specification*
*disconnection specification*
*component declaration*

The following example illustrates a package declaration.

```
package Logic is

   type Three_level_logic is ('0', '1', 'Z') ;
   constant Unknown_value : Three_level__logic := '0' ;
   function Invert (
      Input : Three_level_logic)
```

Scanned with CamScanner

```
        return Three_level_logic ;

    end Logic ;
```

The general form of a package body is

```
    package body identifier is
      declarations
    end identifier;
```

A package body always has the same name as its corresponding package declaration, preceded by the reserved words package body. Declarations that may occur in a package body include:

*basic declaration*
*subprogram body*

As with the package declaration, the package body may be empty, and us closing identifier is optional. Note that a package body contains the subprogram bodies whose corresponding subprogram declarations occur in the package declaration. The following is an example of a body for package Logic given above:

```
    package body Logic is
    -- Subprogram body of function Invert
    function Invert (
      Input : Three_level_logic)
    return Three_level_logic is
    begin
      case Input is
          when '0' =>
            return '1' ;
          when '1' =>
            return '0' ;
          when 'Z' =>
            return 'Z' ;
          end case ;
      end Invert;

    end Logic ;
```

Package bodies contain subprogram bodies and declarations that are not intended to be used by other VHDL units. Thus, package declarations contain the public, visible declarations while package bodies contain the private, invisible declarations.

Scanned with CamScanner

Items declared inside a package declaration are not automatically visible to another VHDL unit. A use clause preceding a unit will make items declared in a package declaration viable in the unit.

```
-- The declaration of package Logic (above) is assumed
-- The following use clause makes the declarations of
-- Three_level_logic and Invert visible to the entity declaration use Logic.
Three_level_logic, Logic.Invert;
entity Inverter is
   port (X : in Three_level_logic ; Y : out Three_level_logic) ;
end Inverter ;

-- The architecture body inherits the visibility of its entity
-- declaration, so the use clause does not need to be repeated, architecture
Inverter_body of Inverter is
begin
   process
   begin
         Y <= Invert (X) after 10 ns; - A function call
         wait on X ;
   end process ;
end Inverter_body ;
```

A use clause may as in the immediately preceding example, name specific items declared in a package. However, an alternate form of the use clause, constructed with the reserved word all, makes all items declared in a package declaration visible.

```
-- The following use clause makes all declarations
-- in package Logic visible to the entity declaration.
use Logic.all ;
entity Inverter is
   port (X : in Three_level_logic; Y : out Three_level_logic);
end Inverter;

-- The architecture body inherits the visibility of
-- its entity declaration
architecture Inverter_body of Inverter is
begin
   process
   begin
         Y <-= Invert (X) after 10 ns ;
         wait on X ;
   end process ;
end Inverter_body ;
```

Scanned with CamScanner

Additional discussion of packages and use clauses can be found in Chapter 7.

## Libraries

VHDL allows a designer to maintain multiple design libraries, and the various pieces of a single design may reside in different libraries. This section explains the notion of a separately-analyzable library unit and explains how the names of library units and the names of libraries are made visible so that they can be referenced by other units. Additional discussion of libraries can be found in Chapter 7.

### Library Units and Order of Analysis

Analysis is the process of checking a VHDL unit for syntactic and semantic correctness and inserting the VHDL unit (if it is correct) into a library. A library unit is a VHDL construction that can be separately analyzed. There are two classes of library units: primary units and secondary units. The primary units are entity declaration, package declaration and configuration declaration (configuration declarations will be explained later); the secondary units are package body and architecture body. In a given library, there can be at one time only one primary unit with a given name, but there may be multiple secondary units with the same name and there may be secondary units with the same name as a primary unit. Secondary units must be analyzed into the same library as their corresponding primary unit. A library unit is separately analyzable in the sense that the VHDL text for a library unit is not contained inside the text for another VHDL unit. However, "separately analyzable" does not mean "analyzable in any order". There are three rules governing the order in which library units may be analyzed. First, a secondary unit can be analyzed only after its corresponding primary unit has been analyzed. Second, any library unit that references a primary unit can only be analyzed only after that primary unit has been analyzed. Third, an architecture body referenced in a configuration declaration must be analyzed before the configuration declaration; likewise, one configuration declaration referenced in a second configuration declaration must be analyzed before that second-configuration declaration.

Since the name of a package body is always the same as the name of its corresponding package declaration, there can never be more than one package body for a given package declaration. Architecture bodies are not subject to this same restriction. The name of an architecture body may be different from the name of its corresponding entity declaration, and it is possible to have multiple architecture bodies corresponding to a single entity declaration. This gives the designer the ability to maintain multiple hardware designs for a single hardware interface or to maintain abstract, behavioral implementations in addition to concrete, structural implementations. Figure 3.4 shows a sample library with five library units.

### Visibility of a Primary Unit and Libraries

The name of a primary unit (for example, a package declaration) not automatically visible to other units. A variant of the use clause (with the name of a library to the left of the dot and the name of a primary unit to the right of the dot)

Scanned with CamScanner

provides the needed visibility. The following example assumes that the package Logic has been analyzed into the library Basic_library.

```
-- The first use clause makes the name of package Logic
-- visible; the second use clause makes the declarations
-- in package Logic visible.
use Basic_library .Logic ;
use Logic.all ;
entity Inverter is
```

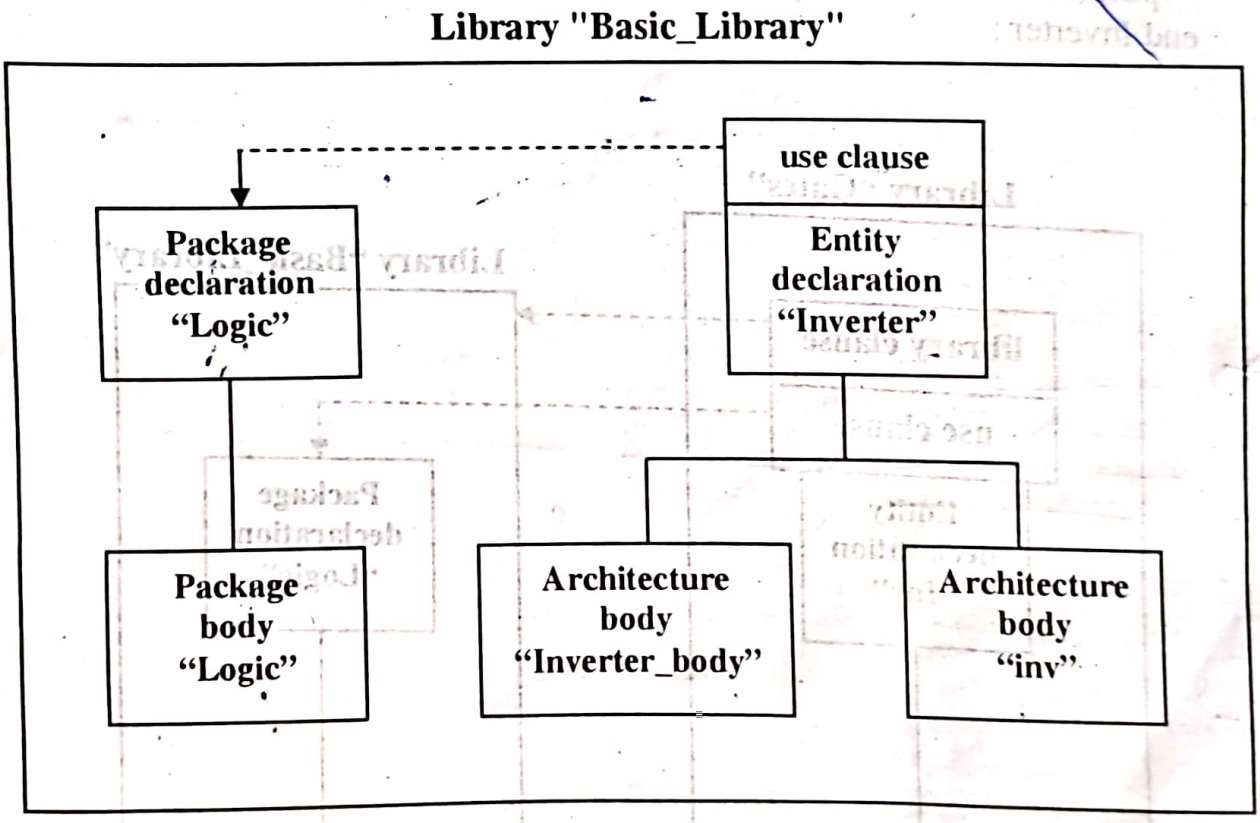



Figure 3.4. Library Units

```
    port (X : in Three_level_logic ; Y : out Three_level_logic) ;
end Inverter;
```

The alternate form of the use clause,

```
use Basic_library.all ;
```

would make visible the names of all primary units in library Basic_library.

Just as an item declared inside a package declaration is not automatically viable and just as the name of a primary unit is not automatically viable, so the name of a library is not automatically visible. But while a use clause provides the needed visibility in the cases of an item declared inside a package declaration and the name of a primary unit, it

Scanned with CamScanner

is a *library clause* that provides the needed visibility in the case of a library. The following VHDL and Figure 3.5 illustrate the use of multiple libraries.

```
-- The following library clause makes the name of library
-- Basic_library visible
library Basic_library ;
use Basic_library.Logic ;
use Logic.all ;
entity Inverter is
   port (X : in Three_level_logic ; Y : out Three_level_logic) ;
end Inverter ;
```

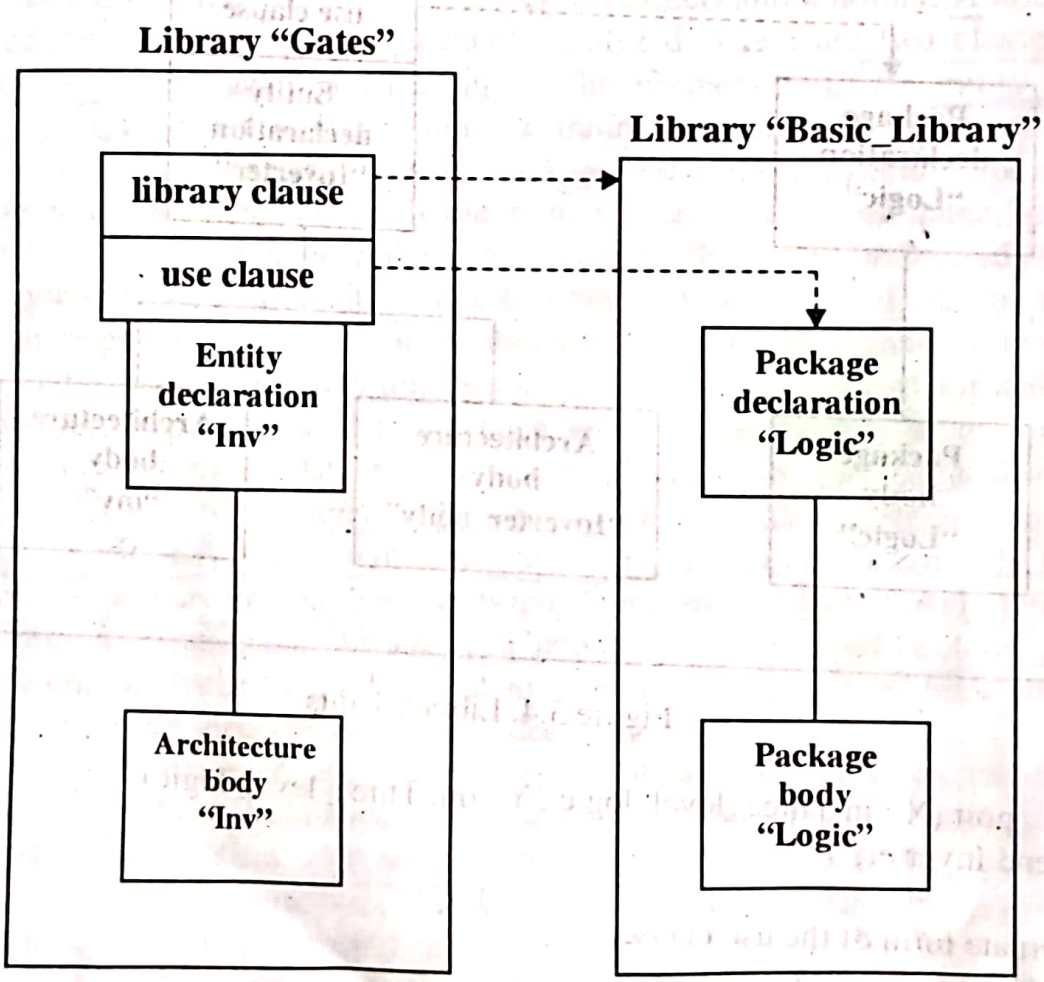



Figure 3.5. Multiple Libraries 1

There is one exception to the rule that the name of library is not automatically visible: the library name "Work" is always visible (just as if the library clause "library Work;" actually preceded every unit) Furthermore, the library name "Work" always designates the library which the current library unit is being analyzed into. "Work" is thus

Scanned with CamScanner

a logical name, whose interrelation is dependent on installation and, within this installation, on a particular en exists at the time a library unit is analyzed.

A library name can be referenced in VHDL, but VHDL does not give the designer a means of creating a library or defining a library name. Since libraries must be defined outside of VHDL, the use of library names in a VHDL description would make the description more installation-dependent and less portable. The logical library name Work makes it possible to write VHDL descriptions that are portable across design environments.

Scanned with CamScanner

# CHAPTER 4
# DATA TYPES

VHDL permits designers to declare a virtually unlimited number of data types for characterizing the values held by signals, variables, and constants. The previous chapter mentioned briefly three data types: integer types, floating point types, and enumeration types. This chapter provides a detailed description of the other major kinds of data types in VHDL (two types not discussed in this chapter, access types and file types, are discussed in Chapter 9). This chapter consists of six sections. The first section of this chapter explains the various kinds of literals that are used in designating values of different types. The second section describes how scalar types are declared. The third section describes how composite types (arrays and records) are declared, how their values are expressed, and how their elements are referenced. The fourth section explains the notion of a subtype. The fifth section presents various predefined attributes that apply to types and array objects. The sixth section covers the various predefined operators that are used to construct expressions.

## Literals

A literal is a symbol (other than a reserved word like begin or an operator symbol like "+") whose value is immediately evident from its presentation. There are six kinds of literals: integer literals, floating point literals, character literals, string literals, bit string literals, and physical literals.

A floating point literal differs from an integer literal in that floating point literal contains a dot (.) while an integer literal never contains a dot. Both floating point literals and integer literals may be expressed in any base from 2 to 16, and both kinds of literals may have exponents in base 10 (floating point literals, but not integer literals, may have negative exponents). Any two adjacent digits in an integer or floating point literal may be separated by a single underscore Q character; this underscore is strictly to improve legibility.

I.F C S B P

Integer literals
Floating-point literals
Character literal
String literals
bit string literal
Physical literals

```
                2   -- base 10 integer literal
                0   -- base 10 integer literal
         77459102   -- base 10 integer literal
             10E4   -- base 10 integer literal
          16#D2#    -- base 16 (hex) integer literal
          8#720#    -- base 8 (octal) integer literal
     2#11010010#    -- base 2 (binary) integer literal
              1.0   -- base 10 floating point literal
              0.0   -- base 10 floating point literal
       63971.333333 -- base 10 floating point literal
    65_971.333_333  -- equivalent to preceding literal
      8#43.6#e+4    -- base 8 (octal) floating point literal
          43.6E-4   -- base 10 floating point literal
```

literals example.

Scanned with CamScanner

A character literal is a single ASCII character enclosed in single quotes (examples were given in Chapter 3). A string literal is sequence of ASCII characters enclosed in quote marks ("). An occurrence of a quote mark within a string literal is indicated by doubling the quote.

"ABC"
"~^!@#$%&*()_+;:'|,.?"
"doubled "" quote"

} character literal

A bit string literal is a sequence of characters from the sets '0' to '9', 'a' to 'f', and 'A' to 'F', enclosed in quote marks and preceded by a single letter indicating the base (B for binary; 0 for octal; X for hex). Adjacent digits in a bit string literal may be separated with the underscore character; as is the case with integer and floating point literals, the underscore has no function other than to improve legibility.

B"11111111"
B"1111_1111"
X"fF"
X"c3"
O"70"

} bit string

A physical literal consists of an integer or floating point literal plus an identifier denoting a scaling factor or unit of measurement. The following are examples of physical values:

15 ft
10 kohm
2.3 sec

} a physical literal is values.

* An identifier denoting a unit of measurement is also a physical literal by itself; in that case, the integer literal 1 is understood to precede the unit identifier.

**Scalar Types**

A *scalar type* is a type whose values cannot be decomposed into more atomic values. The values of a scalar type can be ordered along a single scale (the *less-than* relation is the ordering scale). The scalar data types include integer types, floating point types, enumeration types, and physical types. Integer, floating point, *2nd* enumeration types were introduced in Chapter 3 and will be reviewed here.

The general form for all type declarations is:

type *identifier* is *type-definition* ; — applicable to all type declarations.

For both integer types and floating point types, the type definition is a *range constraint.* A range constraint consists of the reserved word range and a *range specification.* A rang

specification can be constructed in one of two ways. The simplest form of range specification consists of two expressions and a *direction*. The direction is to for a range which is considered to be ascending and downto for a range which is considered to be descending. The following are range specifications constructed with expressions and a direction;

```
-2**31 to 2**31 - 1
7 downto 0
```

The ability to specify the direction of a range is particularly useful in conjunction with the definition of arrays (discussed later in this chapter). A designer may, for example, want to view a register as having bits arrayed from 7 down to 0 rather than as 0 to 7. The direction of a range is of interest for another reason: VHDL has looping constructs which will iterate over values in one order for an ascending range and in the reverse order for a descending range. The second way to construct a range specification is by using the predefined attribute 'Range. This attribute is explained later in this chapter.

The type definition for an enumeration type is a parenthesized list of enumeration literals, each enumeration literal being either an identifier or a character literal. Each identifier or character literal may appear only once in any given enumeration type definition, but a single identifier or character literal may be used in the definition of several different enumeration types.

```
type Two_level_logic is ('0', '1') '
type Three_level_logic is ('0', '1', 'Z') ;
type Four_level_logic is ('X', '0', '1', 'Z') ;
```

The capability to have multiple meanings for an enumeration literal is actually part of a larger capability in VHDL called *overloading*. In general, an enumeration literal, an operator symbol (like "+" and "xor")t or the name of a subprogram may have multiple meanings, all visible in a VHDL description, provided that a reference to the overloaded item, can be disambiguated on the basis of type information in the context of the reference. Overloading will be discussed in Chapter 9.

The fourth kind of scalar type is the physical type. The physical type definition specifies a range constraint, one *base unit*, and zero or more *secondary units*, each secondary unit being an integral multiple of the base unit.

```
type Resistance is range 1 to 10E9
  units
      ohm; -- the base unit
      kohm = 1000 ohm; -- secondary unit, multiple of base unit
  end units ;
```

There is one predefined physical type in VHDL:

Scanned with CamScanner

```
-- The actual range of type Time is allowed to vary from
-- implementation to implementation, but must be
-- at least from -(2**31-1) to (2**31-1) femtoseconds.

type Time is range -(2**31-1) to (2**31-1)
  units
      fs;                -- femtosecond
      ps  = 1000 fs ;    -- picosecond
      ns  = 1000 ps ;    -- nanosecond
      us  = 1000 ns ;    -- microsecond
      ms  = 1000 us ;    -- millisecond
      sec = 1000 ms ;    -- second
      min = 60 sec ;     -- minute
      hr  = 60 min ;     -- hour
  end units ;
```

The term *numeric type* includes integer types, floating point types and physical types. Numeric types are those for which the operations of addition, subtraction, multiplication, division, modulus, remainder, exponentiation, complement and absolute value are defined.

**Composite Types**

A composite type is one whose values can be decomposed into smaller atomic values. There are two kinds of composite types: records and arrays. Records are heterogeneous composite types; that is, the elements of a record can be of various types. A record type definition specifies one or more elements, each element having a different name and possibly a different type.

```
-- An enumeration type and an integer type that will be used in
-- defining a record type:
type Opcode is
   (Add, Add_with_cany, Sub, Sub_with_carry, Complement) ; type Address is
range 16#0000# to 16#FFFF# ;

type Instruction is record
   Opcode_field      : Opcode ;
   Operand_1         : Address ;
   Operand_2         : Address ;
end record ;

type Register_bank is record
   F0 : Real ;
   F1 : Real ;
   R0 : Integer ;
```

Scanned with CamScanner

```
   R1 : Integer ;
   A0 : Address ;
   A1 : Address ;
   IR : Instruction ;
end record ;

-- Names of elements with the same type can be grouped
-- in identifier lists, as in the following:

type Register_bank is record
   F0, F1 : Real ;
   R0, R1 : Integer ;
   A0, A1 : Address ;
   IR : Instruction ;
end record ;
```

Arrays are homogeneous composite types; that is, the elements *of* an array are all of the same type. Arrays may have one, two, or more dimensions. Each dimension has a type, and this type must be a *discrete type* (integer type or enumeration type). An array type definition may be *constrained* or *unconstrained.* In a constrained array type definition, the bounds of the array are specified, while in an unconstrained array type definition, the bounds are not specified. The ability to declare objects of unspecified size is particularly useful in interface lists. This makes it possible, for example, to declare an entity with a port that is a bundle of a variable number of bits; thus a single entity representing a shift register could serve as an 8-bit, 16-bit, or 32-bit register depending on the bounds of the actual array signal that was connected to the unconstrained! port.

