CHAPTER 14

COMPUTER AIDED DESIGN OF DIGITAL SYSTEMS

14.1 INTRODUCTION

The importance of digital techniques, circuits, and systems have been well recognized in all walks of human life. In fact the term 'digital' has now entered every sphere of our activities. This has created a need for designing and developing more and more powerful and complex digital circuits and systems. As we have seen through all the earlier chapters, any digital circuit consists of only a few basic circuits; AND, OR, and NOT gates and a memory element FLIP-FLOP, irrespective of the size and complexity of the circuit. We have also discussed the design of digital circuits using manual methods, such as; simplification of Boolean expressions using Boolean algebraic theorems, graphical method, tabular method, using available SSI and MSI devices, etc. These design (synthesis) methods or tools have served well for understanding the concepts of digital circuits and their design for systems which are relatively small in size and are not complex enough in today's context.

However, the ever increasing size and the complexity of digital systems require design methods which make use of computers, which themselves are complex digital systems. These methods are known as computer aided design (CAD) methods and a number of CAD tools have been developed for this purpose.

One may wonder how the earlier computers were designed having complex electronic circuitry without the help of CAD tools. It was a result of human ingenuity which made it possible to develop a system which has made tremendous effect on the way we think, work, and live. Since, there were no computers at that time, therefore, the question of having CAD tools did not arise. Now with the tremendous progress in semiconductor technology, design of computers and other digital systems have become a very complex process requiring CAD tools without which we can not think of designing powerful and efficient complex digital systems including computers.

CAD tools have not only made it possible to design modern complex logic circuits but have also made the design work much easier. Many tasks in the design process are performed automatically by the CAD tools resulting in faster and efficient design. However, the importance of learning the theory of digital circuits can not be minimized even when CAD tools are employed for design.

A number of hardware description languages (HDLs) have been developed for describing the structure and behaviour of complex digital circuits and a number of HDL based CAD tools have been developed for the design of digital systems. Proliferation of hardware description languages for the design of *Very High Speed Integrated Circuits* (VHSIC) led to accepting the two HDLs, VHDL (VHSIC Hardware Description Language) and Verilog, by IEEE (USA) and their IEEE standards were adopted. The first version of VHDL was IEEE standard 1076–1987 which was updated in 1993 (IEEE 1076–1993), 2000 (IEEE 1076–2000), 2002 (IEEE 1076–2002), 2006 (IEEE 1076–2006), and 2008 (IEEE 1076–2008). Verilog was adopted in 1995 as IEEE standard 1364 (Verilog–95) which was updated in 2001 (Verilog – 2001) and in 2005 (IEEE standard 1364–2005). Both the languages are used in industry. We have chosen VHDL for discussion.

Computer aided design concepts, CAD tools, and VHDL have been discussed here which will help the readers to enter into the most fascinating and useful field of digital systems design.

A number of examples have been included for describing combinational and sequential circuits using VHDL.

14.2 COMPUTER AIDED DESIGN (CAD) CONCEPTS

Figure 14.1 illustrates a basic digital design process to design a digital system that performs certain tasks and meets certain specifications. The design process begins with the specifications such as functionality and

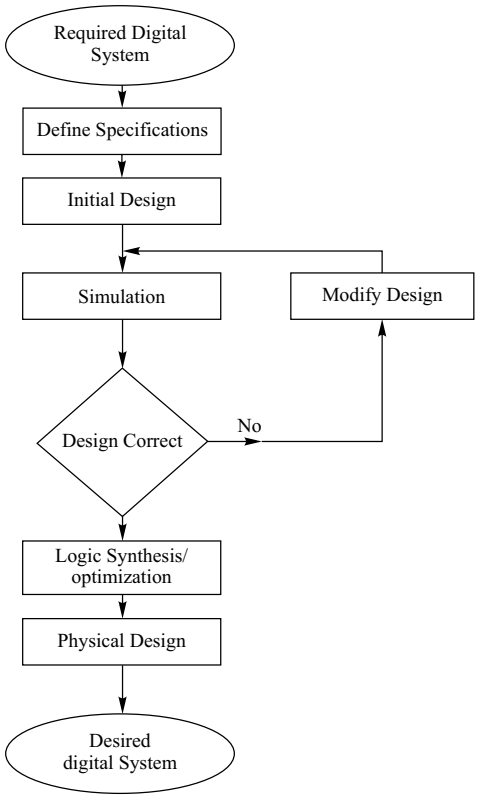

Fig. 14.1 ***Basic Digital Design Process***

input-output behaviour of the system to be designed. From a complete set of specifications, a designer with his/her considerable design experience and intuition makes an initial design by defining the general structure of the intended hardware. The initial design idea has to go through several design stages before its hardware implementation is obtained. At each stage of design process the designer evaluates and verifies the results of that design stage. For the evaluation of the results of a design stage, the design is to be tested by applying test data (stimuli) at the input and finding the output and checking this output and verifying it against the original system specifications. If there is any error then the design must be modified to eliminate the error. The modified design is again put to evaluation and verification. This process is repeated until there is no error in the designed system.

The above procedure indicates that the designed system needs to be implemented in hardware at every stage of design. This is tedious, time consuming, and costly and it may not always be possible to implement it. The development of various CAD tools have made it very convenient and easy to go through this process. CAD tools enable the designers to simulate the behaviour of a design without its hardware implementation. Various *simulation* CAD tools are available. These are used to simulate the design and are also used to determine whether the obtained design meets the required specifications or not. If not, then appropriate modifications are made in the design and the verification of the modified design is repeated through simulation till the correct design is obtained. Once the functionally correct design is obtained, logic synthesis and optimization CAD tools are used to obtain optimized logic expression to suit the target hardware technology to be used for implementing the design. Using these optimal logic expressions, physical design tool is used for obtaining gate level or transistor level design.

As the size and complexity of a digital system to be designed increases, more and more computer-aided design tools are introduced in the process.

14.3 CAD TOOLS

Various computer aided tools have been developed for aiding the designer of digital systems in performing different design tasks. These tools make the design process easy, convenient, accurate, and fast. A variety of such tools are available. In some cases a compatible set of such tools are packaged together forming a CAD system. The CAD tools for performing various steps in the design process are discussed below.

14.3.1 Design Entry

The first stage in the design of any digital system is the conception of what the intended system is required to do and the formulation of its general description. This stage requires designer's intuition and experience and is essentially a manual process. The description of the system conceived is to be entered into the CAD system and this process is known as *design entry.* The system to be designed may be described in one or more of the three methods, accordingly design entry tools are required to handle each one of these descriptions. The three design entry methods are:

- Truth table
- Schematic diagram
- Hardware description language (HDL)

Design Entry Using Truth Table

The truth table of a logic function or a timing waveforms diagram may be specified for designing the necessary hardware, using the components available, in an optimum way. A CAD system capable of transforming the truth table or the timing waveforms into a network of logic gates is required for this purpose. The CAD tool

used for drawing timing diagram and transforming it automatically into a network of logic gates is known as the *waveform editor.*

This method of design entry can be used only for a small number of variables, since truth table method of description is possible only for logic functions with a few variables. For a larger circuit involving relatively large number of variables, this method can be employed for smaller logic functions that are parts of the larger circuits. These smaller circuits can be interconnected by using another CAD tool known as *schematic capture tool*.

Design Entry Using Schematic Capture

The term *schematic* refers to a circuit diagram in which the circuit elements are represented by their graphical symbols and the connections between the circuit elements are drawn as lines. A schematic capture tool known as a *graphic editor* available in a CAD system can be used to draw a schematic diagram. It uses the graphics capabilities of a computer and allows a designer to draw a schematic diagram using a computer mouse.

The schematic tool provides a library of graphical symbols that represent gates of various types with different number of inputs. Some of the commonly required circuits using these gates can be created by a designer and these can also be represented by graphical symbols. These special symbols of the circuits can also be made available in the library and can be imported into the user's schematic. These small circuits and other sub-circuits or sub-systems can be created by the user, using either different entry methods or the schematic tool itself.

The schematic capture design entry method uses the gates and the other circuits created by the user and connects them to create the desired logic circuit. This is a hierarchical design method and is, therefore, a convenient way of dealing with the complexities of larger circuits.

Design Entry Using Hardware Description Languages

Similar to various programming languages, a number of languages have been developed to describe hardware rather than a program to be executed on a computer. The two most commonly used *hardware description languages (HDLs)* in industry are: VHDL (Very High-Speed Integrated Circuits Hardware Description Language) and *Verilog HDL*. Both these HDLs are IEEE (Institution of Electrical and Electronics Engineers, USA) standards. Using a HDL, a logic circuit is represented in its code which is used for design entry. This method of design entry is the most commonly used method for the design of digital systems. A number of CAD tools are commercially available for the design that uses HDLs. This method can be used for efficiently designing small as well as large systems.

Both the IEEE standard HDL languages, VHDL and Verilog HDL, widely used in the industry differ in many ways but they offer similar features. Since VHDL is more popular than Verilog HDL, therefore, we have chosen VHDL here, although the concepts discussed for VHDL can easily be applied when using Verilog HDL.

The three design entry methods can be mixed in a system design for different subcircuits, for example, a schematic capture tool can be used in which a subcircuit in the schematic is described using VHDL.

14.3.2 Initial Synthesis

Synthesis is the process of transforming design entry information of the circuit into a set of logic equations. By using synthesis CAD tools logic equations that describe the logic functions needed to realize the circuit are automatically generated. In general, these logic equations produced by the initial synthesis tools are not in the optimal form, because these equations are generated on the basis of designer's input to the CAD tools.

It is difficult for a designer to manually produce optimal results, especially for large circuits. *Optimization* techniques are used to obtain logic equations which are not only functionally correct but are optimized for the design goal. The process of optimization is performed after functionally correct design is obtained.

14.3.3 Functional Simulation

After the design entry and the initial synthesis are complete, it is necessary to verify the circuit function of the designed circuit with the expected function. For this purpose, a *functional simulator* CAD tool is used. The

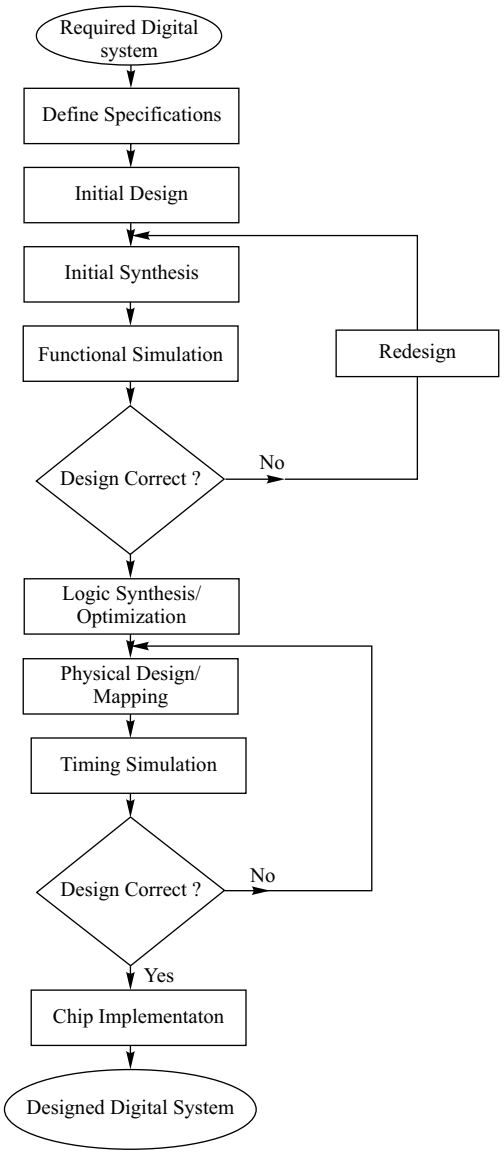

Fig. 14.2 ***Digital Design Process***

functional simulator simulates the circuit function from the logic equations obtained from the synthesis tool and the inputs which are applied. The output of the simulator, which is obtained either in the truth table form or as the timing diagram are examined to verify whether the designed circuit operates as required or not. If the output from the simulation process is not the desired output then the design is to be modified suitably. This process is repeated unless the functionally correct design is obtained as shown in Fig. 14.2.

