



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 ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - I
PROGRAMMING IN HDL – SEC1406

I. CONCEPTS IN VHDL

INTRODUCTION TO 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 :-

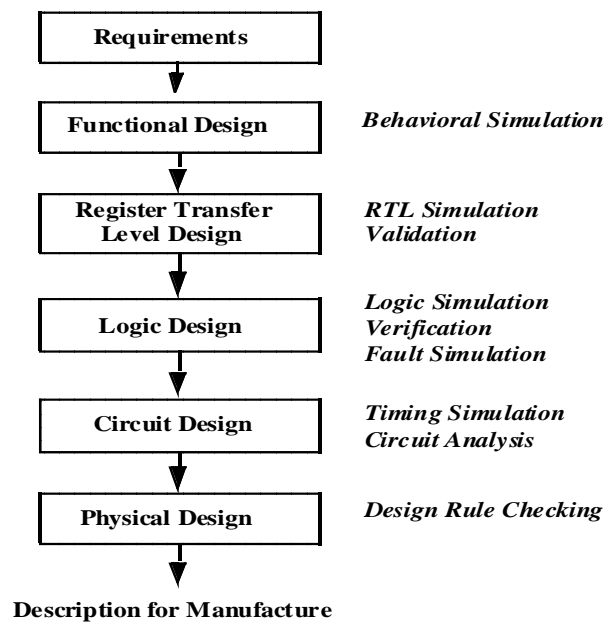


Figure 1.1 : Digital system design process

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:-

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).

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

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.

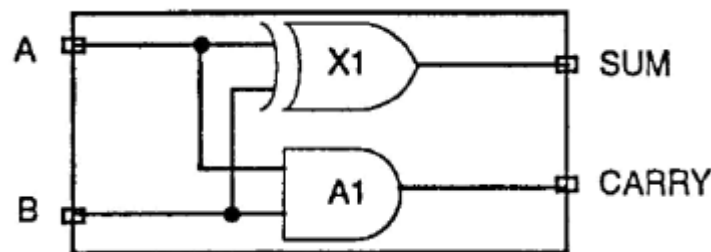


Figure 1.2 : Half adder

Here is an example of an entity declaration for the half-adder circuit

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
entity DECODER2x4 is

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

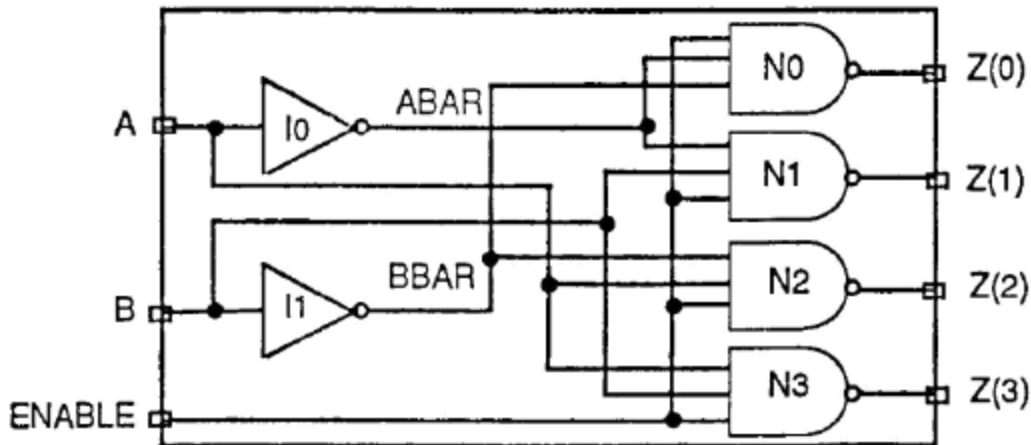


Figure 1.3 : 2*4 decoder

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 (structural modeling)
2. As a set of concurrent assignment statements (dataflow modeling)
3. As a set of sequential assignment statements (behavioral modeling)
4. Any combination of the above three (Mixed modeling)

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 HA-STRUCTURE
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;
```

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;
```

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;

Language elements of VHDL

Basic Language Elements of VHDL are

- Identifiers
- Comments
- Data Objects
- Data Types
- Operators

Identifiers

- Identifiers are used to name items in a VHDL model.

Basic identifier: composed of a sequence of one or more characters, A basic identifier may contain only capital 'A' - 'Z' , 'a' - 'z', '0' - '9', underscore character '_'

- first character must be a letter, last character must NOT be an underscore
- Two underscores cannot occur concurrently
- case insensitive: COUNT, count, Count, counT are all the same
- Keywords can not be used as basic identifiers

Extended identifier: sequence of characters written between two backslashes

Any printable characters can be used including %, \$, *, etc.

- lower case and upper case are distinct
- examples: /2FOR\$/ , /countNow!/ , /&#\$(@#&!!!/

Comments

- Its non executable or readable parameter for understanding purpose.
- The comments to be proceeded by two consecutive hyphens(--)

Example

entity half_adder is

port (a, b: in std_logic; sum, carry: out std_logic);

end half_adder; -- end of entity with entity name

architecture HA-DF of half_adder is

begin

sum <= a xor b; -- a xor with b, result assigned to sum

carry <= a and b; -- a and with b, result assigned to carry

end HA_DF; -- end of architecture with architecture name

Data Objects

– hold a value of a specified type

constant: holds a single value of a specified type and cannot be changed throughout the simulation

constant declaration:

constant RESULT: BIT:=1;

constant FALL_TIME: TIME:=10ns

variable: holds a single value of a specified type but can be changed throughout the simulation

variable ABAR:BIT;

variable STATUS:BIT_VECTOR(3 downto 0);

signal: holds a list of values including current value and a list of possible future values

typically used to model wires and flip-flops

signal DATA_BUS:BIT_VECTOR(0 to 31)

file: same as with any computer file, contains data

Data Types

– Is a name which is associated with a set of values and a set of operations.

Major Data Types:

Scalar Type

Composite Type

Access Type

File Type

There can also be user-defined types and subtypes A *subtype* is a type with a (possibly) added constraint

syntax: subtype subtype_name is base_type range range_constraint;

example: subtype DIGITS is integer range 0 to 9;

Scalar types

Enumeration – defines a type that has a set of user-defined values

type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

'u' unspecified, 'x' unknown, '0' strong zero, '1' strong one

'z' high impedance, 'w' weak unknown, 'l' weak zero, 'h' weak one, '-' don't care

Integer – values fall within the specified integer range

type REG_SIZE is range 0 to 31

subtype WORD is REG_SIZE range 0 to 15

Floating Point –real decimal types

Physical – represent measurement of some physical quantity like time, voltage, or current

Composite Types

– a collection of values

Array Types – collection of values all belonging to a single type

BIT_VECTOR and **STRING** are pre-defined one-dimensional array types

type **DATA_BYTE** is array (0 to 7) of **BIT**;

type **MEMORY** is array (0 to 127) of **DATA_BYTE**;

Record Types – collection of values that may belong to different types

type **MODULE** is

record

SUM : **BIT_VECTOR**(0 to 7);

COUT : **BIT**;

end record;

Access Type

Values belonging to an access type are pointers to a allocated object of some other type.

Example :

 type **PTR** is access **MODULE**;

File Type

Objects of file type represent files in the host environment.

Syntax :

 type **file_type_name** is file of **type_name**;

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

5.1 Addition Operator

5.2 Multiplication Operator

5.3 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:

- 1. NOT**
- 2. AND**
- 3. OR**
- 4. NAND**
- 5. 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: Rotate left**
- 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:

Addition operator (+)

Subtract Operator (-)

Multiplication operator (*)

Division Operator (/)

Modulus (MOD)

Remainder (REM)

Miscellaneous Operator Uses as special cases in VHDL

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

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 behavior),**

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.

architecture HA_CONCURRENT of HALF_ADDER is

begin

sum <= A xor B;

carry <= A and B;

end HALF_ADDER;

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.

architecture 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;

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)

```

variable ABAR,BBAR : std_logic;
begin
ABAR := not A;
BBAR := not B;
Z(0) <= (ABAR and BBAR);
Z(1) <= (ABAR and B);
Z(2) <= (A and BBAR);
Z(3) <= (A and B);
end process;

```

Structural Model

An entity is modeled as a set of components connected by signals, that is, as a net-list. 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. COMPONENT & PORT MAP statements are used to implement structural modeling. The component instantiation statements are concurrent statements, and their order of appearance in the architecture body is therefore not important. A component can be instantiated any number of times. Each instantiation must have a unique component label.