In a *constrained* array definition, the type and range of each, dimension is specified by a *discrete range.* There are two forms of discrete range. The first form is simply a range specification, the same construct that is part of the range constraint in an integer or floating point type definition.

```
type Word is array (15 downto 1) of Bit ;
type Severity_level_stats is array (Note to Failure) of Integer;
```

The second form of discrete range is a *subtype indication. A* subtype indication consists of the name of a type followed optionally by a constraint. The following example declares two integer types, Column and Row, and then declares two two-dimensional constrained array types whose dimensions are specified with subtype indications. In type Matrix, the subtle indications are simply the names of the types Column and Row; thus the array has bounds of 1 to 24 and 1 to 80. In type Reduced_matrix, the subtype indications are the names of the Column and Row with further range constraints; the resulting array has bounds of 1 to 10 and 1 to 40.

```
type Column is range 1 to 80 ;
```

Scanned with CamScanner

```
type Row is range 1 to 24 ;
type Matrix is array (Row, Column) of Boolean ;
type Reduced_matrix is array
   (Row range 1 to 10, Column range 1 to 40) of Boolean ;
```

In an *unconstrained* array type, the type of each dimension is given, but the exact range and direction of each dimension, and hence the size of the array, is left unspecified (using the notation range <> ).

```
type Screen is array
   (Integer range <>, Integer range <>) of Pixel;
   -- type Pixel is defined elsewhere
```

There are two predefined unconstrained array types in VHDL: Bit_vector, whose index type ranges over the natural numbers, and String, whose index type ranges over the positive integers.

```
type Bit_vector is array (Natural range <>) of Bit ;
type String is array (Positive range <>) of Character;
```

**Aggregates and String Literals**

Values of composite types can be given in aggregate notation. An aggregate is a parenthesized list of *element associations,* each element association specifying a value for an element of the array or record. Individual element associations are separated by commas. The following are examples of composite type definitions and aggregates of those types:

```
-- Declaration of type Rational:
type Rational is record
   Numerator : Integer ;
   Denominator : Integer ;
end record;

--Aggregate of type Rational:
  (155, 2077)
-- Declaration of type Conversion_array:
Type Clock_level is (Low, Rising, High, Falling) ;
type Conversion_array is array (Clock_level) of Bit ;

-- Aggregate of type Conversion_array:
  ('0', '1', '1', '0')
```

Scanned with CamScanner

Just as there are alternate notations -- positional and named - for an *association element* in an association list (discussed Chapter 3), so there are alternate notations for an *element association* in an aggregate. The aggregate in the above example is given in positional association since the mapping of each element association to an element of the composite type is given implicitly by the left-to-right order. In the alternate notation, named association, the correspondence is made explicit by naming the field of the record type element or the index value of the array element .Thus the following aggregates with named association are equivalent to those given in the immediately preceding example:

```
(Numerator => 155, Denominator => 2077)
(Denominator => 2077, Numerator => 155)

(Low => '0', Rising => '1', High => '1', Falling => '0')
(High => '1'. Rising => '1', Falling => '0', Low => '0')      1
```

In a named element association, the record element name or array index value to the left of the arrow (=>) is called a *choice*. If several element associations specify the same value, then it is possible to group these several element associations into one. In this case, the vertical bar (I) is used to group those choices which are assigned the same value. Thus

```
(Low => '0', Rising => T, High => T, Falling => *0')
```

is equivalent to

```
(Low I Falling => '0' Rising I High => '1')
```

and

```
(Numerator => 1, Denominator => 1)
```

is equivalent to

```
(Numerator I Denominator => 1)
```

There are two shorthand ways of grouping choices. First, in an array aggregate, a choice may be a discrete range (as explained above, there arc two forms of discrete range: a range specification and a subtype indication).

```
type Hex_letter is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        'A', 'B', 'C', 'D', 'E', 'F') ;
type Hex_index is array (Hex_letter) of Boolean ;
constant Is_decimal_digit : Hex_index :=
        ('0' to '9' => True, 'A' to 'F' => False) ;
```

Scanned with CamScanner

```
-- Discrete ranges (range specifications) as choices

type Eight is range 1 to 8 ;
type Eight_digit is array (Eight) of Hex_letter ;
variable Hex_value : Eight_digit := (Eight => '0') ;
  -- Discrete range (subtype indication consisting of the
  -- name of a type without the additional range constraint)
  -- as a choice
```

Second, in either a record aggregate or an array aggregate, the reserved word others can be used to designate all remaining choices (that is, those choices, if any, not specified so far).

```
constant One : Rational := (others => 1) ;
constant Is_decimal_digit : HexJndex :=
        ('0' to '9' => True, others => False) ;
variable Hex_value : Eight_digit := (others => '0') ;
```

VHDL makes available a special kind of representation — the string literal — for values of one-dimensional arrays whose element type an enumeration type containing character literals. The following example illustrates the use of a string literal to specify a value of a one-dimensional array.

```
type Four_level_logic is ('X', '0', '1', 'Z') ;
type Pair is array (1 down to 0) of Four_level_logic ;
constant Reset: Pair := "OZ" ;
  -- equivalent to (1 => '0', 0 => 'Z')
constant dear : Pair := "ZO" ;
  -- equivalent to (1 => 'Z', 0 => '0')
```

**Referencing Elements of Composites**

An object of a composite type may be referenced in its entirety or by element. The simple name of an array or record is a reference to the entire array or record. An indexed name is used to reference an element of an array. An indexed name consists of the name of the object, followed by a parenthesized list of index expressions, one index expression for each dimension of the array. The type of an indexed name is the element-type of the array.

```
type Column is range 1 to 80 ;
type Row is range 1 to 24 ;
type Matrix is array (Row, Column) of Boolean ;

signal S : Matrix ;
```

Scanned with CamScanner

-- References (indexed names) to elements of signal S:

```
S (1, 1)        -- type is Boolean
S (3, 14)       -- type is Boolean
```

A slice name is a reference to a contiguous subset of elements of a one-dimensional array (it is not possible to construct a slice name for a multi-dimensional array). The slice is constructed by appending a parenthesized discrete range to the name of the one-dimensional array. The type of a slice name is the same as that of the array being sliced.

```
type Byte is array (7 downto 0) of Bit ;
type Memory is array (0 to 2** 16-1) of Byte ;

signal S_byte : Byte ;
signal S_memory : Memory ;

-- References (slice name) to contiguous elements of
-- array signals:

-- slice of three elements
S__byte (3 downto 1)

-- slice of 2** 15 elements
S_memory (2**15-1 to 2**16-1)

-- slice of one element
S_memory (0) (0 downto 0)
```

It may seem counter-intuitive that the slice S_byte (3 downto 1) with three elements should have the same type as the declared array S_byte with eight elements. This is explained by the way VHDL defines a constrained array type such as Byte. A constrained array type declaration is actually defined to be a subtype (subtypes are explained below) of an anonymous unconstrained array type; the type is anonymous since it was not explicitly declared by the designer and therefore has no name. The *base Type* (underlying type) of the slice S_byte.(3 **downto** 1) is this anonymous unconstrained array type, as is the base type of the whole signal S_byte. The difference between the slice arid the whole "signal is a difference in index constraint, not a difference in base type.

A selected name is a reference to an element of a record. A selected name consists of the name of the object, followed by a dot (.), followed by the name of the element. The type of a selected name is the type of its element.

```
type Rational is record
   Numerator : Integer ;
```

Scanned with CamScanner

```
    Denominator : Integer ;
end record;

signal S : Rational;
-- References (selected names) to elements of signal S:
S.Numerator   --type is Integer
S.Denominator        --type is Integer
```

**Subtypes**

A type declaration defines a domain of values that are completely different from the values in the domain defined by any other type declaration. Sometimes it is known that an object will take on values from only a of the values defined by the type. To capture design restrictions, VHDL provides the notion of a subtype. A subtype declaration defines a subset of the values of a type by specifying a constraint on the type; a subtype is a type with a constraint. Since their values are drawn from the same domain, subtypes of a type are fully compatible with each other; however, an object declared to be of a particular subtype may not take on a value that lies outside the subset of values defined for the subtype even if that value is in the domain of the type.

Types and subtypes both provide the hardware designer assistance in checking design correctness. But the type and subtype mechanisms detect different kinds of design errors, and they detect the errors at different times in the development process. A designer can use types to detect incorrect data paths — wire connections that are in error. For example, it is an error to connect a signal of type Integer to a port of type Real (unless some type conversion is applied at the connection; type conversions are discussed in Chapter 7). Detecting such errors requires only a knowledge of the types of the objects that are being connected; a knowledge of the actual values passing along the data paths in simulation time is not required. As a result, errors involving type compatibility can be detected early in the development process, when a design unit is first analyzed. In contrast, a designer can use subtypes to detect incorrect assumptions about the actual values (for example their magnitude or sign) that an object may assume. Suppose a register of type Integer is connected to a signal of type Integer, and suppose, Anther, that the designer has assumed that the register will never hold values outside the range 0 to 127. If, in the course of simulating the design, a value of 128 is loaded into the register, the design has been based on a mistaken assumption. To detect such mistaken assumptions the designer can declare the port on the register to be a subtype off Integer with the restricted range 0 to 127. Checking subtype compatibility involves checking the actual values taken on by an object against a subtype constraint; each time an object assumes a new value, the new value must be checked to see if it falls within the allowed range of values. Thus, in general, subtype checks cannot be performed until a design is simulated.

There are two ways a subtype declaration can constrain a type. The subtype can specify a range constraint that defines a subset of values of a scalar type; a value is in the subset if it falls within the range specified by the bounds and the direction (**to** or **downto** ) of the range constraint. Or the subtype can specify an index constraint that for each

Scanned with CamScanner

dimension of an unconstrained array type, defines a subset of the values defined by the discrete type of the dimension, and thereby determines a subset of the values of the array type.

```
-- Range constraint:
subtype Lowercase_letters is Character range 'a' to 'z' ;

-- Index constraint:
subtype Register is Bit_vector (7 downto 0) ;
```

Since the entire domain of values of a type is a subset of itself, it is also proper to speak of a subtype that imposes no constraint on the type. For this reason, we can say that every type declaration in VHDL also defines a subtype of the same name, and the word "subtype" can refer both to something declared in a subtype declaration and to something declared in a type declaration. The general form of a subtype declaration is

**subtype** *identifier* is *subtype-indication* ;

A *subtype indication* consists of the name of a type or subtype followed by an optional constraint. The optional constraint is either a range constraint, such as occurs in integer and floating point type declarations, or an index constraint, such as occurs in constrained array type declarations. A subtype indication that lacks an explicit constraint denotes a subtype with exactly the same constraint as the constraint on the subtype named in the type mark. Thus

```
subtype ASCII is Character ;
```

simply creates another name for Character.

A subtype indication may also contain the name of a function, immediately preceding the name of the type or subtype. The type returned by the function must be the same as the type named in the subtype indication. Furthermore, such a function must be a *resolution function*, that is, a function that meets the following conditions:

- The function has a single parameter whose type is an unconstrained array.
- The type returned by the function is the same as the element type of its single parameter.

If a subtype indication includes the name of a resolution function, then any signal declared to be of that subtype will be considered a *resolved signal* (see Chapter 5). A resolved signal may have multiple drivers, and the named function will determine the value of the resolved signal.

```
type Base_logic is ('0', '1', 'Z');
type Base_logic_array is array
   (Integer range <>) of Base_logic ;
```

Scanned with CamScanner

```
-- A resolution function declaration:
function Resolve_drivers (
    P : Base_logic_array)
return Base_logic ;

-- A subtype indication containing the name
-- of a resolution function:
subtype Logic is Resolve_drivers Base_logic ;

-- S1 is not a resolved signal
signal S1 : Base_logic ;

-- S2 and S3 are resolved signals of the same base type
signal S2 : Logic ;
signal S3 : Resolve_drivers Base_logic ;
```

There are two predefined subtypes in VHDL: Positive and Natural. Their definition makes use of the VHDL attribute High.

```
subtype Positive is range 1 to Integer'High ;
subtype Natural is range 0 to Integer'High ;
```

**Attributes**

Certain classes of items in VHDL may have attributes. These classes are

1. types, subtypes
2. procedures, functions
3. signals, variables, constants
4. entities, architectures, configurations, packages
5. components
6. statement labels

An attribute is a named characteristic of items belonging to these classes A particular attribute for a particular item may have a value, and if it does have a value, the value may be referenced *by* using an attribute name. The general form of an attribute name is

*name ' attribute_identifier*

Attribute values are completely different from the values of object (signals, variables and constants); an object has only one value at any given time but may have many attributes. VHDL allows a designer to define his own attributes — the *user-defined* attributes — but there are also certain predefined attributes. User-defined attributes will be discussed in Chapter 9. This section discusses only predefined attributes related to types, subtypes,

Scanned with CamScanner

and array objects; other predefined attributes are discussed in Chapter 9 and in Appendix A.

VHDL defines the attributes Left, Right, High and Low for all scalar subtypes.

```
type Bit_position is range 15 down to 0 ;

type Fraction is range -0.999999 to 0.999999 ;

type Opcode is
   (Add, Add_with_carry, Sub, Sub_with_carry, Complement) ;

subtype Adding_opcode is Opcode range Add to Add_with_carry ;

-- Equalities illustrating predefined attributes:

  Bit_position'Left = 15
  Bit_position'Low = 0
  Fraction'Right = 0.999999
  Fraction'High = 0.999999
  Opcode'Left = Add
  Opcode'High = Complement
  Adding_opcode'Right = Add_with_carry
```

The attributes Pos, Val, Succ, Pred, Leftof, and Rightof are defined for any physical subtype or any discrete subtype (a discrete type is an integer type or an enumeration type). Roughly speaking, the attribute Pos maps values of any discrete type onto integers while the attribute Val maps integers onto values of a discrete type. It is important to note that for an enumeration type T, T'Pos(T'Left)) is defined to be 0, while for an integer type T, T'Pos(T'Left)) is equal to T'Left.

```
-- Equalities illustrating predefined attributes:
Opcode'Pos(Add) = 0
Opcode'Tos(Complement) = 4
Opcode'Val(2) = Sub
Opcode'Val(-l) is undefined
Opcode'Val(5) is undefined
Time'Pos(l fs) = 1
Time'Pos(l ps) = 1000
Bit_position'Pos(15) = 15
Bit_position'Val(15) = 15
```

For any physical or discrete type T,

```
T'Succ(X) = T'Val(T'Pos(X)+l)
```

Scanned with CamScanner

T'Pred(X) = T'Val(T'Pos(X)-1)

For a physical or discrete type T, with an *ascending* range,

T'Rightof(X) = T'Succ(X)
T'Leftof(X) = T'Pred(X)

For a physical or discrete type T, with an *descending* range,

T'Rightof(X) = T'Pred(X)
T'Leftof(X) = T'Succ(X)

The attributes Left, Right, High, Low, Length, Range and Reverse_range, are defined on constrained array subtypes and array objects. If A is a constrained array subtype or array object, then the value of A'Left(N) is the same as the value of T'Left where T is the subtype of the Nth index of A. Right, High, and Low are defined analogously. A'Length(N) is equal to the number of elements in the Nth dimension of A. Range and Reverse_range yield ranges, not values. An attribute name of the form A'Range(N) or A'Reverse_range(N) is a *range specification,* which can be used in a range constraint and as a discrete range:

```
type Window is array (1 to 12, 1 to 40) of Pixel ;
-- used as a discrete range:
type Row_info is array (Window'Range(1)) of Boolean ;
-- used as a discrete range:
type Column_info is array (Window'Range(2)) of Boolean ;
-- used in a range constraint:
subtype Column_number is Integer range Window'Range(2) ;
```

Cannot be changed

**Predefined Operators**

VHDL provides a number of predefined operations for the construction of expressions that compute values. The predefined operators fall into four groups: arithmetic, relational, logical, and concatenation.

| Group | Symbol | Function |
|---|---|---|
| arithmetic (binary) | + | addition |
| | - | subtraction multiplication |
| | * | division |
| | / | modulus |
| | mod | remainder exponentiation |
| | rem | |
| | ** | |

Scanned with CamScanner

| | | |
|---|---|---|
| arithmetic (unary) | +<br>-<br>abs | unary plus<br>unary minus absolute value |
| relational | =<br>/=<br><<br>><br><=<br>>= | equal<br>not equal<br>less than<br>greater than<br>less than or equal greater than or equal |
| logical (binary) | and<br>or<br>nand<br>nor<br>xor | logical and<br>logical or complement of and<br>complement of or<br>logical exclusive-or |
| logical (unary) | not | Complement |
| Concatenation | & | Concatenation |

The operands of the addition and subtraction operators must be of the same type and may be any numeric type (integer, floating point, or Physical). The operand of the unary arithmetic operators may also be any numeric type. The operands of the modulus and remainder operators must be of the same type and may be any integer type. The operands of the multiplication and division operators may be both the same integer type or may be both the same floating point type. There are also three special cases involving multiplication or division of physical types:

- Any physical type may be multiplied by an integer literal or floating point literal.
- Any physical type may be divided by an integer literal or floating point literal.
- Any physical type may be divided by the same physical type. f

The left operand of the exponentiation operator can be any integer type or floating point type; the right operand must be an integer literal (a negated integer literal is allowed as the right operand only if the left operand is of a floating point type). If one (or both) of the operands of an arithmetic operator is not a literal, then the resulting expression has the type of that operand, with one exception: the division of one physical type by the same physical type results in an expression of an integer type.

The result of an expression formed with a relational operator is always of type Boolean. The operands of a relational operator must be of the same type. The operand type of the relational operators "=" and "/=" may be any type (except a file type). The operand type of the remaining relational operators may be any scalar type or any one-dimensional array type whose element type is a discrete type (enumeration type or integer type). The following example shows the use of the "<" with one-dimensional array types.

```
type A is array (Integer range <>) of Integer ;
constant Cl  : A := (2 to 1 => 1) ; -- null array
constant C2 : A := (1 to 2 => 2) ; -- (2, 2)
```

```
constant C3 : A := (1 => 1, 2 => 2) ;  -- (1, 2)
constant C4 : A := (1 => 1);   -- array of length 1

-- Given the above declarations, the following relations are true:

  C1 < C4
  C4 < C3
  C3 < C21
```

The operands of the binary logical operators must be of the same type. The logical operators are defined for the predefined types Bit and Boolean and for one-dimensional arrays of Bit and Boolean. The resulting expression has the same type as the type of the operand. All of I the binary logical operators except **xor** are "short-circuit" operators; that is, the right operand is evaluated only if the value of the left operand does not in itself determine the value of the binary expression. Thus, the expression

```
(X = X) or (3/0 = 0)
```

could not generate an error condition since the right operand of the or will never be evaluated whereas the expression

```
(3/0 = 0) or (X = X)
```

will always generate an error condition.

The concatenation operator is a binary operator. Either operand of the concatenation operator may be either a one-dimensional array type or the element type of a one-dimensional array type; the operands must be of the same type or one operand must be the element type of the other operand. The declarations of constants S3, S4, S5, A3, A4, and A5 below illustrate the use of the concatenation operator.

```
constant S1 :  String := "ABC" ;
constant S2      : String := "DEF" ;
constant S3 : String := S1 & S2 ;  -- "ABCDEF"
constant C1 : Character := 'X' ;
constant C2 : Character := 'Y' ;
constant S4 : String := S3 & C1 ;   -- "ABCDEFX"
constant S5 : String := C1 & C2 ;  -- "XY"

type Arr is array (Integer range <>) of Boolean ;
constant A1 : Arr := (1 => True, 2 => False) ;
constant A2 : Arr = (1 => False, 2 => True) ;
constant A3 : Arr = A1 & A2; -- (True, False, False, True)
constant B1 : Boolean := False ;
constant B2 : Boolean := True ;
```

Scanned with CamScanner

**constant A4 : Arr := A3 & B1; -- (True, False, False, True, False)**
**constant A5 : Arr := B1 & B2; -- (False, True)**

# CHAPTER 5
# BEHAVIORAL DESCRIPTION

This chapter explores those aspects of VHDL which relate to the behavioral description of a discrete system. The chapter is followed by a chapter on the structural description of digital devices and a chapter on system design where the description of large-scale designs is explored.

This chapter is divided into three main sections. The first section looks at the process statement, which is the primary behavioral construct in VHDL. The second section looks at the sequential nature of behavioral modeling in VHDL. This section describes the executable statements which can occur within a process statement. The final section deals with the concurrent aspects of behavioral modeling in VHDL and explores how the local sequential nature is integrated into the overall concurrency of the execution model.