14.3.4 Logic Synthesis and Optimisation

After the functionally correct design is obtained logic synthesis and optimisation CAD tools are used to obtain optimised design to achieve design goal such as cost, speed, or the technology of implementation. Use of logic synthesis and optimisation tools before the functionally correct design is obtained does not serve any purpose. The CAD synthesis tools automatically implement the synthesis techniques. In case the logic synthesis tool is *technology independent*, the optimisation is independent of the resources available in the target chip, whereas the *technology mapping* synthesis tool ensures that the circuit designed by it uses the logic resources available in the target chip. For example, if the target chip is a CPLD, then each logic function in the circuit is expressed in terms of the gates available in its macrocells. Similarly, for an FPGA target chip the number of inputs to each logic function is constrained by the size of the look-up tables (LUTs) or type of logic cells available as the case may be.

14.3.5 Physical Design

After the logic expressions are optimized, the next step is to design the circuit using the available logic resources in the target chip. This step is known as *physical design*, or *layout synthesis.* The physical design consists of two operations: *placement* of logic functions in the optimised circuits in the target chip (CPLD or FPGA) and interconnecting the components in the chip, referred to as *routing*. The *placement* and *routing* CAD tools are used for this purpose.

14.3.6 Timing Simulation

The functional simulation process assumes negligible propagation delay time of the logic gates, which is not true for practical circuits. There is always some time needed for signals to propagate through logic gates, therefore, it is necessary to take care of propagation delays while designing digital circuits. For this purpose timing simulation is used, which simulates the actual propagation delays in the technology chosen for implementation of the design. The model to be used for timing simulation must take care of delays associated with the chip, macrocells if the target device is a CPLD and logic cells for an FPGA target device, and the delays through the interconnection wires. *Timing simulator* CAD tools are used for this.

14.3.7 Summary

A typical CAD system for digital design comprises tools for performing the following tasks:

- *Design entry* It allows a designer to enter a description of the desired circuit/system in the form of truth tables, schematic diagrams, or HDL codes such as VHDL.
- *Initial Synthesis* It generates logic expressions based on data entered during the design entry stage.
- *Functional Simulation* It is used to simulate and verify the functionality of the circuit, using the inputs (stimuli) provided by the designer.
- *Logic Synthesis and optimisation* It is used to design optimised circuit.
- *Physical design* It is used to implement the optimised circuit in a given technology, for example, in a CPLD chip or in an FPGA chip.

- *Timing Simulation* It is used to include the effect of propagation delays that are expected in the target technology and ensures that the designed circuit meets the required performance.
- *Chip configuration* It configures the actual chip to realize the designed circuit.

The design process is illustrated in Fig. 14.2.

14.4 INTRODUCTION TO VHDL

VHDL is a Hardware Description Language used for modelling digital systems made of interconnection of components. The complexity of the digital system being modeled may vary from that of a simple gate to a complete electronic circuit/system. VHDL is an acronym for VHSIC Hardware Description Language and VHSIC is an acronym for Very High Speed Integrated Circuits.

VHDL is an industry standard language used to describe hardware from the abstract to the concrete level and has become the universal communication medium of design and for specifiying input and output from various design tools, such as simulation tools, synthesis tools, placement and routing tools, etc. from vendors of CAD work stations, ASICs, CPLDs, and FPGAs etc.

VHDL contains elements which can be used to describe the *behaviour* (*concurrent* or *sequential*) or structure, with or without timing, of any digital system irrespective of the complexity of the sytem. Since it is the most widely used hardware description language, therefore, it is a commonly used method of documenting circuits by the designers, thus making it possible to understand circuits designed by other designers.

For computer aided design of digital systems, VHDL is a very convenient and commonly used HDL for design entry. It supports hierarchical modeling of the system and top-down and bottom-up methodologies of design. Models written can be verified using a VHDL simulator.

When a digital circuit is to be designed, it is required to be described in VHDL. For this we must specify an *entity* and an *architecture* at the top-level, and also specify an entity and architecture for each of the

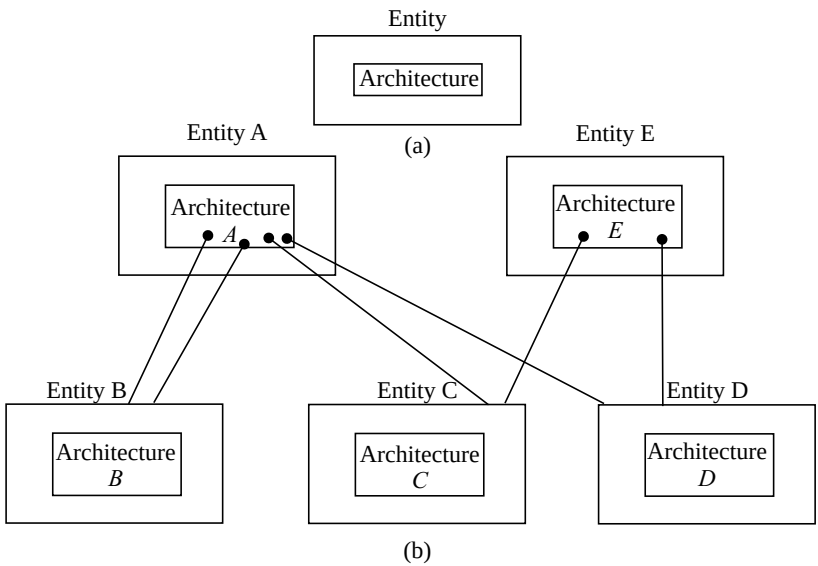

Fig. 14.3 ***(a) Basic Concept of VHDL (b) Use of Lower-level Entities by Higher-level Architectures***

component modules that are part of the circuit. Each entity declaration includes a list of interface signals and their types that can be used to connect it to other modules or to the outside world. The behaviour of the circuit and each of its components is described in the architecture declaration. The behaviour may be described in structural form, i.e. interconnection of components, or as a set of concurrent or sequential statements.

Figure 14.3(*a*) illustrates the basic concept of VHDL. It has one entity declaration and an architecture. In general, a digital system or circuit may be composed of a number of sub-systems or components. Every sub-system or component can be described by its VHDL, i.e., each sub-system or component has its own entity declaration and architecture. The architecture of the digital system or circuit can make use of the entities of its sub-systems or components. This makes hierarchical system design possible. Figure 14.3(*b*) shows a top-level architecture *A* making use of entities *B*, *C*, and *D*. The entity *B* is used two times. In general, a top-level architecture can make multi use of lower-level entity. An entity can also be used by any other architectures. In Fig. 14.3(*b*), the entities *C* and *D* are used by some other architecture *E* also. The architectural details of the lower-level entities are not visible to the higher-level architectures.

A VHDL source code file has two main sections: an entity and an architecture. Similar to any other programming language, VHDL has some rules and characteristics. Some of these are given below:

- VHDL is strongly-typed language. It has some predefined types. Every signal, variable, and constant in a VHDL program must match with the allowed type in the program. The type specifies the set or range of values that the object can take on. There is also a set of operators (boolean and integer) such as `AND`, `OR`, `NAND`, `NOR`, `XOR`, `XNOR`, `NOT`, addition, subtraction, multiplication, and division, etc.
- There are a number of reserved words (or keywords) which have specific meanings and these can not be used for identifiers or as any other name. These are given in Appendix-A1.
- VHDL is a free-form language. Carriage returns, blank lines, and additional blank spaces may be included between words for clarity without any ill effects.
- Long statements can continue over more than one line.
- A line starting with two adjacent hyphens (– –) is treated as comment.
- Concluding semicolon (;) is syntactically required. Its absence will cause error message during the compilation of the code.
- TIME is a predefined type, any of the following time units are available in VHDL:

  fs – femtosecond
  ps – picosecond
  μs – microsecond
  ms – millisecond
  s – second
  min – minute
  hr – hour

  ns is built in VHDL. It need not be specified.
- VHDL allows arrays to be indexed in either direction (ascending or descending) because both conventions are prevalent in hardware.
- VHDL is a data flow language, unlike procedural computing languages such as *C* and assembly code, which run sequentially (one instruction at a time).
- A basic identifier in VHDL is composed of a sequence of one or more alphanumeric characters and underscore (_) character. The first character must be an alphabet, and the last character must not be an underscore. Two consecutive underscores are not allowed.
- VHDL is case insensitive, i.e. upper and lower case letters are considered identical.
- Boolean operators `AND`, `OR`, `NOT`, `NAND`, `NOR`, `XOR`, and `XNOR` are built in VHDL.

- A *system library* ieee library is provided which stores packages for standard logic gates and other commonly used functions. A *user library* can be created by the user.
- Special notation symbols and syntax provided in VHDL are given in Appendix-A2.

14.4.1 Entity

The term *design entity* or just *entity* in VHDL refers to any digital device that possesses some form of intercommunication characteristic. It is a hardware abstraction of an actual digital hardware device. The device to be modeled (entity) may be a single gate, a central processsing unit (CPU), board containing discrete components, or even a complete digital system.

An entity may be decomposed into its constituent lower level entity, or subcomponents or alternatively it may be treated as a building block to construct a high level entity. This means an entity X can be used as a component of another entity Z, or an entity W can be decomposed into its lower level entities P, Q, R, . . . etc. For example, an EX-OR gate shown in Fig. 14.4*a* is an entity which can be decomposed into its lower level enitites AND, OR, and NOT gates as shown in Fig. 14.4*b*, or conversely an EX-OR entity can be considered as composed of AND, OR, and NOT entities.

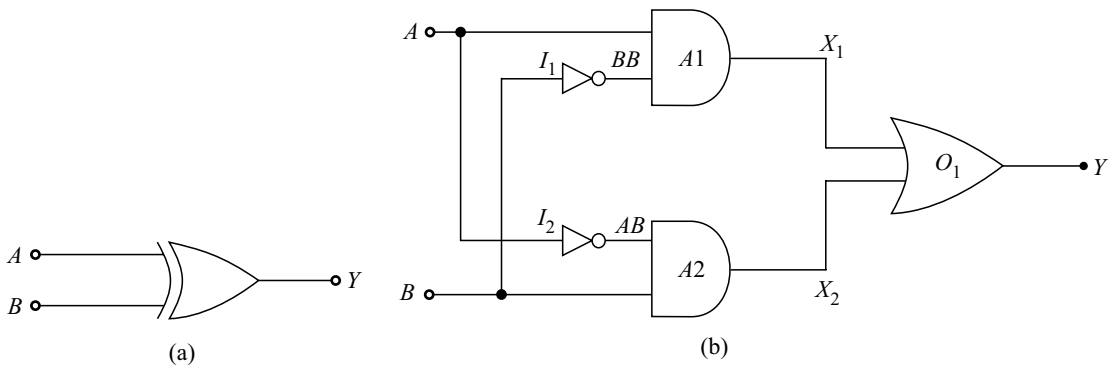

Fig. 14.4 ***(a) An*** EX-OR ***Gate (b) An*** EX-OR ***Function using*** AND, OR***, and*** NOT ***Gates***

In VHDL, all designs are created using entities. A design must have atleast one entity. An entity is the most basic building block in a digital design and since an entity can be decomposed into its lower level entities or can be treated as a building block for higher level entity, therefore, VHDL can support both top-down and bottom-up design methodologies. This methodology is referred to as *hierarchical design* and it is very useful in dealing with the complex circuits.

The term ENTITY is a key-word in VHDL and is a construct used in VHDL code. The keyword ENTITY signifies the start of an entity statement. Each entity is uniquely declared describing its external communication links to the outside world. It specifies the number of ports, the directions of the ports, and the type of ports. Some more information such as timing can be put into the entity.

An entity is assigned a name and the corresponding construct declares its input and output signals, known as *ports*. A port is indentified by the keyword PORT. Each port has an associated *mode* that specifies whether it is an input port (keyword IN), an output port (keyword OUT) or a bidirectional port (keyword INOUT) of the entity. Since each port represents a signal, hence, it has an associated *type*.

The syntax of an entity declaration is:

ENTITY *entity-name* IS

PORT (List of input, output port names and their types);
END *entity-name;*

The words: ENTITY, IS, PORT, IN, OUT, and END used above are the *reserved* words or *keywords* in VHDL. These words have specific meanings for the VHDL compiler, and they can not be used for any other purpose.