Component Declaration

A component in a structural description must first be declared using a component declaration. A component declaration declares the name and the interface of a component (similar to the entity). The interface specifies the mode and the type of ports.

The syntax of a simple form of component declaration is:

```

COMPONENT Component-Name [IS]
    PORT(List-of-Interface-Ports);
END COMPONENT [Component-Name ];

```

The component-name may or may not refer to the name of an entity already existing in a library. If it does not, it must be explicitly bound to an entity. The binding information can be specified using a configuration. 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)

Component Instantiation

A component instantiation statement defines a association of formal and actual parameters. It associates the signals in the entity with the ports of that component.

A format of a component instantiation statement:

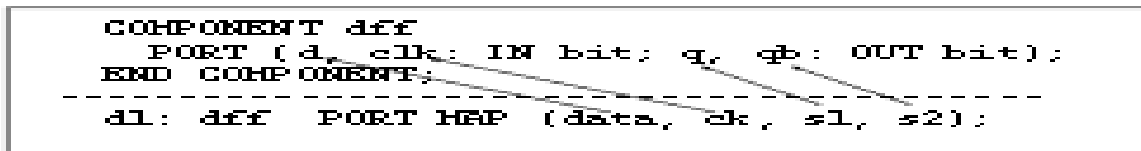
Component-Label: Component-Name PORT MAP (association-list);

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 formals.

There are two ways to perform the association of formals with actuals:

1. Positional association
2. Named association

In positional association, 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.



```
COMPONENT dff
  PORT (d, clk: IN bit; q, qb: OUT bit);
END COMPONENT;
-----
d1: dff PORT MAP (data, ck, s1, s2);
```

The diagram illustrates positional association. It shows a component declaration for 'dff' with four ports: 'd' and 'clk' are inputs of type 'bit', and 'q' and 'qb' are outputs of type 'bit'. Below this, separated by a dashed line, is an instantiation 'd1: dff PORT MAP (data, ck, s1, s2);'. Arrows indicate the mapping: 'd' maps to 'data', 'clk' maps to 'ck', 'q' maps to 's1', and 'qb' maps to 's2'.

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

```
d1 : dff PORT MAP (data, ck, s1, open);
```

In named association, an association-list is of the form:

formal1 => actual1 ,formal2=> actual2, ... formaln=> actualn

For example

```
d1 : dff PORT MAP (d => data, clk => ck, q => s1, qb => s2);
```

In named association, the ordering of the associations is not important since the mapping between the actual and formal is explicitly specified.

Dataflow Model

The Data-Flow modeling is a collections of concurrent statements. All the statements must be write only in the architecture body. There is no meaning to the order of the statements.

There are 3 Data-Flow statement:

- Concurrent Signal Assignment
- Conditional Signal Assignment
- Selected Signal Assignment

Concurrent Signal Assignment

The syntax is:

```
target-signal _1<= expression_1;  
target-signal _2<= expression_2;
```

–Example:

```
entity decoder is  
port (a, b : in std_logic;  
d: out std_logic_vector(0 to 3));  
end decoder;  
architecture DEC_DF of decoder is  
signal s1,s2 : std_logic;  
begin  
s1 <= not a;  
s2 <= not b;  
d(0) <= s1 and s2;  
d(1) <= s1 and b;  
d(2) <= a and s2;  
d(3) <= a and b;  
end DEC_DF;
```

Conditional Signal Assignment Statement

- Also called a When-Else Statement.
- Concurrent statement, thus all signals.
- Similar to a sequential IF-THEN-ELSE statement.

- Select one of several values to drive an output signal.
- Selection based on first condition that is TRUE.

Syntax:

```
target_signal <= value1 when condition1 else
                    value2 when condition2 else
                    ...
                    value9;
```

Example : 2*4 decoder

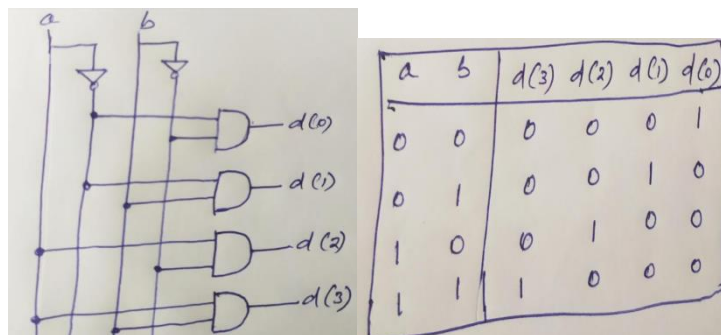


Figure 1.4 : 2*4 decoder logic diagram and truth table

entity decoder is

```
port (a, b : in std_logic;
      d: out std_logic_vector(0 to 3));
end decoder;

architecture DEC_CS of decoder is
begin
  d <= "0001" when (a = '0' and b = '0') else
        "0010" when (a = '0' and b = '1') else
        "0100" when (a = '1' and b = '0') else
        "1000";
end DEC_CS;
```

Selected Signal Assignment Statement

- Also called a With-Select-When statement.
- Concurrent statement, thus all signals.
- Similar to a sequential CASE statement.
- Select one of several values to drive an output signal.
- Selection based on all possible values of a selector expression.

syntax:

with expression select

```
target_signal <= value1 when condition1,  
                value2 when condition2,  
                ...  
                value9 when condition9;
```

Example : 2*4 decoder

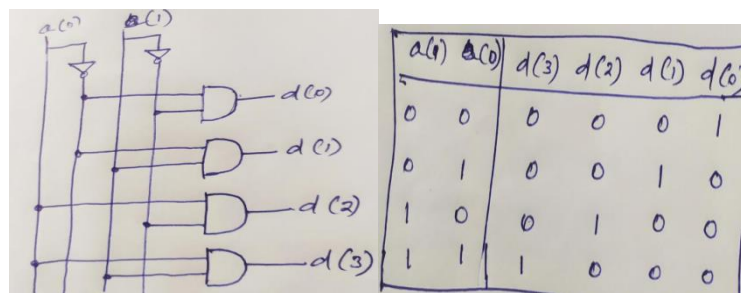


Figure 1.5 : 2*4 decoder logic diagram and truth table

entity decoder is

```
port (a : in std_logic_vector (0 to 1);
```

```
d: out std_logic_vector(0 to 3));
```

```
end decoder;
```

architecture DEC_SS of decoder is

```
begin
```

```
with a select
```

```
d <= "0001" when "00",
```

```
    "0010" when "01",
```

```
    "0100" when "10",
```

```
    "1000" when "11";
```

```
end DEC_SS;
```

Behavioral Model

The process statement contains sequential statements that describe the functionality of an entity in sequential terms. The sensitivity list is a set of signals which will cause the process to execute in sequential order when an event occurs.