## Process Statements

In VHDL there are two levels at which the designer must define behavior of a discrete system: the sequential level and the concurrent The sequential level involves programming the behavior of each process which will be used in the model. The concurrent level involves defining the ~ relationship between these processes, with particular attention to the communication between them. The *process statement* (or process) is the construct which bridges these two levels.

The process statement is a concurrent statement which defines specific behavior to be executed when that process becomes active. It has the basic form:

```
process label :
    process
        declarations
    begin
        statements
    end process;
```

basic form of process statement.

The *process_label* defines a named label for the process; this label is optional. The *declarations* section defines the local data environment needed by the process. The *statements* section of the process is the sequential program which defines the behavior of the process.

Each process statement defines a specific action, or behavior, to be performed when the value of one of its sensitivity signals changes. This action is defined by the sequentially ordered execution statements in the process. The behavior of the process is a function of the signals read b the process. The results of the behavior are applied to the output signals and may be read by other processes. This behavior is performed each time new information is received on a sensitivity signal of the process. When the final statement of the process has been executed, execution returns to the first statement of the process and continues. Execution may be suspended using a wait statement (discussed in

Scanned with CamScanner

the next section A process which is being executed are said to be *active* otherwise the process is said to be *suspended*. Every process in the model may be active at any given point in time and all active processes are execute concurrently with respect to simulation time.

Process statements which do not assign values to signals are said to be *passive*. A passive process has no outputs (because it does not assign to any signals) and, therefore, cannot cause the activation of another process. Passive processes can be placed within the entity declaration of a design entity.

## The Wait Statement: Activation and Suspension

As discussed above, a process is said to be in one of two states any given time during simulation: active or suspended. The change between these states is controlled by a single construct called the wait statement. When a wait statement is executed inside a process suspends and the conditions for its reactivation are set There are three different kinds of conditions: timeout, condition, and signal sensitivity. These kinds of conditions can be mixed together in the wait statement, or defaults may be used. The following forms show the three o: the wait statement plus the default form.

```
wait;
wait on signal-list;
wait until condition;
wait for time-expression;
```

The first form is a wait statement which causes a process to suspend indefinitely. After the execution of this form of the wait statement, the process will not be activated again during the simulation. The second form provides a list of signals which will become the sensitivity channels of the process. Whenever an event occurs on a signal to which the process is sensitive, the process is reactivated. The third form specifies a condition which must be true before the process can be resumed. The final form supplies a maximum delay for the process to be suspended; when the time delay expires, the process is reactivated. These conditions may be mixed together to form more complex conditions. For example, consider the following behavioral description of an AND gate controlled by an enable line.

```
And_Process:
  process
  begin
      wait on A, B until Enable = '1';
      T <= transport A and B;
  end process;
```

The statement after the wait statement is a signal assignment statement which is discussed later on in this chapter. The process will reactivate whenever there is an event

Scanned with CamScanner

on A, B or Enable but only if Enable is equal to '1', otherwise the process remains suspended.

If a process is always sensitive to one set of signals, it is possible to designate the sensitivity signals of a process using a *sensitivity list*. If this is done, it is illegal to include any wait statements in the process (or in subprograms called by the process). For example, the following process could be used to implement a simple OR gate:

```
Or_Process:
  process(In1, In2)
  begin
      Output <= In1 or In2;
  end process;
```

The parentheses after process indicate that the process will be sensitive to the signals In1 and In2. This process is equivalent to the following process:

```
Or_Process:
  Process
  begin
      Output <= In1 or In2;
      wait on In1, In2;
  end process;
```

## Behavioral Modeling - Sequential View

### Declarations

The declarations section is used to define the local data environment of a process statement. The language allows the following declarations in this context:

1. subprogram body
2. variable declaration
3. attribute declaration
4. attribute specification

*or*

5. any of the basic declarations described in Chapter 3 ?

Most of the constructs are discussed elsewhere in the book, so section will focus on those which are peculiar to subprograms, namely, variable-declarations.

Variables are data objects which hold values specific process. The declaration of a variable has the form:

**variable** *identifier_list: subtype_indication*
  := *initial_value;*

} variable syntax.

A variable is declared in the declarative region of a process or subprogram. For example, the following process has a variable declaration for the variable count. The process also assigns to the variable. Variable assignments are discussed later on in the chapter.

```
P1 :
   process
        variable count : Integer := 0;
   begin
        count := count + 1;
        wait;
   end process;
```

NOTE: In this process, and throughout the text, a wait statement with no sensitivity clause, condition clause, or timeout clause is used. This will cause the process to suspend for the duration of the simulation. This type of wait statement is not generally used in practice, but is useful in examples which are used to illustrate specific points.

**Sequential Assignment**

The execution of a process statement may involve assigning values to variables and/or signals.

Many programming languages include global variables, meaning variables which are visible to any number of modules used in the program. VHDL does not allow global variables and it is important to understand the reason for this. As stated above, VHDL is primarily a controlled parallel language: controlled because the asynchronicity of the model by the timing model and parallel because each process in the model can be executing at the same time. Therefore, there is no guarantee that a given process will execute before any other process during a single simulation cycle. If one process assigned to a global variable and another process read the variable, the behavior of the system would be indeterminate. For example consider the following which assumes a global variable V:

```
-- THIS VHDL IS NOT LEGAL

variable V : Integer := 0;

Process_One:
   process
   begin
        V := V + 1;
        Wait;
   end process;

Process_Two:
```

```
process
        variable X : Integer;
begin
        X :=V;
        wait;
end process;
```

If Process_One executes before Process_Two, then X would get the value V + 1, or 1. However, if Process_Two executes before Process_One, X would get the initial value of V, or 0. There is therefore a conflict between updating a global variable and reading a global variable. Such a conflict should not be built into hardware, thus global variables have been excluded from VHDL.

**Signal Assignment**

The signal assignment statement is, perhaps, the most important behavioral statement in the language. Signals are used to transmit information between processes. The mechanisms of this transmission are discussed later on in the chapter. This section examines the form of the signal assignment statement *within* a process and later sections will examine the role of signals *outside* the process.

A signal is comprised of a current value and a projected output waveform. The current value element always holds the value of the signal which is read by any given process. If a signal is not a resolved signal, then there is a single process statement which assigns values to the signal. This process is called the *source* for the signal. The source of a signal may also be a connection to an out, inout, or buffer port in a child component. Component connections are discussed in the next chapter. The source of the signal may assign a number of values to the signal at a given point in time and designate a delay for each of these values. The current value of the signal-is-assigned a new value when the delay for the new value has expired. The collection of values and their delays is called the projected output waveform. It is possible for an element of the waveform to change or be removed before the expiration of the element's delay, causing the value to never be applied to the current value of the signal, hence *projected*. The specifics of the nature of the projected output waveform are discussed in the section off this chapter which deals with the concurrent nature of VHDL behavior. The goal of this section is to examine how a value and delay are supplied by the source process.

The simplest form of a signal assignment statement is:

*signal-name <= value;*

This statement assigns *value* to the current value of the signal at the beginning of the next simulation cycle. The left hand side of the statement is referred to as the *target* of the assignment. When no explicit delay is given in the assignment statement a *delta* delay is implied. The *value* element of the assignment can be any expression which yields the scheduled new value of the signal; e.g. a function call, an arithmetic expression, or a direct assignment of a value.

Scanned with CamScanner

It is possible to associate the assigned value with a delay. The delay is given relative to the current time of the simulation. For example, if the current simulation time is 10 ns, then an assignment giving a delay of 2 ns will occur at 12 ns in the simulation. The signal assignment with a delay has the form:

*signal-name* <= *value* after *time-expression;*

The *time-expression* can be any expression which yields a value of the predefined type TIME.

VHDL includes two delay types for controlling the effect of the assignment on the projected output waveform of the signal: inertial and transport. Inertial delay is used to represent components which require the value on inputs to persist for a given time before the component responds. Flip-flops, where spikes on the inputs might be ignored, are a good example of this. Transport delay is similar to wire delay; *i.e.* the output always occurs no matter what the time duration of its signal. In VHDL, the delay type is always indicated in the assignment to the signal. The default delay type in VHDL is inertial. The user may indicate that the assignment should use transport delay in the following way:

*signal-name* <= transport *value* after *time-expression;*

The details of the effects of transport and inertial delays are explored later in the chapter.

When a number of values need to be assigned to a signal during one activation of its driving process, the following form of the signal assignment may be used:

*signal-name* <=
  *value* after *time-expression,*
  *value* after *time-expression,*
    (...)
  *value* after *time-expression;*

For example, the following process might be used to implement a simple clock.

```
Clock_Process:
  process
  begin
    clock <=
        '1' after 1 ms,
        '0' after 2 ms,
        '1' after 3 ms,
        '0' after 4 ms,
        '1' after 5 ms;
    wait;
  end process;
```

Scanned with CamScanner

One of the most common errors made when writing VHDL processes is to forget the implications of the inherent delay in a signal assignment which does not specify a time clause. It should be remembered that signals are never assigned to directly. There is always at least a delta delay between the execution of an assignment and the updating of the current value of the signal. As an example, consider the following process:

```
signal A : Bit := '0';
signal B : Bit := '1';

PI :
  process
  begin
      B <= A;
      assert (B = A) report "B does not equal A" severity error; wait on B;
  end process;
```

The signal assignment statement will put the current value of A into the projected output waveform of B after a delta delay (one simulation cycle.) At first start of the next simulation cycle *(i.e.* after executing the wait statement) the value will be moved from the projected output waveform of B into the current value of B. An assertion statement provides a means for printing a message if a condition is false; the syntax and semantics of the assertion statement are discussed later in the chapter. Therefore, the condition of the assertion statement will be FALSE because when it is executed the value of A has not been moved into the current value of B. The assertion statement in the following process will, however, evaluate to TRUE and no assertion message will he reported.

```
signal A : Bit := '0';
signal B : Bit := '1';

PI :
  process
  begin
      B <= A;
      wait on B;
      assert (B = A) report "B does not equal A" severity error;
  end process;
```

## Signal Drivers

The values which are traveling over a data pathway at a given time are contained in the *driver* of the signal defining that data pathway. A driver is a collection of value/time pairs which are referred to as *transactions*. A transaction can be represented by a tuple of the form (value, time); e.g. if the value 1 is generated on the data pathway

Scanned with CamScanner

and the pathway will be updated after another 2 ns, the transaction would be designated as (1, 2 ns). Note that the time element is relative to the current simulation time so that, in this example, after one more ns this transaction would be (1, 1 ns).

A driver for a signal is defined by the process which assigns values to the signal. The driver is a special case of a *source* for the signal; the source is a process which assigns to the signal or any connection of the signal to an out, inout, or buffer port. Every process which assigns to signal creates a source for that signal. A signal may have only one source unless the signal is a resolved signal, meaning a resolution function is associated with the signal to *resolve* the sources into a single transaction (resolved signals are discussed later on in this chapter).

For example, the following architecture contains a single source for the signal S:

```
architecture A of E is
   signal S : integer;
begin
   process
   begin
         S <= transport 1 after 1 ns;
         S <= transport 2 after 2 ns;
         wait;
   end process;
end A;
```

Even though the process has two assignments to the signal S, there is only one process which is assigning to S. On the other hand, the following architecture contains two sources for S and, since S is not a resolved signal, is illegal.

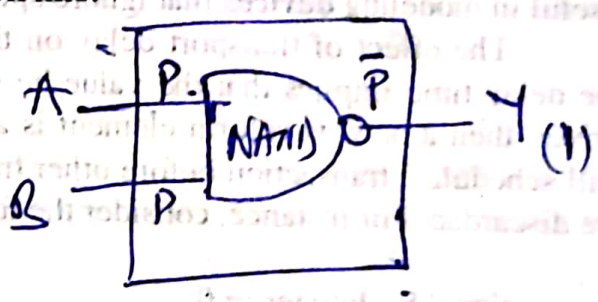
```
-- THIS ARCHITECTURE CONTAINS AN ERROR
architecture A of E is
   signal S : Integer;
begin
   process
   begin
         S <= transport 1 after 1 ns;
         wait;
   end process;
   process
   begin
         S <= transport 2 after 2 ns;
         wait;
   end process;
end A;
```

Scanned with CamScanner

The driver of a signal gets its initial value from the default value on the declaration which is visible to the source process. If the signal declaration and a source process for that signal are in the same architecture (or the signal is a global signal), the initial value of the driver is taken from the default value on the signal declaration. If the source process is in another component, the initial value of the driver is taken from the port declaration through which the signal is passed. This can lead to confusion in initializing models, particularly when a process statement is extracted from an architecture and placed in a child component.

For example consider an architecture which performs the NAND function on two bit values. The entity and architecture might look like:

```
entity NAND_GATE is
  port (
        A, B : in Bit;
        Y : out Bit);
  end NAND_GATE;

  architecture Behavior of NAND_GATE is
        signal Pass_Bit: Bit := '1';
  begin
        Pass_Bit <= transport A and B;
        Y <= transport not Pass_Bit;
  end Behavior;
```



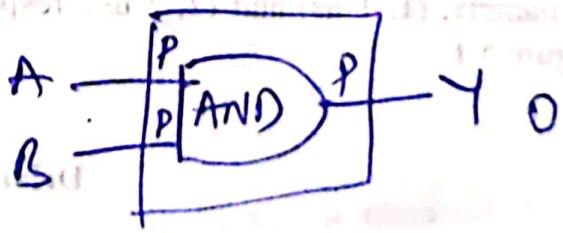

P = Pass_Bit

P̄ = not Pass_Bit.

At the beginning of simulation the initial value of Pass_Bit will be '1', as expected.

Suppose the architecture was to be rewritten to instantiate an AND gate in the following way:

```
entity AND_GATE is
  port (
        A, B : in Bit;
        Y : out Bit := '0');
        end AND_GATE;

architecture Structure of NAND_GATE is
  signal Pass_Bit: Bit := '1';
  component AND_GATE
        port (A, B : in Bit; Y : out Bit);
begin
  And_Phase : AND_Gate
        port map (A, B, Pass_Bit);
  Y <= transport not Pass_Bit;
end Structure;
```




Y ⇐ transport Pass

In this case, the initial value of Pass_Bit is taken from the default value on the out port of the AND_GATE; namely '0'. For a complete discussion of default values on ports see Chapter 6.

## Delay in Signal Assignments

There are two types of delay that can be applied when assigning a time/value pair into the driver of a signal: transport and inertial. Transport delay is analogous to the delay incurred by passing a current through a wire, Inertial delay is used for devices that do not respond unless a value on its input persists for the given amount of time. Inertial delay is useful in modeling devices that ignore spikes on their inputs.

The effect of transport delay on the driver of a signal is faily straightforward. If the delay time implies that the value be scheduled after all the other transactions in the driver, then a new waveform element is added to the end of the driver. If the delay time will schedule a transaction before other transactions in the driver, those later transactions are discarded. For instance, consider the following process:

```
signal S : Integer := 0;

PI :
  process
  begin
      S <= transport 1 after 1 ns;
      S <= transport 2 after 2 ns;
      wait;
  end process;
```

If no waveform elements are on the driver of signal S before execution of this process, the effect of the execution will be to place two time/value pairs in the driver for S; namely, (1, 1 ns) and (2, 2 us), respectively. The effects on the driver are shown in Figure 5.1.

**Driver of signal S:**

S —— [ (1, 1 ns) ]

S <=transport 1 after 1 ns;

S —— [ (1, 1 ns) | (2, 2 ns) ]

S <=transport 2 after 2 ns;

Figure 5.1

If we reverse the statements, however, we get the following process:

```
signal S : Integer := 0;

PI :
   process
   begin
        S <= transport 2 after 2 ns;
        S <= transport 1 after 1 ns;
        wait;
   end process;
```

1st assignment
2nd assignment
Big
Small
Small displacement
Big

This process has a different effect. The first assignment will place a transaction (2, 2 ns) on driver for S. When the second assignment is executed, this transaction will be overwritten by the transaction (1, 1 ns) because the new transaction is scheduled to occur before the transaction (2, 2 ns). The transaction (1, 1 ns) *overrides* the transaction (2, 2 ns) and, thereby, makes the second transaction obsolete. The effects on the driver are shown in Figure 5.2.

Driver of signal S:

S — (2, 2 ns)

displace.

S <=transport 2 after 2 ns;

S — (1,. 1 ns)

S <=transport 1 after 1 ns;

Figure 5.2.

An extension of this example will show the full effects of transport delay. Consider the following process:

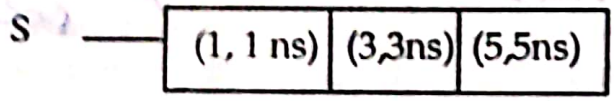
```
signal S Integer := 0;

PI :
   process
   begin
        S <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
        S <= transport 4 after 4 ns;
```

Scanned with CamScanner

```
        wait;
    end process;
```

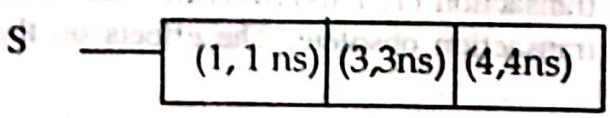
The first assignment in this process adds three transactions to the driver of the signal S : (1, 1 ns), (3, 3 ns), and (5, 5 ns). The second assignment overwrites the last transaction of the previous assignment because the transaction (4, 4 ns) is scheduled before the transaction (5,5 ns). After the execution of this process the driver of signal S will contain the transactions (1, 1 ns), (3, 3 ns) and (4, 4 ns). The effects on the driver are shown in Figure 5.3.

Driver of signal S;




S <=transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;

Driver of signal S:




S <=transport 4 after 4 ns;

Figure 5.3.

INERTIAL DELAY.

Inertial delay is a little trickier because its effect on the driver of a signal is harder to understand at first glance. All transactions which are scheduled to occur after the delay given in the inertial assignment are discarded, as in the transport case. The difference is the effect on the transactions which are scheduled to occur before the new transaction. If the value of a scheduled transaction is different from the value of the new transaction, the old .transaction Ts discarded. If the value is the same as the new transaction, the old transaction is left on the projected output waveform.

Consider the following process:

```
signal S : Integer := 0;

P1:
  process
  begin
      S <= 1 after 1 ns;
      S <= 2 after 2 ns;
      wait;
  end process;
```

Scanned with CamScanner

NOTE: Notice that the reserved word transport is not included on the signal assignments. There is no reserved word inertial because inertial delay is the default delay type for signal assignments.

The first assignment will place the transaction (1, 1 ns) in the signal*s driver. The second assignment overrides the first assignment because a new value is introduced to the driver with inertial delay. One might think of the second assignment as saying S must have the value 2 for at least 2 ns. If the first assignment were not removed, the value of S would get the value 1 after another 1 ns and then the value 2 after another 1 ns. The effects on the driver are shown in Figure 5.4.

Driver of signal S:

Big displaces Small

S — (1,1 ns)

S <=1 after 1 ns;

S — (2,2 ns)

S <=2 after 2 ns;

Figure 5.4.

Reversing the assignments has the same effect as the transport case.

```
signal S : Integer := 0;

P1:
  process
  begin
      S <= 2 after 2 ns;
      S <= 1 after 1 ns;
      wait;
  end process;
```

The second assignment overrides the first because it schedules a transaction to occur before the transaction (2, 2 ns). The effects on the driver are shown in Figure 5.5.

Driver of signal S:

S — (2,2 ns)

S <=2 after 2ns;

S — (1,1 ns)

Figure 5.5

Scanned with CamScanner

S —— (1, 1 ns)

S <=1 after 1 ns;

Figure 5.5.

A final example should show the full effect of inertial delay.

```
signal S : Integer := 0;

PI:
  process
  begin
     S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
     S <= 3 after 4 ns, 4 after 5 ns;
     wait;
  end process;
```

The first assignment places the transactions (1, 1 ns), (3, 3 ns), and (5, 5 ns) on the driver for signal S. The second assignment removes the transaction (1, 1 ns) because the new transaction (3, 4 ns) has a different value component. The transaction (3, 3 ns) is retained on the projected output waveform because the value component is the same as the value component of the new transaction (3, 4 ns). The last transaction (5, 5 ns) is removed because the time component is greater than the time component of the new transaction (3, 4 ns). Finally, the second transaction from the second assignment is added to the end of the projected output waveform of signal S. The resulting projected output waveform consists of the transactions (3, 3 ns), (3, 4 ns), and (4, 5 ns). The effects on the driver are shown in Figure 5.6.

Driver of signal S:

S —— | (1, 1 ns) | (3,3ns) | (5,5ns) |

S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;

S —— | (3, 3 ns) | (3,4ns) | (4,5ns) |

S <= 3 after 4 ns, 4 after 5 ns;

Figure 5.6.