An *entity-name* is composed of a string of one or more alphanumeric characters and the underscore (_) character. The first character of a name must be an alphabet (upper-or-lower case) and the last character must not be an underscore. Also a name can not have two successive underscores and it must not be a VHDL reserved word. VHDL is not case sensitive, which means upper and lower case letters can be freely used, for example, SET_CK_HIGH, Select_Signal, ROM_address are all valid names. Entry name should preferrably be assigned as a word meaningfully related to the function performed by the entity so as to make it convenient for the designer for indentification.

The list of input, output port names and their types describes the input and output signals of the entity and their types. For example, for the circuit of Fig. 14.3*a*, it will appear as

```
PORT   (A, B : IN BIT;
          Y : OUT BIT);
```

A and *B* are the input ports (or signals), and *Y* is an output port (or signal). The signal names are user-selected identifiers and are separated by comma. The input and output signals are of the type BIT which can assume only one of the two binary digits 0 or 1.

The direction of the port can be input (IN), output (OUT), or bidirectional (INOUT) and is referred to as *mode.*

The reserved word END signifies the end of the ENTITY declaration. The symbols colon(:) and semicolon(;) are used as separator and terminator respectively.

From the entry declaration illustrated above, it is clear that an entity declaration describes only the input and output signals of the entity. It does not give any details about the behaviour of the entity or the way the circuit components are connected, i.e. the structure of the circuit. In fact the entity declaration does not and cannot model a device's behaviour. It is merely an outer shell without any functional core.

Example 14.1

Write entity declaration constructs in VHDL for the AND, OR, NOT, and EX-OR gates shown in Fig. 14.5.

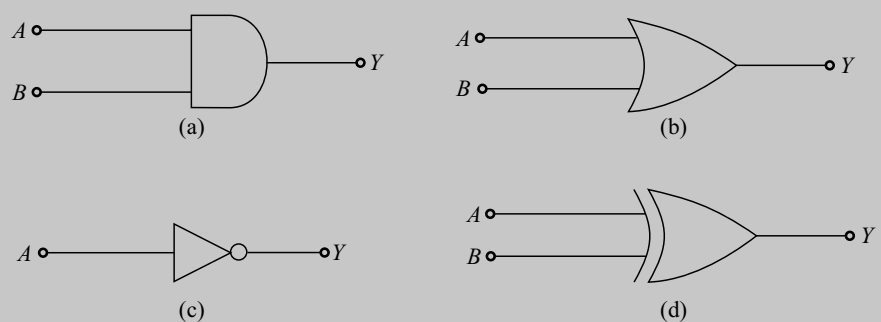

Fig. 14.5 ***(a)*** AND ***(b)*** OR ***(c)*** NOT ***(d)*** EX-OR ***Gates***

Solution

(a) For AND gate

```
ENTITY and2 IS
PORT (A, B: IN BIT;
Y: OUT BIT);
END and2;
```

Here, the name of the entity is chosen as *and2* (a 2-input AND gate), *A* and *B* are the input ports, and *Y* is the output port. The type of signal is BIT.

(b) For OR gate

```
ENTITY or2 IS
PORT (A, B: IN BIT;
Y: OUT BIT);
END or2;
```

(c) For NOT gate

```
ENTITY not IS
PORT (A: IN BIT; Y: OUT BIT);
END not;
```

(d) For EX-OR gate

```
ENTITY xor2 IS
PORT (A, B: IN BIT; Y: OUT BIT);
END xor2;
```

Example 14.2

Write entity construct for the EX-OR circuit of Fig. 14.4*b*.

Solution

Let the name of the entity be Circuit_Fig. It has two input ports *A* and *B* and one output port *Y*. The entity declaration for this circuit will be

```
ENTITY Circuit_Fig IS
PORT (A, B: IN BIT; Y: OUT BIT);
END Circuit_Fig;
```

From this entity declaration, we observe that although this circuit consists of AND, OR, and NOT gates, the circuit itself is an entity and the entity declaration gives no information about the structure or behaviour of the circuit.

Example 14.3

Write entity construct for the R-S FLIP-FLOP circuit shown in Fig. 14.6.

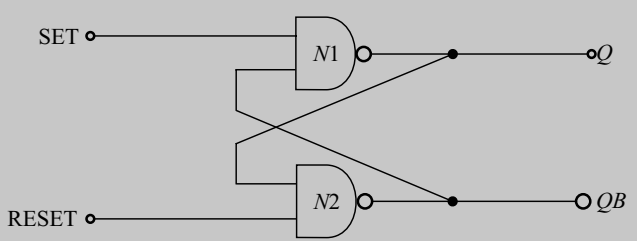

Fig. 14.6 ***R-S*** FLIP-FLOP

Solution

Let the name of the entity be rsff. It has two input ports SET and RESET and two bidirectional ports *Q* and *QB*. The entity construct will be

```
ENTITY rsff IS
PORT (SET, RESET: IN BIT;
Q, QB: INOUT BIT);
END rsff;
```

14.4.2 Architecture

A VHDL design entity is modeled using an entity declaration and its atleast one architecture body. The entity declaration describes the external view of the entity, whereas the architecture body contains the internal description of the entity. The internal description can be specified in the following ways:

- A set of interconnected components that represent the structure of the entity.
- A set of concurrent statements that represent the behaviour of the entity.
- A set of sequential statements that represent the behaviour of the entity.

Each of the above methods of representation can be specified in a different architecture body or mixed within a single architecture body. An architecture body is composed of two parts: *declarative part* and *statement part.* The syntax of architecture body is:

```
ARCHITECTURE architecture-name OF entity-name IS
[declarative part]
BEGIN
[statement part]
END architecture-name
```

An architecture must be given a name consisting of a text string which should be assigned by a designer in a way meaningful to the design.

The words ARCHITECTURE, OF, BEGIN are keywords in VHDL.

The declarative part appears before the keyword BEGIN. It can be used to declare signals, user-defined types, constants, components, function and procedure definitions. The signals and other declarations in an architecture are local to that architecture only. Declarations common to multiple entities can be made in a separate 'package' used by all entities. The 'package' will be discussed later.

The statement part of the architecture is contained between the keywords BEGIN and END. All the statements are *concurrent* statements, which means, these are executed concurrently (simultaneously) and not sequentially as in the case of any programming language.

Structural Modeling

In structural modeling, an entity is described as a set of interconnected components in the architecture body. Let us consider the EX-OR gate shown in Fig. 14.4(*b*) and its entity declaration given in part (d) of Example 14.1. This circuit has three types of components, two 2-input AND gates, one 2-input OR gate, and two NOT gates. Its architecture body is given below:

```
ARCHITECTURE XOR_STRUCTURE OF xor2 IS
COMPONENT and2                          - - Entity and2 has X, Y inputs
```

```
PORT (X, Y: IN BIT; Z: OUT BIT);          - - and Z output.
END COMPONENT;
COMPONENT or2                             - - Entity or2 has P, Q inputs
PORT (P, Q: IN BIT; R: OUT BIT);          - - and R output.
END COMPONENT;
COMPONENT not                             - - Entity not has I input
PORT (I: IN BIT; J: OUT BIT);             - - and J output.
END COMPONENT;
SIGNAL AB, BB, X1, X2: BIT;
BEGIN
I1: not PORT MAP (B, BB);
I2: not PORT MAP (A, AB);
A1: and2 PORT MAP (A, BB, X1);
A2: and2 PORT MAP (AB, B, X2);
O1: or2 PORT MAP (X1, X2, Y);
END XOR_STRUCTURE;
```

In this, the architecture body has been named as *XOR_STRUCTURE* and it is associated with the entity declaration with the name *xor2* and therefore it inherits the list of interface ports from that entity declaration. For each type of component, *component declaration* is required. The component declarations are shown in the declarative part of the architecture body. The architecture body also contains a signal declaration that declares four signals *AB*, *BB,* X_1, and X_2, of type `BIT`. These signals which represent wires, are used to connect the various components that form the `EX-OR` circuit. The scope of these signals is restricted to the architecture body and are not visible outside the architecture body. These signals are referred to as *local signals* or *buried nodes*, and are neither inputs nor outputs of the entity of the architecture. The component declaration lists its name, mode and type of each of its ports.

For each component, *component instantiation* statement is required in the statement part. This requires port map, i.e. input and output ports to be specified. A component instantiation statement is a concurrent statement, therefore, these statements can appear in any order. The structural modeling describes only interconnection of components, without specifying behaviour of the components or the entity they collectively represent.

The component instantiation statements include a label, such as *I*1, *I*2, *A*1, *A*2 . . . etc.; component name, such as and2, or2, not . . . etc.; and association between the actual signals that are visible in the architecture body and the ports of the component being instantiated. The mapping of ports specifies the association between the ports of a component with the ports in the architecture body, which may consist of external input, output ports and internal ports declared through signal statement. For example, *I*1 is a `NOT` gate with port *B* as input and *BB* signal as output, similarly *I*2, *A*1, *A*2, and *O*1 are instantiated. The signals in the `PORT MAP` list are in the same order in which they appear in entity definition of the component.

Example 14.4

Write architecture body of R-S `FLIP-FLOP` shown in Fig. 14.6 using structural modeling. Assume entity *nand*2 with *A*, *B* inputs and *Y* output.

Solution

It consists of two 2-input NAND gates. Let us use the name of the architecture also rsff which is same as the entity name. It is allowed for the architecture body to have same name as entity. The architecture body will be

```
ARCHITECTURE rsff OF rsff IS
COMPONENT nand2
PORT (A, B : IN BIT; Y : OUT BIT);
END COMPONENT;
BEGIN
N1: nand2
PORT MAP (SET, QB, Q);
N2: nand2
PORT MAP (RESET, Q, QB);
END rsff;
```

In this example, there are no signals to be declared.

Data Flow Modeling

In this modeling, the flow of data through the entity is expressed using concurrent signal assignment statements. As the name implies, the statements contained in the model assign values to signals. These statements execute concurrently, i.e. in parallel, not serially, as in the case of a programming language. The structure of the entity is not explicitly specified in this modeling, but it can be implicitly deduced.

A signal assignment statement is of the form:

$$A <= B; \tag{14.1}$$

It means A gets the value of B, i.e. the current value of signal B is assigned to signal A. This statement is executed whenever signal B changes value. Signal B is in the sensitivity list of this statement. A signal assignment statement is executed whenever a signal in its sensitivity list changes value. A transaction is generated when a signal assignment statement is executed. The target signal may or may not change following the execution of the signal assignment statement. If a new value of target signal is generated, then an event is scheduled for the target signal.

Time delay can also be introduced in the signal assignment statements as given below:

$$A <= B \text{ AFTER } 10 \text{ ns}; \tag{14.2}$$

Here, signal B is in the sensitivity list of this signal assignment and according to this statement signal A gets the value of signal B after a lapse of 10 ns. AFTER is a VHDL keyword. The data flow modelling uses simple assignment statements involving logic or arithmetic expressions.

Example 14.5

Write the architecture body of each of the entities of Example 14.1 using concurrent signal assignments, i.e. the data flow model. Assume 10 ns delay.

Solution

(a) For AND gate

```
ARCHITECTURE df_ and2 OF and2 IS
```

```
    BEGIN
    Y <= A AND B AFTER 10ns;
    END df_and2;
```

(b) For OR gate

```
    ARCHITECTURE df_or2 OF or2 IS
    BEGIN
    Y <= A OR B  AFTER  10ns;
    END df_or2;
```

(c) For NOT gate

```
    ARCHITECTURE INV OF not IS
    BEGIN
    Y <= NOT A AFTER 10ns;
    END INV;
```

(d) For EX-OR gate

```
    ARCHITECTURE  df_xor OF xor2 IS
    BEGIN
    Y <= (A AND (NOT B)) OR (B AND (NOT A));
    END df_xor2;
```

Since VHDL does not assume any precedence of logic operators, therefore, parentheses must be used in the expression to avoid compile-time error for the expression.

Example 14.6

Write architecture body of *R-S* FLIP-FLOP shown in Fig. 14.6 using data flow model. Assume 5 ns delay time.

Solution

```
ARCHITECTURE df_rsff OF rsff IS
BEGIN
Q <= NOT (QB AND SET) AFTER 5 ns;
QB <= NOT (Q AND RESET) AFTER 5 ns;
END df_rsff;
```

In Examples 14.5 and 14.6, AND, OR, NOT, and XOR have been used which are in-built boolean operators in VHDL.

