SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

**SCHOOL OF ELECRICAL AND ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

# UNIT – I - BASIC CONCEPTS IN VHDL – SECA1602

# I. Introduction

## VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC is an acronym for Very High-Speed Integrated Circuits). It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modeled in the same description. The VHDL language can be regarded as an integrated amalgamation of the following languages:

- sequential language
- Concurrent language
- net-list language
- timing specifications
- Waveform generation language.

Therefore, the language has constructs that enable you to express the concurrent or sequential behavior of a digital system with or without timing. It also allows you to model the system as an interconnection of components. Test waveforms can also be generated using the same constructs. All the above constructs may be combined to provide a comprehensive description of the system in a single model.

The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is a strongly typed language and is often verbose to write. It inherits many of its features, especially the sequential language part, from the Ada programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. Fortunately, it is possible to quickly assimilate a core subset of the language that is both easy and simple to understand without learning the more complex features. This subset is usually sufficient to model most applications. The complete language, however, has sufficient power to capture the descriptions of the most complex chips to a complete electronic system.

**Digital system design process: -**

Digital Systems have conquered the whole world. Every appliances or equipment's we see today are digital. This is because of the very small element called Transistor invented by John Bardeen, Walter Brattain & William Shockley in 1947 at Bell Labs. This tiny and Powerful transistor changed the future of Electronics. Therefore, it is our responsibility to study the analysis and design of this digital system as an electronic student. In this chapter we will study the Basic Digital IC Design Flow and then we will study what are the tools available for digital design and synthesis. Later we are going to study a special hardware description language (VHDL) which is used to describe the digital systems.

**Digital Design Flow Process: -**

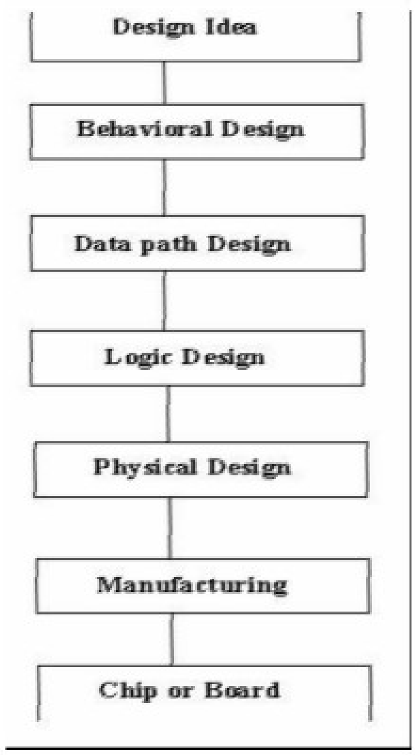


Fig 1.1: Generic IC design flow

Based on the specification given, the design team forms a general idea about the solution to the problem. System level decisions are made regarding the design and a general consensus is reached regarding the major functional blocks that go into the making of the chip. At the end of this stage, a general block diagram solution of the design is agreed upon. CAD tools are generally not needed at this stage.

**Behavioral Design:**

Hardware Description Languages (HDLs) are used to model the design idea (block diagram). Circuit details and electrical components are not specified. Instead, the behavior of each block at the highest level of abstraction is modeled. Simulations are then run to see if the blocks do indeed function as expected and the whole system performs as a whole. Behavioral descriptions are important as they corroborate the integrity of the design idea. Here we don’t have any architectural or hardware details.

**Data Path Design:**

The next Phase in the design process is the design of the system data path. In this phase, the designer specifies the registers and logic units necessary for implementation of the system. These components may be interconnected using either bidirectional or unidirectional buses. Based on the intended behavior of the system, the procedure of controlling the movement of data between registers and logic units through buses are developed. Data components in the data part of circuit communicate via system busses and the control procedure controls flow of data between these components. This phase results in architectural design of the system with specification of control flow.

**Logic Design:**

Logic Design is the next phase in the design process and involves the use of primitive gates and flip-flops for the implementation of data registers, busses, logic units, and their controlling hardware. The result of this design stage is a net list of gates and flip-flops. Components used and their interconnections are specified in this net list.

**Physical Design:**

This stage transforms the net list into transistor list or layout. This involves the replacement of gates and flip-flops with their transistor equivalents or library cells.

**Manufacturing**:

The final step is manufacturing, which uses the transistor list or layout specification to burn fuses of FPGA or to generate masks for Integrated circuit (IC).

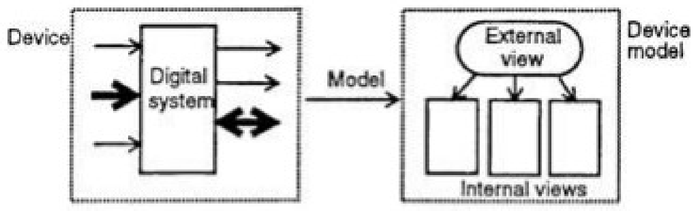
**Levels of Abstraction: -**

**Hardware Abstraction:**

VHDL is used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view of the device specifies the functionality or structure, while the external view specifies the interface of the device through which it communicates with the other models in its environment. Fig1.2 shows the hardware device and the corresponding software model.

The device-to-device model mapping is strictly a one to many. That is, a hardware device may have many device models. For example, a device modeled at a high level of abstraction may not have a clock as one of its inputs, since the clock may not have been used in the description. Also, the data transfer at the interface may be treated in terms of say, integer values, instead of logical values. In VHDL, each device model is treated as a distinct representation of a unique device, called an entityin this text. Fig1.2 shows the VHDL view of a hardware device that has multiple device models, with each device model representing one entity. Even though entity I through N represent N different entities from the VHDL point of view, in reality they represent the same hardware device.

The entity is thus a hardware abstraction of the actual hardware device. Each entity is described using one model that contains one external view and one or more internal views. At the same time, a hardware device may be represented by one or more entities.



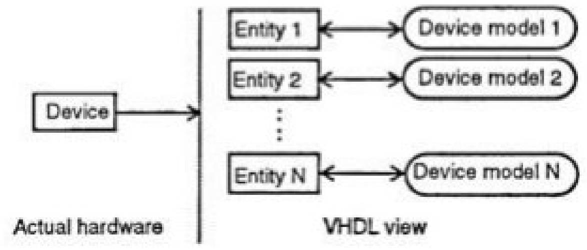


Fig 1.2 A VHDL view of a device

**Basic Terminology:**

VHDL is a hardware description language that can be used to model a digital system. The digital system can be as simple as a logic gate or as complex as a complete electronic system. A hardware abstraction of this digital system is called an entity in this text. An entity X, when used in another entity Y, becomes a component for the entity Y. Therefore, a component is also an entity, depending on the level at which you are trying to model.

To describe an entity, VHDL provides five different types of primary constructs, called design units. They are

1. Entity declaration
2. Architecture body
3. Configuration declaration
4. Package declaration
5. Package body

An entity is modeled using an entity declaration and at least one architecture body. The entity declaration describes the external view of the entity, for example, the input and output signal names. The architecture body contains the internal description of the entity, for example, as a set of interconnected components that represents the structure of the entity, or as a set of concurrent or sequential statements that represents the behavior of the entity. Each style of representation can be specified in a different architecture body or mixed within a single architecture body Fig1.3 shows an entity and its model

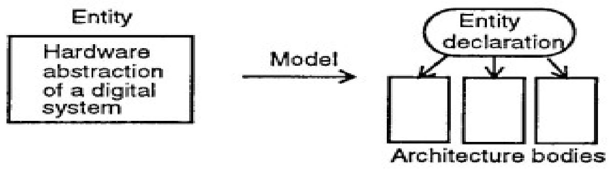


Fig 1.3 An entity and its model.

A configuration declaration is used to create a configuration for an entity. It specifies the binding of one architecture body from the many architecture bodies that may be associated with the entity. It may also specify the bindings of components used in the selected architecture body to other entities. An entity may have any number of different configurations. A package declaration encapsulates a set of related declarations such as type declarations, subtype declarations, and subprogram declarations that can be shared across two or more design units. A package body contains the definitions of subprograms declared in a package declaration. Fig1.4 shows three entities called El, E2, and E3. Entity El has three architecture bodies, EI_AI, EI_A2, and EI_A3. Architecture body EI _AI is a purely behavioral model without any hierarchy. Architecture body EI_A2 uses a component called BX, while architecture body EI_ A3 uses a component called CX. Entity E2 has two architecture bodies, E2_ AI and E2_A2, and architecture body E2_AI uses a component called MI. Entity E3 has three architecture bodies, E3_ AI, E3_A2, and E3_A3. Notice that each entity has a single entity declaration but more than one architecture body.

The dashed lines represent the binding that may be specified in a configuration for entity El. There are two types of binding shown: binding of an architecture body to its entity and the binding of components used in the architecture body with other entities. For example, architecture body, EI_A3, is bound to entity El, while architecture body, E2_AI, is bound to entity E2. Component MI in architecture body, E2_AI, is bound to entity E3. Component CX in the architecture body, EI _A3, is bound to entity E2. However, one may choose a different configuration for entity El with the following bindings:

- Architecture EI_A2 is bound to its entity El
- Component BX to entity E3
- Architecture E3_AI is bound to its entity E3

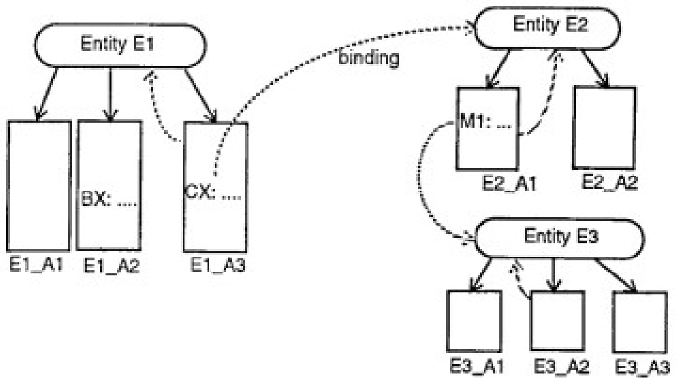


Fig 1.4 A configuration for entity El.

Once an entity has been modeled, it needs to be validated by a VHDL system. A typical VHDL system consists of an analyzer and a simulator. The analyzer reads in one or more design units contained in a single file and compiles them into a design library after validating the syntax and performing some static semantic checks. The design library is a place in the host environment (that is, the environment that supports the VHDL system) where compiled design units are stored. The simulator simulates an entity, represented by an entity-architecture pair or by a configuration, by reading in its compiled description from the design library and then performing the following steps:

1. Elaboration
2. Initialization

3. Simulation

A note on the language syntax. The language is case insensitive, that is, lower case and upper-case characters are treated alike. For example, CARRY, CarrY, or CarrY, all refer to the same name. The language is also free -format, very much like in Ada and Pascal programming languages. Comments are specified in the language by preceding the text with two consecutive dashes (-). All text between the two dashes and the end of that line is treated as a comment. The terms introduced in this section are described in greater detail in the following sections.

**Entity Declaration:**

The entity' declaration specifies the name of the entity being modeled and lists
the set of interface ports. Ports are signals through which the entity communicates with the other models in its external environment.

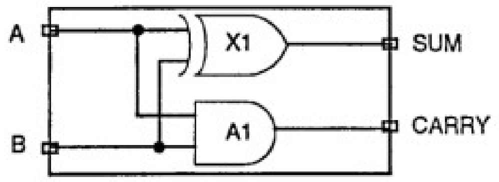


Fig 1.5 A half-adder circuit.

Here is an example of an entity declaration for the half-adder circuit shown in Fig. 1.5.

```
entity HALF_ADDER is
port (A, B: in BIT; SUM, CARRY: out BIT);
end HALF_ADDER;
```

The entity, called HALF_ADDER, has two input ports, A and B (the mode in specifies input port), and two output ports, SUM and CARRY (the mode out specifies output port). BIT is a predefined type of the language; it is an enumeration type containing the character literals '0' and '1'. The port types for this entity have been specified to be of type BIT, which means that the ports can take the values, '0' or '1'.

The following is another example of an entity declaration for a 2-to-4 decoder circuit shown in Fig. 1.6.

```
entity DECODER2x4 is
        port (A, B, ENABLE: in SIT:
                Z: out BIT_VECTOR(0 to 3));
        end DECODER2x4;
```

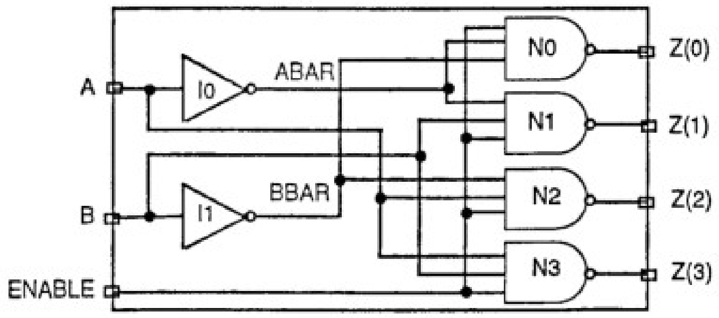


Fig 1.6: A 2-to-4 decoder circuit.

This entity, called DECODER2x4, has three input ports and four output ports. BIT_VECTOR is a predefined unconstrained array type of BIT. An unconstrained array type is a type in which the size of the array is not specified. The range "0 to 3" for port Z specifies the array size.

From the last two examples of entity declarations, we see that the entity declaration does not specify anything about the internals of the entity. It only specifies the name of the entity and the interface ports.

**Architecture Body:**

The internal details of an entity are specified by an architecture body using any of the
following modeling styles:

1. As a set of interconnected components (to represent structure),
2. As a set of concurrent assignment statements (to represent dataflow),
3. As a set of sequential assignment statements (to represent behavior),
4. Any combination of the above three.

**Structural Style of Modeling:**

In the structural style of modeling, an entity is described as a set of interconnected components. Such a model for the HALF_ADDER entity, shown in Fig. 1.5, is described in an architecture body as shown below.

```
architecture HA_STRUCTURE of

          HALF_ADDER is component

          XOR2

                    port (X, Y: in BIT;

                              Z: out BIT);

                    end component;

          component AND2

          port (L, M: in BIT;

                     N: out BIT);

          end component;

          begin

          X1: XOR2 port map (A, B, SUM);

          A1: AND2 port map (A, B, CARRY);

end HA_STRUCTURE;
```

The name of the architecture body is HA_STRUCTURE. The entity declaration for HALF_ADDER (presented in the previous section) specifies the interface ports for this architecture body. The architecture body is composed of two parts: the declarative part (before the keyword begin) and the statement part (after the keyword begin). Two component declarations are present in the declarative part of the architecture body. These declarations specify the interface of components that are used in the architecture body. The components XOR2 and AND2 may either be predefined components in a library, or if they do not exist, they may later be bound to other components in a library.

The declared components are instantiated in the statement part of the architecture body using component instantiation statements. XI and A1 are the component labels for these component instantiations. The first component instantiation statement, labeled XI, shows that signals A and B (the input ports of the HALF_ADDER), are connected to the X and Y input ports of a XOR2 component, while output port Z of this component is connected to output port SUM of the HALF_ADDER entity.

Similarly, in the second component instantiation statement, signals A and B are connected to ports L and M of the AND2 component, while port N is connected to the CARRY port of the HALF_ADDER. Note that in this case, the signals in the port map of a component instantiation and the port signals in the component declaration are associated by position (called positional association). The structural representation for the HALF_ADDER does not say anything about its functionality. Separate entity models would be described for the components XOR2 and AND2, each having its own entity declaration and architecture body.

A structural representation for the DECODER2x4 entity, shown in Fig. 1.6, is shown next.

```
   architecture DEC_STR of
      DECODER2x4 is component
      INV
            port (A: in BIT;
                 Z: out BIT);
   end component;
   component NAND3
   port (A, B, C: in BIT;
            Z: out BIT);
   end component;
   signal ABAR, BBAR: BIT

begin

I0: INV port map (A, ABAR);

I1: INV port map (B, BBAR);

N0: NAND3 port map (ABAR, BBAR, ENABLE, Z(0));

N1: NAND3 port map (ABAR, B, ENABLE, Z (1));

N2: NAND3 port map (A, BBAR, ENABLE, Z (2));

N3: NAND3 port map (A, B, ENABLE, Z (3));

end DEC_STR;
```

In this example, the name of the architecture body is DEC_ STR, and it is associated with the entity declaration with the name DECODER2x4; therefore, it inherits the list of interface ports from that entity declaration. In addition to the two component declarations (for INV and NAND3), the architecture body contains a signal declaration that declares two signals, ABAR and BBAR, of type BIT. These signals, that represent wires, are used to connect the various components that form the decoder. The scope of these signals is restricted to the architecture body, and therefore, these signals are not visible outside the architecture body. Contrast these signals with the ports of an entity declaration that are available for use within any architecture body associated with the entity declaration

A component instantiation statement is a concurrent statement, as defined by the language. Therefore, the order of these statements is not important. The structural style of modeling describes only an interconnection of components (viewed as black boxes) without implying any behavior of the components themselves, nor of the entity that they collectively represent. In the architecture body DEC_STR, the signals A, B, and ENABLE, used in the component instantiation statements are the input ports declared in the DECODER2x4 entity declaration. For example, in the component instantiation labeled N3, port A is connected to input A of component NAND3, port B is connected to input port B of component NAND3, port ENABLE is connected to input port C, and the output port Z of component NAND3 is connected to port Z (3) of the DECODER2x4 entity. Again, positional association is used to map signals in a port map of a component instantiation with the ports of a component specified in its declaration. The behavior of the components NAND3 and INV are not apparent, nor is the behavior of the decoder entity that the structural model represents

**Configuration Declaration:**

A configuration declaration is used to select one of the possibly many architecture bodies that an entity may have, and to bind components, used to represent structure in that architecture body, to entities represented by an entity-architecture pair or by a configuration, that reside in a design library. Consider the following configuration declaration for the HALF_ADDER entity

```
library CMOS_LIB, MY_LIB;

configuration HA_BINDING of

  HALF_ADDER is for HASTRUCTURE

       for X1:XOR2

            use entity

            CMOS_LIB.XOR_GATE(DATAFLOW);

       end for;

       for A1:AND2

            use configuration MY_LIB.AND_CONFIG;

       end for;

       end for; end HA_BINDING;
```

The first statement is a library context clause that makes the library names CMOS_LIB and MY_LIB visible within the configuration declaration. The name of the configuration is HA _BINDING, and it specifies a configuration for the HALF_ADDER entity. The next statement specifies that the architecture body HA_STRUCTURE (described in Sec. 23.1) is selected for this configuration. Since this architecture body contains two component instantiations, two component bindings are required. The first statement (for XI: . . . end for) binds the component instantiation, with label XI, to an entity represented by the entity-architecture pair, XOR_GATE.

The architecture body consists of one signal declaration and six concurrent signal assignment statements. The signal declaration declares signals ABAR and BBAR to be used locally within the architecture body. In each of the signal assignment statements, no after clause was used to specify delay. In all such cases, a default delay of 0ns is assumed. This delay of 0ns is also known as delta delay, and it represents an infinitesimally small delay. This small delay corresponds to a zero delay with respect to simulation time and does not correspond to any real simulation time.

To understand the behavior of this architecture body, consider an event happening on one of the input signals, say input port B at time T. This would cause the concurrent signal assignment statements 1,3, and 6, to be triggered. Their right - hand-side expressions would be evaluated and the corresponding values would be scheduled to be assigned to the target signals at time (T+A). When simulation time advances to (T+A), new values to signals Z(3), BBAR, and Z(1), are assigned. Since the value of BBAR changes, this will in turn trigger signal assignment statements, 2 and 4. Eventually, at time (T+2A), signals Z(0) and Z(2) will be assigned their new values.

The semantics of this concurrent behavior indicate that the simulation, as defined by the language, is event-triggered and that simulation time advances to the next time unit when an event is scheduled to occur. Simulation time could also advance a multiple of delta time units. For example, events may have been scheduled to occur at times 1,3,4,4+A, 5,6,6+A, 6+2A, 6+3A, 10,10+A, 15, 15+A time units.

Entity declaration and the DATAFLOW architecture body, that resides in the CMOS_LIB design library. Similarly, component instantiation Al is bound to a configuration of an entity defined by the configuration declaration, with name AND_CONFIG, residing in the MY_LIB design library.

There are no behavioral or simulation semantics associated with a configuration declaration. It merely specifies a binding that is used to build a configuration for an entity. These bindings are performed during the elaboration phase of simulation when the entire design to be simulated is being assembled. Having defined a configuration for the entity, the configuration can then be simulated. When an architecture body does not contain any component instantiations, for example, when dataflow style is used, such an architecture body can also be selected to create a configuration. For example, the DEC_DATAFLOW architecture body can be selected for the DECODER2x4 entity using the following configuration declaration.

```
configuration DEC_CONFIG of

DECODER2x4 is for

DEC_DATAFLOW

end for;

end DEC_CONFIG;
```

DEC_CONFIG defines a configuration that selects the DEC_DATAFLOW architecture body for the DECODER2x4 entity. The configuration DEC_CONFIG, that represents one possible configuration for theDECODER2x4 entity, can now be simulated

**Package Declaration**

A package declaration is used to store a set of common declarations like components, types, procedures, and functions. These declarations can then be imported into other design units using a context clause. Here is an example of a package declaration.

```
            package EXAMPLE_PACK is
                    type SUMMER is (MAY, JUN, JUL, AUG, SEP);
                    component D_FLIP_FLOP
                            port (D, CK: in BIT; Q,
                                 QBAR:out BIT);
                    end component;
   constant PIN2PIN_DELAY: TIME: =125 ns;
   function INT2BIT_VEC
            (INT_VALUE: INTEGER)
                    return
            BIT_VECTOR;
   end EXAMPLE_PACK;
```

The name of the package declared is EXAMPLE_PACK. It contains type, component, constant, and function declarations. Notice that the behavior of the function INT2BIT _VEC does not appear in the package declaration; only the function interface appears. The definition or body of the function appears in a package body (see next section).

Assume that this package has been compiled into a design library called DESIGN_LIB. Consider the following context clauses associated with an entity declaration.

```
library DESIGN_LIB;
use DESIGN_LIB.EXAMPLE_P
ACK.all; entity RX is . . .
```

The library context clause makes the name of the design library DESIGN_LIB visible within this description, that is, the name DESIGN_LIB can be used within the description. This is followed by a use context clause that imports all declarations in package EXAMPLE_PACK into the entity declaration of RX.It is also possible to selectively import declarations from a package declaration into other design units. For example,

```
   library DESIGN_LIB;
```

```
use

DES[GN_LIB.EXAMPLE_PACK.D_FLIP_ FLOP;

use

DESIGN_LIB.EXAMPLE_PACK.PIN2PIN_DELAY;

architecture RX_STRUCTURE of

RX is . . .
```

The two use context clauses make the component declaration for D_FLIP_FLOP and the constant declaration for PIN2PIN_DELAY, visible within the architecture body. Another approach to selectively import items declared in a package is by using selected names. For example

```
library DESIGN_LIB;

package

ANOTHER_PACKAGE is function

POCKET_MONEY

        (MONTH:DESIGN_LIB.EXAMPLE_PAC K.SUMMER)

        return INTEGER;

        constant TOTAL_ALU: INTEGER; -- A deferred constant.

end ANOTHER_PACKAGE;
```

The type SUMMER declared in package EXAMPLE_PACK is used in this new package by specifying a selected name. In this case, a use context clause was not necessary. Package ANOTHER_PACKAGE also contains a constant declaration with the value of the constant not specified; such a constant is referred to as a deferred constant. The value of this constant is supplied in a corresponding package body.

**Package Body:**

A package body is primarily used to store the definitions of functions and procedures that were declared in the corresponding package declaration, and also the complete constant declarations for any deferred constants that appear in the package declaration. Therefore, a package body is always associated with a package declaration; furthermore, a package declaration can have at most one package body associated with it. Contrast this with an architecture body and an entity declaration where multiple architecture bodies may be associated with a single entity declaration.

A package body may contain other declarations as well. Here is the package body for the package EXAMPLE_PACK declared in the previous section.

```
package body EXAMPLE_PACK is

        function INT2BIT_VEC (INT_VALUE: INTEGER)

return

        BIT_VECTOR is
```

```
                begin
            end INT2BIT_VEC;
        end EXAMPLE_PACK;
```

The name of the package body must be the same as that of the package declaration with which it is associated. It is important to note that a package body is not necessary if the corresponding package declaration has no function and procedure declarations and no deferred constant declarations. Here is the package body that is associated with the package ANOTHER_PACKAGE that was declared in the previous section.

```
    package body ANOTHER_PACKAGE is
        constant TOTAL_ALU: INTEGER: = 10;
        function POCKET_MONEY
        (MONTH:
        DESIGN_UB.EXAMPLE_PACK.SUMMER)
        return INTEGER is
    begin
        case MONTH is
            when MAY => return 5;
            when JUL I SEP => return 6;
            when others => return 2;
        end case; end POCKET_MONEY;
    end ANOTHER_PAC
```

**Data Operators:**

VHDL will support different types of operations. The following are the types of operators available in VHDL

1. Assignment operator
2. Logical Operator
3. Relational Operator
4. Shift operator
5. Arithmetic operator

   Addition Operator

   Multiplication Operator

   Miscellaneous operator

**Assignment Operator**

This operator is used to assign values to signals, variables, and

constants. They are

1. <= Used to assign a value to signal

2.: = Used to assign a variable, constant or generic, used for also

establishing initial values.