Syntax

architecture architecture_name of entity_name is

begin

 process(sensitivity_list)

 [variable_declarations;]

begin

 sequential assignment statements;

end process;

end architecture_name ;

Some of the sequential statements in Behavioral model are

1.Variable Assignment Statement

2.Signal Assignment Statement

3.Wait Statement (wait on, wait until, wait for)

4.If Statement (conditional)

5.Case Statement (Selection)

6.Loop Statement (for loop, while loop, no loop)

7.Null Statement (No operation)

8.Exit Statement (Exit from the loop)

9.Next statement (Exit from one loop, and execute next loop)

10.Assertion Statement (Modelling constrain)

11.Report Statement (Display Message)

12.Procedure call Statement (Linkage)

13.Return Statement

If statement

– selects statements for execution based upon a condition

Syntax

if condition_1 then

sequential_statement_1;

elsif condition_2 then

sequential_statement_2;

:

:

else sequential_statement_n;

end if;

Example : 2*4 decoder

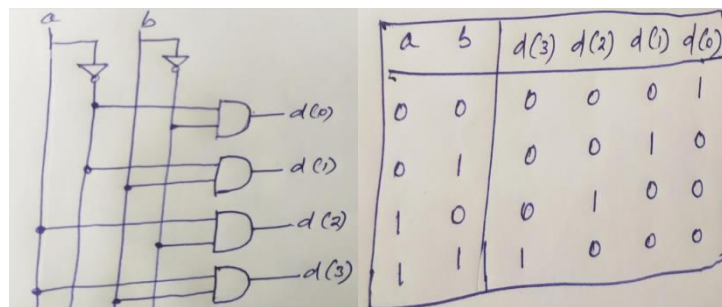


Figure 1.6 : 2*4 decoder logic diagram and truth table

entity decoder is

port (a, b : in std_logic; d: out std_logic_vector(0 to 3));

end decoder;

architecture DEC_IF of decoder is

begin

process (a,b)

begin

if (a = '0' and b = '0') then

d <= "0001";

elsif (a = '0' and b = '1') then

```

    d <= "0010";
elsif (a = '1' and b = '0') then
    d <= "0100";
else d<= "1000";
end if;
end process;
end DEC_IF;

```

Case statement

– selects one branch of execution from a list of many based upon selected expression

Syntax:

case expression is

when choice_1 => statement_1;

when choice_2 => statement_2;

:

when choice_n => statement_n;

end case;

Example : 2*4 decoder

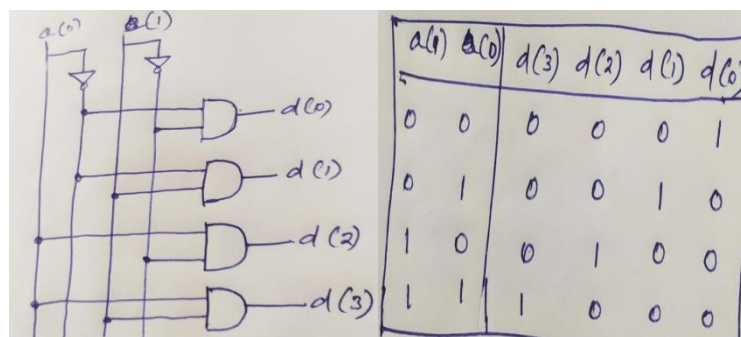


Figure 1.7 : 2*4 decoder logic diagram and truth table

entity decoder is

```
port (a : in std_logic_vector(0 to 1); d: out std_logic_vector(0 to 3));
```

```
end decoder;
```

architecture DEC_CASE of decoder is

begin

process (a)

begin

case a is

when “00” => d <= “0001”;

when “01” => d <= “0010”;

when “10” => d <= “0100”;

when “11” => d <= “1000”;

end case;

end process;

end DEC_CASE;

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.**
- 7. Stuart Sutherland, “RTL Modeling With System Verilog for Simulation and Synthesis: Using System Verilog for ASIC and FPGA Design”,1st Edition, Sutherland HDL,Inc., 2017.**
- 8. Simon Monk, “Programming FPGAs: Getting Started with Verilog”, 1st Edition, Tata McGraw Hill,2016.**
- 9. User Guide – “7 Series FPGAs Configurable Logic Block” - (WWW.XILINX.COM)**



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 ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - II
PROGRAMMING IN HDL – SEC1406

II. 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 2.1 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 gatelevel 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.

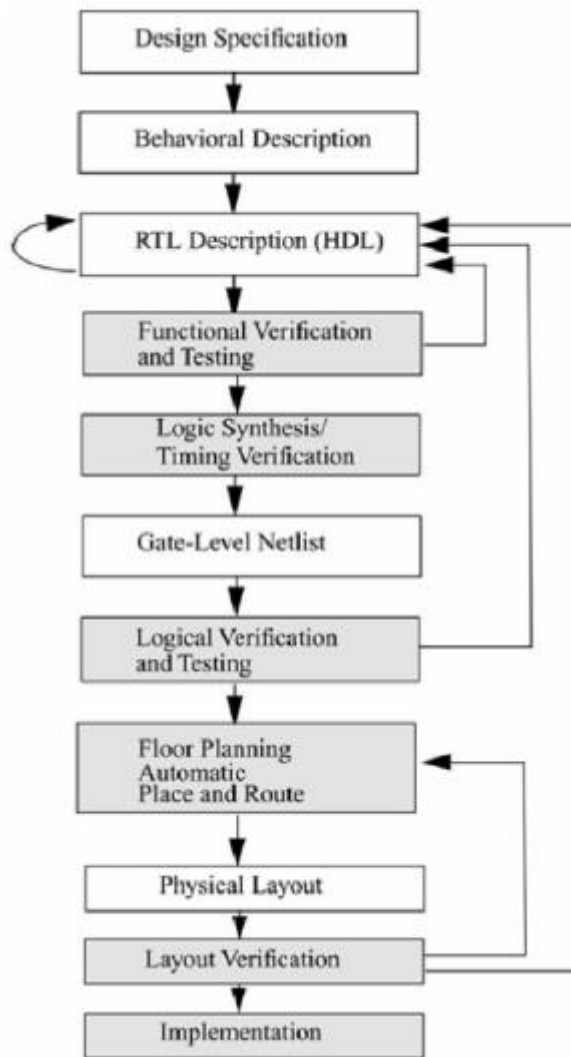


Figure 2.1: Typical Design Flow

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 2.2 shows the top-down design process.

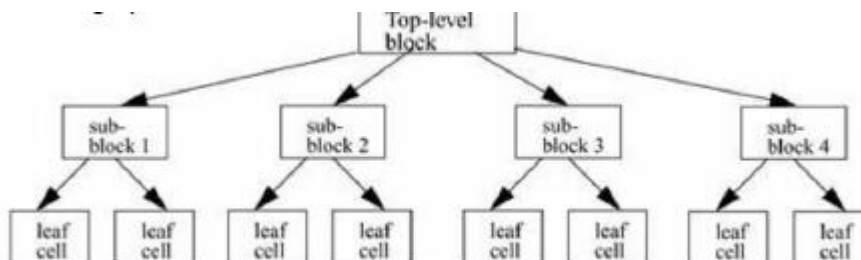


Figure 2.2: Top-down Design Methodology

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 2.3 shows the bottom-up design process

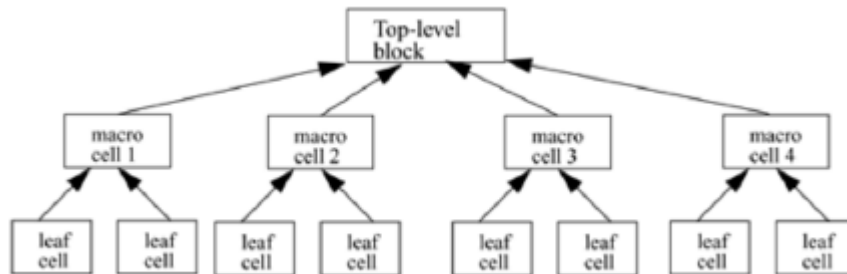


Figure 2.3: Bottom-up Design Methodology

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) have 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

triereg

tri1

tri0

supply0

supply1

Variable types

There are five different kinds of variable types

reg

integer

time

real

realtime

memory

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 3×10^6). 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 2.1 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	~	bitwise	one
	&	negation	two
		bitwise and	two
	^	bitwise or	two
	&	reduction and	one
	~&	reduction nand	one
Reduction		reduction or	one
	~	reduction nor	one
	>>	Right shift	Two
	<<	Left shift	Two
Shift	>>>	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.**
- 7. Stuart Sutherland, “RTL Modeling With System Verilog for Simulation and Synthesis: Using System Verilog for ASIC and FPGA Design”,1st Edition, Sutherland HDL,Inc., 2017.**
- 8. Simon Monk, “Programming FPGAs: Getting Started with Verilog”, 1st Edition, Tata McGraw Hill,2016.**
- 9. User Guide – “7 Series FPGAs Configurable Logic Block” - (WWW.XILINX.COM)**