There are two other types of concurrent signal assignment statements. These are:

- Conditional signal assignment, and
- Selected signal assignment.

Conditional Signal Assignment

A conditional signal assignment allows a signal to be one of several values based on certain conditions to be satisfied. It uses keywords, WHEN and ELSE, boolean operators such as AND, OR, NOT, and XOR, and relational operators = (equality), /= (inequality), > (greater than), >= (greater than or equal to), and <= (less than or equal).

Example 14.7

For a 2 : 1 multiplexer shown in Fig. 14.7, write entity declaration and conditional signal assignment data flow architecture body.

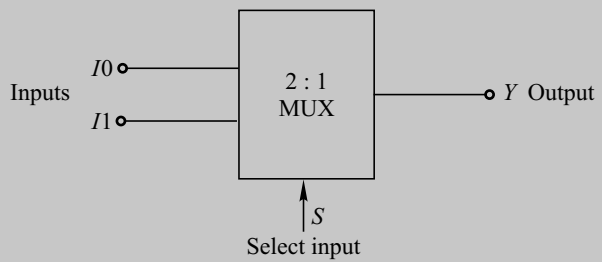

Fig. 14.7 ***A 2 : 1 Multiplexer***

Solution

We shall make use of IEEE library which includes STD_LOGIC type. STD_LOGIC type is a standard data type for representation of logic signals in VHDL.

```
LIBRARY ieee;
USE ieee. std_logic_1164.all;
ENTTY MUX_FIG 13_7 IS
   PORT   (I0, I1, S : IN STD_LOGIC;
                Y : OUT STD_LOGIC);
END MUX_FIG 13_7;
ARCHIRECTURE DFA OF MUX_FIG 13_7 IS
BEGIN
   Y <= I0 WHEN S = '0' ELSE I1;
END DFA;
```

The conditional signal assignment specifies that Y is assigned the value of $I0$ where $S = 0$, or else Y is assigned the value of $I1$.

Selected Signal Assignment

A selected signal assignment allows a signal to be assigned one of several values, based on a selection criterion. The selected signal assignment begins with the keyword `WITH`, specifies the selection criterion and then `SELECT` keyword.

Example 14.8

Repeat Example 14.7 with selected signal assignment data flow architecture body.

Solution

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY MUX_FIG 13_7 IS
```

```
   PORT    (I0, I1, S : IN STD_LOGIC;
                 Y : OUT STD_LOGIC);
END MUX_FIG 13_7;
ARCHITECTURE DFA OF MUX_FIG 13_7 IS
BEGIN
  WITH S SELECT
  Y <=  I0 WHEN '0',
        I1 WHEN OTHERS;
END DFA;
```

Here, the Keyword OTHERS is used for taking into consideration all the other possible values of other than 0. In STD_LOGIC data type, *S* can take on values 0, 1, Z (high-impedance state), and —(don't care).

Behavioural Modeling

In this type of modeling, the behaviour of an entity is expressed using statements which are executed sequentially similar to that of a high-level programming language. A process statement is the main mechanism used to model the behaviour of an entity. The functionality of an entity is described in an algorithmic representation in the process statement.

The process statement starts with keyword PROCESS and ends with the keyword END PROCESS. All the statements between the keywords PROCESS and END PROCESS are considered part of the process statement. A VHDL PROCESS statement can be used anywhere alongwith concurrent statements. Its execution is in parallel with other concurrent statements and other processes.

Statements in a PROCESS are sequentially, evaluated. Any assignments made to the signals inside the process are not visible outside the process until all of the statements in the process have been evaluated. In case there are multiple assignments to the same signal, only the last one will be visible outside. Neither conditional nor selected signal assignments are allowed within a process.

The process statement consists of three parts: senitivity list, declarative part, and statement part.

Sensitivity List

A process statement is always active and executes at all times if not suspended. Following PROCESS keyword, the sensitivity list (signals) in parentheses is specified. The sensitivity list includes all input signals that are used inside the PROCESS. For example, for the 2 : 1 multiplexer of Fig. 14.7, the sensitivity list will have *I*0, *I*1, *S* signals. The process is activated when an event occurs on any one of these signals. When the program flow reaches the last sequential statement, the process becomes suspended until another event occurs on a signal that it is sensitive to. Regardless of the events on the sensitivity list signals, processes are executed once at the beginning of the simulation run.

Declarative Part

The declarative part is used to declare local variables or constants that can be used only inside the process. i.e. such objects are visible only to the process within which they are declared. Signals and constants declared in the declarative part of an architecture that encloses a process statement can be used inside a process. Such signals are the only means of communication between different processes.

Initialization of objects declared in a process is done only once at the beginning of a simulation run. The initializations in a subprogram are performed each time the subprogram is called.

The process declarative part consists of the area between the sensitivity list and the keyword BEGIN.

Statement Part

The statement part of the process consists of the area between the keywords BEGIN and END PROCESS. All the statements are sequential and are executed one after the other in a sequential order.

The statement part of a process is always active. When the program flow reaches the last sequential statement of this part, the execution returns to the first statement in the statement part and continues.

Example 14.9

Write architecture body of *R-S* FLIP–FLOP shown in Fig. 14.6 using behavioural model. Assume 10 ns delay time.

Solution

```
ARCHITECTURE behave_rsff OF rsff IS
BEGIN
PROCESS (SET, RESET)
BEGIN
IF SET = '1' AND RESET = '0' THEN
Q <= '0' AFTER 10 ns;
QB <= '1' AFTER  10 ns;
ELSIF SET = '0' AND RESET = '1' THEN
Q <= '1' AFTER 10 ns;
QB <= '0' AFTER  10 ns;
ELSIF SET = '0' AND RESET = '0' THEN
Q <= '1' AFTER 10 ns;
QB <= '1' AFTER 10 ns;
END IF;
END PROCESS;
END behave_rsff;
```

Let us examine the execution of this architecture when SET changes to a '0' and RESET remains at a '1'. Since, SET is in the sensitivity list of the process statement, the process is invoked and each statement in the process is executed sequentially. The first statement is an IF statement. This statement yields a negative result because SET = '0' and RESET = '1', therefore, the following two signal assignments are not executed. Then the next check is performed which succeeds and therefore, the next two signal assignments are executed.

Architecture Selection

An entity can have more than one architecture. Three different architectures have been described above. One or more than one (mixed) architecture styles can be used in a single architecture body. It depends on the designer as to which architecture should be used to model the entity. If the model is going to be used to drive a layout tool, then the structural architecture will probably be most appropriate. If a structural model is not wanted, then a more efficient model, could be data flow or sequential, from the point of view of memory space required and speed of execution. Depending upon the preference of the designer for concurrent or sequential code, one of these architectures can be selected.

14.4.3 Configuration Declaration

An entity can have multiple architectures. Which of these architecture bodies is to be used in a simulation? The VHDL language has two options available for this purpose. In one option, a simulation run will, by default, use an architecture body that was most recently compiled. Use of default option in not a preferred way because of the requirement of continual recompilation of architecture bodies in order to ensure their inclusion in a simulation and for difficulty in future references. In the other option we can explicitly pronounce as to which one of the architectures is to be simulated. This is done via a configuration statement. The reserved word `CONFIGURATION` announces a user's intention to create a configuration declaration. The binding of component instantiations to design entities is performed by configuration specifications. For example, we consider a configuration statement using structural architecture of the *rsff* entity (Ex. 14.4). The configuration statement is given below.

```
CONFIGURATION rsff_str OF rsff IS
FOR rsff
FOR N1, N2: nand2 USE ENTITY WORK. mynand
(Version 1);
END FOR;
END rsff_str;
```

Here, *rsff_str* is name given to configuration. Following the reserved word `OF` name of the design entity (*rsff*) that we are configuring is written. For the top most entity, architecture *rsff* is used. The two components *N*1 and *N*2 are instantiated in the *rsff* architecture. The architecture of 2-input `NAND` gate is used from library called `WORK` with entity `mynand` and architecture `version 1`. All of the entities now have architectures specified for them.

This binds the architecture body *rsff* to the design entity *rsff*. The first `END` terminates the pronounced binding and the last `END` terminates the configuration declaration. The configuration name is optional with the last `END` reserved word.

A simulation model can be created by compiling the entities, architectures, and the configuration. In case we want to use another architecture for simulation, we need not recompile the complete design, only the new configuration needs to be recompiled.

14.4.4 Package

A *package* is a collection of codes of commonly used data type definitions, declaration of signals, and components. The packages are the mechanisms that allow sharing of definitions of data types, signals, and components (sub-circuits) among design entities which access the package. A package is stored in a computer file system at a location specified by its name. Definitions and declarations provided in the package can be used in any source code file by including the statements

```
LIBRARY library_name,
USE library_name.package_name.ALL
```

The library_name represents the location in the computer file system where the package is stored. A library may either be provided as a part of a CAD system, which is referred to as a *system library*, or may be created by the user, which is referred to as a *user library*. The package is specified by a package_name which may be in a system library or a user library.

In Examples 14.7 and 14.8, we have used the first two statements, in which ieee is the name of the library which is a system library and std_logic_1164 is a package which defines data types. The clause USE is used for accessing the package and ALL is used for having access to the entire package.

A signal declared in a package can be used by any entity that accesses the package. Such signals are global signals, whereas a signal declared in an architecture can only be used inside architecture. Such signals are referred to as local signal.

14.4.5 Generic

It is often necessary to pass certain information to an entity in VHDL. For instance, if an entity is a gate level model with a rise and a fall delay, values for the rise and fall delays could be passed into the entity with generics.

Consider a positive-edge triggered D-type FLIP-FLOP with asynchronous set and reset inputs shown in Fig. 14.8.

Its entity declaration using generics will be

```
ENTITY D_FLIP-FLOP IS
GENERIC (sq_delay, rq_delay, cq_delay: TIME: = 5 ns);
PORT (D, SET, RESET, clk: IN BIT; Q, QB: OUT BIT);
END D_FLIP-FLOP;
```

Here, sq_delay is delay from set to *Q*, rq_delay is delay from reset to *Q*, and cq_delay is delay from clock to *Q*. The delay time is 5 ns. The delay parameters are passed on to the entity through the generic.

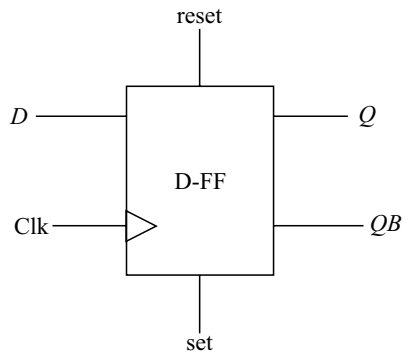

Fig. 14.8 ***Positive-Edge Triggered D-FF***

14.4.6 Data Objects

In VHDL code, information is represented as signals, variables, and constants. These are referred to as *data objects*. The data objects are given names. The rules for the name are the same as the rules for the identifiers. Signals basically represent wires in a circuit. VHDL variables are similar to signals, except that they do not have physical significance in a circuit. Variables are used in VHDL functions, procedures, and processes. All assignments to variables occur immediately, whereas in the case of signals, their values are assigned only when an event to which they are sensitive occurs. Constant objects are names assigned to specific values of a type. It helps whenever a designer modifies the design with a different value of the constant.

Declaration of Data Objects

A signal is declared as:

```
SIGNAL signal_name: type_name;
```

The rules for the signal names are same as the rules for the identifiers (see section 14.4). The signal type specifies the legal values that the signal can have. The various types are discussed below.

VHDL Signal Types

The most commonly used signal types are:

- BIT

- BIT_VECTOR
- STD_LOGIC
- STD_LOGIC_VECTOR
- INTEGER
- ENUMERATION
- BOOLEAN

BIT Type BIT type is a predefined signal type in VHDL. Objects of BIT Type can have the values '0' or '1'. It is declared as

```
SIGNAL A : BIT;
```

It can be used to specify only an individual single bit signal.

BIT_VECTOR Type BIT_VECTOR type is also a predefined type. It is used to declare a group of elements of the same type together as a single object, such as a byte. It is declared as