3. => Used to assign values to individual vector or with others.

**Logical Operators**

Used to perform to logical operations. The data must be of type Bit, Std_logic or std_ulogic. The logical operators are

- NOT
- AND
- OR
- NAND
- NOR, XOR & XNOR

**Relational Operators:**

Used for making comparisons. The data can be of any types listed

above. The relational (Comparison) operators listed below:

1. = Equal to

2. /= not equal to

3. < Greater than

4. > Lesser than

5. <= Greater than

6. >= Lesser than

**Shift Operators**

Used for shifting data.

1. Sll: Shift left logic

2. Sla: shift left arithmetic

3. Srl: Shift right logic

4. Sra: Shift right arithmetic

5. Rol:Rotateleft

6. Ror: Rotate right

**Arithmetic Operators**

Used to perform arithmetic operations. The data can be of integer, signed,

Unsigned or a real.

The different types of arithmetic operations are:

1. Addition operator (+)

2. Subtract Operator (-)

3. Multiplication operator (*)

4. Division Operator (/)

5. Modulus (MOD)

6. Remainder (REM)

**Miscellaneous Operator**

Uses as special cases in VHDL

1. Absolute (ABS):
2. Exponentiation (**)

## DATA TYPES

All of the objects that are discussed in previous section—the signal, the Variable, and the constant—can be declared using a type specification to specify the characteristics of the object. VHDL contains a wide range of types that can be used to create simple or complex objects. To define a new type, you must create a type declaration. A type declaration defines the name of the type and the range of the type. Type declarations are allowed in package declaration sections, entity declaration sections, architecture declaration sections, subprogram declaration sections, and process declaration sections.

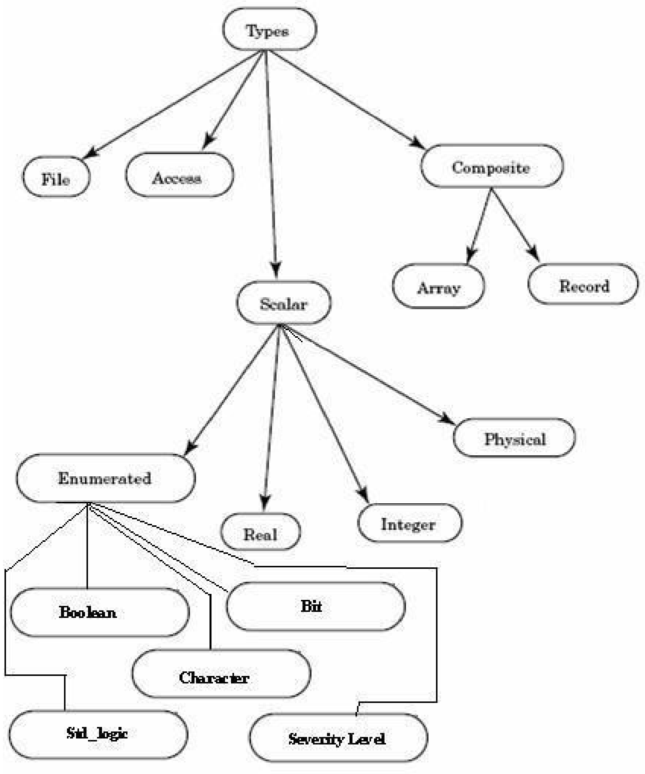


**Fig 1.7: Data Types in VHDL**

### Scalar Types

Scalar types describe objects that can hold, at most, one value at a time. The type itself can contain multiple values, but an object that is declared to be a scalar type can hold, at most, one of the scalar values at any point in time. Referencing the name of the object references the entire object. Scalar types encompass these four classes of types.

1. Integer types
2. Real types
3. Enumerated types
4. Physical types
5. Floating Point

**Enumerated Data Types**

An enumerated type is a very powerful tool for abstract modeling. A designer can use an enumerated type to represent exactly the values required for a specific operation. All of the values of an enumerated type are user-defined. These values can be identifiers or single-character literals. An identifier is like a name.

These are further classified as the following:

1. Boolean
2. Character
3. Bit
4. Std_logic
5. Severity Level

**Boolean**

This data type is used when we need to convey some *true or false* conditions.
For example

```
Architecture …………………..

Beg
in
Pro
cess
(….
)

Variable        temp

:boolean
Begin if a <
b then


temp <=
 True; Else

 temp <= False;

 end if;
```

end process;

**Character**

This daa type is used when we need to use all alpha numeric and special characters.

**Bit**

This data type is used when we need to represent binary values (‘0’ and ‘1’)

**Severity Level**

This data type is used in Complex projects where we need to show warnings, errors in runtime,

Failures in runtime.

**Std_ulogic;**

This data types are declared in std_logic_1164.all package of IEEE Library

U →Uninitialized

X →Forcing
unknown Z
→High Impedence
W →Weak
unknown

‘x‘→don’t care

0 →Forcing 0

1 →Forcing 1

L → Weak 0

H →Weak 1

A typical enumerated type for a four-state simulation value system looks like this:

**Type fourval** is (‘x’, ‘0’, ‘1’, ‘z’);

Character literals are needed for values ‘1’ and ‘0’ to separate these values from the integer values 1 and 0. It would be an error to use the values 1 and 0 in an enumerated type, because these are integer values. The characters X and Z do not need quotes around them because they do not represent any other type, but the quotes were used for uniformity.

**Integer Data type**

Integers are exactly like mathematical integers. All of the normal predefined mathematical functions like add, subtract, multiply, and divide apply to integer types. The VHDL LRM does not specify a maximum range for integers, but does specify the minimum range: from -2,147,483,647 to 12,147,483,647. The minimum range is specified by the Standard package contained in the Standard Library. The Standard package defines all of the predefined VHDL

The Standard Library is used to hold any packages or entities provided as standard with the language.

There are two types of declaration for Integer Data type

1. Type_integer declaration

   Ex: *type <word lengt> is range 0 to 31;*

2. Object_integer declaration

   Ex: *constant <loop number>: <integer><=345;*

**Real Data Type**

Real types are used to declare objects that emulate mathematical real numbers. They can be used to represent numbers out of the range of integer values as well as fractional values. The minimum range of real numbers is also specified by the Standard package in the Standard library, and is from _1.0E_38 to _1.0E_38.

```
Following are a few examples of some real
numbers: Architecture test of test is

Signal a: real;

begin

a <= 1.0;        --ok 1

a <= 1;          --
error 2 a <= -
1.0e10; --ok 3

a <= 1.5e-20;   --ok 4

a <= 5.3 ns;    --error 5

End test;
```

Line 1 shows how to assign a real number to a signal of type **REAL**. All real numbers have a decimal point to distinguish them from integer values. Line 2 is an example of an assignment that does not work. Signal **a** is of type **REAL**, and a real value must be assigned to signal **a**. The value 1 is of type **INTEGER**, so a type mismatch is generated by this line. Line 3 shows a very large negative number. The numeric characters to the left of the character **E** represent the mantissa of the real number, while the numeric value to the right represents the exponent. Line 4 shows how to create a very small number. In this example, them exponent is negative so the number is very small. Line 5 shows how a type **TIME** cannot be assigned to a real signal. Even though the numeric part of the value looks like a real number, because of the units after the value, the value is considered to be of type **TIME**.

**Physical Data types**

Physical types are used to represent physical quantities such as distance, current, time, and so on. A physical type provides for a base unit, and successive units are then defined in terms of this unit. The smallest unit represent able is one base unit; the largest is determined by the range specified in the physical type declaration. An example of a physical type for the physical quantity current is shown here:

```
Type current is range 0 to 1000000000

Units

na; --nano amps

ua = 1000 na; --micro amps

ma = 1000 ua; --milli amps

 a = 1000 ma; --amps

end units;
```

The type definition begins with a statement that declares the name of the type (**current**) and the range of the type (0 to 1,000,000,000). The first unit declared in the **UNITS** section is the base unit. In the preceding example, the base unit is **na**. After the base unit is defined, other units can be defined in terms of the base unit or other units already defined. In the preceding example, the unit **ua** is defined in terms of the base unit as 1000 base units. The next unit declaration is **ma**. This unit is declared as 1000 **ua**. The unit declaration section is terminated by the **END UNITS** clause. More than one unit can be declared in terms of the base unit. In the preceding example, the **ma** unit can be declared as 1000 **ma** or 1,000,000 **na**. The range constraint limits the minimum and maximum values that the physical type can represent in base units. The unit identifiers all must be unique within a single type. It is illegal to have two identifiers with the same name.

## Signal Assignment Statement

Signals are assigned values using a signal assignment statement *The* simplest form of a signal assignment statement is

*signal-object***<=** *expression* [**after** *delay-value* ]**;**

A signal assignment statement can appear within a process or outside of a process. If it occurs outside of a process, it is considered to be a concurrent signal assignment statement. When a signal assignment statement appears within a process, it is considered to be a sequential signal assignment statement and is executed in sequence with respect to the other sequential statements that appear within that process.

When a signal assignment statement is executed, the value of the expression is computed and this value is scheduled to be assigned to the signal after the specified delay. It is important to note that the expression is evaluated at the time the statement is executed (which is the current simulation time) and not after the specified delay. If no after clause is specified, the delay is assumed to be a default delta delay.

Some examples of signal assignment statements are

```
COUNTER <= COUNTER+ "0010"; - Assign after a delta delay.
PAR <= PAR xor DIN after 12 ns;

Z <= (AO and A1) or (BO and B1) or (CO and C1) after 6 ns;
```

## Inertial Delay Model

*Inertial delay* models the delays often found in switching circuits. It represents the time for which an input value must be stable before the value is allowed to propagate to the output. In addition, the value appears at the output after the specified delay. If the input is not stable for the specified time, no output change occurs. When used with signal assignments, the input value is represented by the value of the expression on the right-hand-side and the output is represented by the target signal.

Fig1.8 shows a simple example of a noninverting buffer with an inertial delay of 10 ns.

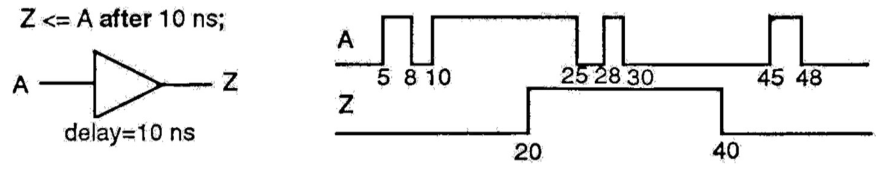


**Fig 1.8: Inertial delay example.**

Events on signal A that occur at 5 ns and 8 ns are not stable for the inertial delay duration and hence do not propagate to the output. Event on A at 10ns remains stable for more than the inertial delay, and therefore, the value is propagated to the target signal Z after the inertial delay; Z gets the value 1' at 20 ns. Events on signal A at

25ns and 28 ns do not affect the output since they are not stable for the inertial delay duration. Transition 1' to '0' at time 30 ns on signal A remains stable for at least the inertial delay duration, and therefore, a '0' is propagated to signal Z with a delay of 10 ns; Z gets the new value at 40 ns. Other events on A do not affect the target signal Z. Since inertial delay is most commonly found in digital circuits, it is the default delay model. This delay model is often used to filter out unwanted spikes and transients on signals.

## Transport Delay Model

*Transport delay* models the delays in hardware that do not exhibit any inertial delay. This delay represents pure propagation delay, that is, any changes on an input is transported to the output, no matter how small, after the specified delay. To use a transport delay model, the keyword transport must be used in a signal assignment statement. Fig 1.9 shows an example of a non-inverting buffer using a transport delay of 10 ns.

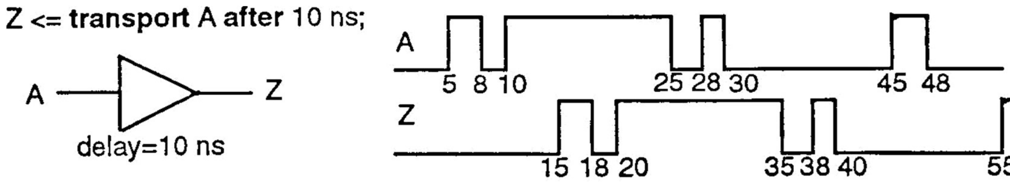


**Fig 1.9: Transport delay example.**

Ideal delay modeling can be obtained by using this delay model. In this case, spikes would be propagated through instead of being ignored as in the inertial delay case. Routing delays can be modeled using transport delay. An example of a routing delay model is

```
                    entity WIRE14 is
port(A: in BIT; Z: out BIT);
                    endWIRE14;

                    architedtureWIRE14_TRANSPORT    of
                    WIRE14 is begin
                        process(
                        A) begin
                                Z <= transport A after
                        0.1 ms; end process;
                    endWIRE14_TRANSPORT;
```

## Concurrent and Sequential assignments

1. As a set of concurrent assignment statements (to represent dataflow),
2. As a set of sequential assignment statements (to represent be-hav.ior),

**Dataflow Style of Modeling(Concurrent assignment**)

In this modeling style, the flow of data through the entity is expressed primarily using concurrent signal assignment statements. The structure of the entity is not explicitly specified in this modeling style, but it can be implicitly deduced. Consider the following alternate architecture body for the HALF ADDER entity that uses this style.

```
architectureHA_CONCURRENTofHALF_
ADDER is
begin
SUM <= A xor B after 8 ns; CARRY <= A
and B after 4 ns;
endHA_CONCURRENT;
```

The dataflow model for the HALF_ADDER is described using two concurrent signal assignment statements (sequential signal assignment statements are described in the next section). In a signal assignment statement, the symbol <= implies an assignment of a value to a signal. The value of the expression on the right-hand-side of the statement is computed and is assigned to the signal on the left-hand-side, called the *target signal.* A concurrent signal assignment statement is executed only when any signal used in the expression on the right-hand-side has an event on it, that is, the value for the signal changes.

Delay information is included in the signal assignment statements using after clauses. If either signal A or B, which are input port signals of HALF_ADDER entity, has an event, say at time T, the right-hand-side expressions of both signal assignment statements are evaluated. Signal SUM is scheduled to get the new value after 8 ns while signal CARRY is scheduled to get the new value after 4 ns. When simulation time advances to (T+4) ns, CARRY will get its new value and when simulation time advances to (T+8) ns, SUM will get its new value. Thus, both signal assignment statements execute concurrently.

Concurrent signal assignment statements are concurrent statements, and therefore, the ordering of these statements in an architecture body is not important. Note again that this architecture body, with name HA_CONCURRENT, is also associated with the same HALF_ADDER entity declaration.

Here is a dataflow model for the DECODER2x4 entity.

```
        architscture dec_dataflow of DECODER2x4
        is signal ABAR, BBAR: BIT;
        begin
Z(3) <=not (A and B and ENABLE);                 -- Statement 1
Z(0) <=not (ABAR and BBAR and ENABLE);    --- Statement 2
BBAR <= not B;                                   -- Statement 3
Z(2) <= not (A and BBAR and ENABLE);             -- Statement 4
ABAR <= not A;                                   -- Statement 5
Z(1 ) <= not (ABAR and B and ENABLE);            -- Statement 6
end DEC_DATAFLOW;
```

The architecture body consists of one signal declaration and six concurrent signal assignment statements. The signal declaration declares signals ABAR and BBAR to be used locally within the architecture body. In each of the signal assignment statements, no after clause was used to specify delay. In all such cases, a default delay of 0ns is assumed. This delay of 0ns is also known as delta delay, and it represents an infinitesimally small delay. This small delay corresponds to a zero delay with respect to simulation time and does not correspond to any real simulation time.

To understand the behavior of this architecture body, consider an event happening on one of the input signals, say input port B at time T. This would cause the concurrent signal assignment statements 1,3, and 6, to be triggered. Their right - hand-side expressions would be evaluated and the corresponding values would be scheduled to be assigned to the target signals at time (T+A). When simulation time advances to (T+A), new values to signals Z(3), BBAR, and Z(1), are assigned. Since the value of BBAR changes, this will in turn trigger signal assignment statements, 2 and 4. Eventually, at time (T+2A), signals Z(0) and Z(2) will be assigned their new values.

The semantics of this concurrent behavior indicate that the simulation, as defined by the language, is event-triggered and that simulation time advances to the next time unit when an event is scheduled to occur. Simulation time could also advance a multiple of delta time units. For example, events may have been scheduled to occur at times 1,3,4,4+A, 5,6,6+A, 6+2A, 6+3A, 10,10+A, 15, 15+A time units.

The after clause may be used to generate a clock signal as shown in the following concurrent signal assignment statement

CLK <= **not** CLK **after** 10 ns;

This statement creates a periodic waveform on the signal CLK with a time period of 20 ns as shown in Fig. 1.10.

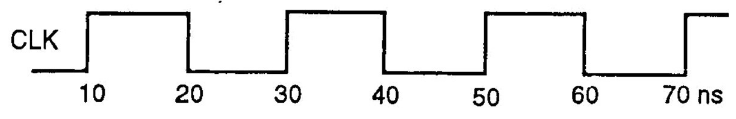


**Fig 1.10: A clock waveform with constant on-off period.**

## Behavioral Style of modeling (Sequential assignment)

In contrast to the styles of modeling described earlier, the behavioral style of modeling specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order. This set of sequential statements, that are specified inside a process statement, do not explicitly specify the structure of the entity but merely specifies its functionality. A process statement is a concurrent statement that can appear within an architecture body. For example, consider the following behavioral model for the DECODER2x4 entity.

```
architecture DEC_SEQUENTIAL of DECODER2x4 is
 begin

 process (A, B, ENABLE)
variable ABAR, BBAR: BIT;
begin

ABAR := not A;     BBAR := not B;                          if (ENABLE = '1')
Then          Z(3) <= not (A and B):              Z(0) <= not (ABAR and BBAR);

Z(2) <= not (A and BBAR);             Z(1 ) <= not (ABAR and B);

Else

Z<="1111";          end if; end process; end;
```

A process statement, too, has a declarative part (between the keywords process and begin), and a statement part (between the keywords begin and end process). The statements appearing within the statement part are sequential statements and are executed sequentially. The list of signals specified within the parenthesis after the keyword process constitutes a sensitivity list and the process statement is invoked whenever there is an event on any signal in this list. In the previous example, when an event occurs on signals A, B, the statements appearing within the process statement are executed sequentially.

Signal assignment statements appearing within a process are called *sequential signal assignmentstatements.* Sequential signal assignment statements, including variable assignment statements, are executed sequentially independent of whether an event occurs on any signals in its right-hand-side expression or not; contrast this with the execution of concurrent signal assignment statements in the dataflow modeling style. In the previous architecture body, if an event occurs on any signal. A, B, statement I which is a variable assignment statement, is executed, then statement 2 is executed, and so on. Execution of the third statement, an if statement, causes control to jump to the appropriate branch based on the value of the signal, ENABLE. If the value of ENABLE is 1', the next four signal assignment statements, 4 through 7, are executed independent of whether A, B, ABAR, or BBAR changed values, and the target signals are scheduled to get their respective values after delta delay. If ENABLE has a value '0', a value of 'V is assigned to each of the elements of the output array, Z. When execution reaches the end of the process, the process suspends itself, and waits for another event to occur on a signal in its sensitivity list.

It is possible to use case or loop statements within a process. The semantics and structure of these statements are very similar to those in other high-level programming languages like C or Pascal. An explicit wait statement can also be used to suspend a process. It can be used to wait for a certain amount of time or to wait until a certain condition becomes true, or to wait until an event occurs on one or more signals. Here is an example of a process statement that generates a clock with a different on-off period. Fig1.11 shows the generated waveform.

```
proces
begin
        CLK  <= '0'

        ; wait for 20
        ns; CLK <=

        '1' ; wait for

        12 ns;

end process;
```

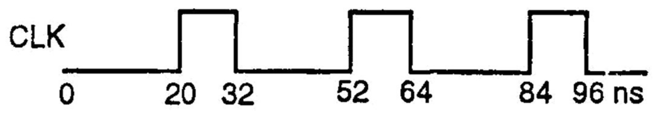


**Fig 1.11: A clock waveform with varying on-off period.**

This process does not have a sensitivity list since explicit wait statements are present inside the process. It is important to remember that a process never terminates. It is always either being executed or in a suspended state. All processes are executed once during the initialization phase of simulation until they get suspended. Therefore, a process with no sensitivity list and with no explicit wait statements will never suspend itself.

A signal can represent not only a wire but also a place holder for a value, that

is, it can be used to model a flip-flop. Here is such an example. Port signal Q models a level-sensitive flip-flop.

```
entityLS_DFF is

    port(Q: out BIT;
        D, CLK:in BIT):
    end LS_DFF;
architectureLS_DFF_BEH        of LS_DFF is
begin
process (D, CLK)
    begin
            if(CLK  = '1')
    then  Q<= D;
            end if;
    end process;
end LS_DFF_BEH;
```

## Delta Delay

A *delta delay is* a very small delay (infinitesimally small). It does not correspond to any real delay and actual simulation time does not advance. This delay models hardware where a minimal amount of time is needed for a change to occur, for example, in performing zero delay simulation. Delta delay allows for ordering of events that occur at the same simulation time during a simulation. Each unit of simulation time can be considered to be composed of an infinite number of delta delays. Therefore, an event always occurs at a real simulation time plus an integral multiple of delta delays. For example, events can occur at 15 ns, 15 ns+IA, 15 ns+2A,15 ns+3A, 22 ns, 22 ns+A, 27 ns, 27 ns+A, and so on.

Consider the AOI_SEQUENTIAL architecture body. Let us assume that an event occurs on input signal D (i.e., there is a change of value on signal D) at simulation time T. Statement 1 is executed first and TEMPI is assigned a value immediately since it is a variable. Statement 2 is executed next and TEMP2 is assigned a value immediately. Statement 3 is executed next which uses the values of TEMPI and TEMP2 computed in statements I and 2, respectively, to determine the new value for TEMPI. And finally, statement 4 is executed that causes signal Z to get the value of its right -hand-side expression after a delta delay, that is, signal Z gets its value only at time T+A; this is shown in Fig. 1.12

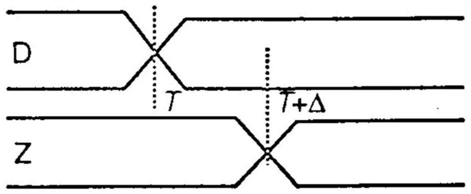


**Fig 1.12: Delta delay.**

Consider the process PZ described in the previous section. If an event occurs on signal A at time *T,* execution of statement I causes VI to get a value, signal Z is then scheduled to get a value at time T+A, and finally statement 3 is executed in which the old value of signal Z is used, that is, its value at time T, not the value that was scheduled to be assigned in statement 2. The reason for this is because simulation time is still at time T and has not advanced to time T+A. Later when simulation time advances to T+A, signal Z will get its new value. This example shows the important distinction between a variable assignment and a signal assignment statement. Variable assignments cause variables to get their values instantaneously while signal assignments cause signals to get their values at a later time (at least a delta delay later).

So far we have seen two examples of sequential statements, the variable assignment statement and the signal assignment statement. Other kinds of sequential statements are described next.

**TEXT / REFERENCE BOOKS**

1. J.Bhaskar, “A VHDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

2. Stphen Brown, "Fundamental of Digital logic with Verilog Design",3rd edition, Tata McGraw Hill, 2008

3. J.Bhaskar, “A Verilog HDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

4. Samir Palnitkar” Verilog HDL: A Guide to Digital Design and Synthesis”, Star Galaxy Publishing; 3rd edition,2005

5. Michael D Ciletti - Advanced Digital Design with VERILOG HDL, 2nd Edition, PHI, 2009.

6. Z Navabi - Verilog Digital System Design, 2nd Edition, McGraw Hill, 2005.

## QUESTION BANK

**PART-A**

1. Distinguish VHDL and Verilog HDL.

2. Data objects are significant in VHDL Justify.

3. List the language elements of VHDL

4. Formulate the syntax of process statement in VHDL.

5. Classify the programming models in VHDL.

6. Wait statement is important in VHDL. Support this statement.

7. Justify how signal declaration is done in VHDL

8. Distinguish concurrent signal assignment and sequential signal assignment.

9. Justify the importance and objects in VHDL

10. List the data types in VHDL

**PART-B**

1. In VHDL, data types and operators are the most significant concept, Explain it.

2. Illustrate the different delay types in VHDL programming.

3. Illustrate the different language elements in VHDL.
4. Discuss concurrent and sequential assignment statements.
5. Identify and illustrate digital system design process and hardware abstraction.

## Behavioral Modeling

This chapter presents the behavioral style of modeling. In this modeling style, the behavior of the entity is expressed using sequentially executed, procedural type code, A process statement is the primary mechanism used to model the procedural type behavior of an entity. This chapter describes the process statement and the various kinds of sequential statements that can be used within a process statement to model such behavior.

Irrespective of the modeling style used, every entity is represented using an entity declaration and at least one architecture body. The first two sections describe these in detail.