Scanned with CamScanner

This last example highlights an important point. The semantics of the inertial delay are such that if a number of elements are given on the waveform of an inertial assignment, as in

S <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;

the elements after the first element are not considered to be inertial assignments. This is because the language requires that the waveform elements in a signal assignment must be ascending, *i.e.* must have ascending time delays. Once the first transaction is on the driver, each subsequent transaction is guaranteed to be scheduled after all other transactions on the driver, which is the same behavior as in the transport case. That makes the above statement different from a process which has the following statements:

```
signal S : Integer := 0;

PI :
  process
  begin
        S <= 1 after 1 ns;
        S <= 3 after 3 ns;
        S <= 5 after 5 ns;
        wait;
  end process;
```

Value time
(a, 5ns)
driver
transaction.

After the execution of this process there will be a *single* transaction on the driver for S because the last assignment will override the first two assignments since its value is different. In order to get all three assignments on S, the following process must be used.

```
signal S : Integer := 0;

PI:
  process
  begin
        S <=1 after 1 ns;
        S <= transport 3 after 3 ns;
        S<= transport 5 after 5 ns;
        wait;
  end process;
```

**Variable Assignment**

Variable assignment replaces the value of a variable object with a new value obtained by evaluating the expression on the left hand side of the assignment. The syntax of a variable assignment takes the form:

*variable_name := expression;*

A variable is declared in a process or a subprogram. When the variable is declared in a process it retains it value throughout the simulation; *i.e.* it is never reinitialized. Variables declared in subprograms are reinitialized whenever the subprogram is called. This is because processes are always executing (either active or suspended) while subprograms are only executed when they are called. If there is a wait statement in a subprogram, then the variables which are active when the process which called the subprogram suspends, retain there values until the process is reactivated and the subprogram is exited.

The following process counts the events on the signal S.

```
signal S : Bit := '0';
PI :
  process
      variable Events_on_S : Integer := 0;
  begin
      wait on S;
      Events_on_S := Events_on_S + 1;
  end process;
```

In this example, the variable Events_on_S is declared in the local data environment of the process PI. The declaration defines the variable to be of type Integer and designates that its initial value be set to 0. Whenever an event occurs on S the process will be activated and the variable Events_on_S will be updated.

It should be noted that, contrary to a signal assignment, there is no delay associated with a variable assignment. The variable object will have the new value immediately following the execution of the assignment statement. This means a sequential statement immediately following the variable assignment which uses the variable will see the new value of the variable.

```
PI :
  process
      variable V : Integer := 0;
  begin
      V := 1;
      if V = 1 then
         Print("This will get printed.");
      end if;
      wait;
  end process;
```

Conditional control (decision making). If a given condition is made, Repeated / iterative control.

## Sequential Control

VHDL provides a number of constructs for controlling the flow M execution within a process statement. Some of these constructs provide conditional control within the process; *i.e.* statements are executed where a given condition is true or not. Others provide iterative control, statements are executed repeatedly until some condition is reached.

## Conditional Control

During the execution of a process, a designer may want to select alternative collections of statements depending on some condition of the model. VHDL provides two forms for this type of conditional control the *if statement* and the *case statement.*

The if statement is used to select one or none of a collection of statements to execute based on the truth of one or more conditions or expressions. Each condition or expression must be boolean; that is it must return either TRUE or FALSE. We can distinguish three types of conditional selection in an if statement.

The first, and most basic, form of the if statement is the selection of a collection of statements for execution based on a single condition For instance, the following process might be used to represent a D flip-flop which assigns D to Q after a delay of 5 ns but only if the Enable port is high.

If Enable port is a 1

```
DFF_Process :
  process
  begin
    if Enable = '1' then
      Q <= D after 5 ns;
    end if;
    wait on D;
  end process;
```

DFF Process

no Internal signal (nothing to declare) D = input

Condition

Statement

Q = output.

If Statement

Case Statement

General Statement of If Statement

Notice the form of the if statement:

```
if condition then
  statements
end if;
```

General form of If Statement

If 'Condition' then
Statement
end if;

The second form of the if statement is the selection of one of two collections of statements for execution based upon a single condition.

For instance, the following process might be used to represent a three state buffer. If the enable port is not enabled (equal to "0") then the buffer outputs a high impedance value ("Z"), otherwise the input is sent to the output.

```
Buffer_Process :
  process
  begin
```

Process description of a Balanced or a Buffer.

2 Collection of Statements
Next Forms & Conditions

```
    if Enable = '0' then
        Output <= 'Z' after Delay;
    else
        Output <= Input after Delay;
    end if;
    wait on Input;
end process;
```

This form of the if statement takes the shape:

```
if condition then
    statements
else
    statements
end if;
```

else joining statements.
General form of if statement with the else joining statements

The third, and final, form of the if statement is really a notational convenience. This form is used when there are a number of alternatives which should be executed based on a number of related or ordered conditions. The following process might be used to represent a 2-input logical AND gate.

```
And_Proeess:
  process
  begin
      if In1 = '0' or In2 = '0' then
          Out <= '0' after Delay;
      else
          if In1 = 'X' or In2 = 'X' then
              Out <='X' after Delay,
          else
              Out <= '1' after Delay;
          end if;
      end if;
      wait on In1, In2;
end process;
```

otherwise assign one to the output

This process is legal VHDL but the nested if structure is awkward and places unwarranted emphasis on the first condition. We can rewrite this process using the third form of the if statement.

```
And_Process:
  process
  begin
```

```
        if In1 = '0' or In2 = '0' then
            Out <= '0' after Delay;
        elsif In1 = 'X' or In2 = 'X' then
            Out <= 'X' after Delay;
        else
            Out <= '1' after Delay;
        end if;
        wait on In1, In2;
    end process;
```

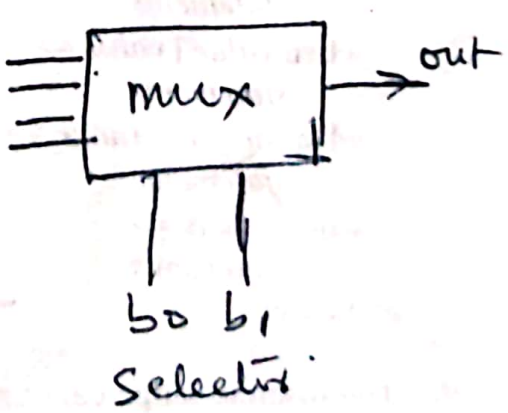
In this form the elsif branch of the if statement can be used any number of times, and with or without the final else branch. So assuming two uses of the elsif branch the third form of the if statement might look like:

```
if condition then
    statements
elsif condition then
    statements
elsif condition then
    statements
end if;
```

General form of if statement with elsif.

The *case statement* is another form of conditional control provided in VHDL. The case statement is used to select a collection of statements based on the range of values of a given expression. It is like the if statement in that a condition is used to choose among a number collections of statements. It is different, however, in that the statements are chosen based on the value of an expression. The designer gives the expression and identifies a collection of expressions for each possible value of the type of the expression. The expression must be of discrete type or a one-dimensional array.

```
Multiplexer_Process:
    process
    begin
        case Selector is
            when "00" =>          -- 1st arm
                Out <= In0 after Delay;
            when "01" =>          -- 2nd arm
                Out <= In1 after Delay;
            when "10" =>          -- 3rd arm
                Out <= In2 after Delay;
            when "11" =>          -- 4th arm
                Out <= In3 after Delay;
        end case;
        wait on Selector;
```



```
end process;
```

The when conditions are called *arms* of the case statement. There may be any number of these but no two arms can have the same value and every value in the range of the type must be represented in order to facilitate the inclusion of every value in the range and to take care of types whose ranges are not finite, VHDL includes two special types of arms. The following example makes use of both of these special types in selecting statements based on an integer expression.

```
Select_Process:
   process
   begin
        case X is
          when 1 =>          — 1st arm
               Out <= 0;
          when 2 | 3 =>      — 2nd arm
               Out <= 1;
          when others =>     — rest
               Out <=2;
        end case;
   end process;
```

The second aim of the case statement is chosen when X evaluates to 2 *or* 3. The third arm of the case statement is chosen when X evaluates to something other than the values given in the other arms. The when others aim must always be the last aim of a case statement if it is included.

The general form of the case statement is:

```
case expression is
   when value =>
        statements
   when value | value =>
        statements
   when discrete-range =>
        statements
   when others =>
        statements
end case;
```

General form of Case Statement.

The discrete range case causes the enclosed statements to execute if the value of the expression is within the range given. The choices must be unique; *i.e.* the value of the expression must evaluate to one and only one of the arms of the case statement. There can be any number of arms in the case statement and the order can be mixed in any way

except for an arm which uses the others keyword; if it is used, it must be the last arm of the case statement and only one such arm is allowed.

**Iterative Control**

Sometimes a collection of statements needs to be executed repeatedly for a specific number of times or until some condition occurs. This type of control is called iterative control because the execution iterates over the statements until some condition is met. VHDL provides iterative control in the form of loop statements or loops. There are three kinds of loop statements: the simple loop, the for loop and the while loop. The exit statement is a sequential statement which is closely associated with loops. The execution of an exit statement causes the loop to be exited.

A simple loop encloses a set of statements in a structure which is set up to loop forever. This structure has the form:

*Loop_label:* loop
  *statements*
end loop *loop_label;*

The *loop_label* is optional. In the syntax of the language a loop statement can be included in the collection of statements and so it is possible to nest loop statements within other loop statements.

```
PI :
  process
      variable A : Integer := 0;
      variable B : Integer,
  begin
      Loopl:
      loop
         A := A + 1;
         B := 20;
         Loop2 :
         loop
            B := B - A;
         end loop Loop2;
      end loop Loop1;
      wait;
  end process;
```

This example will cause an infinite loop when executed since there is no way to get out of either loop The simple form is useful only when used in conjunction with the exit statement; without it, the simulation of the model would never terminate. The exit statement has two general forms:

exit *loop_label;*
exit *loop_label* when *condition;*

The first form exits the enclosing loops with the given loop label. The second form does the same thing but only if the condition is true. In both of these forms the *loop_label* is optional; if it is not given the statement exits the immediately enclosing loop statement.

Going back to the simple *loop* example, we can change the process to exit when certain conditions are reached.

```
PI :
  process
      variable A : Integer := 0;
      variable B : Integer := 1;
  begin
      Loopl :
      loop
        A := A + 1;
        B:=20;
        Loop2 :
            loop
              if B < (A * A) then
                  exit Loop2;
              end if;
              B := B - A;
            end loop Loop2;
            exit Loop1 when A > 10;
        end loop Loopl;
      wait;
  end process;
```

This looping mechanism is simple but it can lead to problems where used and does not give any indication in the loop statement itself as to how the loop will be exited. The for loop is used to control the exit from the loop based on iterating through some number of values in a given range. Taking the simple loop example, we can rewrite the process to reflect the intentions of the loop a little more clearly.

```
PI :
  process
      variable B : Integer := 1;
  begin
      Loop1:
    for A in 1 to 10 loop
        B := 20;
```

```
        Loop2:
        loop
          if B < (A * A) then
              exit Loop2;
          end if;
          B := B - A;
        end loop Loop2;
      end loop Loop1;
      wait;
  end process;
```

This is a much clearer description of the execution of the outer loop because it explicitly states the number of iterations of the enclosed statements which will be executed. The for loop has the general form:

*Loop_label :*
for *loop_variable* in *range* loop
  *statements*
end loop *loop_label;*

The *loop_variable* does not need to be declared; the loop statement includes an implicit declaration for it. The range can be any discreet range (these are denned in Chapter 4).

The final type of loop statement is the while loop. This type of loop statement is used when there is a boolean condition which, when false, causes an exit from the loop. Again, looking at the example we can rewrite the process using a while loop for Loop.

```
PI :
  process
      variable B : Integer := 1;
  begin
      Loop1:
      for A in 1 to 10 loop
          B := 20;
          Loop2:
          while B >= (A * A) loop
              B := B - A;
          end loop Loop2;
      end loop Loop1;
      wait;
  end process;
```

This version of the process is clearly easier to understand and read. The while loop has the general form:

*loop_label* :
while *condition* loop
*statements*
end loop *loop_label*

general form while loop

One other construct which is used with loop statements is the next *statement*. This statement is used to advance control to the next iteration of a loop. The syntax for the next statement is:

next *loop-label* when *condition;*

The *loop-label* and when *condition* are both optional. When this statement is executed, control goes to the bottom of the loop identified by the label (or immediately enclosing loop if no label is given) and the loop is begun again. For a for loop, the loop index is incremented and for a while loop the condition is checked. If the *condition* is present, it is evaluated and if it is TRUE, the current loop iteration is exited and the next iteration is begun, otherwise the next statement is ignored. For instance, the P1 loop above could be rewritten as:

```
PI :
  process
      variable B : Integer := 1;
  begin
      Loop1:
      for A in 1 to 10 loop
        B := 20;
        Loop2:
        loop
            B := B - A;
            next Loop1 when B < (A * A);
        end loop Loop2;
      end loop Loop1;
      wait;
  end process;
```

**Other Sequential Statements**

**The Assertion Statement**

The assertion statement is an interesting feature which is not found in many programming languages, much less hardware description languages. It allows the designer to encode constraints on a model within the VHDL source. The designer supplies a boolean condition which must be met in a particular device during simulation. The condition is checked during the simulation and if it is not met a message is sent to the standard output device, usually the terminal. The designer supplies the condition of the

constraint, the message which will be reported if the condition becomes false, and the severity of the condition. The severity is of the type Severity_Level from package STANDARD which has the values Note, Warning, Error, and Failure. In some VHDLI systems, unmet conditions of severity Error or Failure cause the simulation to terminate. The assertion statement has the following syntax:

```
assert condition
  report message
      severity level;
```

When the *condition* is FALSE the message is sent to the system output with an indication of the severity of the message and the name of the design unit in which the assertion occurred. If no message is given, the default message "Assertion violation" is reported. The default severity level is error.

```
assert Enable /= 'X'
  report "Unknown value on Enable"
      severity Error,
```

This feature is very useful, especially in large-scale designs, because it allows the designer to a) directly encode specific constraints and error conditions directly into the description, b) provide useful messages which indicate the nature of the constraint or error condition, and in some systems, c) stop the simulation when significant constraints are not met or error conditions are detected which cannot be handled by the model. For example, consider the following model of an SR flip flop:

```
entity SRFF is
  port(
      S, R : in Bit;
      Q, QBar : out Bit);
  end SRFF;

architecture Constrained of SRFF is
begin
  process
      variable Internal1, Internal2 : Bit := '0';
      variable Last_State : Bit := '0';
  begin
      assert not (S = '1' and R = '1')
        report "Both S and R equal to '1'"
            severity Error,
      if S = '0' and R = '0' then
        Last_State := Last_State;
      elsif S = '0' and R = '1' then
```

Scanned with CamScanner

```
        Last_State:- '0';
      else -- S = '1' and R = '0'
        Last_State := '1*;
      end if;
      Q <= Last_State after 2 ns;
      wait on R, S;
    end process;
  end SRFF;
```

If both S and R are equal to '1', the assertion is false, the message "Both S and R equal to '1'" is reported, and the simulation may be terminated (because the severity is *Error*).

Other examples of using assertion statements can be found in Chapter 10, particularly in the sections titled "A Device Controller" and "Setup and Hold Timing".

## Procedure Calls

Procedure calls invoke procedures to be executed during a process. These procedures can be thought of a extensions to the behavior of to process statement They serve two main purposes. First, a procedure can be written to isolate complicated sections of a process statement. This provides a further functional breakdown of the behavior and makes the process statement easier to read. Procedures are also used to encapsulate behavioral code which can be used by different processes. The behaviour of the procedure is the same for all calls but the parameters may be different.

The procedure call statement has the form:

*procedure-name(association-list);*

## The Return Statement

The return statement is used to terminate the execution of a subprogram. For a procedure, the only legal form of the statement is

```
return;
```

When this statement is executed, control returns to the point at which the procedure was called.

For a function, the return statement is used to return a value to the data *object* assigned to the function output. For instance,

```
function AND_FUNCTION(X, Y: in BIT) return BIT is
begin
  if X= '1' and Y= '1' then
      return '1';
  else
      return '0';
  end if;
```

Scanned with CamScanner

```
end AND_FUNCTION;
```

This function returns the bit value '1' if both inputs are '1', otherwise it returns '0'.

## The Null Statement

It is sometimes desirable to explicitly define conditions under which no action is to be taken. The null statement is used for this purpose. For example,

```
procedure ModTwo(X : inout Integer) is
begin
   case X is
         when 0 I 1 => null;
         when others => X := X mod 2;
   end case;
end ModTwo;
```

## Behavioral Modeling - Concurrent View

The previous sections examined the local sequential nature of the process statement. The design of a discrete system is largely a matter of integrating these isolated processes into a complete system or subsystem. This section describes the concurrent statements in VHDL which can be used as shorthands for simple process statements; namely, the concurrent assertion statement and the concurrent signal assignment statement. These constructs are important in writing compact, easy to read VHDL Ascriptions. This is followed by a discussion of resolved signals, those signals that are assigned to by more than one process. The section, and the chapter, ends by examining a few simple examples which illustrate low processes communicate with each other.

## Concurrent Statements and Equivalent Processes

There is a class of concurrent statements in VHDL which are a "shorthand" for processes which exhibit the same behaviour. These statements serve to make the description easier to read and, in the case of signal assignments, to allow RTL-like expressions inside of design entities. This class of concurrent statements includes the concurrent assertion statement and concurrent signal assignment statements.

## Concurrent Signal Assignment

A concurrent signal assignment statement represents an equivalent process statement which assigns to (drives) the signals given as the target of the assignment. The concurrent signal assignment is a compact way of describing a process which chooses a waveform based on a set of conditions or a control expression. The syntax of the concurrent signal assignment is similar to that of the sequential signal assignment but it is not enclosed within a process statement. The concurrent signal assignment is sensitive to the signals in its transform; i.e. the *longest static prefix* of every signal in the transform.

Scanned with CamScanner

The longest static prefix is a static signal or the static prefix of a composite signal. For instance, an integer signal S is itself the longest static prefix, whereas the longest static prefix of an array signal element A(I) is the entire composite signal A. A collection of concurrent signal assignment s looks very much like an RTL description of a component. The concurrent signal assignment statement has the general form:

*target <= transform;*

For example, consider an abstract description of a subtraction unit as described by the following design unit:

```
entity Subtractor is
   port (
         In1, In2 : in Integer,
         Out1 : out Integer);
   end Subtractor;

architecture Almost_Simplest of Subtractor is
begin
   process
   begin
         Out1 <= In2 – In1 after 8 ns;
         wait on In1, In2;
   end process;
end Almost_Simplest;
```

Using a concurrent signal assignment, the above architecture could be simplified to read:

```
architecture Simplest of Subtractor is
begin
   Out1 <= In2 – In1 after 8 ns;
end Simplest;
```

The assignment is automatically executed whenever there is an event on In1 or In2.

It should be noted that, as in all concurrent statements, the textual order of two concurrent assignments does not indicate the order in which those assignments execute. For instance, the following two architectures an: functionally equivalent.

```
entity DFF is
   port (
         D : in Bit;
         Q : out Bit);
   constant Delay : Time := 5 ns;
end DFF;
```

Scanned with CamScanner

```
architecture Beh1 of DFF is
begin
   Q <= D after Delay;
   QBar <= not D after Delay;
end Beh1;

architecture Beh2 of DFF is
begin
   QBar <= not D after Delay;
   Q <= D after Delay;
end Beh1;
```

A special form of the concurrent signal assignment is the conditional signal assignment statement. In this statement, the waveform of the assignment is chosen based on a set of boolean conditions given in the assignment statement. The conditions are checked until a condition is found to be true and then the waveform associated with that condition is assigned to the target. The conditional signal assignment statement has the following form:

*target* <= *options*
   *waveform1* **when** *condition1* **else**
      .
      .
      .
   *waveformN-1* **when** *conditionN-1* **else**
   waveformN;

The *target* is the signal(s) which is(are) being assigned to. The *options* can be either guarded, which is discussed in Chapter 9, or transport which is equivalent to the transport delay discussed earlier in the chapter. For the remainder of this section, it is assumed that transport is the only option used. 1

This behavior is similar to that of an if statement and, in fact, the equivalent process for a conditional signal assignment statement includes an if statement.

```
process (signals-in-transform)
begin
   if condition1 then
         target <= options waveform1
      .
      .
      .
   elsif conditionN-1 then
         target <= options waveformN-1;
```

Scanned with CamScanner

```
    else
        target <= options waveformN;
    end if;
end process;
```

For example, consider the behavior of an AND gate which uses the logic states '0' and '1'. This behavior could be written with conditional signal assignment as in the following architecture.

```
architecture Conditional of AND_Gate is
begin
  Y <= transport
        '1' after Delay when A = '1' and B = '1' else
        '0' after Delay;
  end Conditional;
```

This architecture is equivalent to the following architecture:

```
architecture ConditionalEquivalent of AND_Gate is
begin
  process (A, B)
  begin
      if A = '1' and B = '1' then
        Y <= transport T after Delay;
      else
        Y <= transport '0' after Delay;
      end if;
  end process;
end (additionalEquivalent;
```

The other kind of concurrent signal assignment is the selected signal assignment statement. Whereas the conditional signal assignment operates much like an if statement, the selected signal assignment is similar to a case statement. An expression is given and each waveform is associated with a possible value of that expression.

```
with expression select
target <= options
  waveform1 when choices1,
      .
      .
      .
  waveformN when choicesN;
```

The *expression* can be any expression which would be legal in a case statement and the other elements are as in the conditional assignment.