```
SIGNAL byte : BIT_VECTOR (7 DOWN TO 0);
```

Here, the bits of the byte are specified as 7 DOWN TO 0. The number 7 refers to the MSB (left-most) and 0 refers to the LSB (right-most). The assignment statement can be

```
byte <= "11001001";
```

and its individual bits will be

byte (7) = 1, byte (6) = 1, byte (5) = 0, … byte (1) = 0, byte (0) = 1.

A multibit data object can also be declared as

```
SIGNAL A: BIT_VECTOR (1 TO 4);
```

Here, 1 is for the left-most bit and 4 is for the right-most bit. An assignment statement can be

```
A <= "1011";
```

It results in $A(1) = 1$, $A(2) = 0$, $A(3) = 1$ and $A(4)$.

STD_LOGIC Type STD_LOGIC type of data object was added to VHDL standard in IEEE standard 1164. Use of this type was introduced in Examples 14.7 and 14.8. For using this type, the following two statements are to be included

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

This type of data object is an extension of BIT type, and it can have values 0, 1, Z, and —(don't care).

A signal declaration statement can be

```
SIGNAL A, B, C : STD_LOGIC;
```

STD_LOGIC_VECTOR Type The STD_LOGIC_VECTOR type is an extension of BIT_VECTOR type. It represents an array of STD_LOGIC objects.

A signal declaration statement can be

```
SIGNAL A, B, C : STD_LOGIC_VECTOR (3 DOWN TO 0)
```

Use of STD_LOGIC_VECTOR type also requires access to STD_LOGIC_1164 package.

INTEGER Type It is similar to mathematical integer. Mathematical functions, such as add, subtract, divide, and multiply are applicable to integer types. An INTEGER signal represents a binary number of 32 bits and therefore, it can represent numbers from $-(2^{31}-1)$ to $+(2^{31}-1)$. It is also possible to declare integer with fewer bits using the RANGE keyword.

A signal declaration statement can be

```
SIGNAL B : INTEGER RANGE – 127 TO 127;
```

This declaration defines *B* as an eight-bit signed number. Type INTEGER is predefined in VHDL.

ENUMERATION Type An *enumeration type* of signal can have user specified possible values of the signal. Its type declaration has the value-list separated by commas. For example,

```
TYPE traffic-light-state IS (reset, stop, wait, go);
```

BOOLEAN Type An object of type BOOLEAN has two values, true and false, where true is equivalent to 1 and false is 0. In VHDL, BOOLEAN type is a predefined type. An example of declaration of BOOLEAN type is

```
SIGNAL lock_out : BOOLEAN;
```

VARIABLE Data Objects

A variable is declared as:

```
VARIABLE variable-names: variable-type;
```

Variables declared within a process have their scope limited to that process. Variables declared outside a process can be shared by more than one processes. An object of variable type can hold a single value of a given type. A variable can be assigned different values at different time using a variable assignment statement. The assignment operator used for variables is : =. Examples of variable declarations and assignment are given below:

```
VARIABLE temp : INTEGER;
temp : = 0;
```

Here, a variable named 'temp' is declared as an integer and its present assigned value is 0.

```
VARIABLE sum: INTEGER RANGE 0 TO 100 : = 10;
```

In this statement, a variable 'sum' is declared as an integer with value in the range 0 to 100. Its present value is assigned as 10.

```
VARIABLE  found, done: BOOLEAN;
```

In this statement, the variables 'found' and 'done' are declared as boolean type. They can have a value 'true' or 'false'

14.4.7 CASE Statement

A CASE statement is similar to a selected signal assignment. The case statement has a selection signal (or expression) and WHEN clauses for various valuations of the selection signal. Its general form is shown below:

```
CASE expression IS
```

```
        WHEN constant_value =>
            statement;
        WHEN constant_value =>
            statement;
        WHEN OTHERS =>
            statement;
    END CASE;
```

The WHEN clause is followed by the => symbol, the statement will be evaluated for the specified constant_value. There must be a WHEN clause for all the possible valuations of the selection signal.

14.5 DESCRIBING COMBINATIONAL CIRCUITS USING VHDL

Any combinational circuit can be described using VHDL. It may be available in the truth table form or in the logic circuit diagram form. We shall discuss a variety of combinational circuits using different VHDL statements and architectures.

14.5.1 Describing Truth Table in VHDL

Consider the truth table of a 3 input, single output combinational circuit given in Table 5.3. The inputs are *A*, *B*, *C* and the output is *Y* its ENTITY declaration is given below:

```
ENTITY Fig. 5_3 IS
PORT  (A, B, C: IN BIT;
            Y : OUT BIT);
END Fig. 5_3;
```

For creating its architecture body, we need to put the three inputs *A*, *B*, *C* in the form of an array because otherwise it will not be possible to identify which input should be considered as MSB and which input will be LSB. For this purpose, we make use of a VHDL operator and & (concatenation) for connecting the bit variables to form a bit vector. Using the and operator, we can create a signal of bit vector i.e,

comb_cir <= *A* & *B* & *C*

Using this approach, we prepare architecture body as given below:

```
ARCHITECTURE truth_table OF Fig. 5_3 IS
SIGNAL comb_cir : BIT_VECTOR (2 DOWN TO 0);
BEGIN
  comb_cir <= A & B & C;       - - input bits are put in order WITH comb_cir SELECT
        Y <= '0' WHEN "000",
             '1' WHEN "001",
             '1' WHEN "010",
             '0' WHEN " 011",
             '1' WHEN "100",
             '0' WHEN "101",
             '0' WHEN "110",
             '1' WHEN "111";
END truth_table;
```

The data flow modeling with selected signal assignment has been used. Here, we observe that the operands for the concatenation operator are the elements *A*, *B*, and *C* and its result is an array *ABC*. In general, the operands can be either an element type or a one-dimensional array type and the result is always an array type.

For Example,

(i) '0' & '1' results in an array of characters "01",
(ii) '*C*' & '*A*' & '*T*' results in the value of "*CAT*".
(iii) "*BA*" & "*LL*" arrays create an array of characters "*BALL*".

It is useful when many separate signals are to be grouped into one bus.

14.5.2 Arithmetic Circuits in VHDL

Arithmetic circuits discussed in Section 5.8 can be described in VHDL.

Example 14.10

Write VHDL code for the circuit of Fig. 5.19 using

(a) Structural architecture
(b) Data flow architecture. Assume gate delay of 5 ns for AND and 10 ns for XOR.

Solution

(a) The VHDL code is given below:

```
– – We make use of IEEE. STD_LOGIC_1164. ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY H_A IS
PORT (A, B: IN STD_LOGIC; S, C: OUT  STD_LOGIC);
END H_A;
- - H_A is name of the half-adder circuit.
ARCHITECTURE  HA_STR OF H_A IS
- - HA_STR is name of the architecture for entity H_A.
COMPONENT XOR2
PORT (X, Y: IN  STD_LOGIC; Z: OUT STD_LOGIC);
END COMPONENT;
COMPONENT AND2
PORT (P, Q: IN STD_LOGIC; R: OUT STD_LOGIC);
END COMPONENT;
BEGIN
X1: XOR2 PORT MAP (A, B, S);
A1: AND2 PORT MAP (A, B, C);
END HA_STR;
```

(b) The entity construct will be same as that of (a). The VHDL code is given below:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164. ALL;
ENTITY H_A IS
PORT (A, B: IN STD_LOGIC; S, C: OUT STD_LOGIC);
END H_A;
```

```
ARCHITECTURE df_HA OF H_A IS
BEGIN
S <= A XOR B AFTER 10 ns;
C <= A AND B AFTER 5 ns;
END df_HA;
```

Example 14.11

Repeat Example 14.10 for data flow architecture with selected signal assignment.

Solution

The ENTITY will be same as that given in Example 14.10. Its architecture is given below:

```
ARCHITECTURE dfss_HA OF H_A IS
SIGNAL ha_X : BIT_VECTOR (1 DOWN TO 0);
SIGNAL ha_Y : BIT_VECTOR (1 DOWN TO 0);
BEGIN
      ha_X <= A & B;   – – input bits AB
      ha_Y <= C & S;   – – output bits CS
WITH ha_X SELECT
      ha_Y <= "00" WHEN "00",
              "01" WHEN "01",
              "01" WHEN "10",
              "10" WHEN "11";
END dfss_HA;
```

14.5.3 Decoders in VHDL

Consider a 2-to-4 decoder circuit shown in Fig. 14.9. It has two data inputs *A*, *B* and four data output *Z*(0) to *Z*(3). *EN* is an enable input. Its VHDL code using different modeling styles are given in the Examples 14.12 to 14.14.

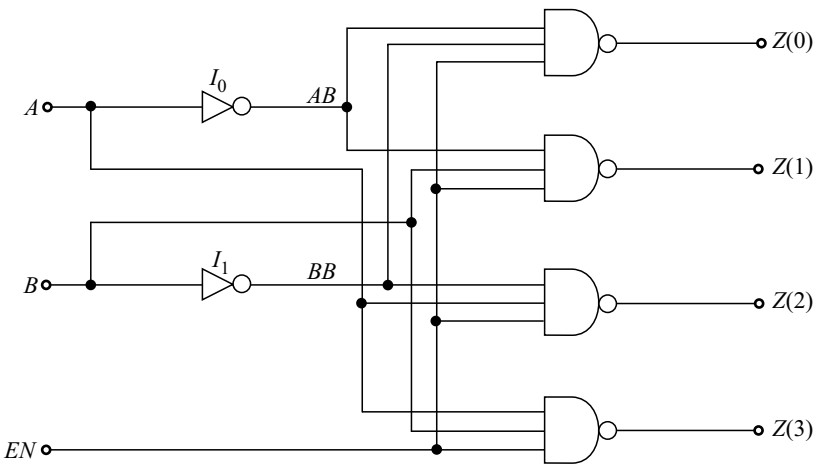

Fig. 14.9 ***2-to-4 Decoder Circuit***

Example 14.12

Write VHDL code for a 2-to-4 decoder circuit shown in Fig. 14.9. Use sequential architecture model.

Solution

```
– – Let the entity name be DECODER 2 to 4
– – Three input ports A, B, and EN
– – Four output ports Z(0), Z(1), Z(2), and Z(3)
– – Signal type is STD_logic for input ports and
– – STD_LOGIC_VECTOR for output ports
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY DECODER2to4 IS
PORT (A, B, EN: IN STD_LOGIC; Z: OUT STD_LOGIC_VECTOR (0 TO 3));
END DECODER2to4;
ARCHITECTURE behave_DEC OF DECODER2to4 IS
BEGIN
PROCESS (A, B, EN)
VARIABLE AB, BB: STD_LOGIC;
BEGIN
AB := NOT A;
BB := NOT B;
IF EN = '1' THEN
Z(3) <= NOT (A AND  B);
Z(2) <= NOT (A AND BB);
Z(1) <= NOT (AB AND B);
Z(0) <= NOT (AB AND BB);
ELSE
Z <= "1111";
END IF;
END PROCESS;
END behave_DEC;
```

The signals *A, B*, and *EN* shown in parenthesis after the keyword process is the sensitivity list. *AB* and *BB* are declared as variables, starting with the key word variable. A variable is different from a signal in that it is always assigned (assignment operator :=) a value instantaneously. Whereas a signal is assigned (assignment operator <=) a value always after a certain delay.

Example 14.13

Write architecture body for the 2-to-4 decode circuit of Fig. 14.9 using selected signal assignment. Use the entity declaration of Example 14.12.

Solution

```
ARCHITECTURE behave_DEC2 of DECODER2to4 IS
SIGNAL ENAB : STD_LOGIC_VECTOR (2 DOWN TO 0);
BEGIN
```

```
        ENAB <= EN & A & B;
WITH ENAB SELECT
            Z <= "1110" WHEN "100",
                 "1101" WHEN "101",
                 "1011" WHEN "110",
                 "0111" WHEN "111",
                 "1111" WHEN OTHERS;
END behave_DEC2;
```

Here, the three inputs *EN*, *A*, and *B* are arranged in *ENAB* order by the signal statement.

Example 14.14

Repeat Example 14.12 using a CASE statement.