An architecture body describes the internal view of an entity. It describes the functionality or the structure of the entity. The syntax of an architecture body is

**architecture** architecture-name **of** entity-name **is**

[ architecture- item-declarations]

**begin**

statements;

**end** architecture-name;

Statements are —>
process-statement
block-statement

concurrent-procedure call concurrent
assertionstatement

concurrent-signal-assignmentstatement

component-instantiation-statement generate-statement

## Process Statement

A process statement contains sequential statements that describe the functionality of a portion of an entity in sequential terms. The syntax of a process statement is

[ process-label:] **process** [(sensitivity-list)]

[process- item-declarations]

**begin**

sequential-statements;

these are ->

variable- assignment-
statement signal-
assignment statement
wait-statement if-
statement

case-
statement
loop-
statement
null-
statement
exit-
statement

next-statement
assertion-statement
procedure-call-
statement return-
statement.

**end process** [process-label];

A set of signals that the process is sensitive to is defined by the sensitivity list. In other words, each time an event occurs on any of the signals in the sensitivity list, the sequential statements within the process are executed in a sequential order, that is, in the order in which they appear (similar to statements in a high-level programming language like C or Pascal). The process then suspends after executing the last sequential statement and waits for another event to occur on a signal in the sensitivity list. Items declared in the item declarations part are available for use only within the process.

The architecture body, AOI _SEQUENTIAL, presented earlier, contains one process statement. This process statement has four signals in its sensitivity list and has one variable declaration. If an event occurs on any of the signals, A, B, C, or D, the process is executed. This is accomplished by executing statement 1 first, then statement 2, followed by statement 3, and then statement 4. After this, the process suspends (simulation does not stop, however) and waits for another event to occur on a signal in the sensitivity list.

**Variable Assignment Statement**

Variables can be declared and used inside a process statement. A variable is assigned a value using the variable assignment statement that typically has the form

variable-object := expression;

The expression is evaluated when the statement is executed and the computed value is assigned to the variable object instantaneously, that is, at the current simulation time. Variables are created at the time of elaboration and retain their values throughout the entire simulation run (like static variables in C high- level programming language). This is because a process is never exited; it is either in an active state, that is, being executed, or in a suspended state, that is, waiting for a certain event to occur. A process is first entered at the start of simulation (actually, during the initialization phase of simulation) at which time it is executed until it suspends because of a wait statement (wait statements are described later in this chapter) or a sensitivity list.

Consider the following process statement.

```
process (A)
        variable EVENTS_ON_A: INTEGER: = 0;
begin
        EVENTS_ON_A              :=
EVENTS_ON_A+1; end process;
```

At start of simulation, the process is executed once. The variable EVENTS_ON_A gets initialized to 0 and then incremented by 1. After that, any time an event occurs on signal A, the process is activated and the single variable assignment statement is executed. This causes the variable EVENTS_ON_A to be incremented. At the end of simulation, variable EVENTS_ON_A contains the total number of events that occurred on signal A plus one.

Here is another example of a process statement.

```
signal A, Z: INTEGER;
. . .
   PZ: process (A)                    --PZ is a label for the
   process.
           variable V1, V2: INTEGER;
   begin
           V1 := A - V2;              --statement 1
           Z  <= - V1;                --statement 2
           V2 := Z+V1 * 2;            -- statement 3
```

**end process** PZ;

If an event occurred on signal A at time T1 and variable V2 was assigned a value, say 10, in statement 3, then when the next time an event occurs on signal A, say at time T2, the value of V2 used in statement 1 would still be 10.

## Signal Assignment Statement

Signals are assigned values using a signal assignment statement The simplest form of a signal assignment statement is

signal-object **<=** expression [**after** delay-value ]**;**

A signal assignment statement can appear within a process or outside of a process. If it occurs outside of a process, it is considered to be a concurrent signal assignment statement. When a signal assignment statement appears within a process, it is considered to be a sequential signal assignment statement and is executed in sequence with respect to the other sequential statements that appear within that process.

When a signal assignment statement is executed, the value of the expression is computed and this value is scheduled to be assigned to the signal after the specified delay. It is important to note that the expression is evaluated at the time the statement is executed (which is the current simulation time) and not after the specified delay. If no after clause is specified, the delay is assumed to be a default delta delay.

Some examples of signal assignment statements are

COUNTER <= COUNTER+ "0010"; - Assign after a delta delay.
PAR <= PAR **xor** DIN **after** 12 ns;

Z <= (AO **and** A1) **or** (BO **and** B1) **or** (CO **and** C1) **after** 6 ns;

## Wait Statement

A process may be suspended by means of a sensitivity list. That is, when a process has a sensitivity list, it always suspends after executing the last sequential statement in the process. The **wait** statement provides an alternate way to suspend the execution of a process. There are three basic forms of the wait statement.

**wait on** sensitivity-list; **wait**
**until** boolean expression;

**wait** **for** time-expression;

They may also be combined in a single wait statement. For example, **wait on** sensitivity-list **until** boolean-expression **for** time-expression-, Some examples of **wait** statements are

```
wait on A, B, C;              -- statement 1
wait until (A = B);           -- statement 2
wait for 10ns;                -- statement 3
wait on CLOCK for 20ns;       -- statement 4

wait until (SUM > 100) for 50 ms; -- statement 5
```

In statement 1, the execution of the wait statement causes the process to suspend and then it waits for an event to occur on signals A, B, or C. Once that happens, the process resumes execution from the next statement onwards. In statement 2, the process is suspended until the specified condition becomes true. When an event occurs on signal A or B, the condition is evaluated and if it is true, the process resumes execution from the next statement onwards, otherwise, it suspends again. When the wait statement in statement 3 is executed, say at time T, the process suspends for 10 ns and when simulation time advances to T+10 ns, the process resumes execution from the statement following the wait statement.

The execution of statement 4 causes the process to suspend and then it waits for an event to occur on the signal CLOCK for 20 ns. If no event occurs within 20 ns, the process resumes execution with the statement following the wait. In the last statement, the process suspends for a maximum of 50 ms until the value of signal SUM is greater than 100. The boolean condition is evaluated every time there is an event on signal SUM. If the boolean condition is not satisfied for 50 ms, the process resumes from the statement following the wait statement.

It is possible for a process not to have an explicit sensitivity list. In such a case, the process may have one or more wait statements. It must have at least one wait statement, otherwise, the process will never get suspended and would remain in an infinite loop during the initialization phase of simulation. It is an error if both the sensitivity list and a wait statement are present within a process. The presence of a sensitivity list in a process implies the presence of an implicit "wait on sensitivity- list" statement as the last statement in the process. An equivalent process statement for the process statement in the AOLSEQUENTIAL architecture body is

```
process                                  -- No sensitivity list.
        variable TEMP1, TEMP2: BIT;
        begin
              TEMP1 :=A and B:
              TEMP2 := C and D;
              TEMP1 := TEMP1 or TEMP2;
               Z<= not TEMP1;
              wait on A, B, C, D;          -- Replaces the sensitivity list.
end process;
```

Therefore, a process with a sensitivity list always suspends at the end of the process and when reactivated due to an event, resumes execution from the first statement in the process.

**If Statement**

An if statement selects a sequence of statements for execution based on the value of a condition. The condition can be any expression that evaluates to a boolean value. The general form of an if statement is

```
if boolean-expressionthen
        sequential-statements
[ elsifboolean-expressionthen        -- elsif clause; if stmt can have 0
or
        sequential-statements ]      -- more elsif clauses.
[ else                               -- else clause.
        sequential-statements ]
end if;
```

The if statement is executed by checking each condition sequentially until the first true condition is found; then, the set of sequential statements associated with this condition is executed. The if statement can have zero or more elsif clauses and an optional else clause. An if statement is also a sequential statement, and therefore, the previous syntax allows for arbitrary nesting of if statements. Here are some examples.

```
                             -- This is a less-than-or-equal-to
if SUM<=100 then             operator.
        SUM := SUM+10;
        end if;
if NICKEL_IN then

        DEPOSITED                 --This"<="   is   a   signal
        <=TOTAL_10;               assignment
elsif DIME_IN then                -- operator.
        DEPOSITED        <=
TOTAL_15;               elsif
QUARTERJN then

        DEPOSITED <= TOTAL_30;
else

end if; DEPOSITED <= TOTAL_ERROR;


if CTRLI='1' then

        if CTRL2 = '0' then

                MUX_OUT<= "0010";

        else

        end if; MUX_OUT<= "0001";


else
```

```
        if CTRL2 ='0' then
                        MUX_OUT <= "1000";
        else end if;

        MUX_OUT <= "0100";
    end if;
```

A complete example of a 2-input nor gate entity using an if statement is shown next.

```
    entity NOR2 is
    port (A, B: in BIT; Z:
    out BIT); end NOR2;

    architecture NOR2 of NOR2 is -- Architecture body
                            can have-- same name
                            as entity.
    begin
        PI: process (A, B)
        constant  RISE_TIME:   TIME
        :=      10      ns;      constant
        FALL_TIME:  TIME  :=  5  ns:
        variable TEMP: BIT;
        begin
            TEMP := A  nor
            B;  if  (TEMP   =
            '1') then
                Z <= TEMP after RISE_TIME;
            else
                Z <= TEMP after FALLJIME;
            end
        if;         end
        process PI;
    end NOR2;
```

**Case Statement**

The format of a case statement is

```
case expression is
        when choices=>sequential-statements          -- branch #1
        when choices=>sequential-statements          -- branch #2
        -- Can have any number of branches.
        [ when others =>sequential-statements ] -- last branch
end case;
```

The case statement selects one of the branches for execution based on the value of the expression. The expression value must be of a discrete type or of a one-dimensional array type. Choices may be expressed as single values, as a range of values, by using I (vertical bar: represents an "or"), or by using the others clause. All possible values of the expression must be covered in the case statement. "The others clause can be used as a choice to cover the "catch-all" values and, if present, must be the last branch in the case statement. An example of a case statement is

```
type WEEK_DAY is (MON, TUE, WED, THU,
FRI, SAT, SUN); type DOLLARS is range 0 to 10;
variable DAY: WEEK_DAY;
variable POCKET_MONEY: DOLLARS;
case DAY is
        when TUE => POCKET_MONEY := 6;       -- branch 1
        when MON I WED =>POCKET_MONEY := 2;      -- branch 2
        when FRI to SUN=>POCKET_MONEY := 7;         -- branch 3
        when others =>POCKET_MONEY := 0;     -- branch 4
end case;
```

Branch 2 is chosen if DAY has the value of either MON or WED. Branch 3 covers the values FRI, SAT, and SUN, while branch 4 covers the remaining value, THU. The case statement is also a sequential statement and it is, therefore, possible to have nested case statements. A model for a 4*1 multiplexer using a case statement is shown next.

```
entity MUX is
        port (A, B, C, D: in BIT; CTRL: in BIT_VECTOR(0 to 1);
                Z: out BIT);
end MUX;

architecture   MUX_BEHAVIOR   of
        MUX          is          constant
        MUX_DELAY: TIME := 10 ns;
```

```
                    begin
PMUX: process (A, B, C, D, CTRL) variable TEMP:
      BIT;
begin
case CTRL is
      when "00" => TEMP
      := A: when "01" => TEMP := B; when "10" =>
      TEMP := C; when "11" => TEMP
      := D;
end case;
Z <= TEMP after
MUX_DELAY; end process
                          PMUX;
                    end MUX_BEHAVIOR;
```

**Null Statement**

The statement

```
null;
```

is a sequential statement that does not cause any action to take place and execution continues with the next statement. One example of this statement's use is in an if statement or in a case statement where for certain conditions, it may be useful or necessary to explicitly specify that no action needs to be performed.

**Loop Statement**

**A loop** statement is used to iterate through a set of sequential statements. The syntax of a loop statement is

```
[ loop-label : ] iteration- scheme
Loop
Sequential -statements
end loop [loop-label] ;
```

There are three types of iteration schemes. The first is the for-iteration scheme that has the form **for** identifier **in** range

An example of this iteration scheme is

```
FACTORIAL := 1;
for NUMBER in 2 to N loop
      FACTORIAL := FACTORIAL *
NUMBER; end loop;
```

The body of the for loop is executed (N-1) times, with the loop identifier, NUMBER, being incremented by I at the end of each iteration. The object NUMBER is implicitly declared within the for loop to belong to the integer type whose values are in the range 2 to N. No explicit declaration for the loop identifier is, therefore, necessary. The loop identifier, also, cannot be assigned any value inside the for loop. If another variable with the same name exists outside the for loop, these two variables are treated separately and the variable used inside the for loop refers to the loop identifier.

The range in a for loop can also be a range of an enumeration type such as

```
type HEXA is ('0', '1', '2', '3', '4', ' 5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E',

'F'):

. . .

for NUM in HEXA'('9') downto HEXA'('0') loop

        -- NUM will take values in type HEXA from '9' through '0'.

        . . .
end loop;


for CHAR in HEXA loop

        -- CHAR will take all values in type HEXA from '0' through 'F'.

        . . .
end loop;
```

Notice that it is necessary to qualify the values being used for NUM [e.g., HEXA'('9')] since the literals '0' through '9' are overloaded, once being defined in type HEXA and the second time being defined in the predefined type CHARACTER. Qualified expressions are described in Chap. 10.

The second form of the iteration scheme is the while scheme that has the form

```
while     boolean-expression
```

An example of the while iteration scheme is

```
J:=0;SUM:=10;

WH-LOOP: while J < 20 loop - This loop has a label,
        WH_LOOP. SUM := SUM * 2;

        J:=J+

3; end loop;
```

The statements within the body of the loop are executed sequentially and repeatedly as long as the loop condition, J < 20, is true. At this point, execution continues with the statement following the loop statement.

The third and final form of the iteration scheme is one where no iteration scheme is specified. In this form of **loop** statement, all statements in the loop body are repeatedly executed until some other action causes it to exit the loop. These actions can be caused by an **exit** statement, **a next** statement, or a return statement. Here is an example.

```
SUM:=1;J:=0;

          L2: loop -- This loop also has a label.
          J:=J+21;

          SUM := SUM* 10;

          exit when SUM > 100;

end loop L2;          -- This loop label, if present, must be the same

                      -- as the initial loop label.
```

In this example, the exit statement causes the execution to jump out of loop L2 when SUM becomes greater than 100. If the exit statement were not present, the loop would execute indefinitely.

**Exit Statement**

The exit statement is a sequential statement that can be used only inside a loop. It causes execution to jump out of the innermost loop or the loop whose label is specified. The syntax for an exit statement is

**exit** [loop-label] [ **when** condition ]**:**

If no loop label is specified, the innermost loop is exited. If the when clause is used, the specified loop is exited only if the given condition is true, otherwise, execution continues with the next statement. An alternate form for loop L2 described in the previous section is

```
SUM := 1; J

:= 0; L3: loop

          J:=J+21;

          SUM := SUM*

          10; if (SUM >

          100) then

                    exit L3;    -- "exit;" also would have been sufficient.

          end
if; end loop
L3;
```

**Next Statement**

The next statement is also a sequential statement that can be used only inside a loop. The syntax is the same as that for the exit statement except that the keyword next replaces the keyword exit. Its syntax is

**next** [loop-label][ **when** condition ];

The **next** statement results in skipping the remaining statements in the current iteration of the specified loop and execution resumes with the first statement in the next iteration of this loop. If no loop label is specified, the innermost loop is assumed.

In contrast to the exit statement that causes the loop to be terminated (i.e., exits the specified loop), the next statement causes the current loop iteration of the specified loop to be prematurely terminated and execution resumes with the next iteration. Here is an example.

```
for J in 10 downto 5 loop
    if      (SUM          <
            TOTAL_SUM)
            then   SUM   :=
            SUM +2;
    elsif       (SUM
            = TOTAL_SUM)
            then next;
    else
    end if; null;

            K:=K
```

+1; **end loop;**

When the next statement is executed, execution jumps to the end of the loop (the last statement, K := K+1, is not executed), decrements the value of the loop identifier, J, and resumes loop execution with this new value of J.

The next statement can also cause an inner loop to be exited. Here is such an example.

```
L4: for K in 10 downto 1 loop

    --statements
    section 1 L5: loop

        -- statements section 2

        next L4 when WR_DONE = '1';

        --statements
    section 3end loop L5;

        --statements section 4

end loop L4;
```

When WR_DONE = 1' becomes true, statements sections 3 and 4 are skipped and execution jumps to the beginning of the next iteration of loop L4. Notice that the loop L5 was terminated because of the result of **next** statement.

**Assertion Statement**

Assertion statements are useful in modeling constraints of an entity. For example, you may want to check if a signal value lies within a specified range, or check the setup and hold times for signals arriving at the inputs of an entity. If the check fails, an error is reported. The syntax of an assertion statement is

```
assert          boolean- expression

[ reportstring- expression]

[ severityexpression]:
```

If the value of the boolean expression is false, the report message is printed along with the severity level. The expression in the severity clause must generate a value of type SEVERTTY_ LEVEL (a predefined enumerated type in the language with values NOTE, WARNING, ERROR, and FAILURE). The severity level is typically used by a simulator to initiate appropriate actions depending on its value. For example, if the severity level is ERROR, the simulator may abort the simulation process and provide relevant diagnostic information. At the very least, the severity level is displayed.

Here is a model of a D-type rising-edge-triggered flip-flop that uses assertion statements to check for setup and hold times.

```
entity DFF is

    port (D, CK: in BIT: Q, NOTQ:
```

```
out BIT); end DFF;

architecture CHECK_TIMES of DFF
        is constant HOLD_TIME:
        TIME: = 5 ns; constant
        SETUP_TIME: TIME:= 3 ns;

begin

        process (D, CK)

                variable LastEventOnD, LastEventOnCk: TIME;

        begin

                --Check for hold
                time: if D'
                EVENT then

                        assert (NOW = 0ns) or

                                ((NOW - LastEventOnCk) >=
                                HOLD_TIME) report "Hold time too
                                short!"

                                severity
                        FAILURE;LastEventO
                        nD := NOW;

                end if;

                -- Check for setup time:

                   if (CK = '1') and CK'EVENT then

                   assert (NOW = 0ns) or

                                ((NOW - LastEventOnD) >= SETUP_TIME)
                                        report "Setup time too short!"

                                severity
                        FAILURE;

                        LastEventOnCk := NOW;
                end if;
```

```
-- Behavior of FF:
if   (CK    =    '1'    )    and
        CK'EVENT      then
        Q<=D;

        NOTQ <= not D;

end if;

end process;

end CHECK_TIMES;
```

EVENT is a predefined attribute of a signal and is true if an event (a change of value) occurred on that signal at the time the value of the attribute is determined. Attributes are described in greater detail in Chap. 10. NOW is a predefined function that returns the current simulation time. In the previous example, the process is sensitive to signals D and CK. When an event occurs on either of these signals, the first if statement is executed. This checks to see if an event occurred on D. If so, the assertion is checked to make sure that the difference between the current simulation time and the last time an event occurred on signal CK is greater than a constant HOLD_TIME delay. If not, a report message is printed and the severity level is returned to the simulator. Similarly, the next if statement checks for the setup time. The last if statement describes the latch behavior of the D-type flip-flop. The setup and hold times have been modeled as constants in this example.

```
package PACK1 is

        constant MIN_PULSE: TIME := 5
        ns; constant PROPAGATE_DLY:
        TIME := 10 ns;

end PACK1;


use
WORK.PACK1.a
ll; entity INV is

        port (A: in BIT; NOT_A:

out BIT): end INV;


architecture    CHECK_INV

of INV is begin
        process (A)

                variable LastEventOnA: TIME := 0 ns;

        begin

                assert (NOW = 0ns) or

                        ((NOW  -  LastEventOnA)  >=
                MIN_PULSE) report "Spike detected on
                input of inverter" severity WARNING;
```

LastEventOnA := NOW:

NOT_A <= **not** **A**
**after**

```
        PROPAGATE_DLY; end process;
    end CHECK_INV;
```

Some other examples of assertion statements are

```
    assert (DATA <= 255)
        report "Data out of range.';

    assert (CLK = '0') or (CLK = '1'); --CLK is of type ('X', '0', 'I ', 'Z').
```

In the last assertion statement example, the default report message "Assertion violation" is printed. The default severity level is ERROR if the severity clause is not specified as in the previous two examples.

## Dataflow Modeling

This chapter presents techniques for modeling the dataflow of an entity. A dataflow model specifies the functionality of the entity without explicitly specifying its structure. This functionality shows the flow of information through the entity, which is expressed primarily using concurrent signal assignment statements and block statements. This is in contrast to the behavioral style of modeling described in the previous chapter, in which the functionality of the entity is expressed using procedural type statements that are executed sequentially. This chapter also describes resolution functions and their usage.

## Concurrent Signal Assignment Statement

One of the primary mechanisms for modeling the dataflow behavior of an entity is by using the concurrent signal assignment statement. An example of a dataflow model for a 2-input or gate, shown in Fig.2.1, follows.

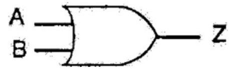


**Fig 2.1 An or gate**

```
    entity OR2 is
        port (signal A, B: in BIT;
        signal Z:out BIT);
        end OR2;
    architecture OR2 of
    OR2 is begin
        Z <= A or B after 9 ns;
        end OR2;
```

The architecture body contains a single concurrent signal assignment statement that represents the dataflow of the or gate. The semantic interpretation of this statement is that whenever there is an event (a change of value) on either signal A or B (A and

B are signals in the expression for Z), the expression on the right is evaluated and its value is scheduled to appear on signal Z after a delay of 9 ns. The signals in the expression, A and B, form the "sensitivity list" for the signal assignment statement. There are two other points to mention about this example. First, the input and output ports have their object class "signal" explicitly specified in the entity declaration. If it were not so, the ports would still have been signals, since this is the default and the only object class that is allowed for ports. The second point to note is that the architecture name and the entity name are the same. This is not a problem since architecture bodies are considered to be secondary units while entity declarations are primary units and the language allows secondary units to have the same names as the primary units.

An architecture body can contain any number of concurrent signal assignment statements. Since they are concurrent statements, the ordering of the statements is not important. Concurrent signal assignment statements are executed whenever events occur on signals that are used in their expressions. An example of a dataflow model for a 1-bit full-adder, whose external view is shown in Fig. 5.2, is presented next.

```
entity FULL_ADDER is

        port (A, B, CIN: in BIT; SUM,
COUT: out BIT); end FULL_ADDER;


architecture FULL_ADDER of FULL_ADDER is

begin SUM<=(A xor B) xor CIN after 15 ns;

COUT <= (A and B) or (B and CIN) or (CIN and A) after 10 ns;

end FULL_ADDER;
```

**Concurrent versus Sequential Signal Assignment**

The signal assignment statements can also appear within the body of a process statement. Such statements are called sequential signal assignment statements, while signal assignment statements that appear outside of a process are called concurrent signal assignment statements. Concurrent signal assignment statements are event triggered, that is, they are executed whenever there is an event on a signal that appears in its expression, while sequential signal assignment statements are not event triggered and are executed in sequence in relation to the other sequential statements that appear within the process. To further understand the difference between these two kinds of signal assignment statements, consider the following two architecture bodies.

```
architecture      SEQ_SIG_ASG      of
        FRAGMENT1 is  -  A,  B  and Z
        are signals.

begin
```

```
          process (B)
          begin -- Following are sequential signal assignment
end;           statements:A<=B;

               Z<=
          A;        end
          process;


architecture CON_SIG_ASG of FRAGMENT2 is
begin -- Following are concurrent signal assignment

          statements:A<=B;

          Z<=A;

end;
```

In architecture SEQ_SIG_ASG, the two signal assignments are sequential signal assignments. Therefore, whenever signal B has an event, say at time T, the first signal assignment statement is executed and then the second signal assignment statement is executed, both in zero time. However, signal A is scheduled to get its new value of B only at time T+Δ (the delta delay is implicit), and Z is scheduled to be assigned the old value of A (not the value of B) at time T+Δ also.

In architecture CON_SIG_ASG, the two statements are concurrent signal assignment statements. When an event occurs on signal B, say at time T, signal A gets the value of B after delta delay, that is, at time T+Δ. When simulation time advances to T+Δ, signal A will get its new value and this event on A (assuming there is a change of value on signal A) will trigger the second signal assignment statement that will cause the new value of A to be assigned to Z after another delta delay, that is, at time T+2Δ. The delta delay model is explored in more detail in the next section.

Aside from the previous difference, the concurrent signal assignment statement is identical to the sequential signal assignment statement.

For every concurrent signal assignment statement, there is an equivalent process statement with the same semantic meaning. The concurrent signal assignment statement:

```
CLEAR <= RESET or PRESET
after 15 ns; -- RESET and PRESET
are signals.
```

is equivalent to the following process statement:.

```
proces
begin

     CLEAR   <=   RESET   or PRESET

     after   15   ns;   wait   on   RESET,
        PRESET;
    end process;
```

An identical signal assignment statement (this is now a sequential signal assignment) appears in the body of the process statement along with a wait statement whose sensitivity list comprises of signals used in the expression of the concurrent signal assignment statement.