As mentioned, the equivalent process contains a case statement to represent the behavior of the assignment.

```
process (signals-in-transform)
begin
  case expression is
      when choices1  =>
      target <= options waveform1;
        .
        .
        .
  when choicesN =>
      target <= options waveformN
  end case;
end process;
```

For example, the following VHDL code defines a decoder circuit described using a selected signal assignment statement

```
entity Decoder is
  port (
      Enable : in Bit;
      Sel : Bit_Vector(2 downto 0);
      DOut: out Bit_Vector(7 downto 0));
  constant Delay : Time := 5 ns;
end Decoder;

architecture Selected of Decoder is
begin
  with Sel select
      DOut <=
        "00000001" after Delay when "000",
        "00000010" after Delay when "001",
        "00000100" after Delay when "010",
        "00001000" after Delay when "011,
        "00010000" after Delay when "100",
        "00100000" after Delay when "101",
        "01000000" after Delay when "110",
        "10000000" after Delay when "111",

end Selected;
```

Scanned with CamScanner

This architecture is equivalent to the following architecture:

```
architecture SelectedEquivalent of Decoder is
begin
    process(Sel)
    begin
        case Sel is
            when "000" =>
                DOut <= "00000001" after Delay;
            when "001" =>
                DOut <= "00000010" after Delay;
            when "010" =>
                DOut <= "00000100" after Delay;
            when "011" =>
                DOut <= "00001000" after Delay;
            when "100" =>
                DOut <= "00010000" after Delay;
            when "101" =>
                DOut <= "00100000" after Delay;
            when "110" =>
                DOut <= "01000000" after Delay;
            when "111" =>
                DOut <= "10000000" after Delay;
            end case;
    end process;
end SelectedEquivalent;
```

As can be seen from these examples, the concurrent signal assignment makes the behavior of certain components easier to read and more compact.

## Concurrent Assertion Statement

The concurrent assertion statement represents a process statement containing an assertion statement with the same expression, report clause and severity clause given in the concurrent statement. The resulting process statement will be passive because it does not contain a signal assignment statement. It is possible, however, to include signals in the expression clause.

The concurrent assertion statement has the same syntax as the sequential assertion statement except that it can have a label and it is not enclosed inside a process. The concurrent assertion statement can be represented by an equivalent process which is sensitive to any signals which are present in the condition of the assertion. The equivalent process for a concurrent assertion statement is a passive -process because toe assertion does not assign to any signals. Therefore, concurrent assertions may be used in entity declarations.

Scanned with CamScanner

As an example, consider the SR flip flop shown above in the discussion of sequential assertion statements. The design unit could be written to include a concurrent assertion statement in the entity of the SR flip flop. This would make the constraint on the-device evident by just looking at the interface. The new design unit is given below.

```
entity SRFF is
  port (
        S. R : in Bit;
        Q, QBar : out Bit);

        SRFF_Constraint_Check:
        assert not (S – '1' and R = '1')
           report "Both S and R equal to '1'"
                 severity Error;

  end SRFF;

architecture Constrained of SRFF is
begin
  process
        variable Internal 1, Internal2 : Bit := '0';
        variable Last_State : Bit := '0';
  begin
        if S = '0' and R = '0' then
           Last_State := Last_State;
        elsif S = '0' and R = '1' then
           Last_State := '0';
        else -- S = '1' and R= '0'
           Last_State := '1';
        end if;
        Q <= Last_State after 2 ns;
        wait on R, S;
  end process;
end SRFF;
```

The above entity is equivalent to the following entity declaration:

```
entity SRFF is
  port (
        S, R : in Bit;
        Q, QBar : out Bit);

        SRFF_Constraint_Check:
        process (S, R)
```

Scanned with CamScanner

```
        begin
            assert not (S = '1' and R = '1')
                report "Both S and R equal to
                    severity Error,
        end process;
    end SRFF;
```

**Resolved Signals**

As mentioned above, a signal normally has a single source which is *driving* the signal. VHDL allows the designer to drive a signal with more than one source, if a *resolution function* is supplied to resolve the multiple sources into a single value for the signal. A good example of when this should be used is a Wired-OR or Wired-AND, where the value to be passed to a component is the result of tying together the output of a number of gates.

A signal which has more than one source is called a *resolved* signal. A resolved signal must have a resolution function associated with it; *i.e.* it is an error if there are multiple sources for a signal which is not a resolved signal. If the subtype indication of the signal declaration includes a resolution function or the declaration of the subtype for the signal contains a resolution function, then the signal is a resolved signal.

A resolution function is a function which takes a one-dimensional, unconstrained array of values of the resolved type and returns a value of the resolved type. As an example consider the following type declarations.

```
type Bit4 is ('X', '0', '1', 'Z');
type Bit4_Vector is array (Integer range <>) of Bit4;
```

We can declare a function which resolves signals of this type. The declaration of such a function might look like:

```
function Wired_Or (Input: Bit4_Vector) return Bit4;
```

To declare a resolved signal using the above resolution function, choices are available.

```
subtype Resolved_Wire is Wired_Or Bit4;
signal Resolved_Signal : Resolved_Wire;

signal Resolved_Signal: Wired_Or Bit4;
```

In the first case, the subtype is made a resolved subtype and the signal is declared to be of that subtype. This case should be used when the same resolution function will be used for a number of signals. In the second case, the resolution function is included directly in the declaration of the signal.

Scanned with CamScanner

The resolution function is invoked every time the value of the resolved signal is updated. If two sources are driving the signal, then the resolution function is invoked with an array of length two which contains the values of the two sources. For instance:

```
Source_1:
  process
      begin
        Resolved_Signal <= '1';
          wait;
      end process;

Source 2:
  process
      begin
          Resolved_Signal <= '0';
          wait;
    end process;
```

At the beginning of simulation these processes will execute and the signal Resolved_Signal will be updated by passing either the Bit4_Vector ('1', '0') or ('0', '1') to the resolution function and assigning the returned value '1' to the current value of Resolved_Signal This brings up an important point: the resolution function should not depend on the order of the input values.

The body of the resolution function should handle any size of the unconstrained array. For example, the function Wired_Or is intended to return the OR function of all the inputs while disregarding any 'Z' inputs. The body of the function might look like:

```
function Wired_Or (Input: Bit4_Vector) return Bit4 is
  variable Result: Bit4 := '0';
begin
  for I in Input'Range loop
        if Input(I) = '1' then
        Result := '1';
        exit;
  esif Input(I) = 'X' then
        Result := 'X';
    else -- Input(I) = 'Z' or '0'
        null;
  end if;
 end loop;
 return Result;
end Wired_Or;
```

The function computes the Wired-OR and returns the result.

**A Counter Element**

This section will examine a very simple example of a behavioral description of an element of a counter. It will be used to show how processes communicate with each other through signals and how those signals operate.

The counter element is a simple circuit which has the following interface:

```
entity Counter_Element is
  port (
        CIn , Clock : in Bit;
        COut, BOut : out Bit);
  end Counter_Element;
```

CIn and COut are the carry in and carry out of the element and BOut is the output bit. The body of the element is very straightforward to construct. One approach is given by the architecture:

```
architecture Pure_Behavior of Counter_Element is
  signal DffOut : Bit := '0';
  signal ExorOut: Bit := '0';
begin
  L1: BOut <= DffOut;
  L2: ExorOut <= DffOut xor CIn;
  L3: COut <= DffOut and CIn;
  L4: process (Clock)
  begin
        if Clock= '1' then
           DffOut <= ExorOut;
        end if;
  end process;

end Pure_Behaviour
```

This architecture has three concurrent signal assignment statements, labeled L1, L2, and L3, and one process statement, labeled L4. The counter element is driven by the clock signal. When the clock changer to '1', a new BOut and COut are computed. During simulation the following actions are taken:

1. At the beginning of simulation, all processes are executed. This means that all the assignments and the process are executed with the initial values of the signals they are reading.
2. *First half of simulation cycle — signals are updated.* Assume there is an event on clock.
3. *Second half of simulation cycle — processes are executed.* The process labeled L4 is sensitive to Clock and is activated. If the clock is equal to T, a transaction is

scheduled for DffOut after a delta delay. If clock is not equal to T, go back to step 2.

4. *First half of simulation cycle — signals are updated.* DffOut gets the value assigned in the last step.

5. *Second half of simulation cycle — processes are executed.* The assignments labeled LI, L2, and L3 are all sensitive to the signal DffOut, which just got a new value, so each of these assignments is executed.

Step 1 happens only at the beginning of the simulation. Steps 2 through 5 are repeated whenever there is an event on the signal Clock.

Scanned with CamScanner

# CHAPTER 6
# STRUCTURAL DESCRIPTION

A structural description of a piece of hardware is a description of what its subcomponents are and how the subcomponents are connected to each other. Structural description is more concrete than behavioral description; that is, the-correspondence between a given portion of a structural description and a portion of the hardware is easier to see than is the correspondence between a given portion of a behavioral description and a portion of the hardware.

While the basic unit of behavioral description is the process statement, the basic unit of structural description is the component instantiation statement. Neither the process statement nor the component instantiation statement can stand by itself; each must be enclosed in an architecture body, which is a separately analyzable library unit. It is an important characteristic of VHDL that a designer can mix behavioral and structural description at any level. At a macro level, behaviorally-described components can be mixed with structurally-described components in a larger hierarchy; at a micro level, process statements and component instantiation statements can be mixed in a single architecture.

This ability to mix description modes offers the designer several advantages. First, the refinement from behavior to structure (if this is how the design process proceeds) need not proceed at the same rate for all portions of the design; thus at some stage a design may contain both abstract behavioral description for unrefined portions and a structural breakdown for portions whose refinement is known. Second, it is not necessary for a designer to simulate everything at a single, low level; portions of the design that have already been verified at a low, structural level can be replaced with behavioral versions for incorporation to larger simulations. In this way, the language allows more efficient use of simulation resources.

This chapter is divided into four sections. The first sections presents the basic features of structural description in VHDL. The second section explains how regular structures can be described in VHDL. The third section explains configuration specifications. The last section treats unconnected ports and default values on ports.

## Basic Features of Structural Description

Ports and their connections are the central matter of structural description. This section first explains how ports are declared in entity declarations and in component declarations and then explains how these ports can be connected by means of component instantiation statements. Three examples, an ALU, a decoder, and data bus, complete the presentation of basic features of structural description.

## Ports in Entity Declarations

The external view of a hardware component is its connection to the outside. These points of connection could be the of a D connector, the pins of an IC, or the inputs and output of a gate. In VHDL, the external view is given by the header of an entity declaration. The header may declare any number of ports representing points of

Scanned with CamScanner

connection. Each port declaration includes the name of the port, the direction of its data flow (the *mode* of the port), and the type of the data that flows through it.

```
entity And2 is
   port (I1 : in Bit ; I2 : in Bit ; O1 : out Bit) ;
end And2 ;
```

Ports are declared in an *interface list*, the general structure of an interface list was discussed in Chapter 3. The only object class permitted in a port list is signal, and as this is the default object class, if need not be specified. The default mode is in. Since several ports having the same mode and the same type may be grouped into a single interface element, the following is equivalent to the preceding example:

```
entity And2 is
   port (I1, I2 : Bit ; O1 : out Bit) ;
end And2 ;
```

**Port Modes and Direction of Data Flow**

When viewing a design at some level of abstraction above the electrical level, it is often convenient to associate a direction of data flow with a point of connection. VHDL provides for the port modes in, out, inout, and buffer for representing data flow into the component (in), out of the component (out), and both into and out of the component (inout, buffer). Assigning a particular mode to a port implies nothing about the type of data passing through the port (the type could be two-level logic, three-level logic, more abstract types like "real or "integer", etc.). Assigning a particular mode to a port simply announces the intention of the designer to read values from the port, write values to the port, or both mad values from and write values to the port. The rules of VHDL do require that values cannot be read from a port unless the mode of that port is in, inout, or buffer and that values cannot be written to a port unless the mode of that port is out, inout, or buffer, The following example demonstrates incorrect reading of an in port and incorrect writing to an out port.

```
entity Wrong_way is
   port (Not_to_be_read : out Bit ; Not_to_be_written : in Bit) ;
end Wrong_way ;

architecture Wrong_way of Wrong_way is
begin
   process
   begin
         -- The Following Line Illustrates TWO ERRORS
         Not_to_be_written <= Not_to_be_read after 5 Ns ;
```

Scanned with CamScanner

```
        wait for 1 us ;
    end process ;
  end Wrong_way ;
```

While buffer ports, like inout ports, can be both read and updated, the two modes are not fully equivalent. The differences between buffer and inout will be explained in the section on component instantiation statements.

**Ports in Component Declarations**

As explained in Chapter 3, ports can be declared not only in entity declarations but also in component declarations.

```
component And2
   port (I1, I2 : Bit ; O1 : out Bit) ;
end component ;
```

There are several fundamental differences between entity declarations and component declarations. First, while an entity declaration, as a separately compilable library unit, never occurs inside another library unit, a component declaration only occurs inside a library unit. A component declaration may occur inside a package declaration or inside an architecture body. Second, an entity declaration declares something that really "exists" in the design library, while a component declaration merely declares a template that does not really "exist" in the design library. This dual definition of external views gives important flexibility. Suppose that a hardware component A consists W three subcomponents B, C, and D connected in a certain way. To describe this in VHDL, the designer would write an architecture body for A that would instantiate B, C, and D. Now an instantiation is a reference to something, and in VHDL a reference is always a reference to a defining occurrence. If an instantiation referred directly to an entity declaration, then the designer could not analyze the description for A until he had first analyzed the descriptions (entity declarations) for B, C, and D. This would place unnecessary restrictions on the way a designer is allowed to work. VHDL removes this restriction by having instantiations refer not to entity declarations but to component declarations; the associations between these component declarations and actual entities can be established separately, in a configuration declaration (configuration declarations are discussed in Chapter 7).

**Component Instantiation Statements**

The component instantiation statement specifies an instance of a component (a child component) occurring inside another component (a parent component). At the point of instantiation, only the external view of the child component (the names, types, and directions of its ports) is visible; signals internal to the child component are not visible. component instantiation statement identifies the child component and specifies which ports or signals in the parent component are connected to which ports in the child component.

Scanned with CamScanner

```
architecture Parent_body of Parent is
   component And2
         port (I1, I2 : Bit ; O1 : out Bit) ;
   end component ;
   signal S1, S2, S3 : Bit ;
begin
   Child : And2 port map (I1 => S1, I2 => S2, O1 => S3) ;
end Parent_body ;
```

The general form of the component instantiation statement is

*label-identifier : component-mark*
generic map *association-list*
port map *association-list*;

The component instantiation statement *must* be preceded by a label. (Two other statements — the *generate statement,* explained later in this chapter, and the *block statement,* explained in Chapter 7 — also require labels. These labels are referenced in *block configurations* and *component configurations* which are discussed in Chapter 7.) The generic map associations are omitted if the corresponding component declaration lacks generics (generics are discussed later in this chapter) and the port map associations are omitted if the component declaration lacks ports. The general structure of an association list was explained in Chapter 3. As is generally true of associations in VHDL, either positional or named association is possible in the association lists of a component instantiation statement. Thus the following is equivalent to &e component instantiation statement in the preceding example:

```
Child : And2 port map (S1, S2, S3) ;
```

The *component mark,* which is either an identifier or a selected name, must reference a component declared in a component declaration. The component declaration need not occur in the architecture body containing the component instantiation but it must be visible at the point of the component instantiation. In the following example the component declaration occurs in a package. The name of the package is made visible by the use clause and the component declaration is made visible by the selected name "Comp_decls.And2" (Chapter 7 contains a general explanation of how declared items are made visible).

```
package Comp_decls is
   component And2
         port (I1, I2 : Bit ; O1 : out Bit) ;
   end component ;
end Comp_decls ;
```

Scanned with CamScanner

```
use Work. Comp_decls ;
architecture Parent_body of Parent is
   signal S1, S2, S3 : Bit ;
begin
   Child : Comp_decls.And2
         port map (I1 => S1, 12 => S2, O1 => S3)
   end Parent_body ;
```

A port on a component declaration is called a *local*. In a component instantiation statement, the port association list must associate an *actual* with each local. This actual must be an object of class signal; an expression or an object of class variable or constant may not be associated with a local. There are two possibilities for an actual:

- The actual may be a signal declared in a signal declaration. In this case, the signal may have been declared in a package, or in the architecture containing the component instantiation statement, or in the entity declaration corresponding to the architecture containing the component instantiation statement.
- The actual may be a- formal port declared in an entity declaration. In this case, the entity declaration must be the entity corresponding to the architecture containing the component instantiation statement

VHDL imposes three kinds of restrictions on the association of an actual with a local. These restrictions are based cm type, mode, and resolvability. First, VHDL requires that the type of the actual be the same as the type of the local (type-mismatches can be bridged with type conversions around actuals and around locals; this is explained in Chapter 7). Second, VHDL requires that if the local is readable, then the actual most be readable, and if the local is writable, than the actual must be writable. Any signal declared in a signal declaration is both readable and writable; so in general, a signal declared in a signal declaration may be associated with a local of any mode. However, a port of mode in is not writable and a port of mode out is not readable. It follows that an actual that is a port of mode in may not be associated with a local that is of mode out or inout and that an actual that is a port of mode out may not be associated with a local that is of mode in or inout. Third, an association with a local of mode out or inout creates source for the actual (sources are explained in Chapter 5). It follow that an actual (either port or signal) that is not a resolved signal may not be associated with a local of node out or inout if this would the actual's having more than one source. The following exam. illustrates erroneous port associations.

```
entity Parent is
   port (Q1 : in Bit ; Q2 : out Bit) ;
end Parent;

architecture Parent_body of Parent is
   component Child
         port (P1 : in Bit ; P2 : out Bit ; P3 : inout Bit) ;
```

Scanned with CamScanner

```
    end component ;
    signal SI : Bit ;   -- SI IS NOT A RESOLVED SIGNAL
  begin
    -- The Following Line Creates a SOURCE For S1
    SI <= not S1 after 5 Ns ;
    C1 : Child port map (
    -- The Following Line Illustrates TWO MISMATCHES
    -- OF MODE
            P1 => Q2, P2 => Q1
    -- The Following Line Would Create an
    -- ADDITIONAL SOURCE FOR S1
            P3 => S1) ;
  end Parent_body ;
```

A port of mode buffer is like a port of mode inout in that they both are readable and writable; since a buffer port is writable, an association of an actual with a local buffer defines a source for the actual just as an association of an actual with a local out or inout defines a source for that actual. Buffer ports differ from inout ports in that a buffer port may never have more than one source (regardless of whether its subtype is resolvable or not) and that the only kind of actual &at may be associated with a buffer port is another buffer port or a signal which has no other source. A buffer port also differs from an inout port in that the effective value of a buffer port is *always* the same as its driving value, regardless of any type conversions that may have applied to the port associations, while the effective value of an inout Port may differ from its driving value.

The following example illustrates errors involving buffer ports.

```
package Types is
   function Resolve_bit (X : Bit_vector) return Bit ;
   subtype Resolved_bit is Resolve_bit Bit ;
end Types ;

use Work. Types.all ;
entity Parent is
   port (Q1 : buffer Resolved_bit; Q2 : in Bit; Q3 : out Bit ;
          Q4 : inout Bit) ;
end Parent;

architecture Parent_body of Parent is
   component Child
         port (P1 : buffer Bit);
   end component ;
   signal S1 : Resolved_bit ;
begin
   -- The Following Line Creates A SOURCE FOR S1
```

Scanned with CamScanner

```
S1 <=not SI after 5 Ns ;
-- The Following Line Creates A SOURCE FOR Q1
Q1 <= not Q1 after 5 Ns ;
-- The Following Three Lines Illustrate
-- MISMATCHES OF MODE
C1 : Child port map (P1 => Q2) ;
C2 : Child port map (P1 => Q3) ;
C3 : Child port map (P1 => Q4) ;
-- The Following Line Would ERRONEOUSLY Associate an
-- ACTUAL BUFFER PORT That Already Has a
-- SOURCE With a LOCAL BUFFER
C4 : Child port map (P1 => Q1) ;
-- The Following Line Would ERRONEOUSLY Associate an
-- ACTUAL SIGNAL That Already Has a
-- SOURCE With a LOCAL BUFFER
C5 : Child port map (P1 => S1) ;
end Parent_body ;
```

**Example: A Simple ALU**

This section contains an example showing how to use the basic features of structural description in VHDL (entity declaration, component declaration, component instantiation statement). The VHDL example specifies a single stage of an ALU (Figure 6.1 is a diagram of the ALU Inputs to the ALU stage .are: four select lines S3, S2, S1, and inverted carry C1; a mode selection bit M; and the two operand and B1. Outputs are the arithmetic or logic result Fl and the out carry C2. Logic functions are generated when M = '1', arithmetic functions when M = '0'. When M = '1' the output carry will always be '1'.