Solution

```
ARCHITECTURE behave_DEC3 of DECODER2to4 IS
SIGNAL AB : STD_LOGIC_VECTOR (1 DOWN TO 0);
BEGIN
AB <= A & B;
PROCESS (AB, EN)
BEGIN
IF EN = '1' THEN
CASE AB IS
WHEN "00" =>
Z <= "1110";
WHEN "01" =>
Z <= "1101";
WHEN "10" =>
Z <= "1011";
WHEN OTHERS =>
Z <= "0111";
END CASE
ELSE Z <= "1111";
END IF;
END PROCESS;
END behaviour DECODER 2 to 4;
```

The last WHEN clause uses the key word OTHERS, which indicates the last option for the select signal.

14.5.4 Multiplexers in VHDL

The 2 : 1 multiplexer shown in Fig. 14.7 has been described in VHDL in Examples 14.7 and 14.8. In Example 14.7, conditional signal assignment has been used, whereas in Example 14.8, selected signal assignment has been used. In both the cases, the statements are executed concurrently, i.e, the order in which the statements appear in VHDL code does not affect the meaning of the code.

We can also describe the same circuit using sequential assignment statements. An example of this type of description is given below.

Example 14.15

Write VHDL code for the multiplexer circuit of Fig. 14.7 using sequential architecture style.

Solution

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;
ENTITY MUX_FIG14_7 IS
PORT   (I0, I1, S  : IN STD_LOGIC;
                Y  : OUT STD_LOGIC);
END MUX_FIG14_7;
ARCHITECTURE SA OF MUX_FIG14_7 IS
BEGIN
PROCESS (I0, I1, S)
BEGIN
IF  S = '0' THEN
    Y <= I0;
ELSE
  Y <= I1;
END IF;
END PROCESS;
END SA;
```

14.5.5 Priority Encoder in VHDL

A decimal-to-BCD priority encoder is given in Fig. 6.34. Its truth table is given in Table 6.18. It has nine inputs and four outputs. The inputs and the outputs are active-low. When none of the inputs is active, its BCD output is 1111. Let its inputs be designated as I_1, I_2, … corresponding to 1, 2, 3, …, 9 respectively, and the outputs be Y_3, Y_2, Y_1, Y_0 corresponding to D, C, B, A respectively.

Its VHDL descriptions in various forms are given in the following examples.

Example 14.16

Write VHDL code for the priority encoder of Fig. 6.34 using conditional signal assignment.

Solution

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY prien IS
PORT   I : IN STD_LOGIC_VECTOR (9 DOWN TO 1);
       Y : OUT STD_LOGIC_VECTOR (3 DOWN TO 0);
END prien;
ARCHITECTURE behave1 OF prien IS
BEGIN
      Y <= "0110" WHEN I(9) = '0' ELSE
           "0111" WHEN I(8) = '0' ELSE
           "1000" WHEN I(7) = '0' ELSE
           "1001" WHEN I(6) = '0' ELSE
```

```
            "1010" WHEN I(5) =  '0' ELSE
            "1011" WHEN I(4) = '0' ELSE
            "1100" WHEN I(3) = '0' ELSE
            "1101" WHEN I(2) = '0' ELSE
            "1110" WHEN I(1) = '0' ELSE
            "1111";
END behave1;
```

In the above example, conditional signal assignment is used with `WHEN` clause. Let us examine how this architecture results in priority encoding. When the switch 9 is pressed, *I*(9) is zero and corresponding to this, the output *Y* becomes 0110 which is decimal 9. If *I*(9) is not zero, then *I*(8) is examined and the execution goes on. In case *I*(9) is zero, then the following `ELSE` keywords do not affect the value of *Y*. In this way, as soon as a pressed key is detected, the other `ELSE` clauses will not be examined. In this way, the function of the priority encoder is achieved. Writing its architecture using selected signal assignment will be quite involved and therefore will not be discussed.

Example 14.17

Write VHDL code for the priority encoder of Fig. 6.34 using sequential signal assignment.

Solution

Its `ENTITY` declaration will be same as that given in Example 14.16. The architecture body is given below, which uses `IF-THEN-ELSE` statement.

```
ARCHITECTURE behave2 OF prien IS
BEGIN
PROCESS (I)
BEGIN
IF      I(9) =  '0' THEN
           Y<= "0110";
ELSIF   I(8) =  '0' THEN
           Y<= "0111";
ELSIF   I(7) =  '0' THEN
           Y<= "1000";
ELSIF   I(6) =  '0' THEN
           Y<= "1001";
ELSIF   I(5) =  '0' THEN
           Y<= "1010";
ELSIF   I(4) =  '0' THEN
           Y<= "1011";
ELSIF   I(3) =  '0' THEN
           Y<=  "1100";
ELSIF   I(2) =  '0' THEN
           Y<= "1101";
ELSIF   I(1) =  '0' THEN
           Y<= '1110';
ELSE
           Y<= "1111";
```

```
END IF;
END PROCESS;
END behave2;
```

Here, the desired priority scheme is described using an IF–THEN–ELSE statement. According to this, if the input $I(9)$ is 0, then the output becomes 0110. This is independent of the other inputs. The other clauses in the IF–THEN–ELSE statement are evaluated only if $I(9) = 1$. The first ELSIF clause states that if $I(8) = 0$, them $Y = 0111$ (decimal 8). If $I(8) = 1$, then the next ELSIF clause is evaluated and the process goes on until a zero input is detected. In case no key is pressed, then the output Y is 1111.

14.5.6 Digital Comparators in VHDL

Table 6.8 gives truth table of a 2-bit comparator. A and B are 2-bit numbers. Let the outputs be named as $A > B$ (*AgtB*), $A = B$ (*AeqB*), and $A < B$ (*AltB*). The VHDL code for this circuit is given in Example 14.18.

Example 14.18

Write VHDL code for the 2-bit comparator circuit.

Solution

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
– – A and B are unsigned numbers.
ENTITY Comparator IS
PORT    (A, B : IN STD_LOGIC_VECTOR (1 DOWN TO 0)
         AgtB, AeqB, AltB : OUT STD_LOGIC);
END comparator;
ARCHITECTURE behave OF comparator IS
BEGIN
        Agt B <= '1' WHEN A > B ELSE '0';
        Aeq B <= '1' WHEN A = B ELSE '0';
        Alt B <= '1' WHEN A < B ELSE '0';
END behave;
```

Here, IEEE.STD_LOGIC_unsigned package is used, because we are dealing with unsigned arithmetic operation. The three conditional signal assignments are used for the three output conditions.

14.5.7 BCD-to-7-Segment Decoder in VHDL

Truth table of a BCD-to-7-segment decoder given in Table 5.15 can be described in VHDL. Here, there are four BCD inputs A, B, C, D, and seven outputs for the 7 segments a, b, c, d, e, f, g. Let us name them as BCD for 4-bit input and LED7 for 7-bit output. Let us assume the 7 outputs to be don't care (–) for any BCD input other than the numerals 0 to 9. VHDL code foe BCD-to-7-segment decoder is given in Example 14.19.

Example 14.19

Write VHDL code for the BCD-to-7-segment decoder of Table 5.15.

Solution

```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;
ENTITY DEC_7_SEG IS
PORT    (BCD : IN STD_LOGIC_VECTOR (3 DOWN TO 0);
         LED7 : OUT STD_LOGIC_VECTOR (1 TO 7));
END DEC_7_SEG;
ARCHITECTURE EX14_19 OF DEC_7_SEG IS
BEGIN
PROCESS (BCD)
BEGIN
CASE BCD IS
    WHEN    "0000"  => LED7  <=  "1111110";
    WHEN    "0001"  => LED7  <=  "0110000";
    WHEN    "0010"  => LED7  <=  "1101101";
    WHEN    "0011"  => LED7  <=  "1111001";
    WHEN    "0100"  => LED7  <=  "0110011";
    WHEN    "0101"  => LED7  <=  "1011011";
    WHEN    "0110"  => LED7  <=  "0011111";
    WHEN    "0111"  => LED7  <=  "1110000";
    WHEN    "1000"  => LED7  <=  "1111111";
    WHEN    "1001"  => LED7  <=  "1110011";
    WHEN    OTHERS  => LED7  <=  "-------";
END CASE;
END PROCESS;
END EX14_19;
```

The last WHEN clause uses the key word OTHERS, which take care of any other combination of inputs. The outputs corresponding to other inputs are all don't cares.

14.5.8 Tristate Buffers in VHDL

A tristate buffer or a set of N tristate buffers forming a bus can be described using VHDL.

Example 14.20

Write VHDL code for a tristate buffer show in Fig. 14.10.

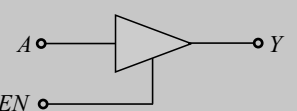

Fig. 14.10 ***A Tristate Buffer***

Solution

The data input and output of the buffer are *A* and *Y* respectively and *EN* is the enable input. Its VHDL code can be within as:

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.ALL;
ENTITY tribuf IS
PORT    (A, EN : IN STD_LOGIC;
              Y : OUT STD_LOGIC);
END tribuf;
ARCHITECTURE behave_tri OF tribuf IS
BEGIN
```

```
    Y <= 'Z' WHEN EN = '0' ELSE
      <= A;
END behave_tri;
```

In this, Y is assigned Hi-Z state when $EN = 0$, otherwise it is assigned the value of A.

Example 14.21

Write VHDL code for an 8-bit tristate buffer circuit shown in Fig. 14.11

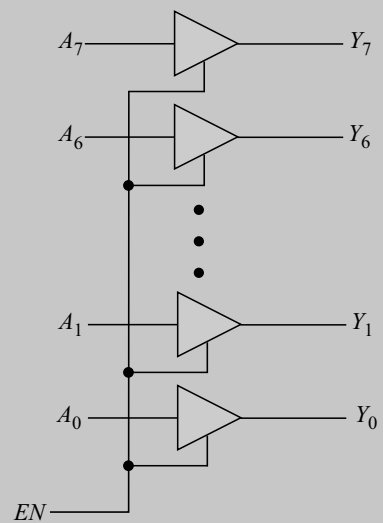

Fig. 14.11 ***An 8-bit Tristate Buffer***

Solution

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL,
ENTITY tribuf8 IS
PORT    (A  : IN STD_LOGIC_VECTOR (7 DOWN TO 0);
         EN : IN STD_LOGIC;
         Y  : OUT STD_LOGIC_VECTOR (7 DOWN TO 0));
END tribuf8;
ARCHITECTURE behave_tribuf OF tribuf8 IS
BEGIN
  Y <= (OTHERS => 'Z') WHEN EN = '0' ELSE A;
END behave-tribuf;
```

Here, the syntax `OTHERS => 'Z'` is used to specify that the output of each buffer is set to Z (Hi-Z state) if EN = 0, otherwise the output Y of each buffer is set to its input A.

Example 14.22

Write VHDL code of an 8-bit buffer shown in Fig. 14.11 using `GENERIC`.

Solution

Here, the number of buffers *N* can be set by using the Generic (see sec. 14.4.5).