**Conditional Signal Assignment Statement**

The conditional signal assignment statement selects different values for the target signal based on the specified, possibly different, conditions (it is like an if statement). A typical syntax for this statement is

```
Target - signal <=[ waveform-elements when condition else]
                  [ waveform- elements
                 When condition else ]
                  . . .
                  waveform-elements;
```

The semantics of this concurrent statement are as follows. Whenever an event occurs on a signal used either in any of the waveform expressions (recall that a waveform expression is the value expression in a waveform element) or in any of the conditions, the conditional signal assignment statement is executed by evaluating the conditions one at a time. For the first true condition found, the corresponding value (or values) of the waveform is scheduled to be assigned to the target signal. For example,

```
Z <= IN0 after 10ns when S0 = '0' and S1 = '0' else
     IN1 after 10ns when S0 = '1' and S1 = '0' else
     IN2 after 10ns when S0 = '0' and S1 = '1' else
     IN3 after 10 ns;
```

In this example, the statement is executed any time an event occurs on signals IN0, IN1, IN2, IN3, S0, or S1. The first condition (S0='0' and S1='0') is checked; if false, the second condition (S0='1' and S1='0') is checked; if false, the third condition is checked; and so on. Assuming S0='0' and S1='1', then the value of IN2 is scheduled to be assigned to signal Z after 10 ns.

For a given conditional signal assignment statement, there is an equivalent process statement that has the same semantic meaning. Such a process statement has exactly one if statement and one wait statement within it. The signals in the sensitivity list for the wait statement is the union of signals in all the waveform expressions and the signals referenced in all the conditions. The equivalent process statement for this conditional signal assignment statement example is

```
proces
begin

if S0 = '0' and S1 = '0'

        then  Z<=  IN0

        after 10 ns;

elsif S0='1'and S1='0' then

Z<= IN1 after 10ns;

        elsif S0='0' and S1 = '1' then Z<= IN2 after 10 ns;

        else end if;

Z<= INS after 10 ns;

wait on IN0, IN1, IN2, IN3, S0, S1;

end process;
```

**Selected Signal Assignment Statement**

The selected signal assignment statement selects different values for a target signal based on the value of a select expression (it is like a case statement). A typical syntax for this statement is

```
with expression select —This is the select
        expression.target-signal <= waveform-
        elements when choices,

                waveform-elements when choices,

                …

                waveform-elements when choices;
```

The semantics of a selected signal assignment statement are very similar to that of the conditional signal assignment statement. Whenever an event occurs on a signal in the select expression or on any signal used in any of the waveform expressions, the statement is executed. Based on the value of the select expression that matches the choice value specified, the value (or values) of the corresponding waveform is scheduled to be assigned to the target signal. Note that the choices are not evaluated in sequence. All possible values of the select expression must be covered by the choices that are specified not more than once. Values not covered explicitly may be covered by an "others" choice, which covers all such values. The choices may be a logical "or" of several values or may be specified as a range of values.

Here is an example of a selected signal assignment statement.

```
type OP is (ADD, SUB,
MUL, DIV); signal
OP_CODE: OP;

. . .

with OP_CODE select
```

Z <= A+B **after** ADD_PROP_DLY **when** ADD, A **-** B **after** SUB_PROP_DLY **when** SUB,

A * B **after** MUL_PROP_DLY **when** MUL, A / B **after** DIV_PROP_DLY **when** DIV;

In this example, whenever an event occurs on signals, OP_CODE, A, or B, the statement is executed. Assuming the value of the select expression, OP_CODE, is SUB, the expression "A - B" is computed and its value is scheduled to be assigned to signal Z after SUB_PROP_DLY time.

For every selected signal assignment statement, there is also an equivalent process statement with the same semantics. In the equivalent process statement, there is one case statement that uses the select expression to branch. The list of signals in the sensitivity list of the wait statement comprises of all signals in the select expression and in the waveform expressions. The equivalent process statement for the previous example is

```
 proces
 begin
case OP_CODE is
        when   ADD   =>   Z<=   A   +B  after ADD_PROP_DLY;
        when  SUB  =>Z<= A-B   after   SUB_PROP_DLY;
        when MUL =>Z<= A * B after MUL_PROP_DLY;
        when DIV => Z <= A /B after DIV_PROP_DLY;
         end case;
         wait on OP_CODE,
 A, B; end process;
```

## Structural Modeling

This chapter describes the structural style of modeling. An entity is modeled as a set of components connected by signals, that is, as a netlist. The behavior of the entity is not explicitly apparent from its model. The component instantiation statement is the primary mechanism used for describing such a model of an entity.

An Example Consider the circuit shown in Fig. 2.2 and its VHDL structural model.

```
 entity GATING is
         port (A, CK, MR, DIN: in BIT; RDY,
 CTRLA: out BIT); end GATING;

 architecture STRUCTURE_VIEW of
         GATING is component AND2
                port (X, Y: in BIT; Z: out BIT);
                end component;
        component DFF
                 port (D, CLOCK:   in   BIT;
                  Q, QBAR: out BIT);
                 end component;
         component NOR2
                 port (A, B: in BIT; Z: out BIT);
                 end component;
                 signal SI, S2: BIT;
 begin

D1: DFF port map (A, CK, SI, S2);
```

A1: AND2 **port map** (S2, DIN, CTRLA);

N1: NOR2 **port map** (SI, MR, RDY);

**end** STRUCTURE_VIEW

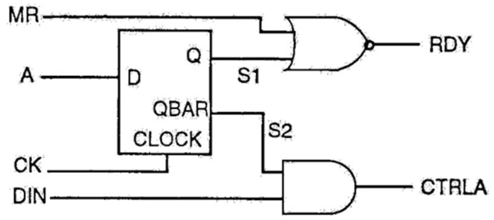


**Fig 2.2: A circuit generating control signals**

Three components, AND2, DFF, and NOR2, are declared. These components are instantiated in the architecture body via three component instantiation statements, and the instantiated components are connected to each other via signals SI and S2. The component instantiation statements are concurrent statements, and therefore, their order of appearance in the architecture body is not important. A component can, in general, be instantiated any number of times. However, each instantiation must have a unique component label; as an example, A1 is the component label for the AND2 component instantiation.

**Component Declaration**

A component instantiated in a structural description must first be declared using a component declaration. A component declaration declares the name and the interface of a component. The interface specifies the mode and the type of ports. The syntax of a simple form of component declaration is

> **component** component-name
>
> **port** (list-of-interface- ports ) **;**
>
> **end component;**

The component-name may or may not refer to the name of an already ex-isfing entity in a library. If it does not, it must be explicitly bound to an entity; otherwise, the model cannot be simulated.

The list-of-interface-ports specifies the name, mode, and type for each port of the component in a manner similar to that specified in an entity declaration. "The names of the ports may also be different from the names of the ports in the entity to which it may be bound (different port names can be mapped in a configuration). An entity of the same name as that of the component already exists and that the name, mode, and type of each port matches the corresponding ones in the component. Some examples of component declarations are

> **component** NAND2
>
> **port** (A, B: **in** MVL; Z:
>
> **out** MVL); **end component;**
>
> **component MP**
>
> **port** (CK, RESET, RON, WRN: **in** BIT;
>
> DATA_BUS: **inout** INTEGER **range** 0 **to** 255;
> ADDR_BUS: **in** BIT_VECTOR (15 **downto** 0));

**end component;**

**component** RX

**port** (CK, RESET, ENABLE, DATAIN,

RD: **in** BIT;

DATA_OUT: **out** INTEGER **range** 0 **to** (2**8 - 1); PARITY_ERROR,

FRAME_ERROR, OVERRUN_ERROR: **out** BOOLEAN);

**end component;**

**Component Instantiation**

A component instantiation statement defines a subcomponent of the entity in which it appears. It associates the signals in the entity with the ports of that subcomponent. A format of a component instantiation statement is component-label: component-name port **map**(association-list) ',

The component-label can be any legal identifier and can be considered as the name of the instance. The component-name must be the name of a component declared earlier using a component declaration. The association-list associates signals in the entity, called actuals, with the ports of a component, called locals. An actual must be an object of class signal. Expressions or objects of class variable or constant are not allowed. An actual may also be the keyword open to indicate a port that is not connected. There are two ways to perform the association of locals with actuals:

**1.** positional association,

**2.**named association.

In positional association, an association-list is of the form

actuali, actualg, actual3, . . ., actual

Each actual in the component instantiation is mapped by position with each port in the component declaration. That is, the first port in the component declaration corresponds to the first actual in the component instantiation, the second with the second, and so on. Consider an instance of a NAND2 component.

--Component declaration:

**component** NAND2

**port** (A, B: **in** BIT; **Z:**

**out** BIT); **end component;**

--Component instantiation:

N1: NAND2 **port map** (S1, S2, S3);

N1 is the component label for the current instantiation of the NAND2 component. Signal S1 (which is an actual) is associated with port A (which is a local) of the NAND2 component, S2 is associated with port B of the NAND2 component, and S3 is associated with port Z. Signals S1 and S2 thus provide the two input values to the NAND2 component and signal S3 receives the output value from the component. The ordering of the actuals is, therefore, important.

If a port in a component instantiation is not connected to any signal, the keyword open can be used to signify that the port is not connected. For example,

N3: NAND2 **port map** (S1, **open,** S3);

The second input port of the NAND2 component is not connected to any signal. An input port may be left open only if its declaration specifies an initial value. For the previous component instantiation statement to be legal, a component declaration for NAND2 may appear like

**component** NAND2

**port** (A, B: **in** BIT := '0'; **Z: out** BIT);

1 Both A and B have an initial value of '0'; however, only

2 the initial value of B is necessary in this case.

**end component;**

A port of any other mode may be left unconnected as long as it is not an unconstrained array. In named association, an association- list is of the form

locale => actual1, local2 => actual2, ..., localn => actualn

For example, consider the component NOR2 in the entity GATING described in the first section. The instantiation using named association may be written as

N1: NOR2 **port map** (B=>MR, Z=>RDY, A=>S1);

In this case, the signal MR (an actual), that is declared in the entity port list, is associated with the second port (port B, a local) of the NOR2 gate, signal RDY is associated with the third port (port Z) and signal S1 is associated with the first port (port A) of the NOR2 gate. In named association, the ordering of the associations is not important since the mapping between the actuals and locals are explicitly specified. An important point to note is that the scope of the locals is restricted to be within the port map part of the instantiation for that component; for example, the locals A, B, and Z of component NOR2 are relevant only within the port map of instantiation of component NOR2.

For either type of association, there are certain rules imposed by the language. First, the types of the local and the actual being associated must be the same. Second, the modes of the ports must conform to the rule that if the local is readable, so must the actual and if the local is writable, so must the actual. Since a signal locally declared is considered to be both readable and writable, such a signal may be associated with a local of any mode. If an actual is a port of mode in, it may not be associated with a local of mode out or inout; if the actual is a port of mode out, it may not be associated with a local of mode in or inout; if the actual is a port of mode inout, it may be associated with a local of mode in, out, or inout.

**Generate Statements**

Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement.

There are two forms of the generate statement.

**1.** Using the for-generation scheme, concurrent statements can be replicated a predetermined number of times.

**2.** With the if-generation scheme, concurrent statements can be conditionally selected for execution.

The generate statement is interpreted during elaboration, and therefore, has no simulation semantics associated with it. It resembles a macro expansion. The generate statement provides for a compact description of regular structures such as memories, registers, and counters.

The format of a generate statement using the for-generation scheme is

generate-label: **for** generale-identifier in discrete- range

**generate**

concurrent-statements

**end generate**[ generate-label];

The values in the discrete range must be globally static, that is, they must be computable at elaboration time. During elaboration, the set of concurrent statements are replicated once for each value in the discrete range. These statements can also use the generate identifier in their expressions and its value would be substituted during elaboration for each replication. There is an implicit declaration for the generate identifier within the generate statement, and therefore, no declaration for this identifier is required. The type of the identifier is defined by the discrete range. Consider the following representation of a 4-bit full-adder, shown in Fig. 2.3, using the generate statement.

```
entity FULL_ADD4 is

    port (A, B: in BIT_VECTOR(3 downto 0); CIN: in BIT;
        SUM: out BIT_VECTOR(3 downto 0); COUT: out
        BIT);

end FULL_ADD4:

architecture    FOR_GENERATE    of
    FULL_ADD4    is    component
    FULL_ADDER

        port (A, B, C: in BIT; COUT,
    SUM: out BIT); end component;

    signal CAR: BIT_VECTOR(4 downto 0);

begin

    CAR(0) <= CIN;

    GK: for K in 3 downto 0 generate

    FA: FULL_ADDER port map (CAR(K),
                  A(K),        B(K),
                  CAR(K+1),SUM(
                  K));
    end    generate
    GK;COUT   <=
    CAR(4);

end FOR_GENERATE
```

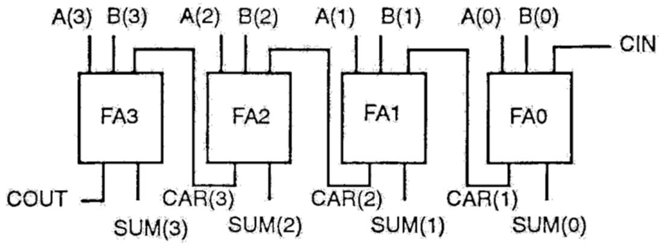


**Fig.2.3: A 4-bit full-adder.**

After elaboration, the generate statement is expanded to

```
FA(3): FULL_ADDER port map (CAR(3), A(3), B(3), CAR(4),
SUM(3));

FA(2): FULL_ADDER port map (CAR(2), A(2), B(2), CAR(3),
SUM(2));

FA(1): FULL_ADDER port map (CAR(1), A(1), B(1), CAR(2),
SUM(1));

FA(0): FULL_ADDER port map (CAR(0), A(0), B(0), CAR(1),
SUM(0));
```

Components in a generate statement can be bound to entities using a generate block configuration. A block configuration is defined for each range of generate labels. Here is an example of such a binding using a configuration declaration.

```
configuration GENERATE_BIND of FULL_ADD4 is
    use WORK.all; -- Example of a declaration in the

                  -- configuration declarative part.

        for FOR_GENERATE

         -- An architecture body block configuration.

        forGK(1)      --A generate block configuration.

            for FA: FULL_ADDER

                use configuration

                WORK.FA_HA_CON;

            end for;

        end for;

        for GK(2 to 3)

            for FA: FULL_ADDER - No explicit binding.

-- Use defaults, i.e., use entity FULL_ADDER-- in working library.
```

```
                    end for;
                end
                for; for
                GK(0)
                    for FA: FULL_ADDER
                        use entity
                    WORK.FULL_ADDER(FA_DA
                    TAFLOW); end for;
                end for;
            end for;
        end GENERATE_BIND;
```

There are three generate block configurations, one each for GK(1), GK(2 to 3), and for GK(0). Each of these block configurations define the bindings for the components valid for that generate index.

The body of the generate statement can also have other concurrent statements. For example, in the previous architecture body, the component instantiation statement could be replaced by signal assignment statements like this

```
G2: for M in 3 downto 0 generate
    SUM(M) <= (A(M) xor B(M)) xor
    CAR(M); CAR(M+1 ) <= (A(M) and
    B(M)) and CAR(M);
end generate G2;
```

The second form of the generate statement uses the if-generation scheme. The format for this type of generate statement is

```
genarate-label: H expression
    generate concurrent-
    statements
end generate [generete-label] ;
```

The if-generate statement allows for conditional selection of concurrent statements based on the value of an expression. This expression must be a globally static expression, that is, the value must be computable at elaboration time.

Here is an example of a 4-bit counter, that is modeled using the if-generate statement.

```
entity COUNTER4 is
port (COUNT, CLOCK: in BIT;
Q: buffer BIT_VECTOR (0 to 3));
end COUNTER4;
```

```
architecture IF_GENERATE of
        COUNTER4 is component
        D_FLIP_FLOP

                port (D, CLK: in BIT; Q:

        out BIT); end component;

        begin

        GK: for K in 0 to 3

        generate GKO: if K= 0 generate

DFF: D_FLIP_FLOP port map (COUNT, CLOCK, Q(K));

end generate GK0;

GK1_3: if K > 0 generate

        DFF: D_FLIP_LOP port map (Q(K-1), CLOCK, Q(K)); end
generate GK1_3;

end generate

        GK; end IF_GENERATE;
```

**Guarded Signals**

A guarded signal is a special type of a signal that is declared to be of a register or a bus kind in its declaration. A general form of a signal declaration is

```
signal list-of-signals: resolution-function

signal-typesignal-kind [: = expression];
```

A guarded signal must be a resolved signal, that is, it must have a resolution function associated with it. Also, the signal can only be assigned values under the control of a guard expression, for example, using a guarded assignment (guarded option used in a concurrent signal assignment statement). This implies that guarded signals can only be assigned values within block statements.

A guarded signal behaves differently from other signals in that when the guard expression is false, the driver to the guarded signal becomes disconnected after a specific time, called the disconnect time. On the other hand, in an unguarded signal, if the guard expression is false, any new events on the signals appearing in the expression do not influence the value of the target signal; the driver continues to drive the target signal with the old value. To understand this difference better, consider the following guarded block BL

```
architecture GUARDED_EX of EXAMPLE is
        signal GUARD_SIG: WIRED_OR
        BIT register; signal
        UNGUARD_SIG: WIRED_AND BIT;

begin

        B1: block ( guard-
        expression )begin
        GUARD_SIG                    <= guardedexpression1 ;
        UNGUARD_SIG                  <=guardedexpression2;
                      end block B1;
                      end GUARDED_EX;
```

Transforming the guarded signal assignment statement into its equivalent process statement, the block B1 now looks like this

```
                B1:   block   (   guard- expression )begin
                      process
                      begin
                            if GUARD then
                                  GUARD_SIG <=expression1;
                            else
                                  GUARD_SIG<=null;
 end if;
                wait on signals-in- expressioni1,GUARD;
                end process;
process
 begin
                            If GUARD then
                                  UNGUARD_SIG <= expression2;
                            end if;
                            wait on signals-in- expressions,
                            GUARD; end process;
                 end block B1;
```

The process statement for the guarded signal, GUARD_ SIG, has an explicit signal assignment statement that disconnects its driver, while there is no such statement for the unguarded signal, UNGUARD_SIG. As this example shows, a driver of a guarded signal can be explicitly disconnected by assigning a null value to the signal. Such a statement is called a disconnection statement.

Let us now explore the differences between a register and a bus signal. A bus signal represents a hardware bus in that when all drivers to the signal become disconnected (as might be the case on a real hardware bus), the value of the signal is determined by calling the resolution function with all the drivers off. A register signal, on the other hand, models a storage component (that is multiply driven) in which if all drivers to the signal become disconnected, the resolution function is not called and the value of the last active driver is retained. With a bus signal, the previous value is lost. Also, bus signals may either be ports of an entity or locally declared signals, whereas register signals can only be locally declared signals.

The disconnect time for a guarded signal can be specified using a disconnection specification. The syntax of a disconnection specification is

**disconnect** guarded-signal-name: signal-type **after** time-expression;

This is an example of a disconnection specification.

**disconnect** GUARD_SIG: BIT **after** 8 ns;

This implies that the driver of signal GUARD_SIG will get disconnected 8 ns after the corresponding GUARD goes false.

The disconnection specification is useful in modeling decay times, for example, capacitance delay on buses. An alternate way of specifying disconnect time is by assigning a value null to the signal in a disconnection statement as shown.

S1 <= **null after** 10 ns;

This statement specifies that the driver of SI will be disconnected after 10 ns. Thereafter, this driver does not contribute to the resolved value of the signal. However, such a statement can appear only as a sequential statement and the target signal must be a guarded signal.

Here is a more comprehensive example.

**use** WORK.RF.PACK.**all;**

-- Package RF_PACK contains functions WIRED_AND and WIRED_OR. **entity** GUARDED_SIGNALS **is**

**port** (CLOCK: **in** BIT;

N: **in** INTEGER);

**end;**

```
architecture EXAMPLE of GUARDED_SIGNALS is
        signal      REG_SIG:      WIRED_AND
        INTEGER   register;
  signal   BUS_SIG: WIRED_OR INTEGER bus;
  disconnect REG_SIG:   INTEGER   after   50   ns;
  disconnect BUS_SIG: INTEGER after  20 ns;
  begin
  BX:   block  (CLOCK='1'   and   (not CLOCK'STABLE))
       begin
               REG_SIG   <=   guarded   N after 15 ns;
               BUS_SIG <= guarded N after 10 ns;
end block
BX; end
```

**TEXT / REFERENCE BOOKS**

1. J.Bhaskar, "A VHDL Primer", Prentice Hall of India Limited. 3rd edition 2004
2. Stphen Brown, "Fundamental of Digital logic with Verilog Design",3rd edition, Tata McGraw Hill, 2008
3. J.Bhaskar, "A Verilog HDL Primer", Prentice Hall of India Limited. 3rd edition 2004
4. Samir Palnitkar" Verilog HDL: A Guide to Digital Design and Synthesis", Star Galaxy Publishing; 3rd edition,2005
5. Michael D Ciletti - Advanced Digital Design with VERILOG HDL, 2nd Edition, PHI, 2009.
6. Z Navabi - Verilog Digital System Design, 2nd Edition, McGraw Hill, 2005.

## QUESTION BANK

### PART-A

1. Justify how arrays are declared in VHDL
2. Classify the types of delays in VHDL.
3. List the styles of description in VHDL.
4. Develop a VHDL program for 2*1 multiplexer.
5. List the operators in VHDL.
6. Develop a VHDL program for 1*2 decoder.
7. Define Data objects in VHDL
8. Develop VHDL code for 2-bit adder.

**PART-B**

1. Distinguish between dataflow modeling and behavioral modeling of VHDL
2. Develop a VHDL code for full adder circuit in different styles of description.
3. Discuss with example behavioral modeling of VHDL
4. Develop a VHDL code for encoder and decoder circuit
5. Develop a VHDL code for Flip flop circuits
6. Distinguish between structural modeling and dataflow modeling of VHDL
7. Discuss with an example structural modeling of VHDL

# UNIT – III - INTRODUCTION TO VERILOG HDL– SECA1602

## INTRODUCTION TO VERILOG HDL

Verilog HDL is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level to the switch level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete electronic digital system, or anything in between. The digital system can be described hierarchically and timing can be explicitly modeled within the same description.

**Typical Design Flow**

A typical design flow for designing VLSI IC circuits is shown in Figure 2.1. Un shaded blocks show the level of design representation; shaded blocks show processes in the design flow. The design flow shown in Figure 3.1 is typically used by designers who use HDLs. In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues.

**Behavioral descriptions are often written with HDLs**

The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level netlist. A gate level netlist is a description of the circuit in terms of gates and connections between them. Logic synthesis tools ensure that the gate-level netlist meets timing, area, and power specifications. The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on a chip.

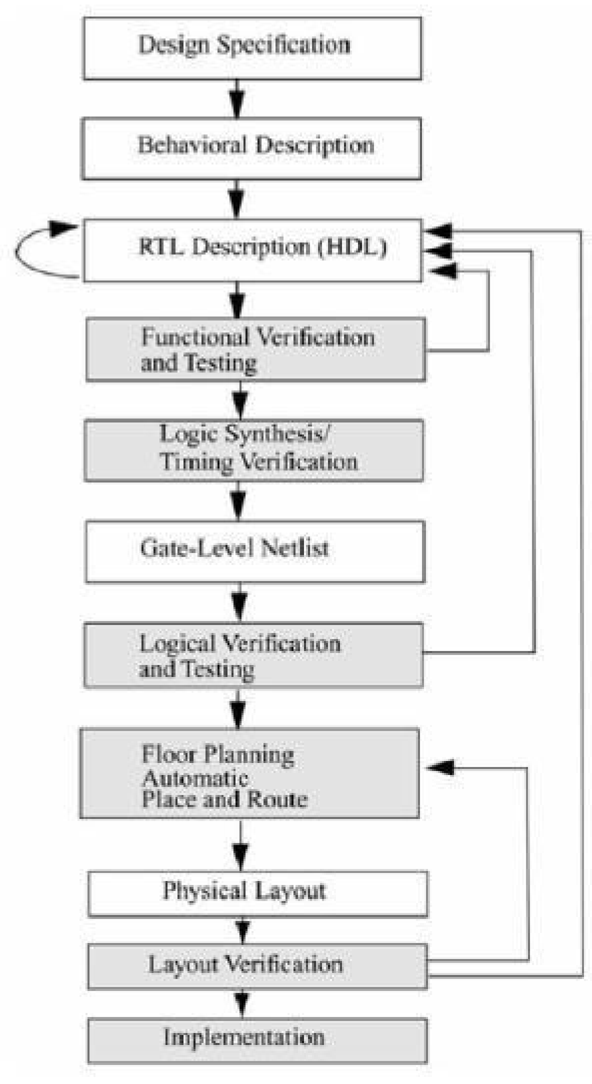

Design Specification
Behavioral Description
RTL Description (HDL)
Functional Verification and Testing
Logic Synthesis/ Timing Verification
Gate-Level Netlist
Logical Verification and Testing
Floor Planning Automatic Place and Route
Physical Layout
Layout Verification
Implementation

Figure 3.1: Typical Design Flow

**Design Methodologies**

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub- blocks until we come to leaf cells, which are the cells that cannot further be divided. Figure 3.2 shows the top-down design process.