The following table shows logic functions and arithmetic functions that would be generated, (on output F1) for selected values of the lines S3 & S2 & S1 & S0.

Scanned with CamScanner

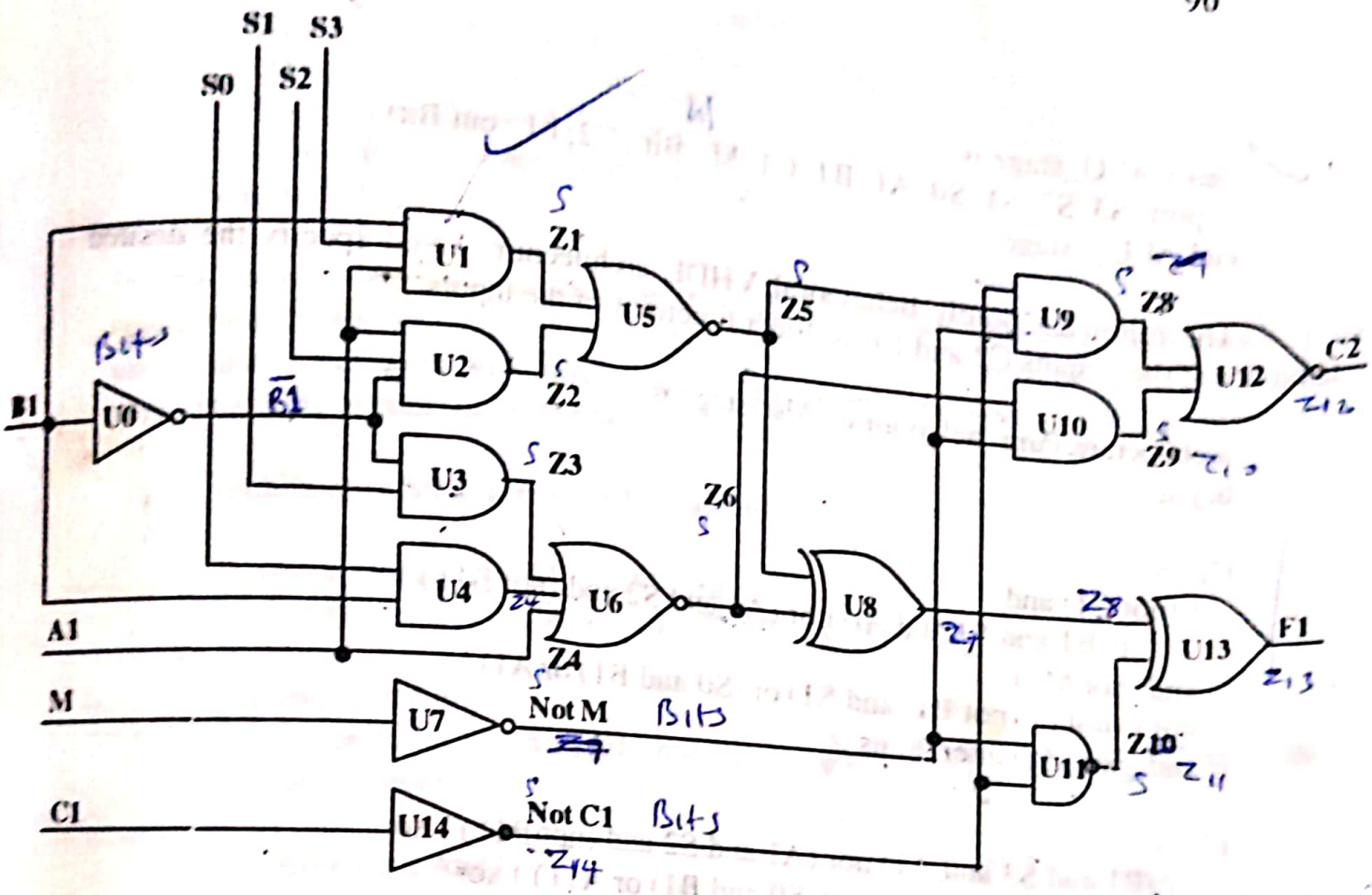



Figure 6.1. ALU Stage

if M = 1

| | |
|---|---|
| "0000" | not A1 |
| "0011" | '0' |
| "0101" | not B1 |
| "0110" | A1 xor B1 |
| "1010" | B1 |
| "1011" | A1 and B1 |
| "1100" | '1' |
| "1110" | A1 or B1 |
| "1111" | A1 |

if M = 0

| | |
|---|---|
| "0000" | A1 + not C1 |
| "0011" | not C1 |
| "0110" | A1 + (1's complement B1) + not C1 |
| "1001" | A1 + B1 + not (C1 |
| "1111" | A1 – not C1 |

Here is the entity declaration for ALU_stage:

```
entity ALU_stage is
   port (S3, S2, S1, S0, A1, B1, C1, M : Bit ; C2, F1 : out Bit) ;
end ALU__stage ;
```

The following purely behavioral VHDL architecture would specify the desired behavior of the outputs C2 and F1 as direct functions of the inputs.

```
architecture Pure_behavior of Alu_stage is
begin

C2<=
   ( (not C1) and
         ( (B1 and S3 and Al) nor (A1 and S2 and (not B1) ) )
   and (not M) )
   nor ( (not ( ( (not B1) and S1) or (S0 and B1) or A1) )
   and (not M) ) after 20 ns ;

Fl <=
   ( ( (B1 and S3 and A1) nor (A1 and S2 and (not B1) ) ) xor
   (not ( ( (not B1) and S1) or (S0 and B1) or A1) ) ) xor
   ( (not C1) nand (not M) ) after 20 ns ;

end Pure_behavior;
```

This purely behavioral specification could be rewritten to intermediate computations as local signals; but the resulting still behavioral since there are no component instantiation statements

```
architecture Behavior of ALU_stage is
   signal NotC1, NotB1, NotM : Bit ;
   signal Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, Z9, Z10 : Bit ;
   constant Delay : Time := 5 Ns ;
begin
   U0 : NotB1 <= not B1 after Delay ;
   U1 : Z1 <= B1 and S3 and A1 after Delay ;
   U2 : Z2 <= A1 and S2 and NotB1 after Delay ;
   U3 : Z3 <= NotB1 and S1 after Delay ;
   U4 : Z4 <= S0 and B1 after Delay ;
   U5 : Z5 <= Z1 nor Z2 after Delay ;
   U6 : Z6 <= not (Z3 or Z4 or A1) after Delay ;
   U7 : NotM <= not M after Delay ;
   U8 : Z7 <= Z5 xor Z6 after Delay ;
   U9 : Z8 <= NotC1 and Z5 and NotM after Delay
```

Scanned with CamScanner

```
    U10 : Z9 <= Z6 and NotM after Delay ;
    U11 : Z10 <= NotC1 nand NotM after Delay ;
    U12 : C2 <= Z8 nor Z9 after Delay ;
    U13 : Fl <= Z7 xor Z10 after Delay ;
    U14   NotC1 <= not C1 after Delay ;
end Behavior;
```

For the immediately preceding version there is a simple structural equivalent: each of the concurrent signal assignments using built-in operators is replaced with a component instantiation statement that instantiates a gate-level primitive (which is assumed to perform the same function as the built-in operator).

```
architecture Structure of ALU_stage is

    signal NotC1, NotB1, NotM : Bit ;
    signal Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8, Z9, Z10 : Bit ;

    component And2
        port (I1, 12 : Bit ; O1 : out Bit) ;
    end component ;

    component And3
        port (I1, 12, 13 : Bit ; 01 : out Bit) ;
    end component ;

    component Nor2
        port (I1, 12 : Bit ; O1 : out Bit) ;
    end component ;

    component Nor3
        port (I1, 12,13 : Bit ; O1 : out Bit) ;
    end component ;

    component Inverter
        port (I1 : Bit ; O1 : out Bit) ;
    end component;

    component Xor2
        port (I1. I2 : Bit ; O1 : out Bit) ;
    end component;

    component Nand2
        port (I1, I2 : Bit ; O1 : out Bit) ;
    end component ;
```

Component And 2
Port (I1, I2 : Bit; O1, out Bit)
end Component;
Component And 3
port (I1, I2, I3 : Bit; O1 : out Bit
end Component;
Cw

```
begin
   U0 : Inverter port map (B1, NotB1) ;
   U1 : And3 port map (B1, S3, A1, Z1) ;
   U2 : And3 port map (A1, S2, NotB1, Z2) ;
   U3 : And2 port map (NotB1, S1, Z3) ;
   U4 : And2 port map (S0, B1, Z4) ;
   U5 : Nor2 port map (Z1, Z2, Z5) ;
   U6 : Nor3 port map (Z3, Z4, A1, Z6) ;
   U7 : Inverter port map (M, NotM) ;
   U8 : Xor2 port map (Z5, Z6, Z7) ;
   U9 : And3 port map (NotCl, Z5, NotM, Z8) ;
   U10 : And2 port map (Z6, NotM, Z9) ;
   U11 : Nand2 port map (NotCl, NotM, Z10) ;
   U12 : Nor2 port map (Z8, Z9, C2) ;
   U13 : Xor2 port map (Z7, Z10, F1) ;
   U14 : Inverter port map (C1, NotC1) ;
end Structure
```

End page 99

**Example: A Decoder**

This section contains a further example showing how to use the basic features of structural description. We first present an abstract behavioral specification of the decoder.

```
entity Decoder is
   port (Enable : Bit ;
         Sel : Bit_vector (2 downto 0) ;
         Dout : out Bit_vector (7 downto 0) ) ;
   constant Delay : Time := 5 Ns ;
end Decoder;

architecture Behavior of Decoder is
begin
   with Sel select
         Dout <=
            "00000001" after Delay when "000" ,
            "00000010" after Delay when "001" ,
            "00000100" after Delay when "010" ,
            "00001000" after Delay when "011" ,
            "00010000" after Delay when "100" ,
            "00100000" after Delay when "101" ,
            "01000000" after Delay when "110",
            "10000000" after Delay when "111",
   end Behavior;
```

Now we present the structural description of the decoder.

Scanned with CamScanner

```
architecture Structure of Decoder is
  component And3
      port (I1, I2, I3 : Bit ; O1 : out Bit) ;
  end component ;

  component Inverter
      port (I1 : Bit ; O1 : out Bit) ;
  end component ;

  signal Sel_bar : Bit_vector (2 downto 0);
begin
  Inv_0 : Inverter
      port map (Sel(0), Sel_bar(0)) ;
  Inv_1 : Inverter
      port map (Sel(1), Sel_bar(1)) ;
  Inv_2 : Inverter
      port map (Sel(2), Sel_bar(2));
  And_0 : And3
      port map (Sel_bar(0), Sel_bar(1), Sel_bar(2), Dout(0)) ;
  And_1 : And3
      port map (Sel_bar(0), Sel_bar(1), Sel      (2), Dout(1)) ;
  And_2 : And3
      port map (Sel_bar(0), Sel      (1), Sel_bar(2), Dout(2)) ;
  And_3 : And3
      port map (Sel_bar(0), Sel      (1), Sel      (2), Dout(3)) ;
  And_4 : And3
      port map (Sel      (0), Sel_bar (1), Sel_bar(2), Dout(4));
  And_5 : And3
      port map (Sel      (0), Sel_bar (1), Sel      (2), Dout(5));
  And_6 : And3
      port map (Sel      (0), Sel      (1), Sel_bar(2), Dout(6));
  And_7 : And3
      port map (Sel      (0), Sel      (1), Sel      (2), Dout(7)) ;

end Structure;
```

Scanned with CamScanner

**Example: Data Bus**

The two preceding examples, the ALU and the decoder, were both gate-level descriptions. As an example of higher-level structural description we will give a top-level sketch of a system consisting of several devices connected by a bus. If multiple devices are to write to a single bus, their access must be controlled by a protocol. In this example, each device that is to write to the bus is connected to a poller by one request line (port Req) and one acknowledge line (port Ack); each device is also connected to the bus (port D_bus). The device neither reads nor writes any of these ports directly; rather, each device instantiates a subcomponent Protocol which encapsulates the entire bus protocol. In this way, the common bus interface is separated from the peculiarities of each device. Each device passes the request line, acknowledge line and bus to its Protocol subcomponent. Each device has two additional connections with its Protocol subcomponent: the Write port is the means by which the device sends a write command to the protocol, and the Data port is the actual data that the device is attempting to put on the bus. Figure 6.2 shows two devices wired in the manner just described. The complete description consists of the package Bus_logic, and three design entities: entity Device_1 and its architecture Device_body; entity Device_2 and its architecture Device_body; entity Devices_on_bus and its architecture Devices_on_us_body. The Poller and Protocol are present as component declarations; however, as their internal structure is not developed their entity declarations and architecture bodies are not given.

Package Bus_logic contains the necessary type definitions. Since there will be multiple sources for the bus, it must be a resolved signal; the type of the bus will be type Logic, a resolved subtype of type Base_logic. Type Base_logic_array, an array of the unresolved Base_logic, will be used as the type of the array of request lines and acknowledge lines on the Poller subcomponent; the arrays will be split with one wire from the array going to each device. The package also contains the component declarations for the Poller and Protocol.

```
package Bus_logic is
  type Base_logic is ('0', '1', 'Z') ;
  type Driver_array is array (Natural range <>) of Base_logic ;
  -- a resolution function
  function Resolve_drivers (P : Driver_array) return Base_logic ;
  -- resolved subtype for the bus
  Subtype Logic is Resolve_drivers Base_logic ;
  constant Number_of_devices : Positive := 2 ;
  -- Base_logic_array will be used to declare the array of request
  -- and acknowledge lines on the Poller, in this example, there will
  -- be two devices.
  type Base_logic_array is array (1 to Number_of_devices)
        of Base_logic ;

  component Protocol
        port (D_bus : inout Logic := 'Z' ; Data : Base_logic ;
```

Scanned with CamScanner

```
        Write, Ack : Base_logic ; Req : out Base_logic) ;
    end component ;

    component Poller
        port (Req : Base_Logic_array ;
            Ack : out Base_logic_array) ;
    end component;
end Bus_logic ;
```

In this example, the entity declarations for Device_1 and Device_2 have identical port declarations. In a more detailed example, each device would have additional ports peculiar to its own function. The three ports D_bus, Req, and Ack represent the information needed by the Protocol subcomponent which each device will instantiate.

```
use Work.Bus_logic.all ;
entity Device_1 is
    port (D_bus : inout Logic := 'Z' ;
        Req : out Base_logic ;
        Ack : in Base_logic
        -- other ports peculiar to Device_1
        ) ;
end Device_1 ;

use Work.Bus_logic.all ;
entity Device_2 is
    port (D_bus : inout Logic := 'Z' ;
        Req : out Base_logic ;
        Ack : in Base_logic
        -- other ports peculiar to Device_2
        ) ;
end Device_2 ;
```

The body of each device declares a local signal Write and a local signal The signal Data holds the current value that the device wishes to put on the bus; the signal Write instructs the Protocol subcomponent to actually put the value on the bus. The device's ports D_bus, Ack, and Req are simply passed down to the Protocol subcomponent. The architecture bodies of Device_1 and Device_2 are identical in this simplified example. Additional logic, peculiar to each device, would appear in a more detailed example.

```
architecture Device_body of Device_1 is :
    signal Write : Base_ogic := '0' ;
    signal Data : Base_logic := '0' ;
begin
    -- instantiation of the Protocol subcomponent
```

Scanned with CamScanner

```
    Bus_protocol : Protocol
        port map (D_Bus, Data, Write, Ack, Req) ;
    -- architecture also contains logic peculiar to Device_1
end Device_body ;

architecture Device_body of Device_2 is
    signal Write : Base_logic := '0' ;.
    signal Data : Base_logic := '0' ;
begin
    -- instantiation of the Protocol subcomponent
    Bus_protocol : Protocol


        port map (D-bus, Data, Write, Ack, Req) ;
    -- architecture also contains logic peculiar to Device_2
end Device_body;
```

The top-level design unit, Devices_on_bus, instantiates the Poller and the two devices. It also contains the signal declarations for the request lines, the acknowledge lines, and the actual bus.

```
use Work.Bus_logic.all ;
entity Devices_on_bus is
end Devices_on_bus ;

architecture Devices_on_bus_body of Devices_on_bus is

  component Device_1
      port (D_bus : inout Logic := 'Z' ;
        Req : out Base_logic ;
        Ack : in Base_logic
        -- other ports peculiar to Device_1
        ) ;
end component;

component Device_2
  port (D_bus : inout Logic := 'Z' ;
      Req : out Base_logic ;
      Ack : in Base_logic
      -- other ports peculiar to Device_2
      ) ;
end component ,

signal D_bus : Logic := 'Z' ;
```

Scanned with CamScanner

```
    signal Req, Ack : Base_logic_array := (others => '0')

begin

    U1 : Device_1 port map (D_bus, Req (1), Ack (1)) ;
    U2 : Device_2 port map (D_bus, Req (2), Ack (2)) ;

    Poll_devices: Poller port map (Req, Ack) ;

end Devices_on_bus_body ;
```

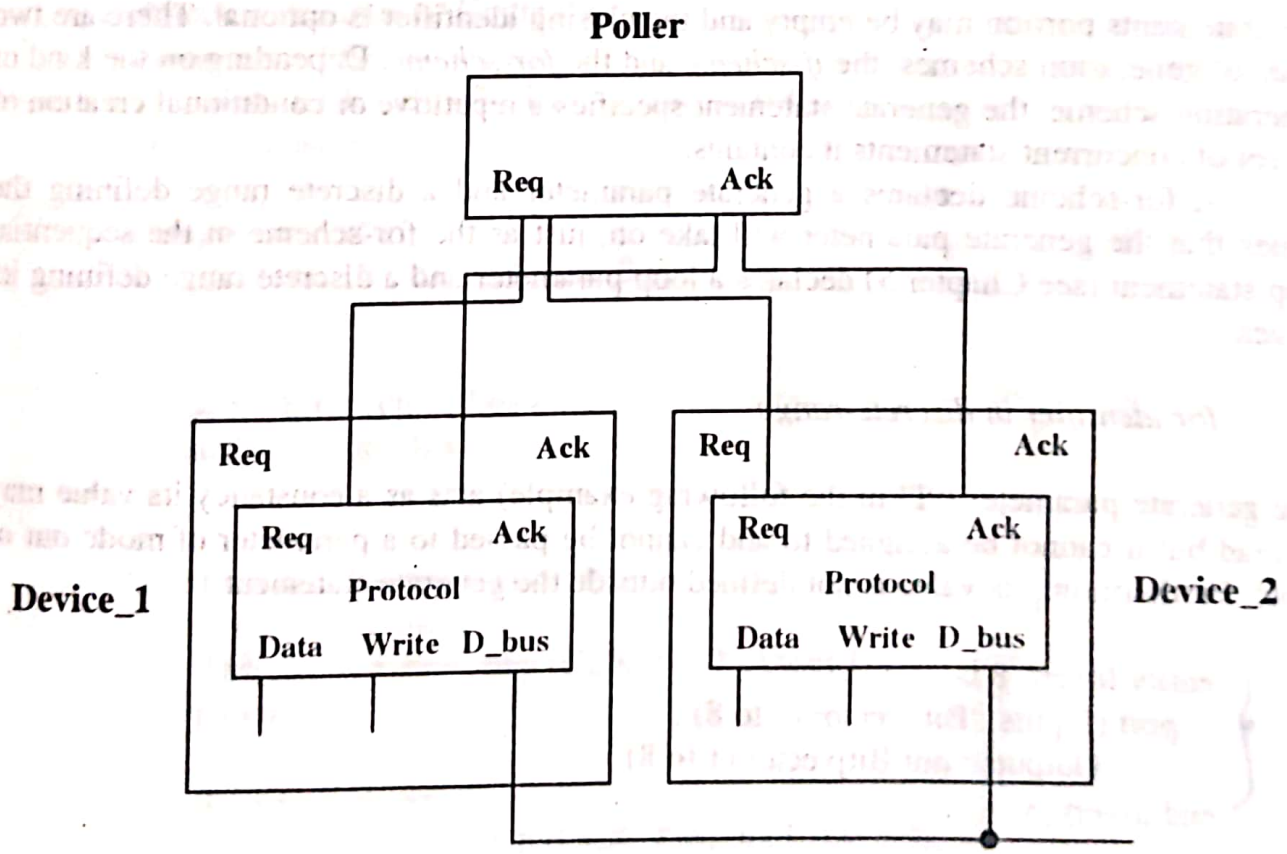



Figure 6.2, Data Bus with Two Devices

**Regular Structures**

Useful hardware often can be described as multiple instances of the same subcomponent, connected in some regular pattern. This section explains the generate statement, which is the means provided by VHDL to describe regular structures. This section also introduces the VHDL generic, which, when used in conjunction with the generate statement, allows the designer to specify regular structures that are general (not specific) in size.