```
LIBRARY IEEE; USE IEEE. STD_LOGIC_1164.ALL;
ENTITY tribuf8 IS
GENERIC (N : INTEGER : = 8);
PORT    (A  : IN STD_LOGIC_VECTOR (N-1 DOWN TO 0)
         EN : IN STD_LOGIC;
         Y  : OUT STD_LOGIC_VECTOR (N-1 DOWN TO 0));
END tribuf8;
```

The ARCHITECTURE body will be same as that of Example 14.21

14.6 DESCRIBING SEQUENTIAL CIRCUITS USING VHDL

The combinational circuits can be described in VHDL using either concurrent or sequential assignment statements, whereas the sequential circuits can be described using only sequential assignment statement. Most of the sequential circuits that are modeled using VHDL are clocked synchronous circuits using edge-triggered D-type FLIP–FLOPs. To describe the edge-triggered behaviour, either the EVENT attribute or WAIT statement is normally used.

14.6.1 EVENT Attribute

The EVENT attribute when attached to a signal name, such as *Clock*, yields a value of type boolean, that is 'true' or 'false' depending upon the occurrence of the event or not respectively. It is expressed as *Clock' EVENT*. Its value is 'true' when a change occurs in the value of *clock* and 'false' when there is no change. The *Clock*' *EVENT* AND *Clock* = '1' statement describe LOW-to-HIGH transition of the clock signal when a change occurs in the value of *Clock*, since the value of *Clock* is now 1. This statement is used to describe the positive edge-triggered FLIP–FLOPs.

14.6.2 WAIT Statement

A WAIT statement is used to suspend the sequential execution of a process or subprogram. The suspended process or subprogram can be resumed by specifying the WAIT statement in the following three ways.

- WAIT ON signal changes
- WAIT UNTIL an expression is true
- WAIT FOR a specific amount of time

The WAIT FOR statement is used for suspending the execution of the process for the time specified either directly or by an expression. The execution continues on the statement following the WAIT statement after the specified time. For example,

```
WAIT FOR 10 ns;
```

suspends execution for 10 ns, after which the execution continues with the statement following the WAIT statement. Here, the wait time is specified directly. If it is specified by an expression such as (*A* + *B* * *C*); then the time is calculated by evaluating the expression.

The WAIT ON statement is used for suspending the execution of the process for the time specified in terms of an event occurring on a signal, specified in the WAIT statement. For example, in a clocked FLIP–FLOP with an asynchronous reset input WAIT ON reset, clock; statement causes the process to suspend execution

until an event occurs on either reset or clock input. The WAIT ON clause can be used in clocked synchronous circuits description.

The WAIT UNTIL statement suspends execution of the process until the expression specified in the statement returns a value true. For example, for a clocked FLIP-FLOP, the WAIT UNTIL statement given below causes resumption of execution of the process when a rising edge occurs on the clock input.

WAIT UNTIL *Clock* = '1' AND *Clock*' *EVENT*;

This statement is often used for describing the clocked synchronous sequential circuits. When WAIT UNTIL clause is used, the sensitivity list is omitted because it implicitly specifies that the sensitivity list consists of only the clock signal.

14.6.3 Latches and FLIP-FLOPs in VHDL

Latches and FLIP-FLOPs can be described in VHDL using sequential assignment statements. The D-type of FLIP-FLOPs are the most commonly used FLIP-FLOPs in digital design using various types of programmable devices. In the case of clocked (or gated) circuits, edge-triggering is normally used.

Example 14.23

Write VHDL code for a gated *D* latch shown in Fig. 14.12.

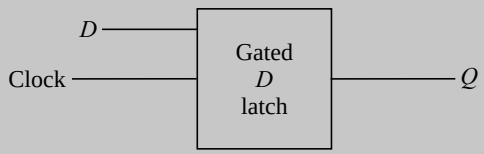

Fig. 14.12 ***A Gated D Latch***

Solution

It has two inputs *D* and Clock and *Q* is the output. Its clock input is active-high.

```
LIBRARY IEEE; USE STD_LOGIC_1164.ALL;
ENTITY latch IS
PORT   (D, Clock : IN STD_LOGIC;
                Q : OUT STD_LOGIC);
END latch;
ARCHITECTURE behave-DL OF latch IS
BEGIN
PROCESS (D, Clock)
BEGIN
IF  Clock  =  '1' THEN
        Q <= D;
END IF;
END PROCESS;
END behave-DL;
```

Example 14.24

Write VHDL code for a positive edge-triggered D-type FLIP-FLOP shown in Fig. 14.13.

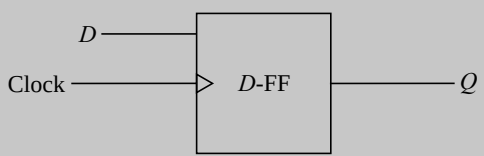

Fig. 14.13 ***A Positive Edge-Triggered D-Type*** FLIP-FLOP

Solution

The entity declaration of this is similar to the entity declaration of a gated *D* latch. Let the name of the entity be *ET-DFF*. We shall write its architecture in two different ways.

Architecture I

```
ARCHITECTURE behave_DFF OF ET_DFF IS
BEGIN
PROCESS (Clock)   -- Q <= D only if clock is active
BEGIN
IF Clock' EVENT AND Clock = '1' THEN
   Q <= D;
END IF;
END PROCESS;
END behave_DFF;
```

Here, *Q* is assigned the value of *D* input when positive edge of the clock occurs.

Architecture II

```
ARCHITECTURE behave_DFF OF ET_DFF IS
BEGIN
PROCESS   -- clock is implicitly included in the list
BEGIN
WAIT UNTIL Clock' EVENT AND Clock = '1';
   Q <= D ;
END PROCESS;
END behave_DFF;
```

Here, the WAIT UNTIL statement has been used.

Example 14.25

Write VHDL code for a positive edge-triggered *D*-type FLIP-FLOP with an active-high *Clear* input.

Solution

In this FLIP-FLOP, there are three inputs, *D*, *Clock*, and *Clear*. When *Clear* input is 1, the output *Q* will be 0 irrespective of the values of *D* and *Clock* inputs in case of asynchronous clear. In case of synchronous clear, *Clock* = 1 will be effective only at the rising edge of the clock. The VHDL entity declaration will be same in both the cases, but their ARCHITECTURE will be different.

```
LIBRARY IEEE; USE STD_LOGIC_1164.ALL;
ENTITY ACD_FF IS
PORT      (D, Clock, Clear : IN STD_LOGIC;
                        Q : OUT STD_LOGIC);
END ACD_FF;
```

Asynchronous clear

```
ARCHITECTURE behave_ACDFF OF ACD_FF IS
BEGIN
PROCESS (Clear, Clock)
```

```
BEGIN
IF   Clear = '1' THEN
        Q <= '0';       -- when Clear =1, then Q = 0
ELSIF Clock' EVENT AND Clock = '1' THEN
  Q <= D;
END IF;
END PROCESS;
END behave_ACDFF;
Synchronous Clear
ARCHITECTURE behave_SCDFF OF ACD_FF IS
BEGIN
PROCESS
BEGIN
WAIT UNIL Clock' EVENT AND Clock = '1';
IF Clear = '1' THEN Q <= '0'
-- At the positive edge of the clock, if Clear = 1, then Q = 0
ELSE
  Q <= D;
END IF;
END PROCESS;
END behave_SCDFF;
```

It is possible to make use of the WAIT ON clause also for an asynchronous clear D-type FLIP-FLOP. The PROCESS statement for this is given below:

```
PROCESS
BEGIN
IF   Clear = '1' THEN
        Q <= 0;
ELSIF Clock' EVENT AND Clock = '1' THEN
  Q <= D;
END IF;
WAIT ON Clear, Clock;
END PROCESS;
```

The WAIT ON statement causes the process to halt execution until an event occurs on either the *Clear* or *Clock* input. The IF statement is then executed and if *Clear* = 1, the FLIP-FLOP is asynchronously cleared ($Q = 0$), otherwise the *Clock* is checked for a rising edge with which to assign the *D* input to the *Q* output.

14.6.4 Registers in VHDL

The registers consists of FLIP-FLOPs and gates and can be described in VHDL either by using the FLIP-FLOPs and gates as subcircuits or by using the sequential assignment statements.

Example 14.26

Write VHDL code for a 5-bit shift register shown in Fig. 14.14.

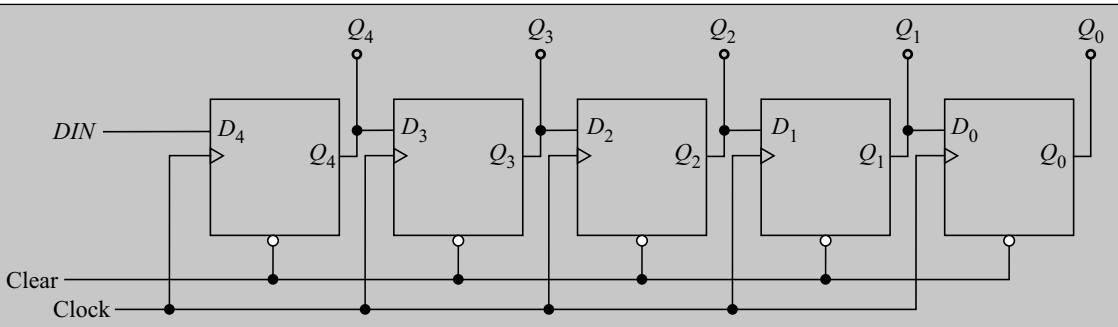

Fig. 14.14 ***A 5-bit Shift Register with Active-Low Asynchronous Clear***

Solution

```
LIBRARY IEEE; USE IEEE. STD_LOGIC_1164.ALL;
ENTITY shift5 IS
PORT   (DIN : IN STD_LOGIC;
        Clock, Clear : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR (5 DOWN TO 0));
END shift5;
ARCHITECTURE shreg5 OF shift5 IS
BEGIN
  PROCESS (Clear, Clock)
BEGIN
IF Clear = '0' THEN Q <= "00000";
ELSIF Clock' EVENT AND Clock = '1' THEN
          Q(4)  <= DIN;
          Q(3)  <= Q(4);
          Q(2)  <= Q(3);
          Q(1)  <= Q(2);
          Q(0)  <= Q(1);
END IF;
END PROCESS;
END shreg5;
```

Example 14.27

Write VHDL code for an 8-bit register using D-type FLIP–FLOPs, asynchronous clear, positive edge-triggered, with parallel-in parallel-out.

Solution

We can write the code introducing the number of bits N through GENERIC clause.

```
LIBRARY IEEE; USE IEEE. STD_LOGIC_1164.ALL;
ENTITY Regn IS
GENERIC (N : INTEGER : = 8);
PORT   (D : IN STD_LOGIC_VECTOR (N-1 DOWN TO 0);
        Clear, Clock : IN STD-LOGIC;
```

```
        Q : OUT STD_LOGIC_VECTOR (N-1 DOWN TO 0));
END Regn;
ARCHITECTURE PIPON OF Regn IS
BEGIN
  PROCESS (Clear, Clock)
BEGIN
IF  Clear = '0' THEN
        Q <= (OTHERS => '0');   – – Assigns 0 bit to each Q
ELSIF Clock' EVENT AND Clock = '1' THEN
  Q <= D ;
END IF;
END PROCESS;
END PIPON;
```

14.6.5 Counters in VHDL

Digital counters are designed using FLIP–FLOPs. The counters can be asynchronous or synchronous, UP or DOWN or UP/DOWN, provision for clearing, parallel loading, and enable, etc. There can be described in VHDL.

Example 14.28

Write VHDL code for a 4-bit synchronous counter shown in Fig. 14.15.

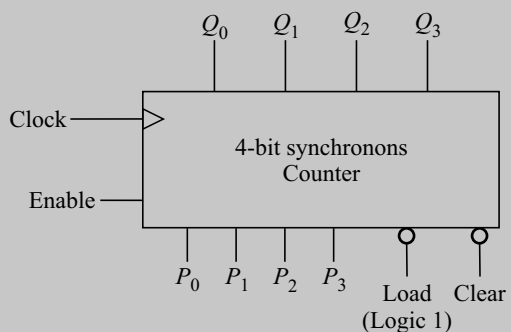

Fig. 14.15 ***A 4-bit Synchronous Counter***

Solution

It is a 4-bit synchronous UP counter. When Clear = 0, the 4-bit output $Q_3\ Q_2\ Q_1\ Q_0$ = 0000. With *Enable* = 1 and *Clear* = 1, it will now count in the UP direction from 0000 to 1111. Its entity and architecture body are given below. We shall use UNSIGNED data type by including the IEEE package.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_ LOGIC_UNSIGNED.ALL;
ENTRY Count4 IS
PORT     (Clock, Clear, Enable : IN STD_LOGIC;
                        Q : OUT STD_LOGIC_VECTOR (3 DOWN TO 0));
END Count4;
ARCHITECTURE Syn_counter OF Count4 IS
SIGNAL Count : STD_LOGIC_VECTOR (3 DOWN TO 0);
BEGIN
  PROCESS (Clock, Clear)
BEGIN
IF Clear = '0' THEN Count <= "0000"
ELSIF Clock' EVENT AND Clock = '1' THEN
IF Enable = '1' THEN
```

```
   Count <= Count + 1;
ELSE
   Count <= Count;
END IF;
END PROCESS;
   Q <= Count;   – – Output Q is assigned Count
END syn_counter;
```

Example 14.29

Assume that counter of Fig. 14.15 is loaded with an integer by making *Load* = 0 before starting counting. Write its VHDL code.