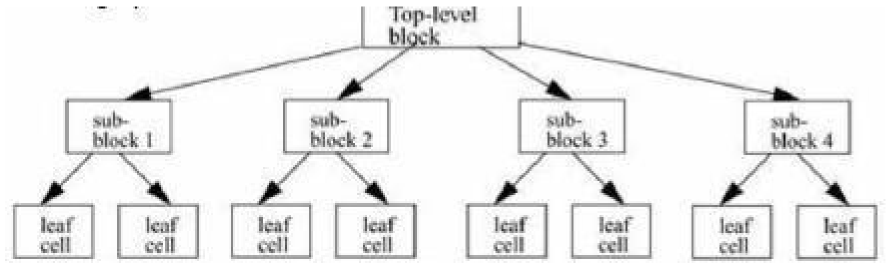


Figure 3.2: Top-down Design Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Figure 3.3 shows the bottom-up design process

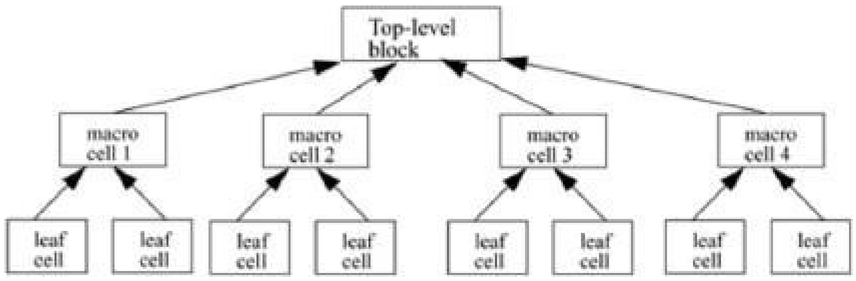


Figure 3.3: Bottom-up Design Methodology

Levels for design description

Verilog supports designing at many different levels of abstraction. Three of them are very important:

- Behavioral level
- Register-Transfer Level
- Gate Level

**Behavioral Level**

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

**Register-Transfer Level**

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers.An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

### Gate Level

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.

## Language Elements

### Identifiers

Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($). Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore.

They cannot start with a digit or a $ sign

```
reg value; // reg is a keyword; value is an identifier

input clk; // input is a keyword, clk is an identifier
```

### Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment
/* This is a multiple line comment */
/* This is /* an illegal */ comment */
/* This is //a legal comment */
```

### Format

Verilog HDl is case sensitive. Identifiers differing only in their case are distinct. Verilog HDL, is free format, constructs may be written across multiple lines, or on one line. White space (newline, tab, and space characters) has no special significance.

**System Tasks and Functions**

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form $<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.

**Compiler Directives**

Compiler directives are provided in Verilog. All compiler directives are defined by using the ‘<keyword> construct. We deal with the two most useful compiler directives.

‘define

The ‘define directive is used to define text macros in Verilog.

The Verilog compiler substitutes the text of the macro wherever it encounters a ‘<macro_name>. This is similar to the #define construct in C. The defined constants or text macros are used in the Verilog code by preceding them with a ‘ (back tick).

```
//define a text macro that defines default word
size //Used as ’WORD_SIZE in the code ’define
WORD_SIZE 32
```

‘include

The ‘include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This works similarly to the #include in the C programming language. This directive is typically used to include header files, which typically contain global or commonly used definitions.

Example ‘include Directive

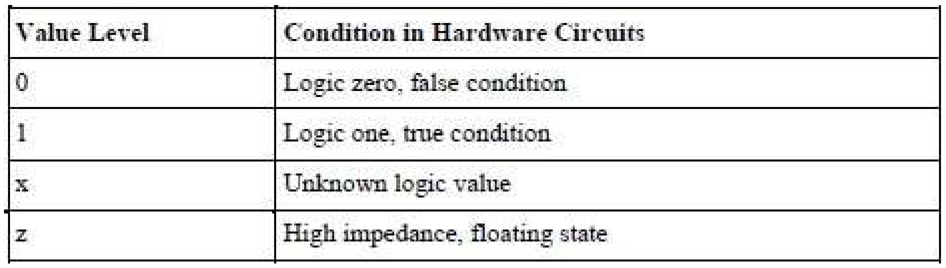
```
// Include the file header.v, which contains declarations in the
// main verilog file design.v.
’include header.v
...
...
<Verilog code in file design.v>
...
...
```

Two other directives, ‘ifdef and ‘timescale, are used frequently.

**Value set**

Verilog supports four values and eight strengths to model the functionality of real hardware.

Strength levels



| Value Level | Condition in Hardware Circuits |
|---|---|
| 0 | Logic zero, false condition |
| 1 | Logic one, true condition |
| x | Unknown logic value |
| z | High impedance, floating state |

**Data types**

Verilog HDL has two groups of data types

**(i) Net type**

A net type represents a physical connection between structural elements. Its value is determined from the value of its drivers such as a continuous assignment or a gate output. If no driver is connected to a net, the net defaults to a value of z.

**(ii) Variable type**

A variable type represents an abstract data storage element. It is assigned values only within an always statement or an initial statement, and its value is saved from one assignment to the next. A variable type has a default value of x.

**Net types**

Here are the different kinds of nets that belong to the net data type wire
tri wor

trior wand triand trireg tri1 tri0

supply0

supply1

**Variable types**

There are five different kinds of variable types

reg

integer

time

real

realti

me

**Register**

Registers represent data storage elements. Registers retain value until another value is placed onto them. Register data types are commonly declared by the keyword reg. The default value for a reg data type is x.

Example of Register

```
reg reset; // declare a variable reset that can hold its value
begin
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = 1'b0; // after 100 time units reset is de asserted.
end
```

**Integer**

An integer is a general-purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting. The default width for an integer is the host-machine word size, which is implementation- specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial counter = -1; // A negative one is stored in the counter
```

## Real

Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 106). Real numbers cannot have a range declaration, and their default value is 0. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
real delta; // Define a real variable called delta
initial

begin
delta = 4e10; // delta is assigned in scientific notation delta = 2.13;
// delta is assigned a value 2.13
end

integer i; // Define an integer i initial
i = delta; // i gets the value 2 (rounded value of 2.13)
```

## Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation specific but is at least 64 bits. The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time initial
save_sim_time = $time; // Save the current simulation time
```

## Arrays

Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types. Multi-dimensional arrays can also be declared with any number of dimensions. Arrays of nets can also be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net. Arrays are accessed by

<array_name>[<subscript>]. For multi-dimensional arrays, indexes need to be provided for each dimension.

```
integer count[0:7]; // An array of 8 count variables
reg bool[31:0]; // Array of 32 one-bit boolean register variables time
chk_point[1:100]; // Array of 100 time checkpoint variables

reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide
```

**Parameters**

Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types and sizes can also be defined.

parameter port_id = 5; // Defines a constant port_id

parameter cache_line_width = 256; // Constant defines width of cache line parameter

signed [15:0] WIDTH; // Fixed sign and range for parameter WIDTH

**Expressions**

An expression is formed using operands and operators. An expression can be used wherever a value is expected.

**Operands**

Operands can be constants, integers, real numbers, nets, registers, times, bitselect (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls.

integer count, final_count;

final_count = count + 1;//count is an integer operand

real a, b, c;

c = a - b; //a and b are real operands

reg [15:0] reg1,

reg2; reg [3:0]

reg_out;

reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are //part-select register operands reg

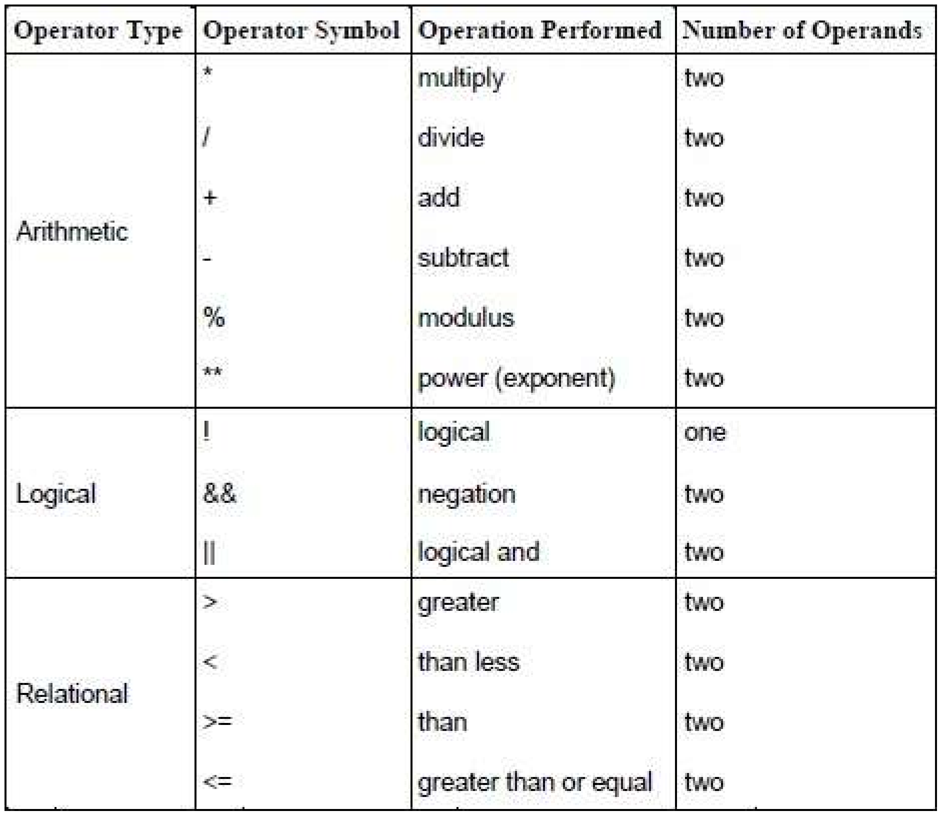
ret_value;

ret_value = calculate_parity(A, B);//calculate_parity is a //function type operand

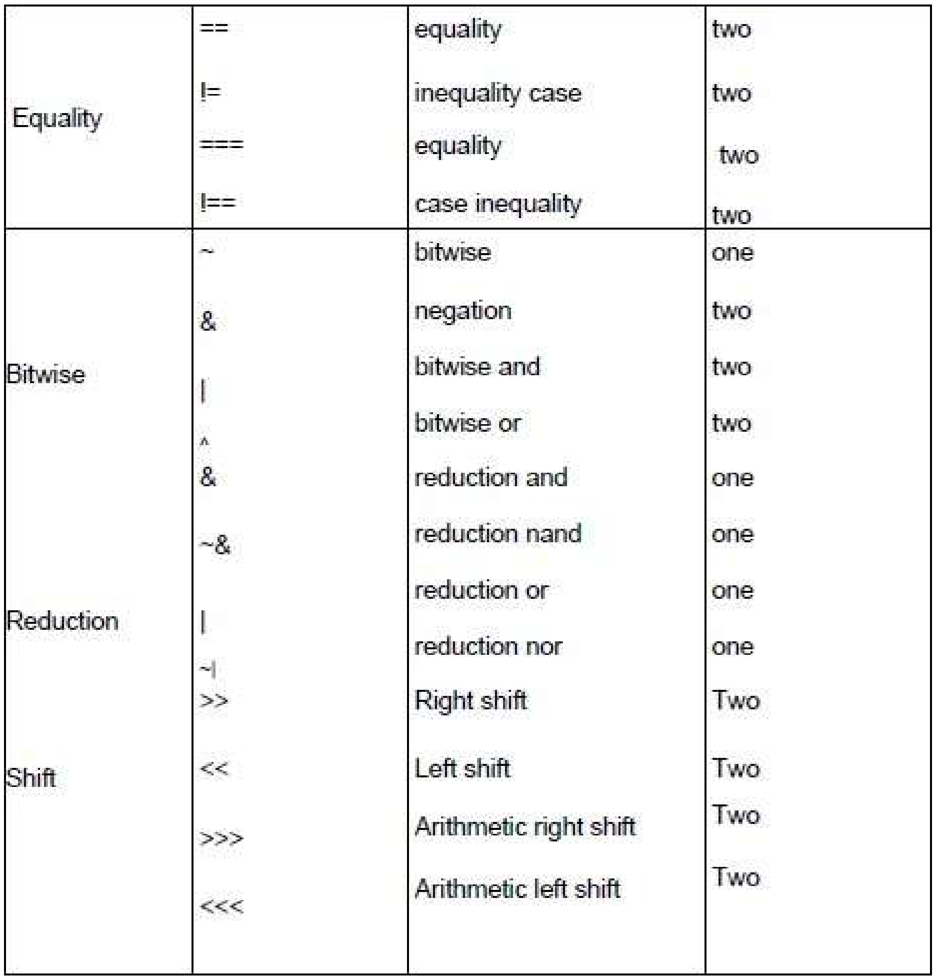
**Operator Types**

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The table 2.1 shows the complete listing of operator symbols classified by category.

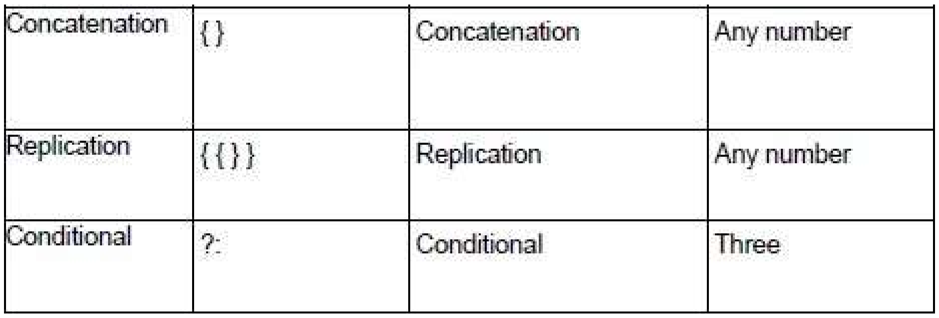
Table 3.1 Operators



| Operator Type | Operator Symbol | Operation Performed | Number of Operands |
|---|---|---|---|
| Arithmetic | * | multiply | two |
| | / | divide | two |
| | + | add | two |
| | - | subtract | two |
| | % | modulus | two |
| | ** | power (exponent) | two |
| Logical | ! | logical | one |
| | && | negation | two |
| | \|\| | logical and | two |
| Relational | > | greater | two |
| | < | than less | two |
| | >= | than | two |
| | <= | greater than or equal | two |



| | | | |
|---|---|---|---|
| Equality | == | equality | two |
| | != | inequality case | two |
| | === | equality | two |
| | !== | case inequality | two |
| | ~ | bitwise | one |
| | & | negation | two |
| Bitwise | \| | bitwise and | two |
| | ^ | bitwise or | two |
| | & | reduction and | one |
| | ~& | reduction nand | one |
| | | reduction or | one |
| Reduction | \| | reduction nor | one |
| | ~\| | | |
| | >> | Right shift | Two |
| Shift | << | Left shift | Two |
| | >>> | Arithmetic right shift | Two |
| | <<< | Arithmetic left shift | Two |



| | | | |
|---|---|---|---|
| Concatenation | { } | Concatenation | Any number |
| Replication | { { } } | Replication | Any number |
| Conditional | ?: | Conditional | Three |

**Module**

The basic unit of description in Verilog is the module. A module describes the functionality or structure of a design and also describes the ports through which it communicates externally with other modules. The structure of a design is described using switch-level primitives, gate- level primitives and user-defined primitives; data flow behavior of a design is described using continuous assignments; sequential behavior is described using procedural constructs. A module can also be instantiated inside another module.

module module_name

(port_list );

Declarations:

reg, wire,

    parameter, input, output, inout, function ,

    task, …. Statements :

    Initial

    statement

    Always

    statement

    Module

    instantiation

    Gate

    instantiation

    UDP

    instantiation

    Continuous

    assignment

    Generate

    statement

    end module

## TEXT / REFERENCE BOOKS

1. J.Bhaskar, “A VHDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

2. Stphen Brown, "Fundamental of Digital logic with Verilog Design",3rd edition, Tata McGraw Hill, 2008

3. J.Bhaskar, “A Verilog HDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

4. Samir Palnitkar” Verilog HDL: A Guide to Digital Design and Synthesis”, Star Galaxy Publishing; 3rd edition,2005

5. Michael D Ciletti - Advanced Digital Design with VERILOG HDL, 2nd Edition, PHI, 2009.

6. Z Navabi - Verilog Digital System Design, 2nd Edition, McGraw Hill, 2005.

## QUESTION BANK

## PART-A

1. Justify how arrays are declared in Verilog HDL.
2. Classify the types of delays in Verilog HDL.
3. List the parameters in Verilog HDL.
4. Formulate the value set of Verilog HDL.
5. List the language elements of Verilog HDL.
6. Develop a Verilog HDL program for 2*4 decoder.
7. Define operands in Verilog HDL
8. Distinguish between inter assignment delay and intra assignment delay.
9. Justify the importance of module in Verilog HDL
10. Develop Verilog HDL code for 2-bit subtraction.

## PART-B

1. Compare procedural constructs and assignments
2. Illustrate the different language elements in Verilog HDL.
3. Operators in verilog are of different types. Support the statement.
4. Develop a Verilog HDL code for encoder and decoder circuit
5. Illustrate the different delay types in Verilog HDL programming.

# UNIT – IV - STYLES OF MODELLING – SECA3007

## STYLES OF MODELING

### Gate Level Modeling:

The Built-in Primitive Gates:

The following built-in primitive gates are available in Verilog HDL.

I. Multiple-input gates: and, nand, or,nor,xor,xnor
II. Multiple-output gates: buf, not
III. Tristate gates: buflfO, bufifl, notifO, notifl
IV. Pull gates: pullup, pulldown 70 Multiple-input
V. MOS switches: cmos, nmos, pmos, rcmos, rnmos, rpmos
VI. Bidirectional switches: tran, tranifO, tranifl, rtran, rtranifO,rtranifl

A gate can be used in a design using a gate instantiation. Here is a simple format of a gate instantiation.

gate_type[ instance_name ] ( terml , term2 , . . . , termN);

Note that the instance_name is optional; gate type is one the gates listed earlier. The terms specify the nets and registers connected to the terminals of the gate. Multiple instances of the same gate type can be specified in one construct. The syntax for this is the following.

gate_type

[ instance_namel ] ( termll , terml2 , . . . , termlN),

[ instance_name2 ] ( term.21 , term22,. . . , term2N),

………

[ instance_nameM ] ( termMl , termM2 , . . . , terwMN);

**Multiple-input Gates:**

The multiple-input built-in gates are: and nand nor or xorxnor. These logic gates have only one output and one or more inputs. Here is the syntax of a multiple-input gate instantiation.

multiple_input_gate_type I instance_name ] ( OutputA , Input 1 , Input2,..., InputN );

The first terminal is the output and all others are the inputs. Here are some examples. The logic diagrams are shown in figure 4.1.

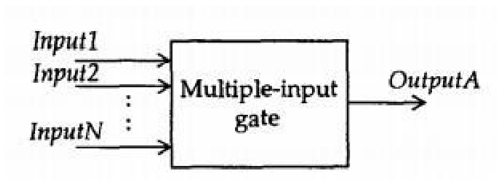


Figure 4.1: Multiple Input Gates

and A1 (Outl, Inl, In2);

and RBX (Sty, Rib, Bro, Qit, Fix) ;

xor (Bar, Bud[0],Bud[l],Bud[2]),

(Car, Cut[0], Cut[l]),

(Sar, Sut[2], Sut[l], Sut[0], Sut[3]);

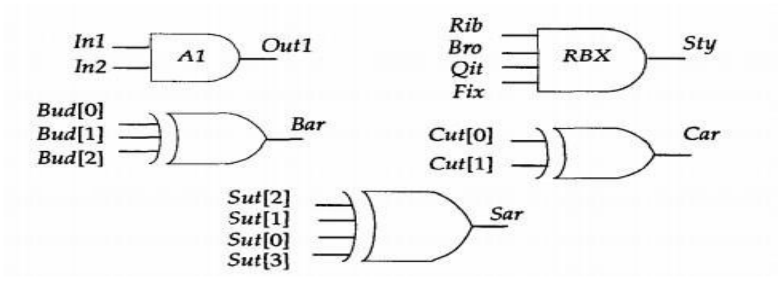
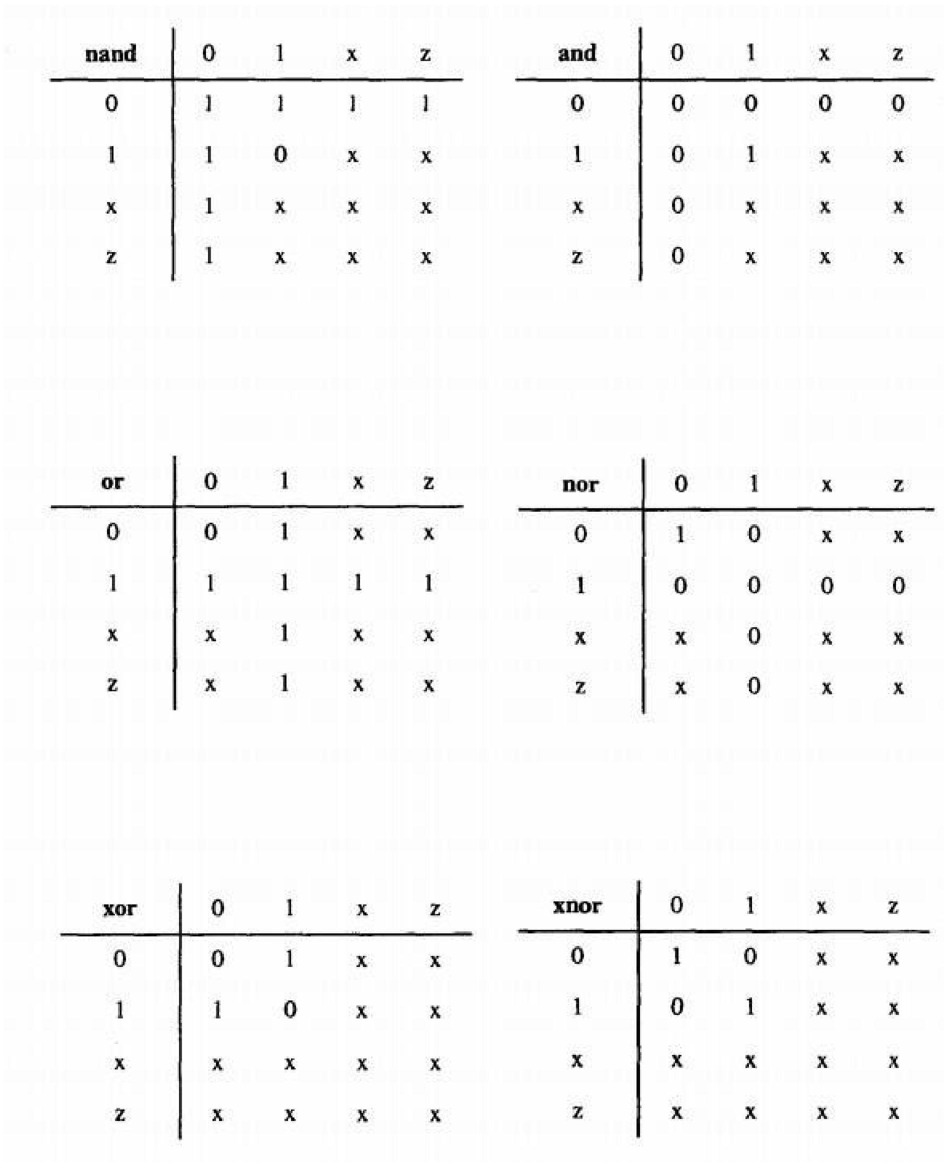


Figure 4.2: Multiple Input Gate examples

The first gate instantiation is a 2-input and gate with instance name Al, output Outl and with two inputs, Inl and Inl. The second gate instantiation is a 4-input and gate with instance name RBX, output Sty and four inputs, Rib, Bro, Qit and Fix. The third gate instantiation is an example of anxor gate with no instance name. Its output is Bar and it has three inputs, Bud[0],Bud[1] and Bud[2]. Also, this instantiation has two additional instances of the same type.

The truth tables for these gates are shown next. Notice that a value z at an input is handled like an x; additionally, the output of a multiple-input gate can never be a z.



| nand | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| x | 1 | x | x | x |
| z | 1 | x | x | x |

| and | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

| or | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | x | 1 | x | x |

| nor | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 0 | x | x |
| 1 | 0 | 0 | 0 | 0 |
| x | x | 0 | x | x |
| z | x | 0 | x | x |

| xor | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 0 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

| xnor | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 0 | x | x |
| 1 | 0 | 1 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

Figure 4.3: Truth table Multiple input gates

**Multiple-output Gates:**

The multiple-output gates are: buf & not

These gates have only one input and one or more outputs. The basic syntax for this gate instantiation is:

multiple_output_gate_type

[instance_name] (Outl, 0ut2 , . . . , OutN, InputA );

The last terminal is the input; all remaining terminals are the outputs.