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 ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - III
PROGRAMMING IN HDL – SEC1406

III. 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
- 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 ] ( term1 , 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_name1 ] ( term11 , term12 , . . . , term1N ),  
[ instance_name2 ] ( term21 , term22 , . . . , term2N ),  
.....  
[ instance_nameM ] ( termM1 , termM2 , . . . , termMN );
```

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 3.1.

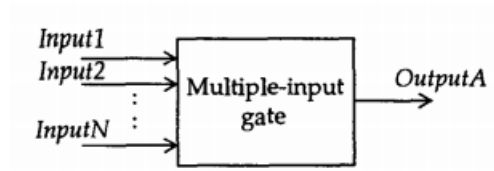


Figure 3.1: Multiple Input Gates

and A1 (Out1, In1, In2);
 and RBX (Sty, Rib, Bro, Qit, Fix) ;
 xor (Bar, Bud[0],Bud[1],Bud[2]),
 (Car, Cut[0], Cut[1]),
 (Sar, Sut[2], Sut[1], Sut[0], Sut[3]);

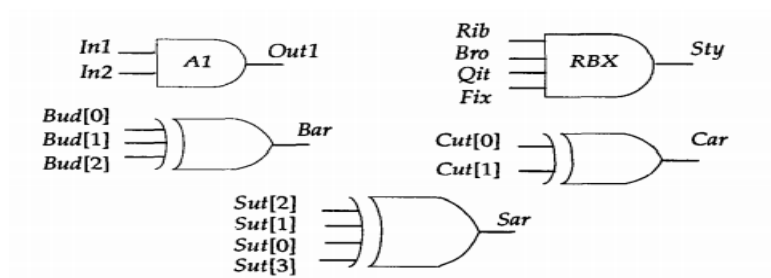


Figure 3.2: Multiple Input Gate examples

The first gate instantiation is a 2-input and gate with instance name A1, output Out1 and with two inputs, In1 and In1. 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 an xor 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 3.3: 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] (Out1, Out2, ..., OutN, InputA);`

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

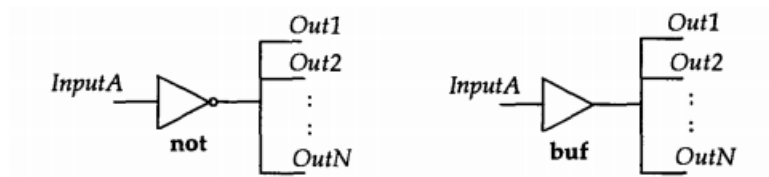


Figure 3.4: Multiple Output Gates

Here are some examples.

```
bufB1 [Fan[0], Fan[1], Fan[2], Fan[3], Clk);
```

```
notN1 {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	not	0	1	x	z
(output)	0	1	x	x	(output)	1	0	x	x

Figure 3.5: 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. For a notifO gate, 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.

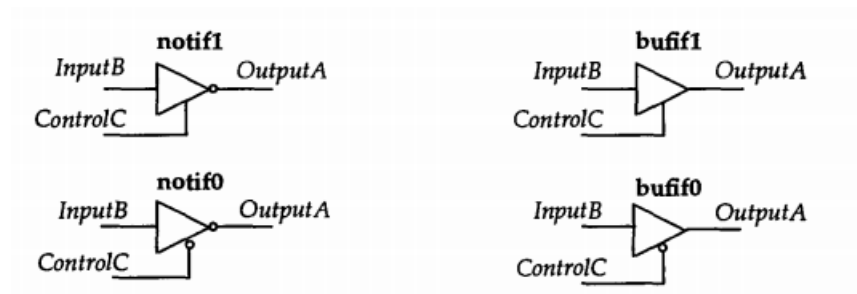


Figure 3.6: Tristate Gates

Here are some examples.

`bufif1 BF1 [Dbus, MemData, Strobe];`

`notif0 NT2 {Addr, Abus, Probe};` The `bufif1` gate `BF1` drives the output `Dbus` to high-impedance state when `Strobe` is 0, else `MemData` 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
Data	0	0	z	0/z	0/z
	1	1	z	1/z	1/z
	x	x	z	x	x
	z	x	z	x	x

bufif1		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	x	x	x

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

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

Figure 3.7: 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 pull down 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 control is 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.

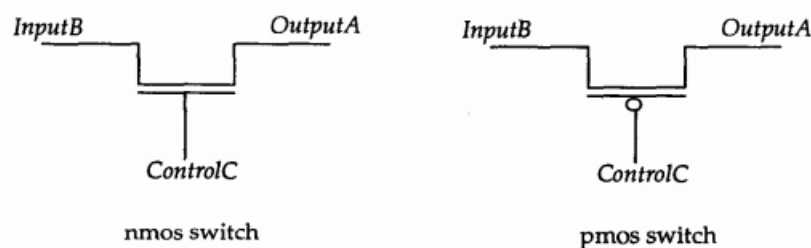


Figure 3.8: nMOS and pMOS switches

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 rmos		Control				nmos rnmos		Control			
		0	1	x	z			0	1	x	z
Data	0	0	z	0/z	0/z	Data	0	z	0	0/z	0/z
	1	1	z	1/z	1/z		1	z	1	1/z	1/z
	x	x	z	x	x		x	z	x	x	x
	z	z	z	z	z		z	z	z	z	z

Figure 3.9: Truth table of MOS switches

The CMOS (Complimentary 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 (rmos) and an nmos (rnmos) switch with common outputs and common inputs.

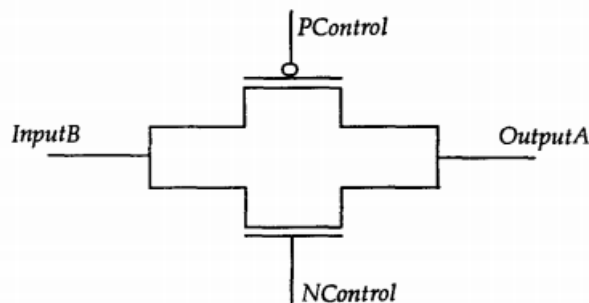


Figure 3.10: (r)cmos switch

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 1 for 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.

Examples:

4 X 1 Multiplexer:

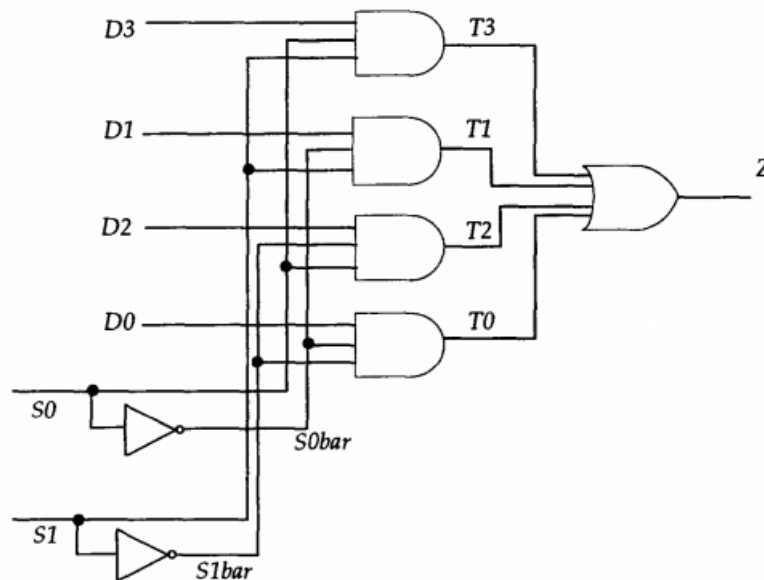


Figure 3.11: 4 X 1 Multiplexer

```
module MUX4x1 (Z, D0, D1, D2, D3, S0, S1);
```

```
output Z;
```

```

input DO, D1, D2, D3, SO, S1;
and (TO, DO, SObar, Slbar),
(Tl, D1, 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:

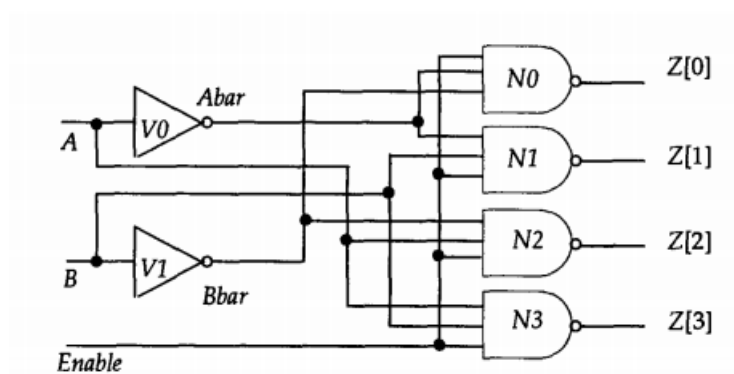


Figure 3.12: 2 to 4 Decoder