Solution

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY Count4L IS
PORT     (P : IN INTEGER RANGE 0 to 15;
          Clock, Clear, Load : IN STD_LOGIC;
          Q : OUT STD_LOGIC_VECTOR (3 DOWN TO 0));
END Count4L;
ARCHITECTURE syn_PICOUNT OF Count IS
BEGIN
   PROCESS (Clock, Clear)
BEGIN
IF Clear = '0' THEN Q <= '0';
ELSIF Clock' EVENT AND Clock = '1' THEN
IF Load = '0' THEN Q <= P;
ELSE Q <= Q + 1;
END IF;
END IF;
END PROCESS;
END Syn_PICOUNT;
```

In this, on the positive edge of the clock if *Load* = 0, then the FLIP-FLOPs in the counter are loaded in parallel form from the *P* inputs.

Example 14.30

Write VHDL code for a mod 8 DOWN counter using positive edge-triggered FLIP-FLOPs with active-high *Enable* input and active-low *Load* input. When the Load input is Low, the FLIP-FLOPs are loaded with integer 7 (mod-1).

Solution

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY downcount IS
GENERIC (mod : INTEGER : = 8);
```

```
PORT    (Clock, Load, Enable  : IN STD_LOGIC;
                              Q  : OUT INTEGER RANGE 0 TO mod-1);
END downcount;
ARCHITECTURE counter_mod 8 OF downcount IS
SIGNAL COUNT : INTEGER 0 TO mod-1;
BEGIN
PROCESS
BEGIN
WAIT UNTIL Clock' EVENT AND Clock = '1';
IF Enable = '1' THEN
IF Load = '0' THEN
COUNT <= mod-1;
ELSE
COUNT <= COUNT-1;
END IF;
END IF;
END PROCESS;
  Q <= COUNT;
END Counter_mod 8;
```

Here, the starting count is mod-1. It is defined as GENERIC parameter named mod. Since the Load input is active-low, therefore, on the positive clock edge, if *Load* = 0, the counter is loaded with the value mod-1. On the other hand, if Load = 1, the count is decremented by 1 on every positive edge of the clock input.

SUMMARY

The computer aided design of digital systems is widely used in industry for designing complex digital systems. Therefore, it has become necessary to learn the various concepts involved in computer aided design techniques and the CAD tools. The basic principles of digital theory and practice are essentially required even when CAD tools are used. The most popular hardware description language VHDL has been briefly introduced in this chapter.

Since VHDL is a vast subject and therefore, it is not possible to cover all the details of VHDL in this book. However, some of the essential features of VHDL have been covered and a large number of examples have been discussed covering various combinational and sequential circuits. The concepts of VHDL introduced here will be helpful in learning more details of the language and its applications from the texts devoted to VHDL.

GLOSSARY

AHDL (Altera Hardware Description Language) A HDL developed by Altera corporation of U.S.A for programming their PLDs.

ARCHITECTURE A key word in VHDL to define the operation of a circuit declared using an ENTITY.

Architecture body A body associated with an entity declaration to describe the internal organization or operation of a design entity. It is used to describe the behaviour, data flow, or structure of a design entity.

Behavioural modelling A method of describing a digital circuit/system which gives the behaviour of the circuit/system to its inputs.

Binding The process of associating a design entity and, optionally, an architecture with an instance of a component. It can be specified in an explicit or default binding indication.

BIT The data object type representing a single bit in VHDL.

Bit Array Representation of a group of bits and assigning it a name.

BIT_VECTOR The data object type representing a bit array in VHDL.

BOOLEAN The data object type in VHDL representing 'true' or 'false'.

Buried Node A point identified in a circuit that is not accessible from outside this circuit. Also known as a local signal.

CAD (Computer-aided design) Design methodologies using computers.

CAD tools Various programs used to design systems using computers. Each tool performs a specific task in the design process.

CASE A VHDL construct that selects one of several options when describing a circuit's operations based on the value of a data object.

Character literal A character of the literal type formed by enclosing one of the graphic characters (including a space and non breaking space characters) between two apostrophe (') characters.

Character type An enumeration type with atleast one character literal among its enumeration literals.

Component A VHDL keyword used to provide information about a component declared in a library.

Component instantiation statement A VHDL concurrent statement that instantiates a component in the architecture body.

Concatenation A term used to describe the linking of two or more data objects into an ordered set.

Concurrent Assignment Statement A statement in VHDL used to describe a circuit that works concurrently with other concurrent statements.

Conditional signal assignment A VHDL concurrent construct that evaluates a series of conditions sequentially to determine the first true condition evaluated and assigns its value to a signal.

Configuration A construct that defines how component instances in a given block are bound to design entities in order to describe how design entities are put together to form a complete design.

Constant An object whose value may not be changed.

Declaration A construct that defines a declared entity and associates an identifer with it.

Declarative part A syntactic component of certain declarations or statements such as entity declarations and architecture bodies.

Declarative region A semantic component of certain declarations or statements.

Design entity An entity declaration together with an associated architecture body. Different design entities may share the same entity declaration.

Design library A host-dependent storage facility for intermediate form representations of analyzed design units.

Design unit A construct that can be independently analyzed and stored in a design library. A design unit can be an entity declaration, an architecture body, a configuration declaration, a package declaration, or a package body declaration.

ELSE A control structure used in conjunction with `IF`/`THEN` to perform an alternate action if the condition is 'false'.

ELSIF A control structure which can be used multiple times following an `IF` statement to select one of several options based on whether the associated expressions are 'true' or 'false'.

ENTITY A keyword in VHDL used to define a circuit about its input(s), output(s) and signal types.

Entity declaration A definition of the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture.

Enumeration literal A literal of an enumeration type. It may be either an identifier or a character literal.

Enumeration type A type whose values are defined by listing (enumerating) them. The values of the types are represented by enumeration literals.

EVENT A change in the current value of a signal, which occurs when the signal is updated with its effective value.

Generic An interface constant declared in the block header of a block statement, a component declaration, or an entity declaration.

HDL (Hardware description language) A coding language used to describe hardware rather than a programming language.

Hierarchical Design A method of designing a system/circuit by breaking it into its constituent modules, each of which can be broken further into simpler constituent modules.

Identifier It defines an alias for some other name.

IF/THEN A control structure used to evaluate a condition and takes decision on the basis of its coming 'true' or 'false'.

Instance A subcomponent of a design entity whose prototype is a component declaration, design entity, or configuration declaration. A component instantiation statement whose instantiated unit denotes a component that creates an instance of the corresponding component.

Integer literal A literal of the type integer.

Integer type A discrete scalar type whose values represent integer numbers with in a specified range.

Layout synthesis Same as physical design.

Library See design library.

Local signal Same as buried node.

Mode The direction of signal flow through a port.

Object Various ways of representing data in the HDL's code.

Optimisation Process of designing a circuit making optimum use of available resources,

PACKAGE A VHDL keyword used to define a set of global elements that are available to other modules.

PORT MAP A VHDL keyword used to list the connections between components.

PROCESS A VHDL keyword used to define a set of sequential statements that describes the functionality of a portion of an entity.

Routing Process of interconnecting logic blocks.

Selected signal assignment A VHDL signal assignment statement that selects among a number of options to assign a value to the signal.

Sensitivity list A list of signals used to invoke the sequence of statements in a `PROCESS`.

Sequential statements Statements that execute in sequence in the order in which they appear. These are used for algorithmic descriptions.

STD_LOGIC A data type similar to BIT type, having possible values as 0, 1, Z, —(don't care), in VHDL.

STD_LOGIC_VECTOR A data type similar to BIT_VECTOR type in VHDL that can have a number of possible values.

Structural modeling A method of describing a digital circuit/system using various physical components of the circuit.

Synthesis Same as design.

Target chip A semiconductor chip such as a CPLD or FPGA which is to be used for physical design.

Technology mapping Transforming design on the technology of the device being used.

Verilog HDL A hardware description language.

VHDL Very high speed integrated circuits Hardware Description Language.

VHSIC Acronym for Very High Speed Integrated Circuits.

REVIEW QUESTIONS

14.1 In a CAD system, the most convenient method of design entry for a large system use is _____.
14.2 _____ tool is used after functionally correct simulation of the circuit is obtained.
14.3 The CPLD or FPGA device to be used for design implementation is known as _____ chip.
14.4 `IN`, `OUT`, and `INOUT` are three possible _____ of a port.
14.5 A line starting with – – signifies _____ in VHDL.
14.6 An entity can have more than _____ architecture body.
14.7 Data type BIT can have values _____.
14.8 _____ declaration specifies only the outer description of a circuit/component.
14.9 VHDL stands for _____.
14.10 VHSIC stands for _____.
14.11 _____ specifies the behaviour of a circuit to be designed.
14.12 In data flow modeling statements are executed _____.
14.13 The statements with in PROCESS are executed _____.
14.14 The signals written in parentheses after the keyword PROCESS is _____.
14.15 _____ is used to bind a component instance to an entry-architecture pair.
14.16 An operator used for signal assignment is _____.

14.17 An operator used for variable assignment is _____.

14.18 Conditional signal assignment is _____ inside PROCESS in VHDL.

14.19 PROCESS statement itself is a _____ statement.

14.20 STD_LOGIC_1164 is a _____ in VHDL IEEE library.

14.21 In a VHDL entity declaration, `PORT` (A : `IN` STD_LOGIC); A can have a value _____.

14.22 *Clock'EVENT* `AND` Clock = '1' is used in VHDL to represent _____.

14.23 `IN` VHDL IEEE LIBRARY; USE IEEE.STD_LOGIC_1164.ALL; statement allows _____ package to be used.

14.24 In the `ARCHITECTURE` body of a D-type `FLIP-FLOP`, PROCESS contains statements

```
WAIT UNTIL Clock = '1';
    Q <= D;
```

the clock is _____.

PROBLEMS

14.1 Are the following names valid for entities? Justify your answer.

(a) rp_jain (b) R.P. JAIN (c) 3_input

(d) Third_Edition (e) Edition-3 (f) Circuit_ _4

14.2 Write entity declarations for the gates shown in Fig. 14.16.

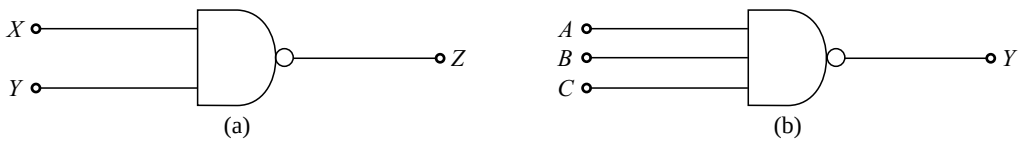

Fig. 14.16

14.3 Write entity declaration for a 4:1 multiplexer circuit

14.4 Write architecture declaration for the gates shown in Fig. 14.16.

14.5 Write VHDL code for a full adder shown in Fig. 14.17. Refer to Fig. 5.21 for its `NAND` gate realization.

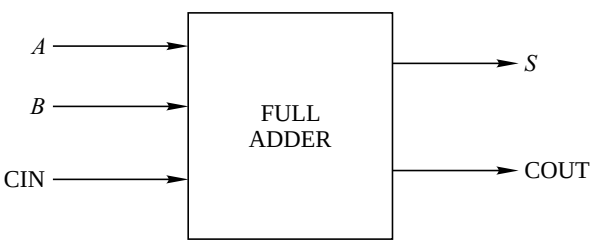

Fig. 14.17 ***Block Diagram of a Full Adder***

14.6 Write VHDL code for the full adder shown in Fig. 14.17. Refer to logic expressions 5.35 for data flow architecture.

14.7 Write VHDL code for the truth table given in Table 5.4

14.8 Write VHDL code for the following circuits:

(i) half-subtractor circuit of Fig. 5.22.

(ii) full-subtractor circuit of Fig. 5.23.

14.9 Write VHDL code for a 3-to-8 decoder using `CASE` statement.

14.10 Repeat Prob. 14.9 using selected signal assignment.

14.11 Write VHDL code for 2 : 1 multiplexer of Fig. 14.7. Use `CASE` statement.

14.12 Write VHDL code for a 4 : 1 multiplexer. Use selected signal assignment.

14.13 Write VHDL code for BCD-to-7-segment decoder using selected signal assignment.

14.14 Write VHDL code for a 4-bit synchronous counter with synchronous clear input.