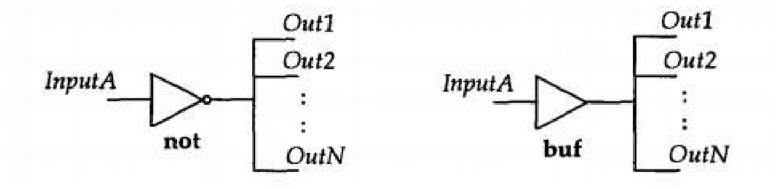


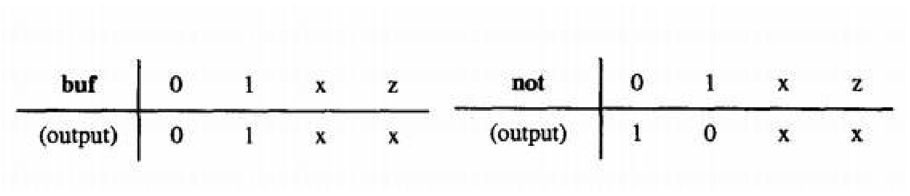
Figure 4.4: Multiple Output Gates

Here are some examples.

bufBl [Fan[0], Fan[l], Fan[2],Fan[3],Clk);

notNl {PhA, PhB, Ready);

In the first gate instance, Clk is the input to the buf gate; this gate instance has four outputs, Fan[0] through Fan[3]. In the second gate instance, Ready is the only input to the not gate. This instance has two outputs, PhA and PhB. The truth table for these gates are shown next.



| buf | 0 | 1 | x | z |
|---|---|---|---|---|
| (output) | 0 | 1 | x | x |

| not | 0 | 1 | x | z |
|---|---|---|---|---|
| (output) | 1 | 0 | x | x |

Figure 4.5: Truth table of Multiple output Gates

**Tristate Gates:**

The tristate gates are: bufifO, bufifl, notifO, notifl

These gates model three-state drivers. These gates have one output, one data input and one control input. Here is the basic syntax of a tristate gate instantiation.

tristate_gate[ instance_name] (OutputA, InputB, ControlC);

The first terminal OutputA is the output, the second terminal InputB is the data input, and the control input is ControlC. Depending on the control input, the output can be driven to the high-impedance state, that is, to value z. For a bufifO gate, the output is z if control is 1,else data is transferred to output. For a bufifl gate, output is a z if control is 0.Fora notifOgate, output is at z if control is at 1 else output is the invert of the input data value. For notifl gate, output is at z if control is at 0.

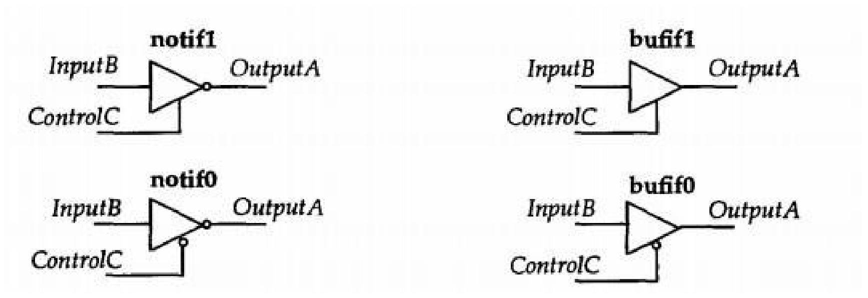
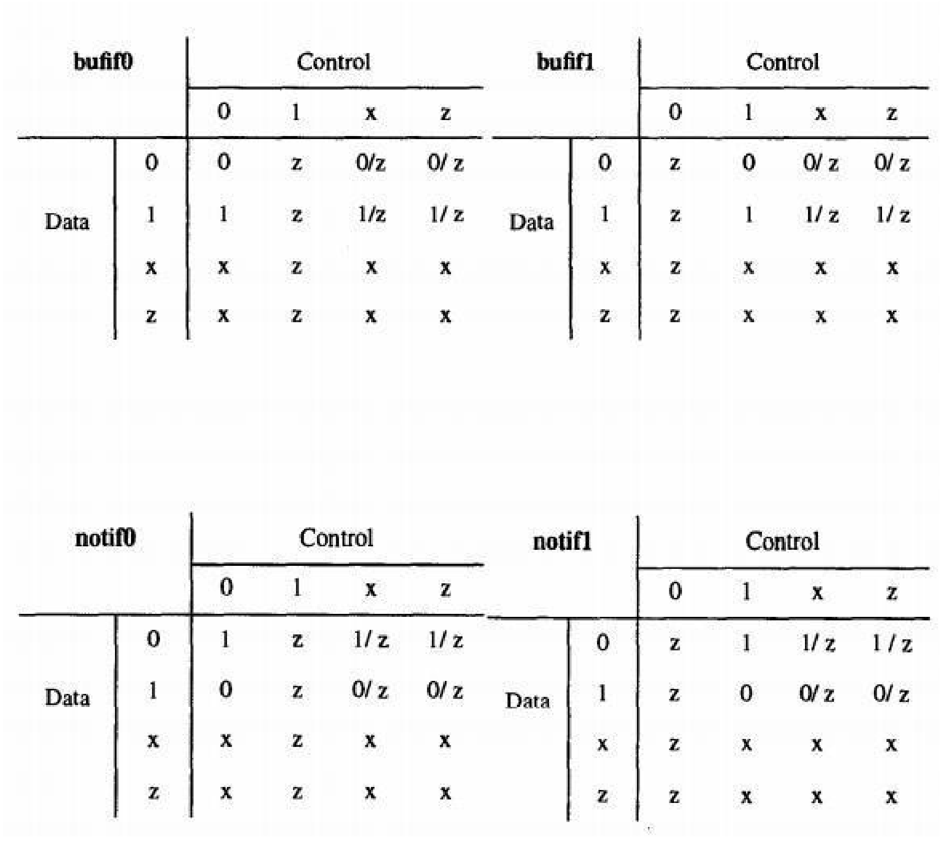


Figure 4.6: Tristate Gates

Here are some examples.

bufifl BF1 [Dbus, MemData, Strobe);

notifO NT2 {Addr, Abus, Probe); The bufifl gate BF1 drivesthe output Dbus to high- impedance state when Strobe is 0, elseMemData is transferred to Dbus. In the second instantiation, when Probe is 1,Addr is in high-impedance state, else Addr gets the inverted value of Abus. The truth tables for these gates are shown next. Some entries in the table indicate alternate entries. For example,0/z indicates that the output can either be a 0 or a z depending on the strengths of the data and control values.



| bufif0 | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| | 0 | 0 | z | 0/z | 0/z |
| Data | 1 | 1 | z | 1/z | 1/z |
| | x | x | z | x | x |
| | z | x | z | x | x |

| bufif1 | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| | 0 | z | 0 | 0/z | 0/z |
| Data | 1 | z | 1 | 1/z | 1/z |
| | x | z | x | x | x |
| | z | z | x | x | x |

| notif0 | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| | 0 | 1 | z | 1/z | 1/z |
| Data | 1 | 0 | z | 0/z | 0/z |
| | x | x | z | x | x |
| | z | x | z | x | x |

| notif1 | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| | 0 | z | 1 | 1/z | 1/z |
| Data | 1 | z | 0 | 0/z | 0/z |
| | x | z | x | x | x |
| | z | z | x | x | x |

Figure 4.7: Truth table for Tristate Gates

**Pull Gates:**

The pull gates are: pullup & pulldown

These gates have only one output with no inputs. A pull up gate places a 1 on its output. A pulldown gate places a 0 on its output. A gate instantiation is of the form:

pull_gate I instance_name ] ( Outputs );

The terminal list of this gate instantiation contains only one output. Here is an example.

pullup PUP (Pwr);

This pullup gate has instance name PUP with output Pwr tied to 1.

**MOS Switch:**

The MOS switches are: cmos, pmos, nmos, rcmos, rpmos, rnmos.

These gates model unidirectional switches, that is, data flows from input to output and the data flow can be turned off by appropriately setting the control input(s).

The pmos(p-type MOS transistor), nmos (n-type MOS transistor), rnmos ('r' stands for resistive) and rpmos switches have one output, one input and one control input. The basicsyntax for an instantiation is:

gate_type[ instance_name ] ( Outputs , InputB , ControlC );

The first terminal is the output, the second terminal is the input and the last terminal is the control. If controlis 0 for nmos and rnmos switches and 1 for pmos and rpmos switches, the switch is turned off, that is, output has value z; if control is 1, data at input passes to output; see Figure 5-5. The resistive switches (rnmos and rpmos) have a higher impedance(resistance) between the input and output terminals as compared to the non-resistive switches (nmos and pmos). Thus when data passes from input to output, a reduction in strength occurs for resistive switches.

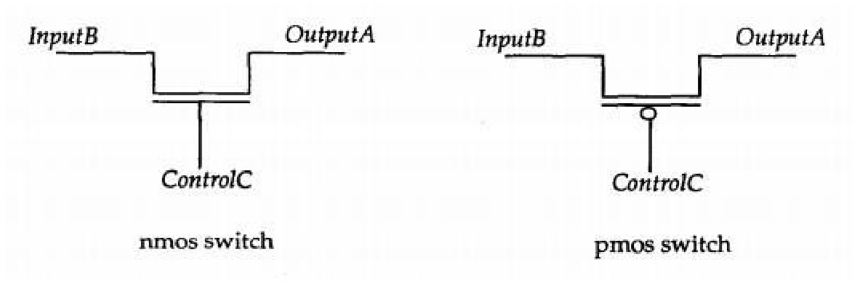


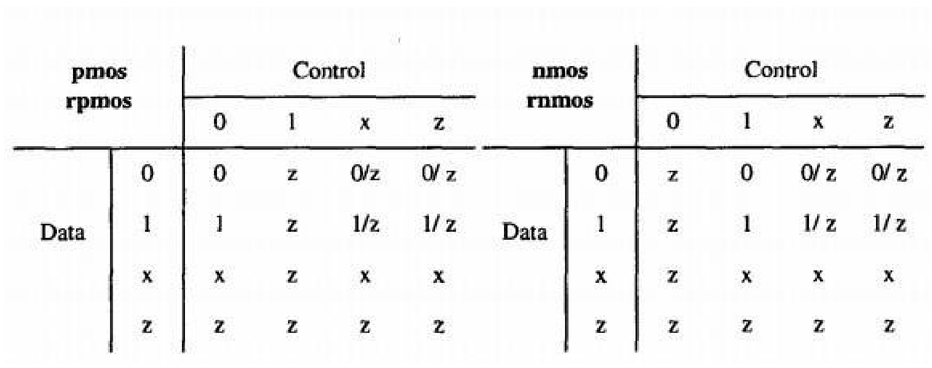
Figure 4.8: nMOS and pMOS switches

Here are some examples.

pmos P1 {BigBus, SmallBus, GateControl);

rnmos RN1 [ControlBit, ReadyBit, Hold);

The first instance instantiates a pmos switch with instance name P1. The input to the switch is SmallBus and the output is BigBus and the control signal is Gate Control. The truth tables for these switches are shown next. Some entries in the table indicate alternate entries. For example, 1/z indicates that the output can be either 1 or z depending on the input and control.



| **pmos rpmos** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | 0 | z | 0/z | 0/ z |
| | 1 | 1 | z | 1/z | 1/ z |
| | x | x | z | x | x |
| | z | z | z | z | z |

| **nmos rnmos** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | z | 0 | 0/ z | 0/ z |
| | 1 | z | 1 | 1/ z | 1/ z |
| | x | z | x | x | x |
| | z | z | z | z | z |

Figure 4.9: Truth table of MOS switches

The CMOS (Complementary MOS) and rcmos (resistive version of cmos) switches have one data output, one data input and two control inputs. The syntax for instantiating these two switches is of the form:

(r)cmos [ instance_name ] ( OutputA , InputB , NControl , PControl);

The first terminal is the output, the second is the input, the third is the n channel control input and the fourth terminal is the p-channel control input. A cmos (rcmos) switch behaves exactly like a combination of a pmos (rpmos) and an nmos (rnmos) switch with common outputs and common inputs.

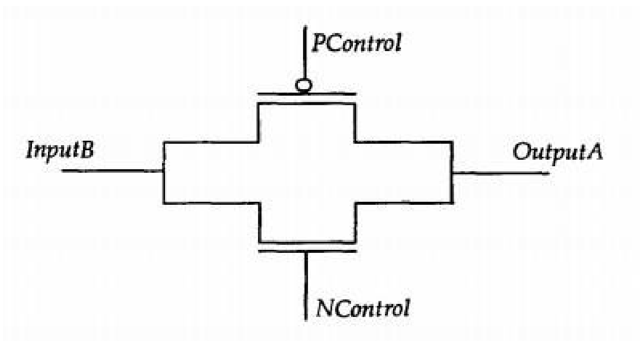


Figure 4.10: (r)cmos switch

**Bidirectional Switch:**

The bidirectional switches are: - tran, rtran, tranifO, rtranifO, tranifl, rtranifl

These switches are bidirectional, that is, data flows both ways and there is no delay when data propagates through the switches. The last four switches can be turned off by setting a control signal appropriately. The tran and rtran switches cannot be turned off. The syntax for instantiating a tran or a rtran (resistive version of tran) switch is:

```
(r)tran [ instance_name ] ( SignalA , SignalB);
```

The terminal list has only two terminals and data flows unconditionally both ways, that is, from SignalA to SignalB and vice versa. The syntax for instantiating the other bidirectional switches is:

```
gate type[ instance_name ] ( SignalA , SignalB , ControlC);
```

The first two terminals are the bidirectional terminals, that is, data flows from SignalA to SignalB and vice versa. The third terminal is the control signal. If ControlC is 1for tranifO and rtranifO, and 0 for tranifl and rtranifl, the bidirectional data flow is disabled. For the resistive switches(rtran, rtranifO and rtranifl), the strength of the signal reduces when it passes through the switch.

Exam
ples:

**4 X 1 Multiplexer:**

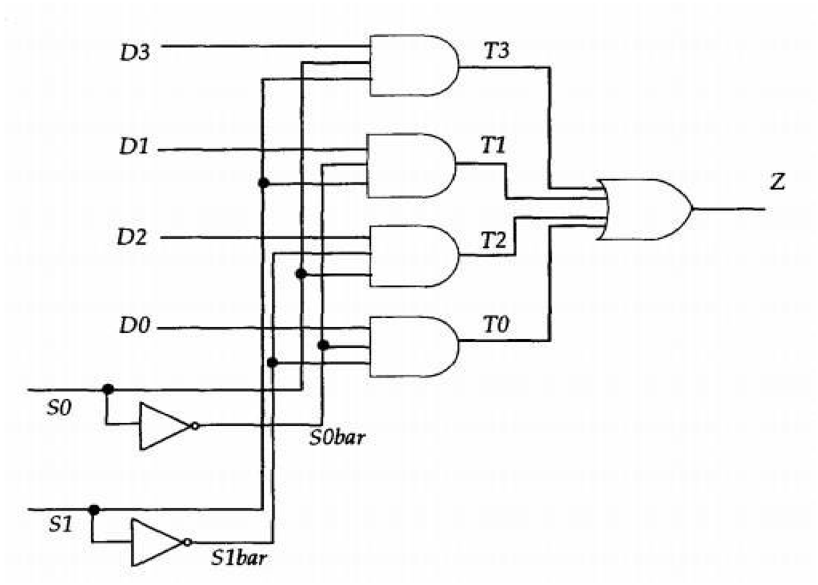


Figure 4.11: 4 X 1 Multiplexer

```
moduleMUX4x1 (Z, DO, Dl, D2, D3, SO, Si);
output Z;input DO, Dl, D2, D3, SO, SI;

and (TO, DO, SObar, Slbar), (Tl, Dl, SObar, S1),

(T2, D2, SO, Slbar), (T3, D3, SO, S1);

not (SObar,SO), (Slbar, S1);

or (Z, TO, Tl, T2, T3);

endmodule
```

**2 to 4 Decoder:**

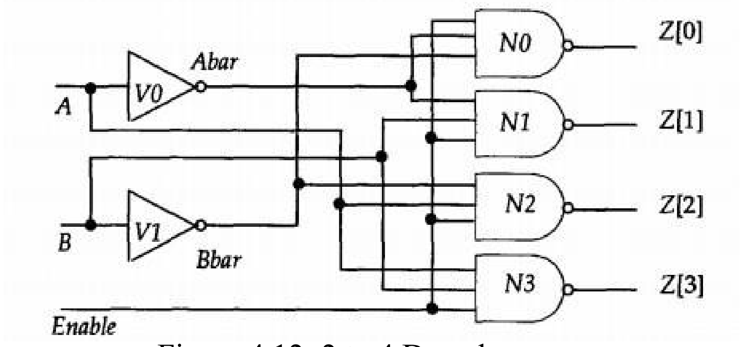


Figure 4.12: 2 to 4 Decoder

```
module DEC2x4 {A, B, Enable, Z) ;
 input A, B, Enable;
 output [0:3] Z;
 wireAbar, Bbar;

 not

  V0 (Abar, A) , V1 (Bbar, B);nand
      NO (Z[0], Enable, Abar, Bbar),
      N1 (Z[l], Enable, Abar, B),

      N2 (Z[2], Enable, A, Bbar),
      N3 (Z[3] , Enable, A, B);
  endmodule
```

**Master Slave Flip-flop:**

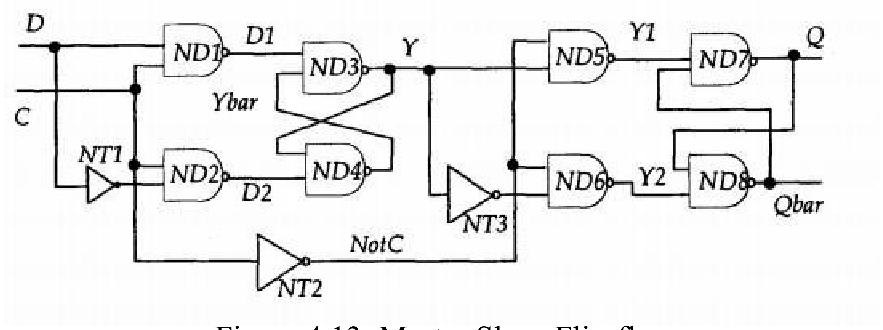


Figure 4.13: Master Slave Flip-flop

```
 module MSDFF (D, C, Q, Qbar);
 input D, C; output Q, Qbar;
 not
NT1 (NotD, D),

NT2 (NotC, C),

NT3 (NotY, Y);
```

```
  nand
        ND1 (Dl, D, C),

        ND2 (D2, C, NotD),

        ND3 (Y, Dl, Ybar),

        ND4 (Ybar, Y, D2),

        ND5 (Yl, Y, NotC),

        ND6 (Y2, NotY, NotC),
        ND7 (Q, Qbar, Yl),
        ND8 (Qbar, Y2, Q);
 endmodule
```

Parity Generator:

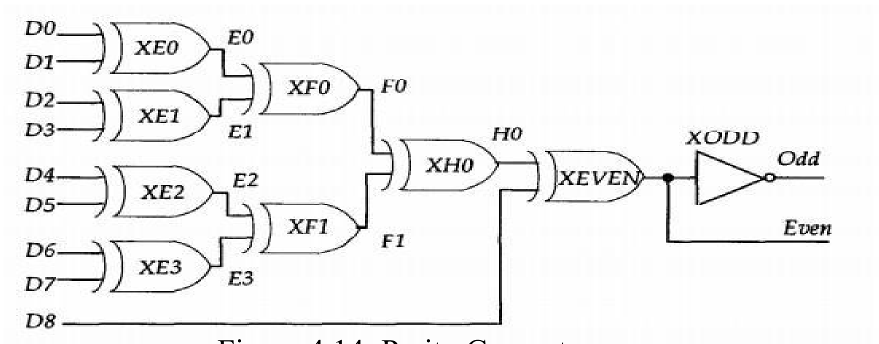


Figure 4.14: Parity Generator

```
 module Parity_9_Bit (D, Even, Odd);
 input [0:8] D;
 output Even, Odd;
 xor

XEO (E0, D[0] , D[l]),

XE1 (El, D[2], D[3]) ,

XE2 (E2,D[4], D[5]) ,

XE3 (E3, D[6], D[7]),

 XFO (F0, E0, El),

 XF1 {Fl, E2, E3),
 XHO {HO, FO, Fl),

 XEVEN {Even, D[8], HO) ;

  not
        XODD {Odd, Even);
 endmodule
```

**USER-DEFINED PRIMITIVES (UDP):**

The primitives available in Verilog are the entire gate or switch types. Verilog has the provision for the user to define primitives –called “user defined primitive (UDP)” and use them. The designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User- Defined Primitives (UDP). These primitives are self-contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate level primitives. UDPs are basically of two types – combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

**Combinational UDPs:**

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An inout declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module.

```
primitiveudp_and(out,
a, b);
output
out;
input
a, b;
table

       // a b: Out;
        0 0: 0;
        0 1: 0;
        1 0: 0
        1
1: 1;
endtabl
e
```

endprim

itive

**Sequential UDPs:**

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state. A sequential UDP can accommodate all these.

```
primitive latch(q, d, clock, clear); // d-
latch

output q; reg q; //q declared as reg to create internal storage

input d, clock, clear;

initial q = 0; //initialize output to
value 0

table

//state table

//d clock clear: q : q+ ;

? ? 1 : ? : 0 ;        //clear condition;

    1 1 0 : ? : 1;         //latchq =data=1

    0 1 0 : ? : 0;        //latchq =data=0

    ? 0 0 : ? : - ;       //retain original state if

clock = 0 endtable

endprimitive
```

**Dataflow Modeling:**

For small circuits, the gate-level modeling approach works very well because the numbers of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent.

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

**Continuous Assignments:**

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

continuous_assign ::= assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;

list_of_net_assignments ::= net_assignment { , net_assignment }

net_assignment ::= net_lvalue = expression

Notice that drive strength is optional and can be specified in terms of strength levels. The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Delay specification is discussed in this chapter. Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.

4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

**Examples of Continuous Assignment:**

Continuous assign - Out is a net. i1 and i2 are

nets. assign out = i1 & i2;

Continuous assign for vector nets - addr is a 16-bit vector net addr1 and addr2 are 16-bit vector registers.

assignaddr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

Concatenation - Left-hand side is a concatenation of a scalar net and a

vector net. assign {c_out, sum [3:0]} = a [3:0] + b[3:0] + c_in;

**Implicit Continuous Assignment:**

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

//Regular continuous

assignment wire out;

assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment

wire out = in1 & in2;

**Implicit Net Declaration**

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

wire
i1, i2;

assign out = i1 & i2; //Note that out was not declared as a
wire

//but an implicit wire declaration for out //is done by the
simulator

**Delays**

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

**Regular Assignment Delay**

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10-time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10-time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

assign #10 out = in1 & in2; // Delay in a continuous
ssign

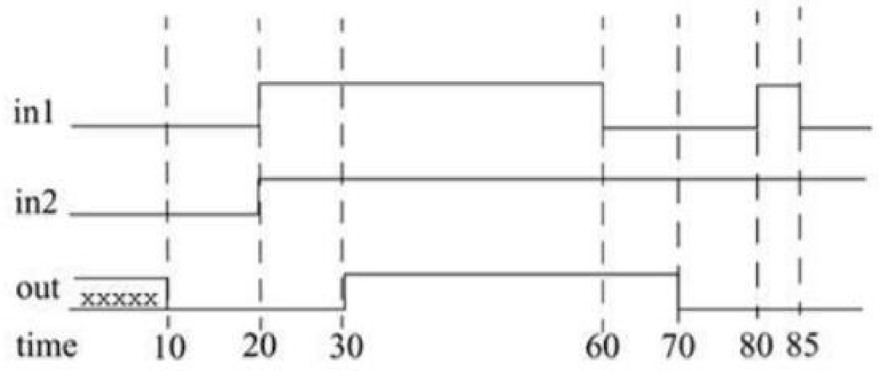


Figure 4.15:
Delays

The above waveform is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30). When in1 goes low at 60, out changes to low at 70.

However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed. Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

**Implicit Continuous Assignment Delay**

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment

delay wire #10 out = in1 & in2;

//same as wire out;

assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

**Net Declaration Delay:**

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net
Delays

wire # 10 out;

assign out = in1 & in2;

//The above statement has the same effect as the following.

wire out;

assign #10 out = in1 & in2;
```

**Examples**

Master Slave Flip-flop:

```
module MSDFF_DF (D, C, Q, Qbar) ;

input D, C; output Q, Qbar;

wireNotC, NotD, NotY, Y, Dl, D2, Ybar, Yl, Y2;

assignNotD = ~ D;

assign Note = ~ C;

assignNotY = ~ Y;

assign D1= - (D & C) ;

assign D2 = ~ (C &NotD);

assign Y = ~ (Dl St Ybar);
```

assignYbar = ~ (Y & D2);

assignYl = ~ (y & Note);

assign Y2 = - (NotY&NotC);

assign Q = ~ (Qbar&Yl);

assignQbar = ~ (Y2 & Q);

endmodule

8 bit Magnitude Comparator: moduleMagnitudeComparator

(A, B, AgtB, AeqB, AltB) ; parameter BUS= 8;

parameter EQ_DELAY = 5, LT_DELAY = 8, GT_DELAY = 8;

input [1 : BUS]A, B;

outputAgtB, AeqB, AltB;

assign %EQ_DELAY AeqB = A == B;

assign $GT_DELAY AgtB = A > B; assign $LT_DELAY AltB = A < B; endmodule

**Behavioral Modeling:**

Behavioral modeling is the highest level of abstraction in the Verilog HDL. The other modeling techniques are relatively detailed. They require some knowledge of how hardware or hardware signals work. The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that a designer need is the algorithm of the design, which is the basic information for any design.

Most of the behavioral modeling is done using two important constructs: initial and always. All the other behavioral statements appear only inside these two structured procedure constructs.

**Procedural Constructs:**

Initial Construct:

The statements which come under the initial construct constitute the initial block. The initial block is executed only once in the simulation, at time 0. If there is more than one initial block, then all the initial blocks are executed concurrently. The initial construct is used as follows:

initial begin reset=1'b0; clk=1'b1; end

or initial

clk = 1'b1;

In the first initial block there is more than one statement hence they are written between begin and end. If there is only one statement then there is no needs to put begin and end.

**Always Construct:**

The statements which come under the always construct constitute the always block. The always block starts at time 0, and keeps on executing all the simulation time. It works like a infinite loop. It is generally used to model a functionality that is continuously repeated.