```

module DEC2x4 {A, B, Enable, Z} ;
input A, B, Enable;
output [0:3] Z;
wire Abar, Bbar;
not
    V0 (Abar, A) ,
    V1 (Bbar, B);
nand
    N0 (Z[0], Enable, Abar, Bbar),
    N1 (Z[1], Enable, Abar, B),
    N2 (Z[2], Enable, A, Bbar),

```

```

    N3 (Z[3] , Enable, A, B);
endmodule

```

Master Slave Flip-flop:

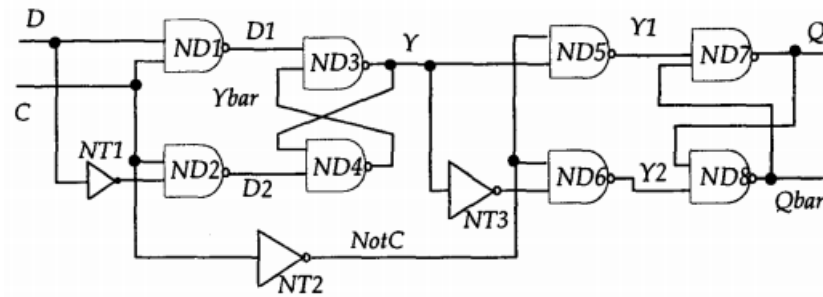


Figure 3.13: Master Slave Flip-flop

```

module MSDFFF (D, C, Q, Qbar) ;
input D, C; output Q, Qbar;
not
    NT1 (NotD, D),
    NT2 (NotC, C),
    NT3 (NotY, Y);
nand
    ND1 (D1, D, C),
    ND2 (D2, C, NotD),
    ND3 (Y, D1, Ybar),
    ND4 (Ybar, Y, D2),
    ND5 (Y1, Y, NotC),
    ND6 (Y2, NotY, NotC),
    ND7 (Q, Qbar, Y1), ND8 (Qbar, Y2, Q);
endmodule

```


Parity Generator:

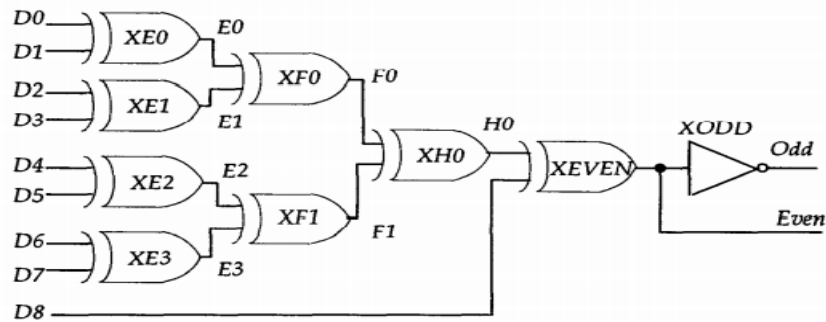


Figure 3.14: Parity Generator

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

xor
    XEO (E0, D[0], D[1]),
    XE1 (E1, D[2], D[3]),
    XE2 (E2, D[4], D[5]),
    XE3 (E3, D[6], D[7]),
    XFO (F0, E0, E1),
    XF1 {F1, E2, E3}, XHO {H0, F0, F1},
    XEVEN {Even, D[8], H0};

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;
```

```
endtable
```

```
endprimitive
```

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.

```
assign addr[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 assign

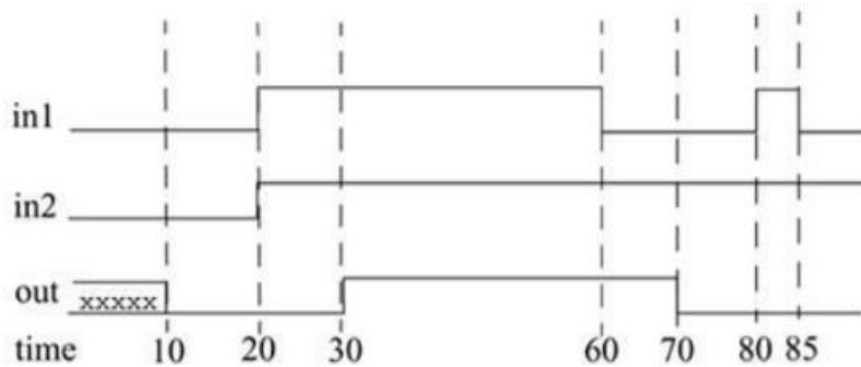


Figure 3.15: Delays

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 MSDFD_DF (D, C, Q, Qbar) ;
input D, C; output Q, Qbar;
wireNotC, NotD, NotY, Y, D1, D2, Ybar, Y1, Y2;
assignNotD = ~ D;
assign Note = ~ C;
assignNotY = ~ Y;
assign D1= - (D & C) ;
assign D2 = ~ (C &NotD);
assign Y = ~ (D1 St Ybar);
assignYbar = ~ (Y & D2);
assignY1 = ~ (y & Note);
assign Y2 = - (NotY&NotC);
assign Q = ~ (Qbar&Y1);
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, $N = \sim N$;
- 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. Repeat-loop
- iii. While-loop
- iv. For-loop

i. Forever loop

The syntax for this form of loop statement is:

forever

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 specifies the 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:

```

module module_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 register. 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;
```

```
// Redclarations:
```

```
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_DELAY 01 (Cout, CI, C2) ;  
endmodule
```

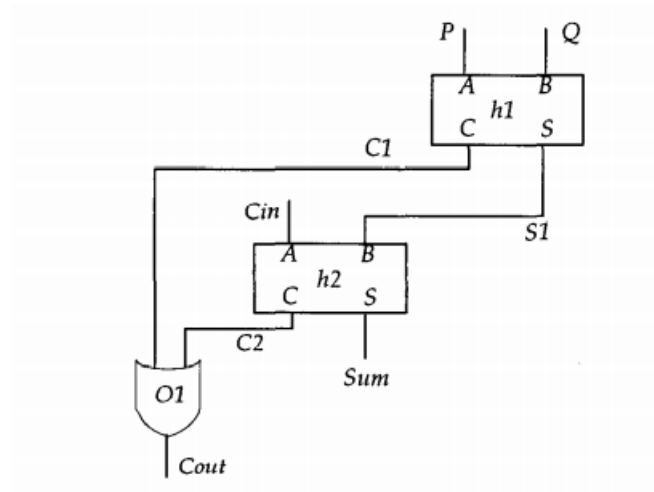


Figure 3.16: Full Adder using Two Half Adders

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[1] and Mpr[4] is connected to Ppy[2].

Examples:

Decade counter:

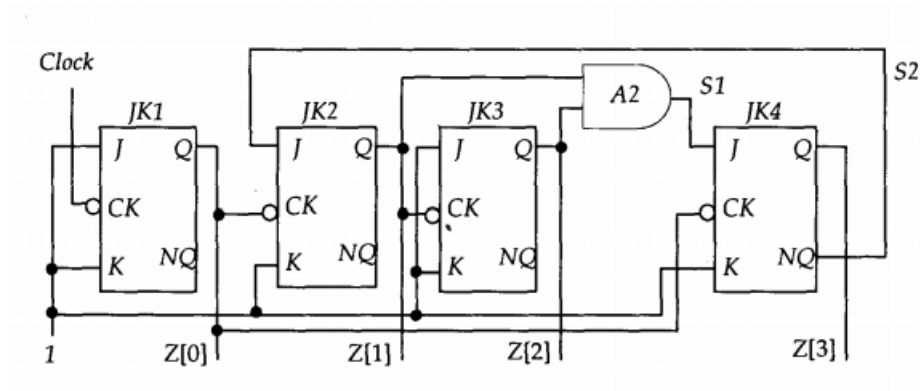


Figure 3.17: Decade counter