Scanned with CamScanner

## Generate Statements

The *generate statement* consists of a *generation scheme* and a set of enclosed concurrent statements. Any VHDL concurrent statement — process statement, block statement, concurrent assertion statement; concurrent signal assignment statement, concurrent procedure call, component instantiation statement, and another generate statement — may be enclosed by a generate statement. The general form of a generate statement is

*label-identifier: generation-scheme* generate
  *concurrent-statements*
end generate *identifier* ;

The statements portion may be empty and the closing identifier is optional. There are two kinds of generation schemes: the *if-scheme* and the *for-scheme*. Depending on the kind of generation scheme, the generate statement specifies a repetitive or conditional creation of the set of concurrent statements it contains.

A for-scheme declares a generate parameter and a discrete range defining the values that the generate parameter will take on, just as the for-scheme in the sequential loop statement (see Chapter 5) declares a loop parameter and a discrete range defining its values.

*for identifier in discrete-range*

The generate parameter ("I" in the following example) acts as a constancy its value may be read but it cannot be assigned to and cannot be passed to a parameter of mode out or inout. Furthermore, its value is not defined outside the generate statement.

entity

```
entity Invert_8 is
  port (Inputs : Bit_vector (1 to 8) ;
        Outputs : out Bitjvector (1 to 8) ) ;
end Invert_8 ;
```

architecture
behavioral

```
architecture Invert_8 of Invert is
  component Inverter
        port (I1 : Bit; O1 : out Bit) ;
  end component ;
begin
  G : for I in 1 to 8 generate
        Inv : Inverter port map (Inputs(I), Outputs(I)) ;
  end generate ;
end Invert_8 ;
```

There are no concurrent statements analogous to the sequential exit-statement and next-statement (see Chapter 5), so a generate statement with a for-scheme will always iterate

Scanned with CamScanner

over all values in the discrete range. However, it is possible for a for-scheme to specify a null range.

Regular structures often show some regularity at the extremes. The if-generate copes with these irregularities. The following simple counter consists of T flip-flops and AND gates. But the low-bit stage and the high-bit stage are slightly different (Figure 6.3 shows the schematic).

```
entity Counter is
   port (
         CLK : Bit ; Carry : Bit ;
         Dout : buffer Bit_vector (7 downto 0) ) ;
end Counter;

architecture Counter of Counter is
   component TFF
         port (CLK : Bit ; T : Bit ; Q : buffer Bit) ;
   end component ;

   component And2
         port (I1, 12 : Bit ; O1 : out Bit) ;
   end component;

   signal S : Bit_vector (7 dawnto 0) ;
   signal Tied_high : Bit := '1' ;
begin

   G1 : for I in 7 downto 0 generate
   G2 : if I = 7 generate
         TFF_7 : TFF port map (CLK, S(I-1), Dout(I) ;
   end generate ;

   G3 : if I = 0 generate
         TFF_0 : TFF port map (CLK, Tied_high, Dout(I) ;
         S(I) <= Dout(I) ;
   end generate ;

   G4 : if I > 0 and I < 7 generate
         And_1 : And2 port map (S(I-1), Dout(I), S(I)) ;
         TFF_1 : TFF port map (CLK, S(I-1), Dout(I)) ;
   end generate ;

 end generate ;
end Counter ;
```

Scanned with CamScanner

Unlike the sequential if-statement, the if-generate cannot have else or elsif branches

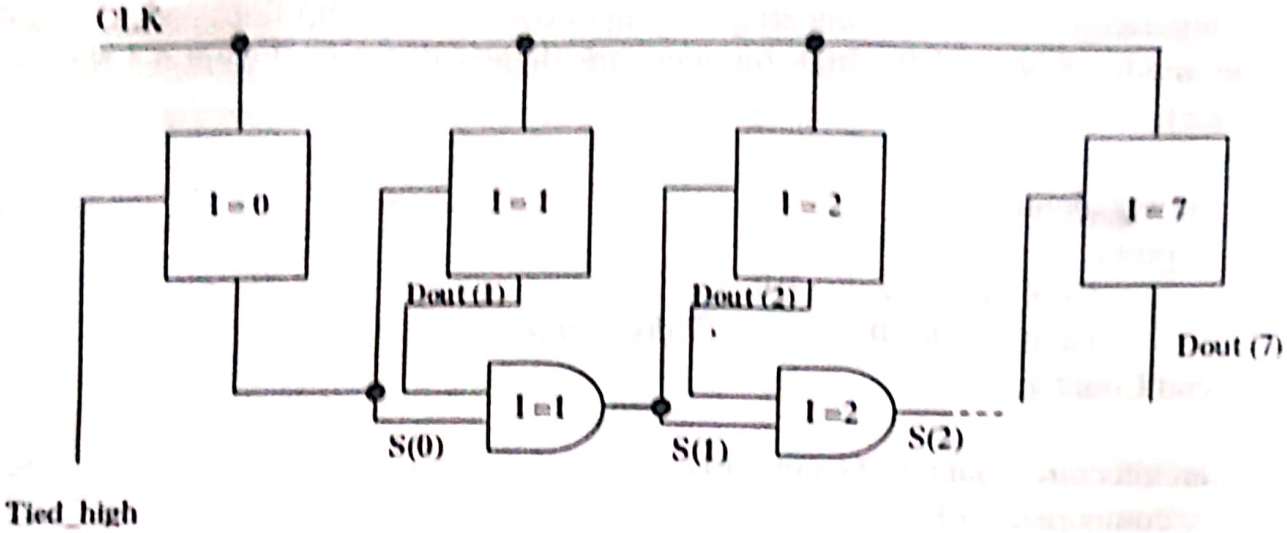



Figure 6.3. Generated Counter

An earlier example described a decoder, which exhibits a regular structure. However, the VHDL of that example did not capture this regularity. Using nested generate statements we can exhibit the basic regularity of the decoder. In the generated version, there is only component instantiation statement in the VHDL text, and this statement] describes the pattern of connections that holds for each component instance by the nested generate statements.

```
architecture Generated_structure of Decoder is
   component And3
         port (I1, I2, I3 : Bit; O1 : out Bit) ;
   end component;

   component Inverter
         port (I1 : Bit ; O1 : out Bit) ;
   end component;

   signal T : Bit_vector (5 downto 0) ;
begin

   Array_select_lines:
   for M in 2 downto 0 generate
         T(M*2+1) <= Sel(M) ;
         Inv_0 : Inverter port map (Sel(M), T(M*2) ) ;
   end generate ;

   two_exp_2 :
   for I in 0 to 1 generate
```

Scanned with CamScanner

```
        Two_exp_l :
        for J in 0 to 1 generate

          Two_exp_0 :
          for K in 0 to 1 generate
              And_0 : And3 port map (T(2*2+l), T(2*I+J), T(2*0+K),
              Dout ((2**2)*I + (2**1)*J + (2**0)*K) ) ;
          end generate;
        end generate;
    end generate;
  end Generated_structure ;
```

The following example shows how to construct an 8-bit ALU from the description of the single ALU stage given in an earlier example. The body of the outer generate (GO) will be elaborated eight limes, once for each bit-slice. The three inner generates (G1, G2 and G3) distinguish the low-order stage and the high-order stage from the other six stages; this is necessary because only the low-order stage is connected to the external carry-in and only the high-order stage is connected to the external carry-out

```
package Reg_pkg is
   constant Size : Positive :» 8 ;
   type Reg is array (Size-1 downto 0) of Bit ;
   type Bit4 is array (3 downto 0) of Bit ;
rend Reg_pkg ;

use Work.Reg_pkg.alI ;

entity ALU is
  port (
        Sel : Bit4 ;
        Rega, Regb : Reg ;
        C, M : Bit ;
        Cout: out Bit ;
        Result : out Reg) ;
end ALU ;

architecture Generated_structure of ALU is
  signal Carry : Reg ;
  component ALU_stage
       port (
          S3, S2, S1, SO, A1, B1, C1, M : Bit ;
          C2, F1 : out Bit) ;
  end component;
```

Scanned with CamScanner

```
begin

G0 : for I in 0 to Size-1 generate

  G1 : if I = 0 generate
       U1 :  ALU_stage      port map (Sel (3), Sel (2), Sel (1), Sel (0)
                            Rega (I), Regb (I), C, M,
                            Carry (I), Result (I) ) ;
  end generate ;

  G2 : if I > 0 and I < Size-1 generate
       U2 :  ALU_stage port map (Sel (3), Sel (2), Sel (1), Sel (0)
                            Rega (I), Regb (I), Carry (I-1), M,
                            Carry (I), Result (I) ) ;
  end generate ;

  G3 : if I = Size-1 generate
       U3 :  ALU_stage port map (Sel (3), Sel (2), Sel (1), Sel (0)
                            Rega (I), Regb (I), Carry (I-1), M,
                            Cout, Result (I) ) ;
  end generate ;
 end generate ;
end Generated_structure ;
```

**Generics**

Entity declarations and component declarations may declare generics in addition to ports. Generics provide a means for an instantiating (parent) component to pass values to an instantiated (child) component. Typical uses of generics are to parameterize timing, the range of subtypes, the number of instantiated subcomponents, and the size of array objects (in particular the size of ports), or simply to document physical characteristics such as temperature. In the following examples, the generics N and Size are used to constrain the size of array ports.

```
entity Decoder is
  generic (N : Positive) ;
  port (Enable : Bit ;
        Sel : Bit_vector (N-1 downto 0) ;
        Dout : out Bit_vector ((2**N)-1 downto 0) ) ;
  constant Delay : Time := 5 Ns ;
end Decoder;

entity ALU is
  generic (Size : Positive) ;
  port (Sel : Bit_vector (3 downto 0) ;
```

Scanned with CamScanner

```
        Rega, Regb : Bit_vector (Size-1 downto 0) ;
        C, M : Bit ; Cout : out Bit ;
        Result : out Bit_vector (Size-1 downto 0) ) ;
end ALU;
```

Like ports, generics are declared in interface lists. The only object class permitted in a generic list is constant, and as this is the default object class, it need not be specified. The only mode permitted is in. Since a generic is a constant of mode in, its value can be read but it cannot be assigned to or passed to a parameter of mode out or inout. Like ports, generics appear in both entity declarations and component declarations.

```
component Decoder
   generic (N : Positive) ;
   port (Enable : Bit ;
         Sel : Bit_vector (N-1 downto 0) ;
         Dout : out Bit_vector ((2**N)-1 downto 0)) ;
end component;

component X
   generic (Rising_delay, Falling_delay : Time ;
         Labl : String) ;
end component;
```

There are two uses of generics that are of particular relevance to structural description: using generics to control generate statements and using generics to index into arrays or define index constraints. The following example is another version of the decoder (Figure 6.4 shows the schematic). This version is generic in size; the same VHDL will describe a 1x2, 2x4, 3x8 ... decoder, depending only on the value of the generic N. Note that this example takes advantage of the fact that VHDL structure can be recursive; that is, an architecture A of an entity E can itself instantiate architecture A of E (obviously this instantiation must be inside a conditional generate or there would be no means to end the recursion).

```
entity Decoder is
   generic (N : Positive) ;
   port (Sel : Bit_vector (1 to N) ;
         Dout : out Bit_vector (1 to 2**N)) ;
end Decoder ;

architecture Generic_structure of Decoder is

   signal Sel_bar : Bit ;
   component And2
         port (I1, I2 : Bit ; O1 : out Bit) ;
```

Scanned with CamScanner

```
    end component ;

    component Inverter
          port (I1 : Bit ; O1 : out Bit) ;
    end component ;

    component Decoder
          generic (N : Positive) ;
          port (
             Sel : Bit_vector (1 to N) ;
             Dout : out Bit_vector (1 to 2**N)) ;
          end component ;

  begin

    Invert_select:
    Inverter port map (Sel (N), Sel_bar) ;

    Not_recursive :
    if N = 1 generate

          Dout (N) <= Sel (N) ;
          Dout (2**(N-1)+1) <= Sel_bar ;
    end generate ;

    Recursive :
    if N > 1 generate
          B1 : block
             signal Temp : Bit_vector (1 to 2**(N-1) ) ;
          begin
             N_minus_1 : Decoder generic map (N - 1)
                                 port map (Sel (1 to N-1), Temp) ;
             For_each_output_from_minus_1 :
             for I in 1 to 2**(N-1) generate
                   And_each_minus_1_with_Sel :
                      And2 port map (Temp (I), Sel (N), Dout (2*(I-1)+1)) ;
                   And_each_N_minus_1_with_Sel_bar :
                      And2 port map (Temp (I), Sel_bar, Dout (2*1) ) ;
             end generate ;
          end block ;
    end generate;

  end Generic_structure;
```

Scanned with CamScanner

The ALU can also be made generic in size, merely by making the constant Size a generic.

## Configuration Specifications

It was explained in the section on component declarations that a component instantiation statement refers to a component declared in a component declaration, not an entity declared in an entity declaration. At some point in the design process a designer will wish to specify, for each component instance, exactly which entity declaration in which design library and which architecture of that design entity is to be selected. The *configuration specification* is the construct that allows the designer to specify the selection of entity declaration and architecture body for each component instance. The following example illustrates a simple use of the configuration specification. The example contains four library, units, all assumed to be analyzed into one design library: (a) entity Inverter; (b) entity Inverter_user; (c) architecture Inverter_user of Inverter_user, and (d) architecture Inverter_body of Inverter. Entity Inverter_user contains a component declaration for a component Inv. Architecture Inverter_user contains a component instantiation statement that instantiates Inv and (preceding the component instantiation statement) a configuration specification that binds the instance of Inv to

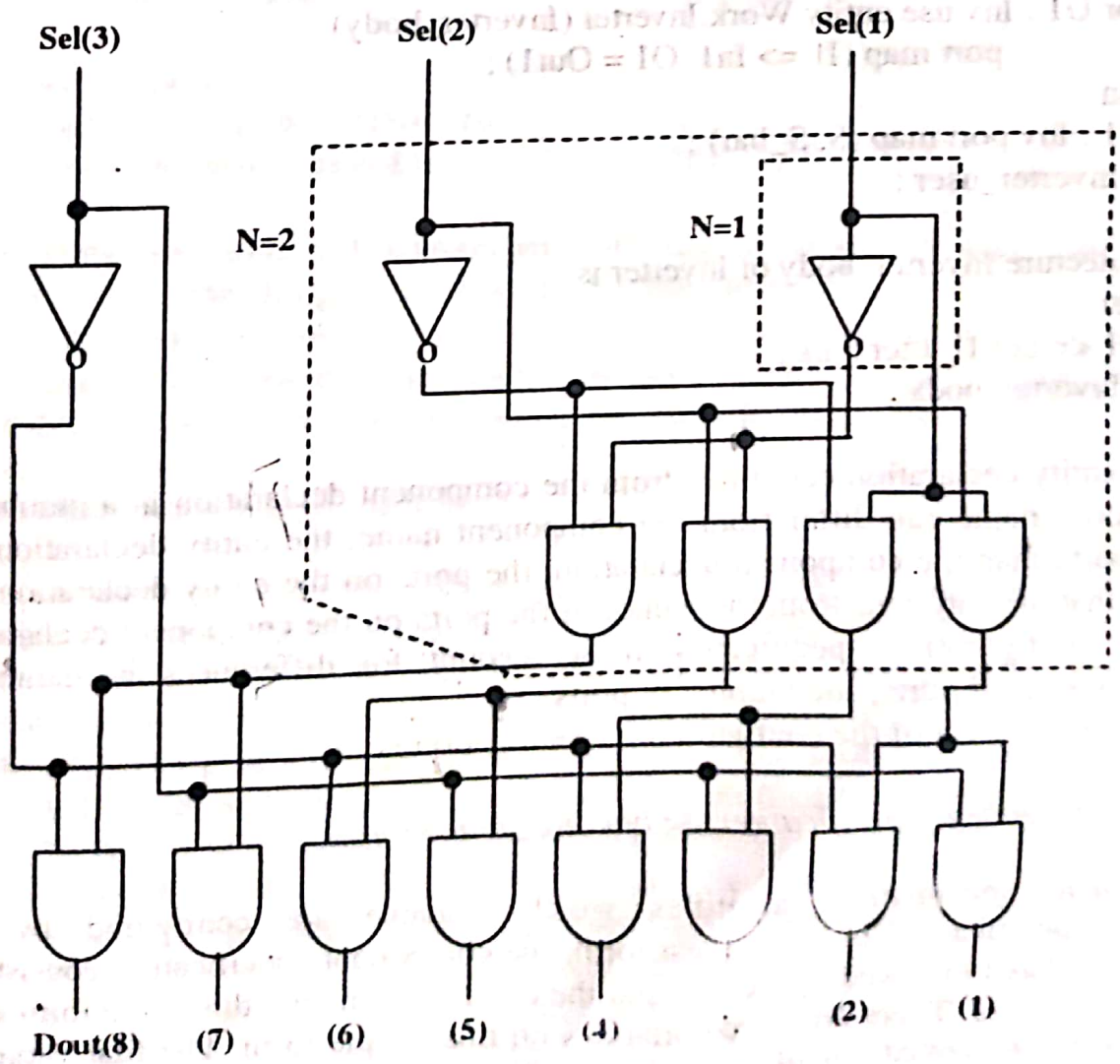



Figure 6.4. Generic Decoder

Scanned with CamScanner

the entity Inverter and to its architecture Inverter_body. Note that it is permissible for the binding indication of a configuration specification reference an architecture body (Inverter_body) that has not yet be analyzed.

```
entity Inverter is
   port (I1 : in Bit ; O1 : out Bit) ;
end Inverter ;

entity Inverter_user is
end Inverter_user ;

architecture Inverter_user of Inverter_user is
   signal S : Bit ;
   signal S_bar : Bit ;
   component Inv
         port (In1 : in Bit ; Out1 : out Bit) ;
   end component ;
   for U1 : Inv use entity Work.Inverter (Inverter_body)
            port map (I1 => In1, O1 = Out1) ;
begin
   U1 : Inv port map (S, S_bar) ;
end Inverter_user ;

architecture Inverter_body of Inverter is
begin
   O1 <= not I1 after 5 ns ;
end Inverter_body ;
```

The entity declaration can differ from the component declaration in a number of ways: the entity name can differ from the component name; the entity declaration can have more ports than the component declaration; the ports on the entity declaration ~au have names that are different from the names of the ports on the component declaration. Clearly, the configuration specification must account for differences in names of components, names of ports, and number of ports.

The general form of the configuration specification is

for *component_specification* use *binding_indication* ;

The *component specification* identifies which instance are configured by the configuration specification. In its simplest form, the component specification consists of an instantiation label followed by a colon and the component name; this is the form used in the example above. There arc three variations on this simple form. The first variation allows a comma-separated list of instantiation labels instead of a single instantiation

Scanned with CamScanner

label; clearly all instantiation labels in the list must label instantiations of the same component.

for U2, U3 : Inverter use entity Work.Inv1 (Inv1_body) ;

A second variation uses the reserved word all in place of the instantiation label; such a configuration specification applies to all instantiations of the given component.

for all : And_gate use entity Work.And_gate1 (And_gate1) ;

A third variation uses the reserved word others in place of the instantiation label; such a configuration specification applies to all instantiations of the given component except for those instances whose labels appear in other (preceding) configuration specifications.

for others : Inverter use entity Work.Inv2 (Inv2.body) ;

The *binding indication* specifies the mapping between component declaration and entity declaration. The general form of a binding indication is

*entity-aspect*
generic map *association-list*
port map *association-list*

Every binding indication will contain an *entity aspect* whose function is to name the entity declaration the designer wishes the instance to be bound to. The entity aspect consists of the reserved word entity followed by the name of the entity. The name of the entity is optionally followed by the simple indentifier (in parentheses) of the architecture body the designer wishes the component instance to be bound to.

entity *entity-mark* ( *identifier* )

Note that the name of the entity may be qualified by the name of the library in which it resides, but that the name of the architecture botly may not be so qualified. This restriction simply reflects the fact that the architecture body must be in the same library as its entity declaration. There are actually two other forms of the entity aspect. First, there may already be in a design library a *configuration declaration* that configures a subcomponent hierarchy; if this is the case, the entity aspect consists of the reserved word configuration followed by the name of this configuration declaration (this is discussed in Chapter 7). Second, the designer may wish to leave a component instantiation explicitly unbound; in this case the entity aspect consists of the reserved words entity open.

Two instantiations of the same component may be configured so that they are bound to different entity declarations. Conversely, two instantiations of two different components may be configured so that they are bound to the same entity declaration.

Scanned with CamScanner

Figure 6.5, 6.6 and 6.7 illustrate several possible ways in which components may be bound to entities.