```
always
#5 clk=~clk;
initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10 units. The initial blocks initiates the clk value to 0 at time 0. Then after every 5 units of time it toggled, hence we get a time period of 10 units. This is the way in general used to generate a clock signal for use in test benches.

```
always@(posedgeclk, negedge reset)

begin
a = b + c; d = 1'b1; end
```

In the above example, the always block will be executed whenever there is a positive edge in the clk signal, or there is negative edge in the reset signal. This type of always is generally used in implement a FSM, which has a reset signal.

```
always @(b, c, d)
begin
a = (b + c)*d;
e = b | c;
end
```

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the sensitivity list.

In the Verilog 2000, we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks are used for implementing the combination logic.

**Operations & Assignments**

The design description at the behavioral level is done through a sequence of assignments. These are called ‘procedural assignments’ – in contrast to the continuous assignments at the data flow level. Though it appears similar to the assignments at the data flow level discussed in the last chapter, the two are different. The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand
- specified on the left side of the "=" sign – for example, N = ~N;
- Here the content of reg N is complemented and assigned to the reg N itself. The assignment is essentially an updating activity.
- The operation on the right can involve operands and operators. The operands can be of different types – logical variables, numbers – real or integer and so on.

**Procedural Assignments**

Procedural assignments are used for updating reg, integer, time, real, realtime, and memory data types. The variables will retain their values until updated by another procedural assignment. There is a significant difference between procedural assignments and continuous assignments. Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value. Whereas procedural assignments update the value of variables under the control of the procedural flow constructs that surround them.

The LHS of a procedural assignment could be:

- reg, integer, real, realtime, or time data type.
- Bit-select of a reg, integer, or time data type, rest of the bits are untouched.
- Part-select of a reg, integer, or time data type, rest of the bits are untouched.
- Memory word.

Concatenation of any of the previous four forms can be specified. When the RHS evaluates to fewer bits than the LHS, then if the right-hand side is signed, it will be sign-extended to the size of the left- hand side. There are two types of procedural assignments: blocking and non- blocking assignments.

**Blocking assignments:**

Blocking assignment statements are executed in the order they are specified in a sequential block. The execution of next statement begins only after the completion of the present blocking assignments. A blocking assignment will not block the execution of the next statement in a parallel block. The blocking assignments are made using the operator =.

initial begin

a = 1; b = #5 2; c = #2 3;

end

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 7.

**Non-blocking assignments:**

The non-blocking assignment allows assignment scheduling without blocking the procedural flow. The non-blocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other. Non- blocking assignments are made using the operator <=.

Note: <= is same for less than or equal to operator, so whenever it appears in expression it is considered to be comparison operator and not as non-blocking assignment.

Initial begin a

<= 1;

b <= #5 2; c <= #2 3; end

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 2 (because all the statements execution starts at time 0, as they are non-blocking assignments).

**Conditional (if-else) Statement:**

The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords if and else are used to make conditional statement. The conditional statement can appear in the following forms.

if (conditional_1)

procedural_statement__1

{else if ( condition_2)

procedural_statement_2}

{else procedural_statement_3 }

Conditional (if-else) statement usage is similar to that if-else statements of C programming language, except that parenthesis is replaced by begin and end.

If conditional evaluates to a non-zero known value, then the procedural_statement_1 is executed. If conditional evaluates to a value 0, x or z, the procedural_statement_1 is not executed, and an else branch, if it exists, is executed. Here is an example.

```
if (Sum < 60)
begin
Grade = C; Total_C = Total_C + 1;
end
else if [Sum < 75)
begin
Grade = B; Total_B = Total_B + 1 ;
end else begin
Grade = A; Total_A = Total_A + 1;
end
```

**Loop Statements**

There are four kinds of loop statements. These are:

- **i.** Forever-loop
- **ii.** ii. Repeat-loop
- **iii.** While-loop
- **iv.** For-loop

**procedural_statement**

This loop statement continuously executes the procedural statement. Thus, to get out of such a loop, a disable statement may be used with the procedural statement. Also, some form of timing controls must be used in the procedural statement, otherwise the forever-loop will loop forever in zero delay. Here is an example of this form of loop statement.

```
initial
begin
Clock
= 0;
#5 forever
# 10 Clock = ~ Clock;
end
```

This example generates a clock waveform; Clock first gets initialized to 0 and stays at 0 until 5 time units. After that Clock toggles every 10 time units.

**Repeat loop**

This form of loop statement has the form:

repeat (loop_count)

procedural_statement

It executes the procedural statement the specified number of times. If loop count expression is an x or a z, then the loop count is treated as a 0. Here are some examples.

repeat (Count) Sum = Sum + 10;

repeat (ShiftBy)

P_Reg = P_Reg<<1;

The repeat-loop statement differs from repeat event control. Consider, repeat

(Count) // Repeat-loop statement.

@ (posedgeClk) Sum = Sum + 1;

which means for Count times, wait for positive edge of Clk and when this occurs, increment Sum. Whereas,

Sum = repeat (Count) @ (posedgeClk) Sum + 1; // Repeat event
control

means to compute Sum + 1 first, then wait for Count positive edges on Clk, then assign to left-hand side.

**While Loop:**

The syntax of this form of loop statement is:

while
(condition)

procedural_statement

This loop executes the procedural statement until the specified condition becomes false. If the expression is false to begin with, then the procedural statement is never executed. If the condition is an x or a z, it is treated as a 0 (false). Here are some example

while (By> 0)

begin

Acc = Acc<< 1; By = By - 1;

end

**For-loop
Statement:**

This loop statement is of the form:

```
for (initial_assignment;
condition;step_assignment)

procedural_statement
```

A for-loop statement repeats the execution of the procedural statement a certain number of times. The initial_assignment specifies the initial value of the loop index. The condition specifies the condition when loop execution must stop. As long as the condition is true, the statements in the loop are executed. The step_assignment specifiesthe assignment to modify, typically to increment or decrement, the step count.

```
integer K;
for (K = 0; K < MAX_RANGE; K = K + 1)
begin
if {Abus[K] == 0)
Abus[K) = 1;
else if (Abus[K] == 1) Abus[K] = 0;

else
$display (\"Abus[K] is an x or a z\");

end
```

**Examples:**

**4x1 Multiplexer**

```
module mux4( input a, b, c, d
input [1:0] sel,

output out );
always @(a or b or c or d or sel)
begin
if(sel==0)
out = a;

else if (sel==1)
out = b;
```

```
  else if ( sel == 2 )
  out = c ;
  else if ( sel == 3 )
  out = d;
  end
  endmodule
```

**D flip-flop**

```
  module  RisingEdge_DFlipFlop(D,clk,Q);
  input D;                          // Data input
  input clk;              // clock input

  output Q;               // output Q

  always @(posedgeclk)
begin
                Q <= D;
  end
```

```
endmodule
```

Shift Register (Serial In Serial Out)

```
module shift (C, SI, SO);
input C,SI; output
SO; reg [7:0] tmp;

always @(posedge C)
    begin
        tmp = tmp<< 1;
        tmp[0] = SI;
   end
assign SO = tmp[7];
endmodule
```

**Structural Modelling:**

The structural model of Verilog HDL is described using:

- Gate instantiation
- UDP instantiation
- Module instantiation

**Module**

A module defines a basic unit in Verilog HDL. It is of the form:

```
modulemodule_name ( port_list );
Declarations_and_Statements
endmodule
```

The port list gives the list of ports through which the module communicates with the external modules.

**Ports**

A port can be declared as input, output or inout. A port by default is a net. However, it can be explicitly declared as a net. An output or an inout port can optionally be redeclared as a regregister. In either the net declaration or the register declaration the net or register must have the same size as the one specified in the port declaration. Here are some examples of declarations.

```
module Micro {PC, Instr, NextAddr);
```

// Port declarations:

input [3:1] PC;

output [1:8] Instr;

inout [16:1] NextAddr;

// Redeclarations:

wire [16:1] NextAddr;

//Optional; but if specified must have same range as in its port declaration. reg

[1:8] Instr;

/* Instr has been redeclared as a reg so that it can be assigned a value within an always statement or an initial statement. */

endmodule

**Module Instantiation**

A module can be instantiated in another module, thus creating hierarchy. A module instantiation statement is of the form:

module_name instance_name( port_associations);

Port associations can be by position or by name; however, associations cannot be mixed. A port association is of the form:

port_expr            // By position.

.PortName (port_expr )// By name.

Where port_expr can be any of the following:

**i.** an identifier (a register or a net)

**ii.** a bit-select

**iii.** a part-select

**iv.** a concatenation of the above

**v.** an expression (only for input ports)

In positional association, the port expressions connect to the ports of the module in the specified order. In association by name, the connection between the module port and the port expression is explicitly specified and thus the order of port associations is not important. Here is an example of a full-adder built using two half-adder modules.

```
Half Adder:
module HA (A, B, S, C);
input A, B;
output S, C;
parameter AND_DELAY = 1, XOR_DELAY = 2;
assign #XOR_DELAY s=A ^ B;
assign #AND_ DELAY C= A & B;
endmodule

Full Adder:
module FA (P, Q, Cin, Sum, Cout);
input P, Q, Cin;
output Sum, Cout; parameter
OR_DELAY = 1; wire SI, CI,
C2;

//Two module instantiations:
HA h1 (P, Q, S1, C1);

// Associatingby position.
HA h2 (A(Cin), S(Sum), B(S1), .C(C2));       //Associating by name.
// Gate instantiation:
or #OR_DELY 01 (Cout, CI, C2) ;
endmodule
```

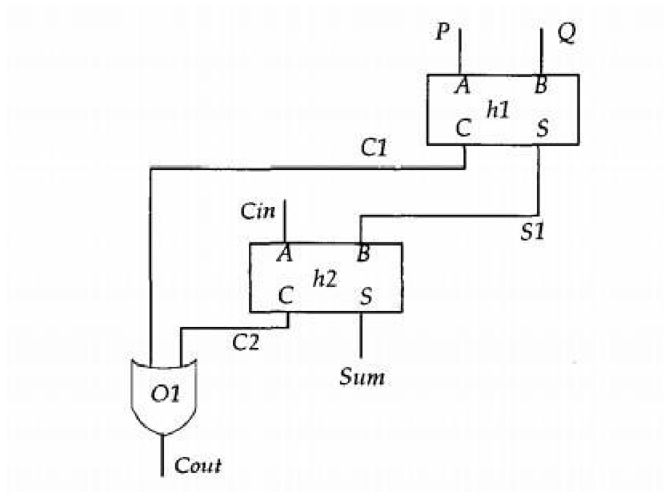


Figure 4.16: Full Adder using Two Half Adders

In the first module instantiation, HA is the name of the module, h1 is the instance name and ports are associated by position, that is, P is connected to module (HA) port A, Q is connected to module port B, S1 to S and C1 to module port C. In the second instantiation, the port association is by name, that is, the connections between the module (HA) ports and the port expressions are specified explicitly.

Different port length:

When a port and the local port expression are of different lengths, port matching is performed by (unsigned) right justification or truncation. Here is an example of port matching.

```
moduleChild (Pba, Ppy) ;

input [5:0] Pba;
output [2:0] Ppy;
endmodule

module Top;
wire [1:2] Bdl;
wire [2:6] Mpr-, Child
C1 {Bdl, Mpr);
endmodule
```

In the module instantiation for Child, Bdl[2] is connected to Pba[0] and Bdl[1 ]is connected to Pba[1]. Remaining input ports, Pba[5], Pba[4], Pba[3] are not connected and therefore have the value z. Similarly,Mpr[6] is connected to Ppy[0], Mpr[5] is connected to Ppy[l] and Mpr[4] is connected to Ppy[2].

Examples:

Decade counter:

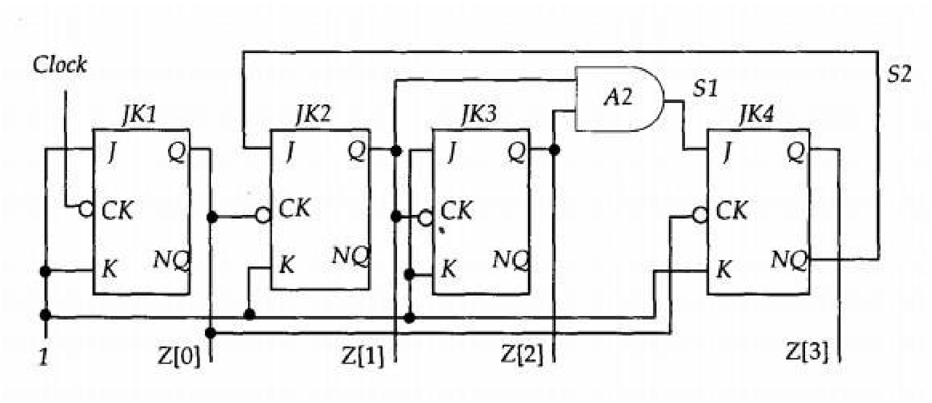


Figure 4.17: Decade counter

module Decade_Ctr(Clock, Z);

input Clock;

output [0:3] Z;

wire SI, S2;

and A1 {SI, Z[2], Z[l]);// Primitive gate instantiation.

// Four module instantiations:

JK_FF JK1 (.J(1’b1), .K(1’b1), .ck(Clock), .Q(z[0]), .NQ ( )),

JK2 (.J(1’b1), .K(1’b1), .ck(Z[0]), .Q(z[1]), .NQ ()),

JK3 (.J(1’b1), .K(1’b1), .ck(Z[1]), .Q(z[2]), .NQ ()),

JK4 (.J(1’b1), .K(1’b1), .ck(Z[0]), .Q(z[1]), .NQ (S2));

endmodule

3 bit UP-DOWN counter

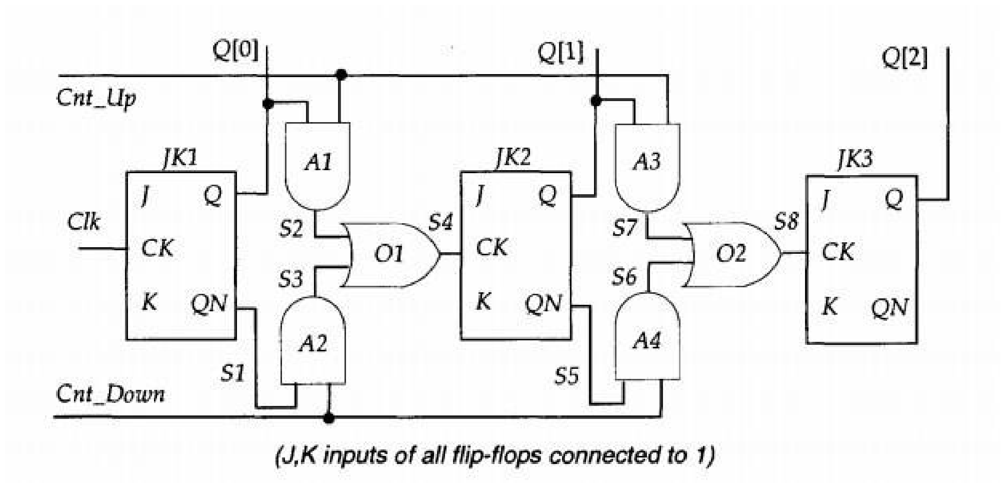


Figure 4.18: 3 bit UP-DOWN counter

```
moduleUp_Down {Clk, Cnt_Up, Cnt_Down, Q);
inputClk, Cnt_Up, Cnt_Down;
output [0:2] Q;
wire S1, S2, S3, S4, S5,S6, S7, S8;
JK_FF JK1 (l'bl, l'bl, Clk, Q[Q], S1),
JK2 (l'bl, l'bl, S4, Q[l], S5),

JK3 (l'bl, l'bl, S8, Q[2], );
and A1 (S2, Cnt_Up, Q[Q]),
        A2 (S3, SI, Cnt_Down),
        A3 (S7, Q[l] ,Cnt_Up),
        A4 (S6, S5, Cnt_Down);

    or 01 (S4, S2, S3),
            02 (S8,S7, S6);
endmodule
```

**TEXT / REFERENCE BOOKS**

1. J.Bhaskar, “A VHDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

2. Stphen Brown, "Fundamental of Digital logic with Verilog Design",3rd edition, Tata McGraw Hill, 2008

3. J.Bhaskar, “A Verilog HDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

4. Samir Palnitkar” Verilog HDL: A Guide to Digital Design and Synthesis”, Star Galaxy Publishing; 3rd edition,2005

5. Michael D Ciletti - Advanced Digital Design with VERILOG HDL, 2nd Edition, PHI, 2009.

6. Z Navabi - Verilog Digital System Design, 2nd Edition, McGraw Hill, 2005.

**PART-A**

1. Justify the importance of gate primitives in Verilog HDL.
2. List the user defined primitives in Verilog HDL.
3. Distinguish gate level modeling and dataflow modeling.
4. Develop a Verilog HDL program for Full Adder using gate level modeling.
5. List the conditional statements in Verilog HDL.
6. Develop a Verilog HDL program for 2*4 decoder using dataflow modeling.
7. Develop a verilog HDL program for XOR gate using switch level modeling.
8. Formulate any one loop statement in Verilog HDL.
9. Classify the types of delays in verilog HDL.
10. Develop Verilog HDL code for 1 bit comparator.

**PART-B**

1. Develop a program in Verilog HDL to design a multiplexer using if and case statement.
2. Discuss with example structural modeling of Verilog HDL.
3. Develop a Verilog HDL program for SISO and SIPO shift registers
4. Distinguish between dataflow modeling and behavioral modeling of Verilog HDL

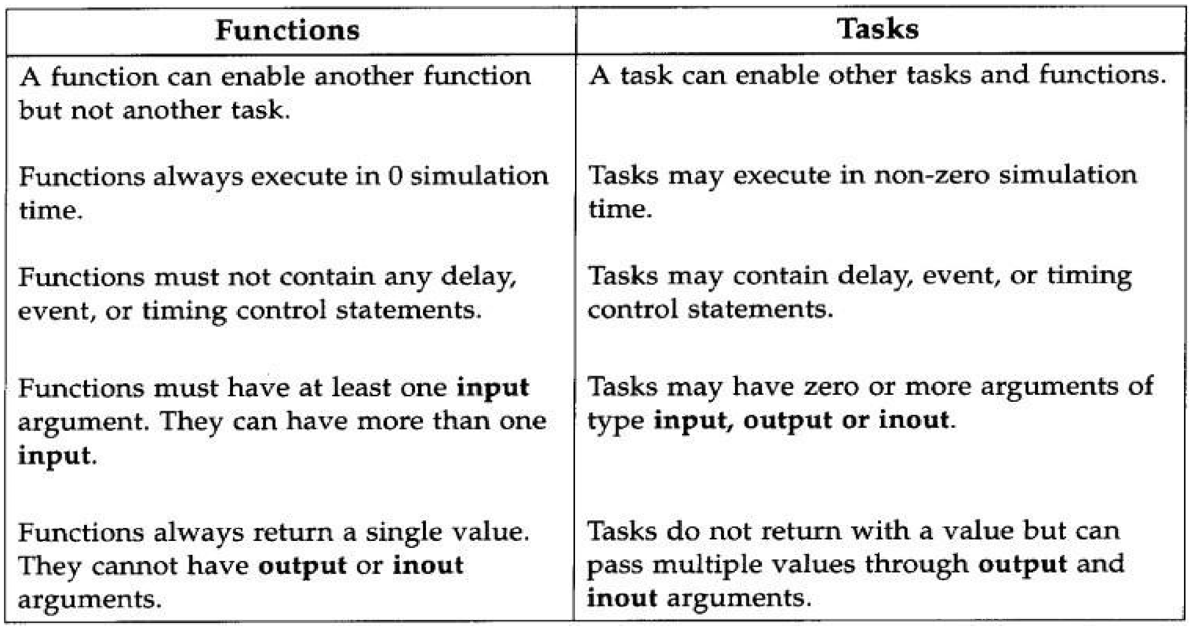
# UNIT – V - FEATURES IN VERILOG HDL– SECA3007

## FEATURES IN VERILOG HDL

### Tasks and Functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. Input, output, and inout argument values can be passed into both tasks and functions.

Differences between Functions and Tasks



| Functions | Tasks |
|---|---|
| A function can enable another function but not another task. | A task can enable other tasks and functions. |
| Functions always execute in 0 simulation time. | Tasks may execute in non-zero simulation time. |
| Functions must not contain any delay, event, or timing control statements. | Tasks may contain delay, event, or timing control statements. |
| Functions must have at least one **input** argument. They can have more than one **input**. | Tasks may have zero or more arguments of type **input, output or inout**. |
| Functions always return a single value. They cannot have **output** or **inout** arguments. | Tasks do not return with a value but can pass multiple values through **output** and **inout** arguments. |

The following rules distinguish tasks from functions:

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value. The purpose of a function is to respond to an input value by returning a single value. A task can support multiple goals and can calculate multiple result values. However, only the output or inout argumentspass result values back from the invocation of a task. A Verilog model uses a function as an operand in an expression; the value of that operand is the value returned by the function.

Task and function declarations specify the following:•
local variables

- I/O ports

- registers

- times
- integers
- real
- events

These declarations all have the same syntax as for the corresponding declarations in a module definition. If there is more than one output, input, and inout port declared in a task these must be enclosed within a block.

Task

- A task begins with keyword task and ends with keyword endtask
- Inputs and outputs are declared after the keyword task.
- Local variables are declared after input and output

  declaration. Task declaration and invocation

Task Declaration syntax

task <task_name>; <I/O declarations> <variable and event declarations>

begin

<statement(s)>

end

endtask

Task invocation syntax

<task_name>; <task_name>(<arguments>);

begin

temp_out = (9/5) *( temp_in+ 32)

end endtask

endmadule

Function

A Verilog HDL function is the same as a task, with very little differences, like function cannot drive more than one output, can not contain delays.

- functions are defined in the module in which they are used. It is possible to define functions in separate files and use compile directive 'include to include the function in the file which instantiates the task.

- functions can not include timing delays, like posedge, negedge, # delay, which means that functions should be executed in "zero" time delay.
- functions can have any number of inputs but only one output. The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.
- functions can take, drive, and source global variables, when no local variables are used.
- When local variables are used, basically output is assigned only at the end of function execution.
- functions can be used for modeling combinational logic.
- functions can call other functions, but cannot call tasks. Syntax
- A function begins with keyword function and ends with keyword endfunction.
- inputs are declared after the keyword function.

**Function Rules**

Functions are more limited than tasks. The following five rules govern their usage:

- A function definition cannot contain any time-controlled statements—that is, any statements introduced with #, @, or wait.
- Functions cannot enable tasks.
- A function definition must contain at least one input argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.

- A function definition can't contain an inout declaration or an output declaration

Function Declaration and Invocation

Declaration syntax:

```
function <range_or_type> <func_name>;
<input declaration(s)>
<variable_declaration(s)>

begin
<statements>
end
endfunction
```

Invocation syntax:

<func_name> (<argument(s)>);

Example

```
module   simple_function();
function  myfunction;  input
a, b, c, d;

begin
myfunction = ((a+b) + (c-d));

end
endfunction
endmodule
```

**SYSTEM TASKS AND FUNCTIONS**

Verilog contains the pre-defined system tasks and functions, including tasks for creating output from a simulation. All system tasks appear in the form $.Operations such as displaying the screen, monitoring values of nets, stopping and finishing are done by system tasks.

**DISPLAY TASKS**

**$display**

$display displays information to standard output and adds a newline character to the end of its output.

**$monitor**

$monitor continuously monitors and displays the values of any variables or expressions specified as parameters to the task. Parameters are specified in the same format as for $display.

$monitoron -$monitoron controls a flag to re-enable a previously disabled $monitor.

Syntax: $monitoron;

$monitoroff-$monitoroff controls a flag to disable monitoring.

Syntax: $monitoroff;

**$write**

$write displays information to standard output without adding a newline character to the end of its output.

Syntax: $write (list_of_arguments);

The default format of an expression argument that has no format specification is decimal. The companion $writeb, $writeo, and $writeh tasks specify binary, octal and hex formats, respectively.

**FILE I/O TASKS**

$fclose

$fclose closes the channels and prevents further writing to the closed channels.

Syntax: file_closed_task ::= $fclose ;

$fdisplay

$fdisplay is the counterpart of $display; it is used to direct simulation data to a file.

Syntax: $fdisplay ([multi_channel_descriptor], list_of_arguments);

$fopen

$fopen opens the file specified by a parameter and returns a 32-bit unsigned MCD (integer multi-channel-descriptor) uniquely associated the file. $fopen rturns 0 if the file could not be opened.