```

module Decade_Ctr(Clock, Z);
input Clock;
output [0:3] Z;
wire S1, S2;
and A1 {S1, Z[2], Z[1]}; // 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

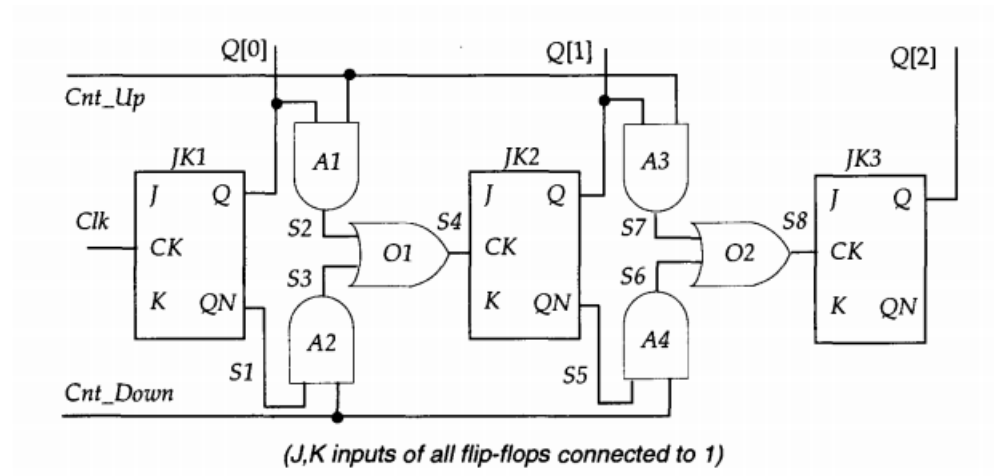


Figure 3.18: 3 bit UP-DOWN counter

```

module Up_Down (Clk, Cnt_Up, Cnt_Down, Q);
input Clk, Cnt_Up, Cnt_Down;
output [0:2] Q;
wire S1, S2, S3, S4, S5, S6, S7, S8;
JK_FF JK1 (1'b1, 1'b1, Clk, Q[0], S1),
JK2 (1'b1, 1'b1, S4, Q[1], S5),
JK3 (1'b1, 1'b1, S8, Q[2], );
and A1 (S2, Cnt_Up, Q[0]),
    A2 (S3, S1, Cnt_Down),
    A3 (S7, Q[1], Cnt_Up),
    A4 (S6, S5, Cnt_Down);
or O1 (S4, S2, S3),
    O2 (S8, S7, S6);
endmodule

```

TEXT / REFERENCE BOOKS

- 1. J.Bhaskar, “A VHDL Primer”, Prentice Hall of India Limited. 3rd edition 2004**
- 2. Stephen 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.**
- 7. Stuart Sutherland, “RTL Modeling With System Verilog for Simulation and Synthesis: Using System Verilog for ASIC and FPGA Design”, 1st Edition, Sutherland HDL, Inc., 2017.**
- 8. Simon Monk, “Programming FPGAs: Getting Started with Verilog”, 1st Edition, Tata McGraw Hill, 2016.**
- 9. User Guide – “7 Series FPGAs Configurable Logic Block” - (WWW.XILINX.COM)**



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 ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - IV
PROGRAMMING IN HDL – SEC1406

IV. 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 arguments pass 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>);

Example

module simple_ task();

task convert;

input [7:0] temp_in;

output [7:0] temp_out;

```

begin
temp_out = (9/5) *( temp_in+ 32)
end
endtask
endmodule

```

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** returns 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 an 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.

We can declare these data types for the test bench of the AND module.

```
reg A, B;  
wire C;
```

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.

Start with declaring the module as for any Verilog file. We can name the module as `and_tb`

```
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;
```

```
wire C;
```

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.

Let's get to applying the stimulus.

```
initial
```

```
begin
```

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

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

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

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

```
end
```

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@(posedge clk)
begin
if(rst == 1)
begin
q <= 0;
qbar <= 1;
end
else
begin
q <= d;
qbar <= ~d;
end
end
end module
```



```
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;
```

```
wire Q, QBAR;
```

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
```

```
end
```

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

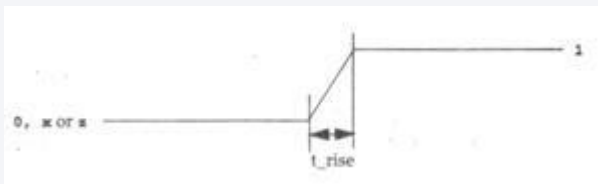
In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

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

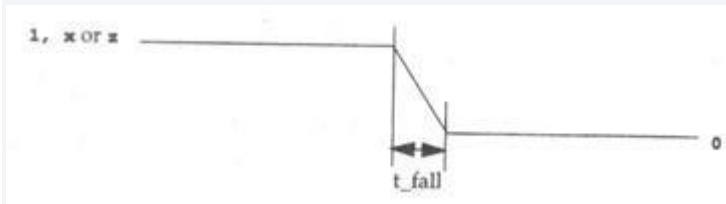
Rise delay

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



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



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 assignment 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;
```

```
endmodule
```

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;
```

```
    a = #10 0;
```

```
    b = a;
```

```
endmodule
```

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 **@**.

Example

```
event received_data;  
always @(posedge clock)  
begin  
if (last_data_packet)  
□received_data;  
end  
always @(received_data)  
data_buf={data_pkt[0],data_pkt[1]};
```

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)  
q=d;  
end
```

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.

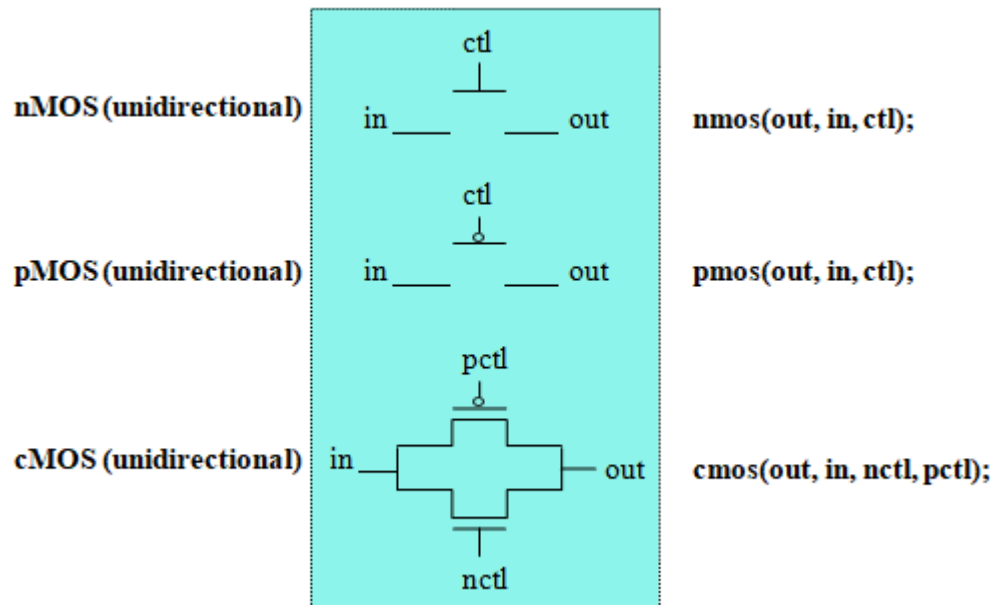


Figure 4.1 : MOS switches

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.

nmos		control			
		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

pmos		control			
		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.

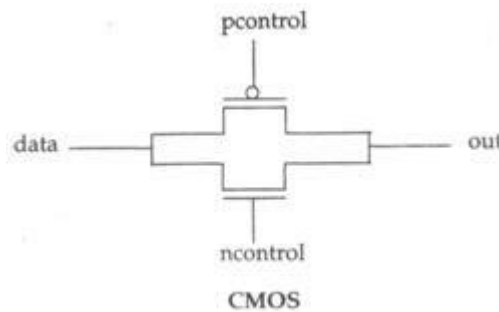


Figure 4.2 : CMOS switch

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.

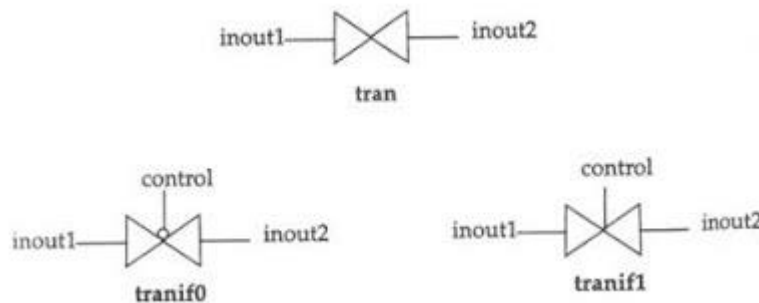


Figure 4.3 : Bidirectional switches

The *tran* switch acts as a buffer between the two signals *inout1* and *inout2*. Either *inout1* or *inout2* can be the driver signal. The *tranif0* switch connects the two signals *inout1* 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 *tranif1* switch conducts if the *control* signal is a logical 1.

These switches are instantiated as shown in below.