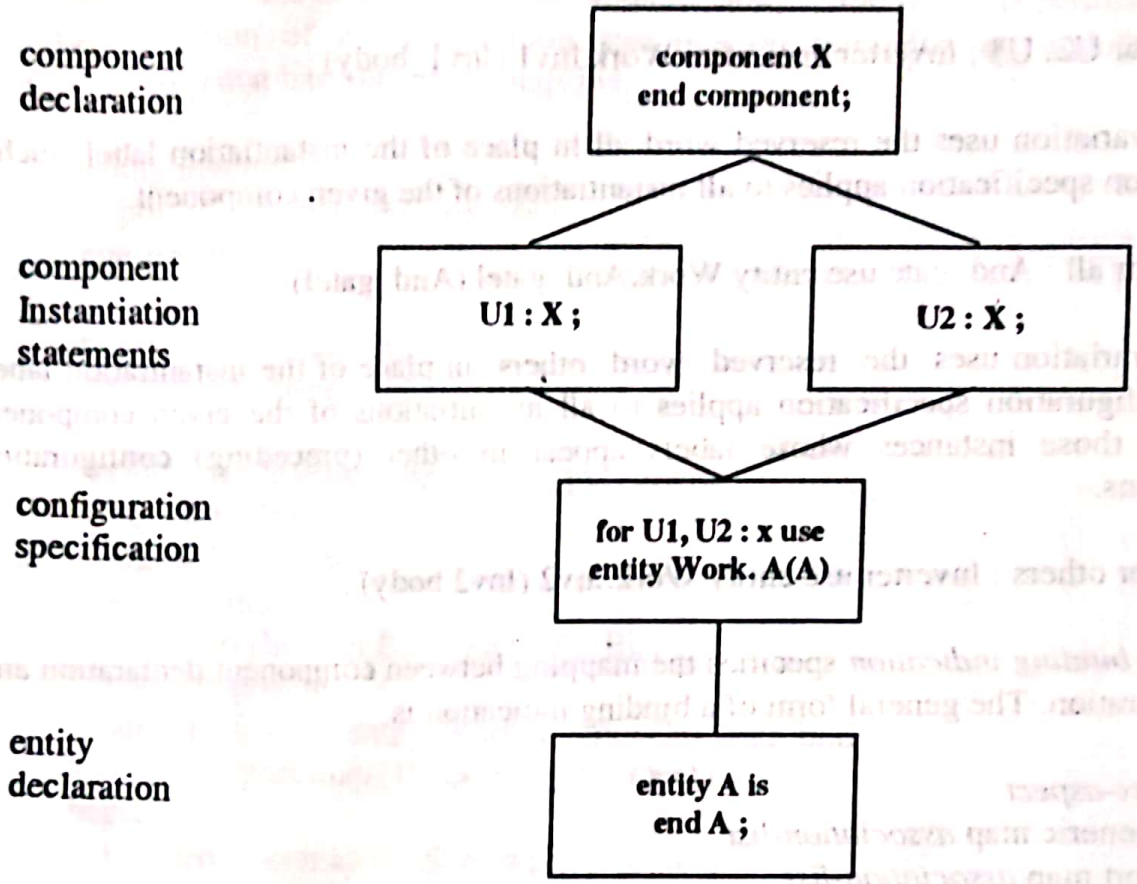



Figure 6.5. Configurations: One Component, One Entity

In addition to the entity aspect, the binding indication may also contain a generic association list and a port association list. The generic and port association lists account for differences between the generics and ports of-the component declaration and the generics and ports of the entity declaration just as the entity aspect accounts for the difference between the name of the component and the name of the entity. If there is no port association list in the binding indication, then for each local port in the component declaration there must be in the entity declaration a formal port having the same name and same type as the local port. If there is no such simple match, then an explicit port association list is required. An analogous rule governs the necessity of an explicit generic association list. The syntax of the port (generic) association list in a binding indication is identical to the syntax of the port (generic) association list in the component instantiation statement The difference is that the port or generic association list of the binding indication maps actuals to *formals* (ports or generics on an entity declaration) while the port or generic association list of a component instantiation statement maps actuals to *locals*.

Scanned with CamScanner

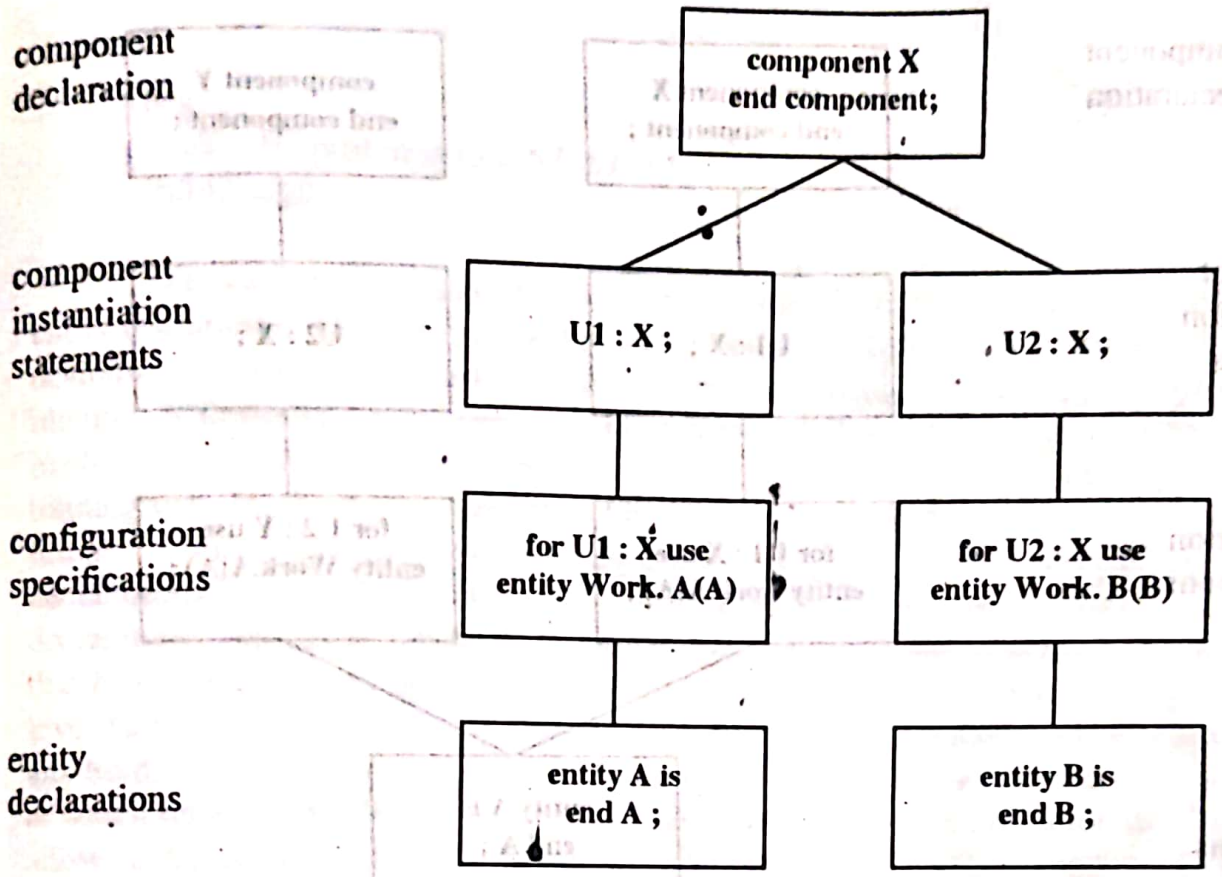



Figure 6.6. Configurations: One Component, Two Entities

The actuals in a port (or generic) association list in a binding indication are typically the local ports (or generics) of a component declaration. However, in case the entity has formals which have no corresponding locals in the component declaration, then a port (or generic) association list in a binding indication may map other signal objects to the entity formals. Consider the following example in which a signal declared in a package is associated with a formal port. In this example, the component FF is declared with only four ports. When the component instantiation statement is bound to the entity JKFF, the tending indication of the configuration specification associates the four locals of FF with four formals of JKFF and in addition associates actual signals from package Global_signals with the three additional ports of I JKFF.

```
entity JKFF is
   port (GLK, PRESET, CLEAR, J, K : Bit ;
     Q, Q_bar : out Bit ) ;
end JKFF;
```

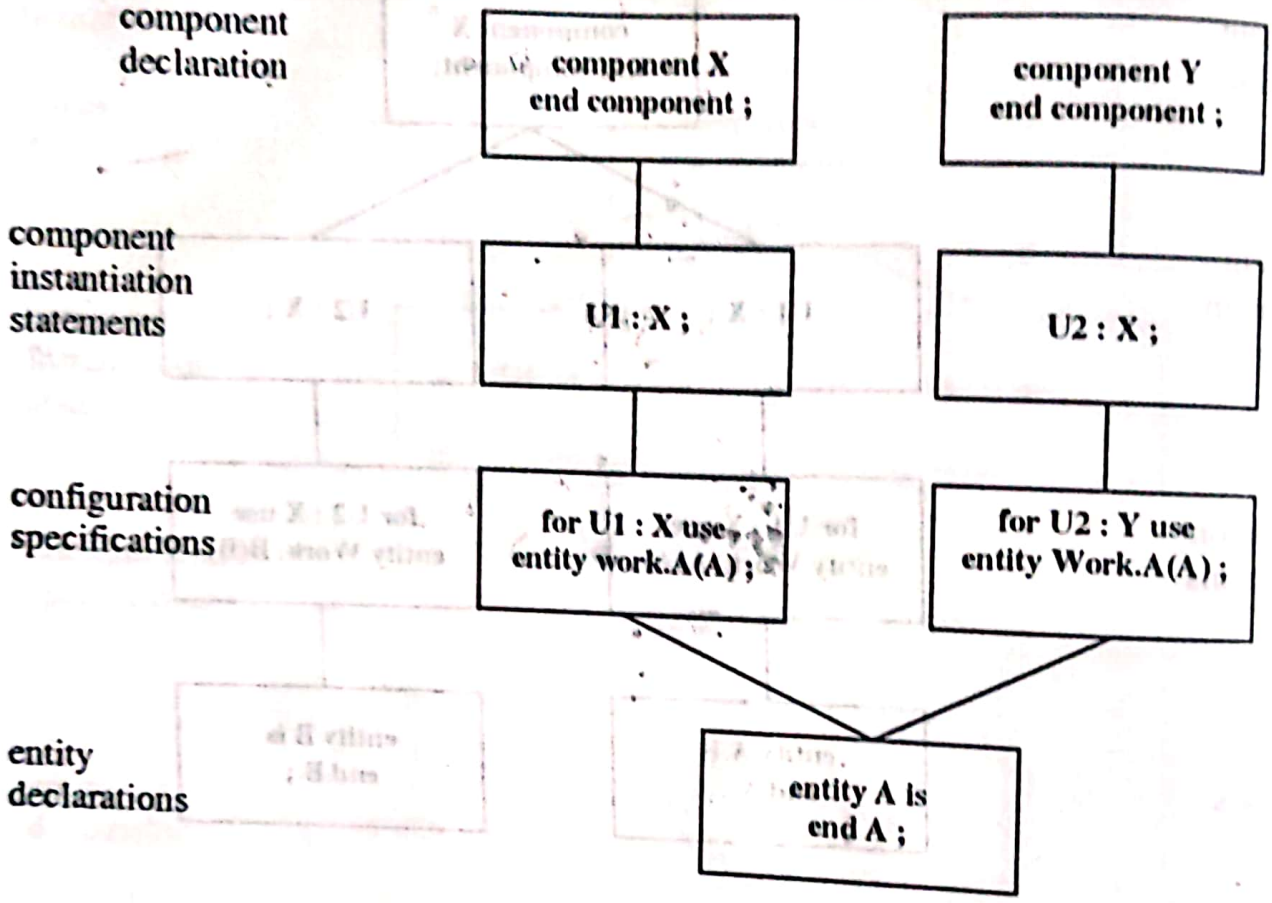



Figure 6.7. Configurations: Two Components, One Entity

```
package Global_signals is
   signal CLK : Bit ;
   signal PRESET : Bit ;
   signal CLEAR : Bit ;
end Global_signals ;

entity Design is
end Design ;

use Work.Global_signals ;
architecture Design of Design is
   signal S1, S2, S3, S4 : Bit ;
   component FF
         port (J, K : Bit Q, Q_bar : out Bit) ;
   end component;
   for U0 : use entity Work.JKFF
         port map (CLK => Global_signals.CLK,
                   PRESET => Global_signals.PRESET,
                   CLEAR => Global_signals.CLEAR,
                   J => S1, K => S2, Q => S3, Q_bar => S4) ;
```

Scanned with CamScanner

```
begin
   U0 : FF port map (S1, S2, S3, S4) ;
end Design ;
```

This section has explained the separate step of binding a component instance to an entity declaration. However, it has not attempted to explain why this separate binding is a desirable feature for a hardware description language. To understand why separate binding is desirable, it is necessary to understand that the binding need not be made explicit in the architecture in which the component instantiation occurs. Instead, the binding can occur after the architecture is written and analyzed, in an entirely separate library unit, the *configuration declaration.* Chapter 7 will discuss configuration declarations in detail; for now it is sufficient to know that a single configuration declaration; can apply bindings to component instantiations occurring in architectures that have already been analyzed into design libraries, and can thus tie together a multi-level hierarchy of design units. The designer may write the separate architectures without any bindings to entities, deferring the bindings until such time as all interfaces are known, at which time configuration declaration can be written. The configuration declaration thus allows a designer to delay specifying the exact interface of a subcomponent. This has two benefits: on the one hand, it provides support for a top-down design methodology; on the other hand, it allows a designer to take advantage of a library of reusable subcomponents.

The separate binding mechanism of VHDL is a powerful but a designer may choose not to take advantage of the flexibility i affords. For example, a designer may know exactly what existing unit he wishes to instantiate. In that case, the designer may prefer to bind the instance right in the instantiating architecture. The configuration specification is provided for such circumstances, as means by which a designer can bind an instance without having to write a separate configuration declaration.

## Default Values and Unconnected Ports

### Default Values

The driver (drivers are discussed in Chapter 5) of every signal is defined to have an implicit default value, according to the following rules:

- If the signal is of a scalar subtype, then the implicit default value of each driver is defined to be T'LEFT, where T denotes the scalar subtype of the signal.
- If the signal is of a composite subtype, then each scalars of the composite is a signal, and each driver of each of these scalar subelements is defined to have an implicit default value by the previous rule.

Drivers of ports are defined in the same way (only ports of modes out, inout, and buffer can have drivers).

It is also possible to explicitly specify a default value in the declaration of the signal or port In that case, the value of the explicit default expression overrides the implicit default value given by the above rules. The following examples illustrate sig l

and port declarations with explicit default expressions (the port declarations could occur in an entity declaration, in a component declaration, or in a block statement.

```
signal S : Bit := '1' ;

port (Enable : in Bit := '0' ;
      Sel : in Bit_vector (2 down to 0) := "111" ;
      Outputs : out Bit_vector (7 downto 0) ;= "10000000"
```

If a port of mode out, inout, or buffer does not have any sources, then a default expression on the port declaration (the port on the entity declaration, not the port on the component declaration) serves to define the initial value of the source of any actual that is connected to that port. Specifying a default expression on the declaration of a port of mode in serves to define the initial value of the port in case the port is unconnected (see the following section).

Initial values of drivers and signals are computed during the elaboration phase, just prior to the start of the execution phase of simulation. As pan of this process, the value of a port of mode out, inout or buffer is propagated "upward" to contribute, as a source, in the initial value of some root signal. The value of this signal is then propagated "downward" to become the value, for reading operations, of lower-level connections of mode in, inout or buffer. While this concise summary appears innocuous, results may sometimes not be obvious. The following example, which attempts to construct a circuit that oscillates every 10 nanoseconds by means of an inverter and buffer. The inverter and buffer were constructed to use a general three-level logic, and the ports of the inverter and buffer were declared with an default expression.

```
package Three_level_logic is
   type Three is ('X', '0', '1') ;
   function Inv (Q : Three) return Three ;
end Three_level_logic ;

package body Three_level_logic is
   function Inv (Q : Three) return Three is
   begin
      case Q is
         when 'X' =>
               return 'X' ;
         when '0' =>
               return '1' ;
         when '1' =>
               return '0' ;
      end case;
   end Inv ;
end Three_level_logic ;
```

Scanned with CamScanner

```
use Work.Three_level_logic.all ;
entity Inverter is
   port (P1: Three := 'X' ;
         P2: out Three := 'X') ;
end Inverter;

use Work.Three_level_logic.all ;
entity Buf is
   port (P1: Three := 'X' ;
         P2: out Three := 'X') ;
end Buf;

architecture Inverter of Inverter is
begin
   P2 <= Inv (P1) after 5 ns ;
end Inverter;

architecture Buf of Buf is
begin
   P2 <= P1 after 5 ns ;
end Buf;

use Work.Three_level_logic.all ;
entity Oscillator is
end Oscillator ;

architecture Oscillator of Oscillator is

   component Inverter
         port (PI : Three; P2 : out Three) ;
   end component;

   component Buf
         port (P1 : Three ; P2 : out Three) ;
   end component;

   for all : Inverter use entity Work.Inverter (Inverter) ;
   for all : Buf use entity Work.Buf (Buf) ;

   signal S1 : Three := '0' ;
   signal S2 : Three := '1' ;
begin
```

Scanned with CamScanner

```
    U1 : Inverter port map (S1, S2) ;
    U2 : Buf port map (S2, S1) ;

end Oscillator;
```

If the intention of the designer of the above circuit is for the circuit to begin oscillating at simulation startup and continue indefinitely, then the design is in error. Both signal SI and signal S2 are connected to ports of mode out; hence the initial values of SI and S2 will be taken from the values of the ports they are connected to. These ports, named P2 in each case, are declared with the default expression 'X'; hence the initial value of the driver of each P2 will be 'X'. Consequently, the initial value of the signals SI and S2 will be 'X', despite the default expressions '0' and T on the signal declarations for SI and S2. And since these values will be propagated downward, it follows that the initial value on each of the ports named P1 will be also be 'X'. But since the inversion of 'X' is 'X', the initial execution of the concurrent signal assignment statements in Inverter and Buf will not result in any change of state, and the simulation will terminate after 5 nanoseconds. A solution to this problem is change the Inv function so that the inversion of 'X' is either '0' or T. Alternatively, the explicit default expression on the out ports of the two entities could be changed to '0' or '1'.

Note that a local port of mode out, inout, or buffer may also play a role in determining the initial value of a signal with which the port has been associated. If the subtype of the local port is a resolved subtype, then the resolution function will be invoked during the upward propagation of values through the local port, and it is this resolved value that will be propagated as a source for the actual signal.

**Unconnected Ports**

It may be that a component is declared to have one or more ports that are not needed by an instantiating architecture. For example, a JK flip-flop may have been declared-with inputs CLK, Preset, CLR, J, and K and outputs Q and Q_bar, while a particular instantiation may have no need of the Q_bar output. Nevertheless, the rules of VHDL require that in a component instantiation there be an association for each local port. In such cases, the designer can always create a local dummy signal and associate this otherwise unused dummy signal with the extra port.

```
package Flip_flops is
   component port JKFF (
         CLK, PRESET, CLEAR, J, K : Bit ;
         Q, Q_bar : out Bit);
   end component ;
end Flip_flops ;

use Work.Flip_Flops ;
architecture FF_user of FF_user is
   signal CLK, Preset, CLR, In_1. In_2, Out_1 : Bit ;
```

Scanned with CamScanner

```
    signal Dummy : Bit ;
  begin
    C : Flip-flops JKFF
          port map (CLK, Preset, CLR, In_1, In_2, Out_1, Dummy) ;
  end FF_user ;
```

While this solution will simulate correctly, it is awkward. In particular, it is indistinguishable from a design error in which a left dangling. A better alternative is to leave the extra port explicitly unconnected-by associating with it the reserved word open.

```
use Work.Flip_flops ;
architecture FF_user of FF_user is
  signal CLK, Preset, CLR, In_1, In_2, Out_1 : Bit ;
begin
  C : Flip_flops.JKFF
        port map (CLK, Preset, CLR, In_1, In_2, Out_1 open) ;
end FF_user ;
```

Any port of mode out, inout, or buffer may be unconnected provided its type is no: an unconstrained array type. A port of mode in may be unconnected only if its declaration includes an explicit default expression.

The reserved word open may also be used in the port association list of a binding indication. In the following example, open is associated with the formal port Q_bar of entity JKFF.

```
package Flip_flops is
  component port JKFF (CLK, PRESET, CLEAR, J, K : Bit ;
        Q : out Bit) ;
  end component ;
end Flip_flops ;

entity JKFF is
  port JKFF (CLK, PRESET, CLEAR, J, K : Bit;
        Q, Q_bar : out Bit) ;
  end JKFF ;

use Work.Flip_flops ;
architecture FF_user of FF_user is
  signal CLK, Preset, CLR, In_1, In_2, Out_1 : Bit ;
  for C : Flip_flops.JKFF use entity Work.JKFF
        port map (CLK, PRESET, CLEAR, J, K, Q, open) ;
begin
  C : Flip_flops.JKFF
        port map (CLK, Preset, CLR, In_1, In_2, Out_1) ;
```

Scanned with CamScanner

end FF_user;

VHDL also defines a default port association list which associates each local port with a formal port of the same name (the type and mode of each local port must be the same as the type and mode of the formal port is associated with) and associates open with each formal port for which there is no corresponding local port. If, in the immediately preceding example, the port association list were omitted from the binding indication, then the default port association list would apply, with the same results as obtained with the explicit port association list.

Scanned with CamScanner