Syntax: file_open_function ::= integer multi_channel_descriptor = $fopen("[name_of_file]");

$readmemb

$readmemb reads binary numbers from a text file and loads them into a Verilog memory, or sub-blocks of a memory, specified by an identifier.

Syntax: $readmemb ("filename", memory_name [, start_addr [, finish_addr]]);

**SIMULATION CONTROL TASKS**

**$finish**

$finish terminates simulation, and returns control to the host operating system.

Syntax: $finish;

**$stop**

$stop suspends simulation, issues an interactive prompt, and passes control to the user.

$stop(n) suspends simulation, issues and interactive prompt, and takes the following action, depending on the diagnostic control parameter, n:

n = 0 Prints nothing.n = 1 Prints the simulation time and location

n = 2 Prints simulation time and location

**Modeling a Test bench**

Whenever we design a circuit or a system, one step that is most important is "testing". Testing is necessary to verify whether the designed system works as expected or not.

If we find some error in an IC after fabrication, we are looking at a great loss because now we have to re-do the entire chip manufacturing process from scratch right from designing the circuit to fabrication.

Test benches are used to test the RTL (Register-transfer logic) that we implement using HDL languages like Verilog and VHDL.

Verifying complex digital systems after implementing the hardware is not a wise choice. It is ineffective in terms of time, money, and resources. Hence, it is essential to verify any design before finalizing it. Luckily, in the case of FPGA and Verilog, we can use test benches for testing Verilog source code.

Now we are going to learn how we can use Verilog to implement a test bench to check for errors or inefficiencies. We'll first understand all the code elements necessary to implement a test bench in Verilog. Then we will implement these elements in a stepwise to truly understand the method of writing a test bench.

**Design Under Test (DUT)**

A design under test, abbreviated as DUT, is a synthesizable module of the functionality we want to test. In other words, it is the circuit design that we would like to test. We can describe our DUT using one of the three modeling styles in Verilog, Gate level, Dataflow level and Behavioral level.

```
For example,
module and_gate(c,a,b);
input a,b;
output c;
assign c = a & b;
endmodule
```

We have described an AND gate using Dataflow modeling. It has two inputs (a,b) and an output (c). We have used continuous assignment to describe the functionality using the logic equation. This AND gate can be our DUT.

So, to test our DUT, we have to write the test bench code. Why
do we have to take the trouble to write another code?

With a test bench, we can view all the signals associated with the DUT. No need for physical hardware.

Writing a test bench is a bit trickier than RTL coding. Verifying a system can take up around 60-
70% of the design process.

**Implementation of test bench**

Let's learn how we can write a test bench. Consider the AND module as the design we want to test.

Like any Verilog code, start with the module declaration.

```
module and_gate_test_bench;
```

**Reg and wire declarations**

Usually, we declare the input and output ports. But, in a test bench, we will use two signal types for driving and monitoring signals during the simulation.

The reg datatype will hold the value until a new value is assigned to it. This data type can be assigned a value only in the always or initial block. This is used to apply a stimulus to the inputs of DUT.

The wire datatype is similar to that of a physical connection. It will hold the value that is driven by a port, assign statement, or reg. This data type cannot be used in initial or always blocks. This is used to check the output signals *from* the DUT.

```
reg A, B;
```

**DUT Instantiation**

The purpose of a test bench is to verify whether our DUT module is functioning as we wish. Hence, we have to instantiate our design module to the test module. The format of the instantiation is:

<dut_module> <instance name>(.<dut_signal>(test_module_signal),…)

```
and_gate dut(.a(A), .b(B), .c(C));
```

We have instantiated the DUT module and_gate to the test module. The signals with a dot in front of them are the names for the signals inside the and_gate module, while the wire or reg they connect to in the test bench is next to the signal in parenthesis.

**Test bench for AND Gate**

We have already written the Verilog file for an AND gate. Let's see how to write a test bench for that DUT.

```
module and_tb;
```

Then, let's have the reg and wire declarations on the way. The input from the DUT is declared as reg and wire for the output of the DUT. It is through these data types we can apply the stimulus to the DUT. Using upper case letters for signals in the test bench avoids confusion.

```
reg A,B;
```

Then comes the part of performing instantiation.

```
and_gate dut(.a(A), .b(B), .c(C));
```

We have linked our test bench to the DUT.

```
initial
begin

#5 A =0; B=0;

#5 A =0; B=1;

#5 A =1; B=0;
```

So our final testbench code will be:

```
module and_tb;

reg A,B;

wire C;

and_gate dut(.a(A), .b(B), .c(C));

initial
begin
#5 A =0; B=0;

#5 A =0; B=1;

#5 A =1; B=0;

#5 A =1; B=1;

end

end module
```

**Testbench for D-flip flop**

For sequential circuits, the clock and reset signals are essential for its functioning.

Let’s test the Verilog code for D-flip flop. Here’s the DUT:

```
module dff_behave(clk,rst,d,q,qbar); input
clk,rst,d;

output reg q,qbar;
always@(posedgeclk) begin
if(rst == 1) begin

q <= 0;
```

```
qbar <= 1; end else
begin q <= d; qbar <=
~d;
```

```
end
end

endmodule
```

Let’s start writing a testbench for the above :

As usual start with the module declaration. Naming the module as dff_tb

```
module dff_tb
```

Moving on with the reg and wire declaration:

```
reg D,CLK,RST;
```

Time for DUT instantiation:

```
dff_behave dut(.clk(CLK), .rst(RST), .d(D), .q(Q), .qbar(QBAR));
```

As we said, a clock signal is essential for working of the flip flop. So, here’s how we create a clock stimulus for our testbench

```
always

#10 CLK = ~CLK
```

The above clock will have a 20 ns pulse width. Therefore, we have generated a 50 MHz clock. Let’s

apply the stimulus for our DUT:

```
initial
begin
RST = 1;

#10 RST = 0;

#10 D = 0;

#10 D = 1
```

Finally, our testbench code is:

```
module dff_tb; reg
CLK = 0; reg
D,RST; wire
Q,QBAR;

dff_behave dut(.clk(CLK), .rst(RST), .d(D), .q(Q), .qbar(QBAR));

 always

 #10 CLK = ~CLK;

 initial begin RST =1;

 #10 RST = 0;

 #10 D = 0;

 #20 D = 1

 end endmodule
```

**Test Bench for Half Adder**

```
module half_adder_verilog_tb; reg
a, b;

wire s, c;

halfadder8 dut (.a(a), .b(b), .s(s), .c(c));

initial
begin  a  =
0; b = 0;

#50;

a = 0;

b = 1;
```

```
#50;

a = 1;

b = 0;

#50;

a = 1; b =
1; end

endmodule
```

**Concepts of Timing and Delays in Verilog**

The concepts of timing and delays within circuit simulations are very important because they allow a degree of realism to be incorporated into the modeling process. In Verilog, without explicit specification of such constraints, the outputs of pre-defined primitives and user- defined modules are all assumed to resolve instantaneously. Some designs, such as high speed microprocessors, may have very tight requirements that must be met. Failure to meet these constraints may result in the design failing to work at all, or possibly even producing invalid outputs. Thus, the aim of the designer may be to produce a circuit that functions correctly, and it is equally important that the circuit also conforms to any timing constraints required of it.

**Delays**

Delays can be modelled in a variety of ways, depending on the overall design approach that has been adopted, namely gate-level modelling, dataflow modelling and behavioural modelling.

Gate level modeling

In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

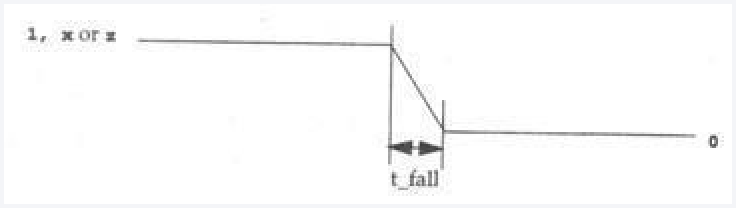
**Rise, Fall, and Turn-off Delays**

There are three types of delays from the inputs to the output of a primitive gate

**Rise delay**

The rise delay is associated with a gate output transition to a 1 from another value.

**Fall delay**

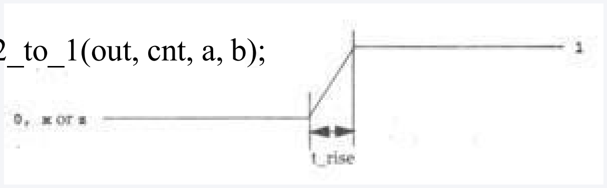


**Turn-off delay**

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

**If the value changes to X, the minimum of the three delays is considered.**

0, 1, x and z take their usual meanings of logic low, logic high, unknown and high impedance. Any or all of these delays can be specified for each gate by use of the delay token #. If only one value is specified, it is used for all these delays. If two are given, they are used for the rise and fall delays respectively. The turn-off delay (the time taken for the output to go to a high impedance state) is taken to be the minimum of these values. Alternatively, all three values can be explicitly set. The use of delays is illustrated for the 2-input multiplexer.



```
module multiplexor_2_to_1(out, cnt, a, b);

  /*

   * A 2-1 1-bit multiplexor

   */
output out;

input cnt, a, b;

wire        not_cnt, a0_out, a1_out;

not # 2    n0(not_cnt, cnt);         /* Rise=2, Fall=2, Turn-Off=2 */ and

#(2,3) a0(a0_out, a, not_cnt); /* Rise=2, Fall=3, Turn-Off=2 */ and
```

```
#(2,3) a1(a1_out, b, cnt);

or #(3,2) o0(out, a0_out, a1_out); /* Rise=3, Fall=2, Turn-Off=2 */

endmodule /* multiplexor_2_to_1 */
```

**Dataflow modeling**

**Net Declaration Delay**

The delay to be attributed to a net can be associated when the net is declared. Thereafter any changes of the signals being assigned to the net will only be propagated after the specified delay.

*e.g.* wire #10 out;

assign out = in1 & in2;

If either of the values of in1 or in2 should happen to change before the assigment to out has taken place, then the assignment will not be carried out, as input pulses shorter than the specified delay are filtered out. This is known as *inertial delay*.

**Regular Assignment Delay**

This is used to introduce a delay onto a net that has already been declared.

*e.g.* wire out; assign #10 out = in1 & in2;

This has a similar effect to the code above, computing the value of in1 & in2 at the time that the assign statement is executed, and then storing that value for the specified delay (in this case 10 time units), before assigning it to the net out.

**Implicit Continuous Assignment**

Since a net can be implicitly assigned a value at its declaration, it is possible to introduce a delay then, before that assignment takes place.

*e.g.* wire #10 out = in1 & in2;

It should be easy to see that this is effectively a combination of the above two types of delay, rolled into one.

**Behavioural modelling**

Regular Delay or Inter-assignment delay

This is the most common delay used - sometimes also referred to as *inter-assignment delay control*.

*e.g.* #10 q = x + y;

It simply waits for the appropriate number of timesteps before executing the command.

**Intra-Assignment Delay Control**

With this kind of delay, the value of x + y is stored at the time that the assignment is executed, but this value is not assigned to q until after the delay period, regardless of whether or not x or y have changed during that time.

*e.g.* q = #10 x + y;

This is similar to the delays used in dataflow modeling.

**Timing controls**

Timing controls provide a way to specify the simulation time at which procedural statements will execute.

There are three methods of timing control

- Delay based timing control
- Event based timing control
- Level-sensitive timing control

**Delay based timing control**

Delay-based timing control in an expression specifies the time duration between the statement is encountered and when it is executed. Delays are specified by the symbol #.

There are three types of delay control for procedural assignments

- Regular delay control
- Intra-assignment delay control
- Zero delay control

**Regular delay control**

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown below example,

```
module clk_gen;

reg clk, reset;

clk = 0;

reset = 0;

#2 reset = 1;

#5 reset = 0;
#10 $finish;
```

**Intra-assignment delay control**

Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Usage of intra-assignment delay control is shown in below example,

```
module intra_assign;

reg a, b;
```

```
 a = 1;

b = 0;
```

**Difference between the intra-assignment delay and regular delay**

Regular delays defer the execution of the entire assignment. Intra-assignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable. Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

**Zero delay control**

Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation in that simulation time are executed. This is used to eliminate race conditions. However, if there are multiple zero delay statements, the order between them is nondeterministic. Usage of zero delay control is shown in below example,

```
initial
begin
x=0;
y=0;
end

initial
begin

#0 x=1;

#0 y=1;

end
```

Above four statements x=0,y=0,x=1,y=1 are to be executed at simulation time 0. However since x=1 and y=1 have #0, they will be executed last. Thus, at the end of time 0,x will have value 1 and y will have value 1.

**Event based timing control**

An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. There are four types of event-based timing control.

- Regular event control
- Named event control
- Event OR control
- Level-sensitive timing control

**Regular event control**

The @ symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value. The keyword posedge is used for a negative transition as shown in below example,

```
module edge_wait_example();

reg enable, clk, trigger;


always @ (posedge enable)

begin

 trigger = 0;

 // Wait for 5 clock cycles
 repeat (5) begin

  @ (posedge clk) ;

 end

 trigger = 1;
```

```
end
```

**Named event control**

Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event. The event does not hold any data. A named event is declared by the keyword event. An event is triggered by the symbol . The triggering of the □event is recognized by the symbol @.

*data_buf={data_pkt[0],data_pkt[1]};*

**Example**

```
[illegible]
always @(posedge clock)

begin
□
if (last_data_packet)
end
```

**Event OR control**

Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a sensitivity list. The keyword or is used to specify multiple triggers as shown in below example,

```
always @(reset or clock or d)

begin
if(reset)
q=1’b0;
else if (clock)
```

**Level-Sensitive Timing control**

Verilog allows a level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword wait is used for level-sensitive constructs.

**Example**

```
always

wait (count_enable) #20 count=count+1;
```

From the above example, the value of count_enable is monitored continuously. If count_enable is 0, the statement is not entered. If it is logical 1, the statement count=count+1 is executed after 20 time units. If count_enable stays at 1, count will be incremented every 20 time units.

## SWITCH LEVEL MODELING

Usually, transistor level modeling is referred to model in hardware structures using transistor models with analog input and output signal values. On the other hand, gate level modeling refers to modeling hard-ware structures with digital input and output signal values between these two modeling schemes is referred to as switch level modeling. At this level, a hardware component is described at the transistor level, but transistors only exhibit digital behavior and their input, and output signal values are only limited to digital values. At the switch level, transistors behave as on-off switches- Verilog uses a 4 value logic value system, so Verilog switch input and output signals can take any of the four 0, 1, Z, and X logic values.

### Switch level primitives

Switches are unidirectional or bidirectional and resistive or nonresistive. For each group those primitives that switch on with a positive gate {like an NMOS transistor} and those that switch on with a negative gate {like a PMOS transistor}. Switching on means that logic values flow from input transistor to its input. Switching off means that the output of a transistor is at Z level regardless of its input value. A unidirectional transistor passes its input value to its output when it is switched on.

A bidirectional transistor conducts both ways. A resistive structure reduces the strength of its input logic when passing it to its output. In addition to switch level primitives, pull-primitives that are used as pull-up and pull-down resistors for tri-state outputs.

**MOS Switches**

Two types of MOS switches can be defined with the keywords nmos and pmos. Keyword nmos is used to model NMOS transistors, Keyword pmos is used to model PMOS transistors. The symbols for nmos and pmos switches are shown in figure.

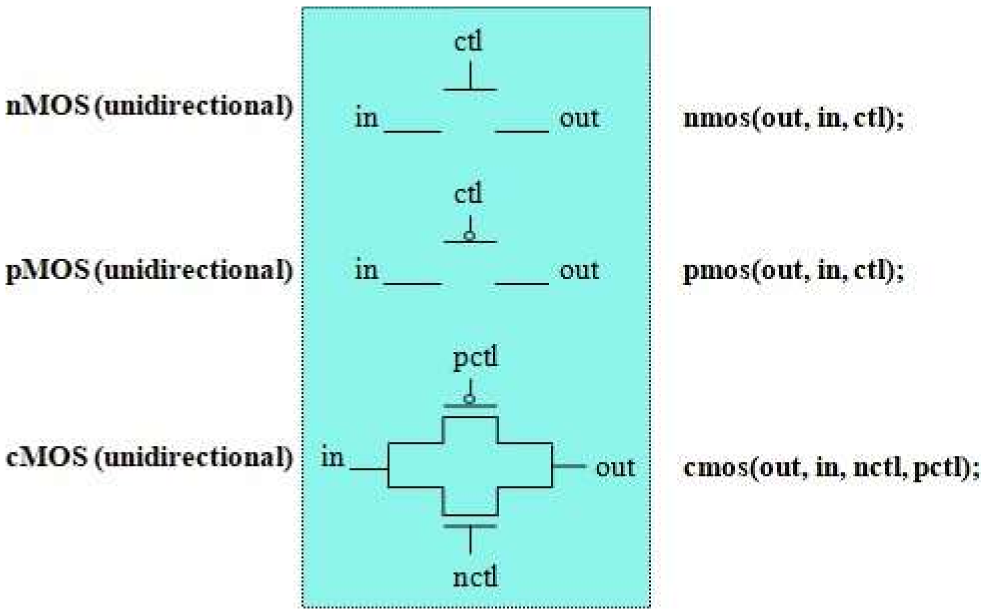


Figure 5.1 : MOS switches

**PMOS and NMOS Switches**

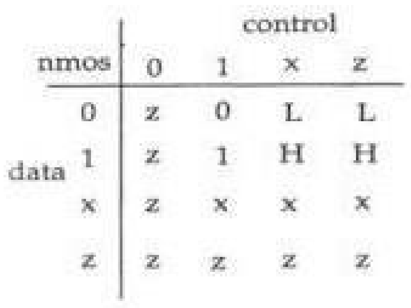
In Verilog nmos and pmos switches are instantiated as shown in below

```
nmos n1(out, data, control); // instantiate a nmos switch
pmos p1(out, data, control); // instantiate a pmos switch
```

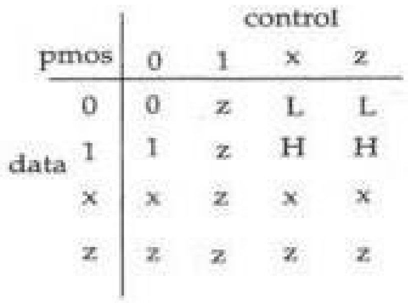
Since switches are Verilog primitives, like logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name

```
nmos (out, data , control); // instantiate nmos switch ; no instance name
pmos (out, data, control); // instantiate pmos switch; no instance name
```

Value of the *out* signal is determined from the values of data and control signals. Logic tables for out are shown in table. Some combinations of data and control signals cause the gates to output to either a 1 or 0 or to an z value without a preference for either value. The symbol L stands for 0 or Z; H stands for 1 or z.



| | | control | | | |
|---|---|---|---|---|---|
| | nmos | 0 | 1 | x | z |
| data | 0 | z | 0 | L | L |
| | 1 | z | 1 | H | H |
| | x | z | x | x | x |
| | z | z | z | z | z |



| | | control | | | |
|---|---|---|---|---|---|
| | pmos | 0 | 1 | x | z |
| data | 0 | 0 | z | L | L |
| | 1 | 1 | z | H | H |
| | x | x | z | x | x |
| | z | z | z | z | z |

Logic Tables of NMOS and PMOS

Thus, the nmos switch conducts when its control signal is 1. If control signal is 0, the output assumes a high impedance value. Similarly a pmos switch conducts if the control signal is 0.

CMOS Switches

CMOS switches are declared with the keyword cmos. A cmos device can be modeled with a nmos and a pmos device. The symbol for a cmos switch is shown in figure.

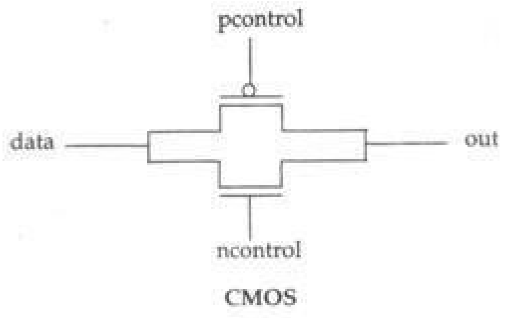


Figure 5.2 : CMOS switch

**CMOS switch**

A CMOS switch is instantiated as shown in below,

***cmos cl(out, data, ncontrol, pcontrol);//instantiate cmos***

***gate or***

***cmos (out, data, ncontrol, pcontrol); //no instance name given***

The ncontrol and pcontrol are normally complements of each other. When the ncontrol signal is 1 and pcontrol signal is 0, the switch conducts.

***nmos (out, data, ncontrol); //instantiate a nmos***

***switch pmos (out, data, pcontrol); //instantiate a***

***pmos switch***

Since a cmos switch is derived from nmos and pmos switches, it is possible derive the output value

from Table, given values of *data, ncontrol,* and *pcontrol* signals.

**Bidirectional switches**

*NMOS, PMOS* and *CMOS* gates conduct from drain to source. It is important to have devices that conduct in both directions. In such cases, signals on either side of the device can be the driver signal. Bidirectional switches are provided for this purpose. Three keywords are used to define bidirectional switches: tran, tranif0, and tranif1.

Symbols for these switches are shown in figure below.

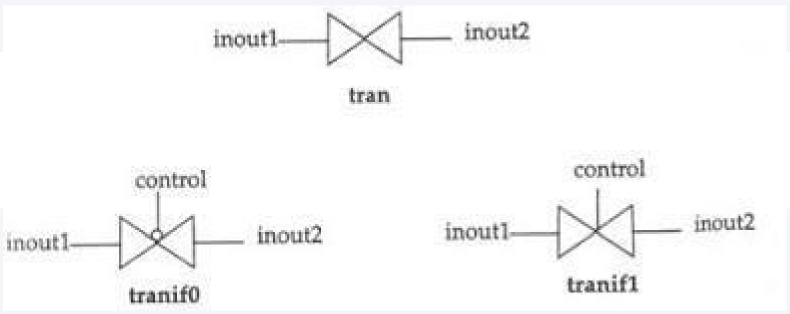


Figure 5.3 : Bidirectional switches

*The* tran switch acts as a buffer between the two signals *inoutl* and *inout2.* Either inoutl or *inout2* can be the driver signal. The *tranif0* switch connects the two signals *inoutl* and *inout2* only if the *control* signal is logical 0. If the *control* signal is a logical 1, the nondriver signal gets a high impedance value z. The driver signal retains value from its driver. The tranifl switch conducts if the *control* signal is a logical 1.

These switches are instantiated as shown in
below.

```
tran tl(inoutl, inout2); //instance name tl is
optional

tranifO (inoutl, inout2, control); //instance name is not
specified
```

Resistive switches reduce signal strengths when signals pass through them. The changes are shown below. Regular switches retain strength levels of signals from input to output. The exception is that if the input is of supply, the output is of strength strong. Below table shows the strength reduction due to resistive switches.

| Input strength | |
|---|---|
| **supply** | **pull** |
| **strong pull** | **pull** |
| **weak** | **weak** |
| **medium** | **large** |
| **medium** | |
| **medium** | **small** |
| **small** | **small** |
| **high** | **high** |

Example-CMOS NAND

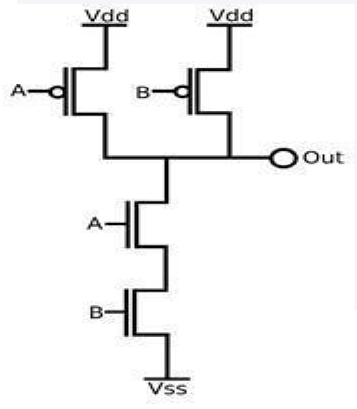


Figure 5.4 : CMOS NAND

```
module my_nand (Out,A,B); input A,B;

ouput Out;
wire C;
supply1 Vdd;
supply0 Vss;

pmos (Out,A,Vdd)
pmos (Out,B,Vdd);
nmos (Out,A,C);
nmos(C,Vss,B);

endmodule
```

## TEXT / REFERENCE BOOKS

1. J.Bhaskar, “A VHDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

2. Stphen Brown, "Fundamental of Digital logic with Verilog Design",3rd edition, Tata McGraw Hill, 2008

3. J.Bhaskar, “A Verilog HDL Primer”, Prentice Hall of India Limited. 3rd edition 2004

4. Samir Palnitkar” Verilog HDL: A Guide to Digital Design and Synthesis”, Star Galaxy Publishing; 3rd edition,2005

5. Michael D Ciletti - Advanced Digital Design with VERILOG HDL, 2nd Edition, PHI, 2009.

6. Z Navabi - Verilog Digital System Design, 2nd Edition, McGraw Hill, 2005.

## QUESTION BANK

### PART-A

1. Define system task
2. List the tristate gates
3. Distinguish between system task and system function.
4. Distinguish between unary operators and ternary operators.
5. Formulate the syntax of event construct
6. Define functional register
7. Define path delay
8. Define net delay
9. Formulate the syntax of repeat construct.
10. List the key words in Verilog HDL

### PART-B

1. Write a model for a 4-Bit shift register with serial in data, serial out data using a for loop with an always Statement.
2. Design a moore FSM with an example, Mention the state transition diagram for it.
3. Design a mealy FSM with an example. Mention the state transition diagram for it.
4. Develop a verilog code for 4-Bit ALU also obtain its test bench and simulation results.
5. Design Verilog module for an edge triggered D Flip flop in the data flow model.