```
tran tl(inout1, inout2); //instance name tl is optional
```

```
tranif0 (inout1, 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

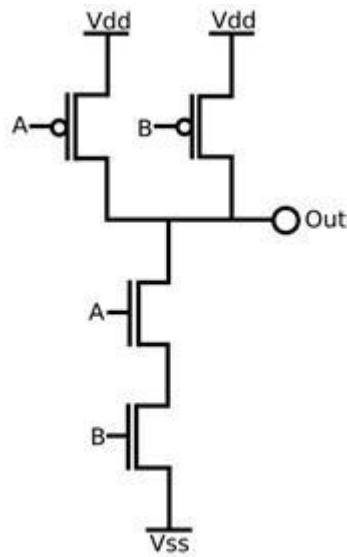


Figure 4.4 : CMOS NAND

```
module my_nand (Out,A,B);  
input A,B;  
output 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. Stephen 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.**
- 7. Stuart Sutherland, “RTL Modeling With System Verilog for Simulation and Synthesis: Using System Verilog for ASIC and FPGA Design”,1st Edition, Sutherland HDL,Inc., 2017.**
- 8. Simon Monk, “Programming FPGAs: Getting Started with Verilog”, 1st Edition, Tata McGraw Hill,2016.**
- 9. User Guide – “7 Series FPGAs Configurable Logic Block” - (WWW.XILINX.COM)**



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 ELECTRICAL AND ELECTRONICS

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

UNIT - V
PROGRAMMING IN HDL – SEC1406

V. REALIZING APPLICATIONS IN FPGA

FPGA Design Flow

The ISE® design flow comprises the following steps: design entry, design synthesis, design implementation, and Xilinx® device programming. Design verification, which includes both functional verification and timing verification, takes places at different points during the design flow. This section describes what to do during each step. For additional details on each design step, click on a link below the following figure.

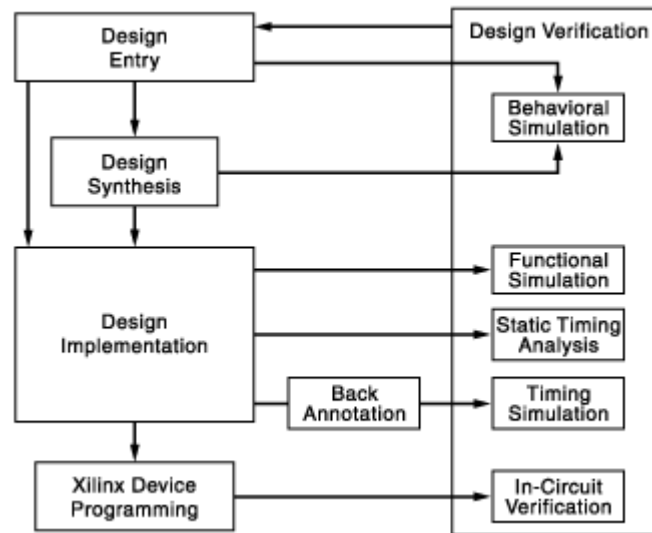


Figure 5.1 : ISE design flow

- Design Entry
- Design Synthesis and Verification
- Design Implementation and Verification
- Device Programming
- In-Circuit Verification

Design Entry

1. Create an ISE project as follows:
2. Create a project.
3. Create files and add to project, including a user constraints (UCF) file.
4. Add any existing files to project.
5. Assign constraints such as timing constraints, pin assignments, and area constraints.

Functional Verification

We can verify the functionality of our design at different points in the design flow as follows:

- **Before synthesis, run behavioral simulation (also known as RTL simulation).**
- **After Translate, run functional simulation (also known as gate-level simulation), using the SIMPRIM library.**
- **After device programming, run in-circuit verification.**

Design Synthesis

The synthesis process will check code syntax and analyze the hierarchy of our design which ensures that our design is optimized for the design architecture that we have selected. The resulting netlist is saved to an NGC file (for Xilinx® Synthesis Technology (XST)) or an EDIF file (for Precision, or Synplify/Synplify Pro).

The synthesis process can be used with the following synthesis technology tools. Select one of the following for information about running your synthesis tool:

- **Xilinx Synthesis Technology (XST)**
- **Precision from Mentor Graphics Inc.**
- **Synplify and Synplify Pro from Synplicity Inc.**

Design Implementation

Implementation of design as follows:

1. **Implement design, which includes the following steps:**
 - **Translate**
 - **Map**
 - **Place and Route**
2. **Review reports generated by the Implement Design process, such as the Map Report or Place & Route Report, and change any of the following to improve our design:**
 - **Process properties**
 - **Constraints**
 - **Source files**
3. **synthesize and implement our design again until design requirements are met.**

Timing Verification

We can verify the timing of our design at different points in the design flow as follows:

Run static timing analysis at the following points in the design flow:

- **After Map**
- **After Place & Route**

Run timing simulation at the following points in the design flow:

- **After Map (for a partial timing analysis of CLB and IOB delays)**
- **After Place and Route (for full timing analysis of block and net delays)**

Xilinx Device Programming

Program Xilinx device as follows:

- **Create a programming file (BIT) to program our FPGA.**
- **Generate a PROM or ACE file for debugging or to download to device. Optionally, create a JTAG file.**
- **Use iMPACT to program the device with a programming cable.**

Xilinx Artix-7 architecture

The Artix-7 FPGA consists of Logic Blocks, Block RAM, DSP blocks, and a global routing network. We will spend most of our time discussing the Logic Blocks. But before we do, realize that modern reconfigurable logic exists because logic designs can easily be expressed in terms of medium scale logic building blocks such as registers, shift registers, multiplexers, counters, adders, subtractors, and comparators. Consider the following output from the Xilinx ISE during the synthesis of Lab 4 on the Spartan-6 FPGA.

ISE decomposed my VHDL design into basic building blocks. This is one reason we insisted on certain coding practices throughout the semester - they increase the likelihood that our design will be efficiently mapped into these basic building blocks. However, consider how the actual FPGA can be configured to realize these basic building blocks. This complicated process is what the Xilinx software does.

Logic Blocks

A configurable logic block (CLB) is a basic block used to implement the logic behind the VHDL designs we have been working on all semester. In FPGAs, hundreds or thousands of CLBs are laid out in an array (commonly a switch matrix) known as the global routing

network. All of the CLBs on the FPGA are connected to each other. On the Artix-7 (and other Xilinx 7-series boards), each CLB contains two Logic Slices (discussed in the following section). The logical layout of a CLB can be seen in the Figure below.

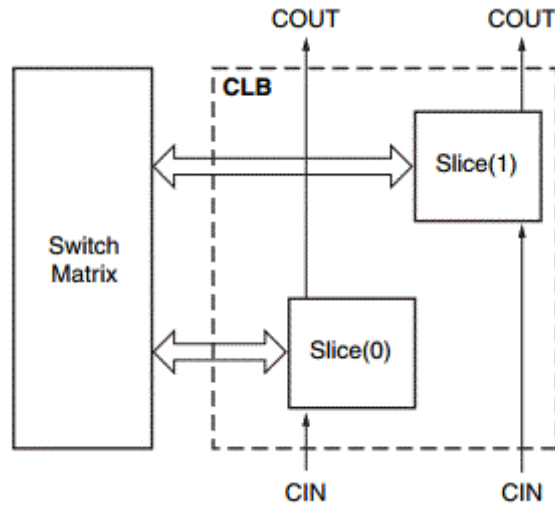


Figure 5.2 : CLB diagram

Logic Slices

The Artix-7 on our board (Artix-7 7A200T) has a total of 33,650 logic slices (16,825 CLBs). Each logic slice contains four 6-input LUTs and eight flip-flops. This corresponds to 134,600 total 6-input LUTs.

There are three possible types of logic slices: SLICEM, SLICEL, and SLICEX. However, in the Artix-7, SLICEX slices are unused; of the 33,650 logic slices, 22,100 are SLICEL and 11,550 are SLICEM.

In the logic slices, three major SLICEM subsystems can be seen: 1. the four 6-input LUTs, 2. the eight flip-flops, and 3. the fast carry logic.

1. Look-up tables

If you need a refresher on how a hardware LUT works, In a SLICEM, there are four 64x1 RAMs which are used to realize 5 or 6-variable functions; the truth table for the function is stored in the RAM and the inputs are used as the input addresses. As an example, let's try to realize a full adder using RAM. In class, we will derive the truth table for sum and carry and show how they can be inserted into a LUT. It is important for the further development of the lecture to point out that $\text{sum} = a \oplus b \oplus c$ and that you can represent $\text{cout} = ((a \oplus b) \text{ and } c)$ or $(a \text{ and } b)$. This last form is pretty nutty, but is also very useful.

2. Flip Flops

There are 8 flip flops in each logic slice.

3. Fast Carry Logic

The fast carry logic is designed explicitly to realize a variation of a carry look-ahead adder. Consider the construction of a 4-bit adder with inputs $A=a_3,a_2,a_1,a_0$, $B=b_3,b_2,b_1,b_0$, and a carry in c_0 . Each slice of the adder can either generate a carry bit or propagate its carry in to the carry out.

- Propagate -- p_i is equal to 1 when the inputs to a bit slice are such that any carry in will be propagated.
- Generate -- g_i is equal to 1 when the inputs to a bit slice are such that a carry will be generated.

We can represent the cout of a slice as $c_{out} = g + p \cdot c_{in}$. This arrangement is effectively what is happening in the carry logic block in the middle of each logic slice.

Interconnect

A logical figure of how the CLBs on the Artix-7 are interconnected to each other can be seen in the Figure 5.3

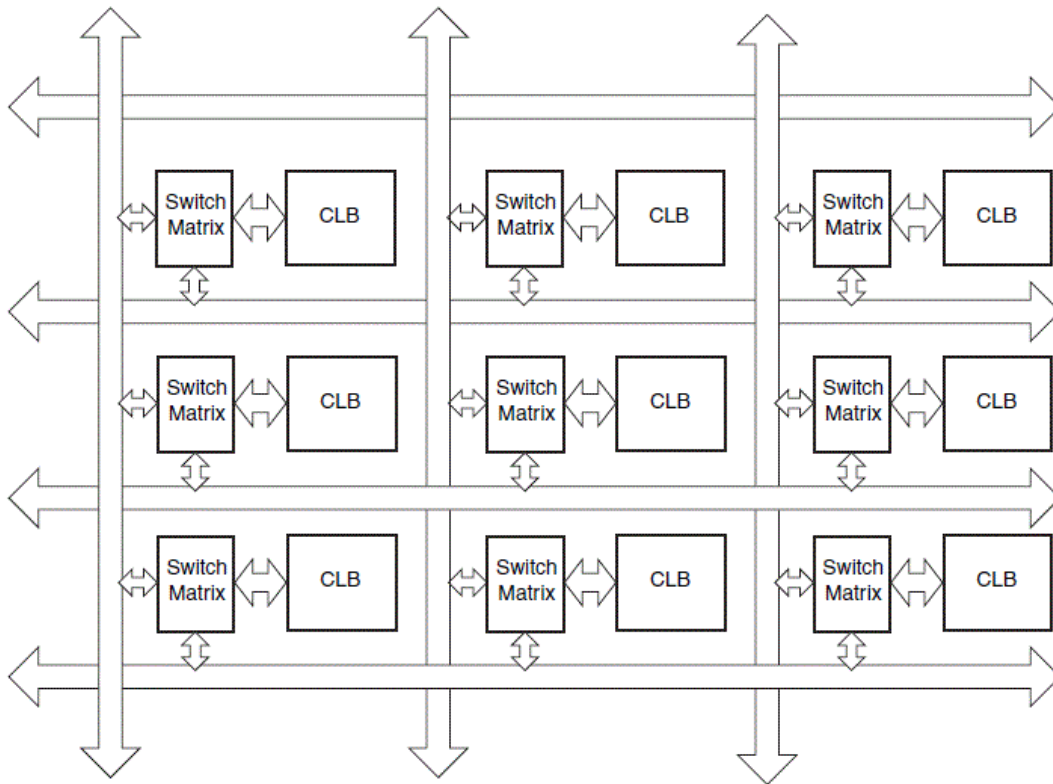
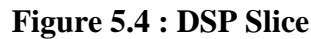


Figure 5.3 : Interconnect diagram

Apart from the slices which make up the CLBs discussed above, the Artix-7 also contains DSP slices. The Artix-7 we are using contains 700 DSP48E1 slices. Each DSP48E1 slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator. A picture of a DSP slice can be seen in the Figure 5.4.



The 7 series configurable logic block (CLB) provides advanced, high-performance FPGA logic:

- CLBs are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix (shown in Figure 5.5). A CLB element contains a pair of slices.

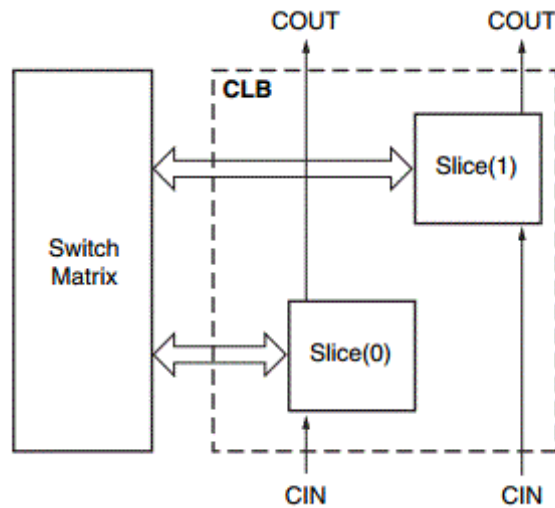


Figure 5.5 : CLB diagram

The LUTs in 7 series FPGAs can be configured as either a 6-input LUT with one output, or as two 5-input LUTs with separate outputs but common addresses or logic inputs. Each 5-input LUT output can optionally be registered in a flip-flop. Four such 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB. Four flip-flops per slice (one per LUT) can optionally be configured as latches. In that case, the remaining four flip-flops in that slice must remain unused. Approximately two-thirds of the slices are SLICEL logic slices and the rest are SLICEM, which can also use their LUTs as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two SRL16s. Modern synthesis tools take advantage of these highly efficient logic, arithmetic, and memory features. Expert designers can also instantiate them.

7 Series CLB Features

The 7 series CLB is identical to that in the Virtex®-6 FPGA family. The CLB is very similar to that of the Spartan®-6 FPGA family with these differences:

- Columnar architecture
- Scales easily to higher densities
- More routing between CLBs
- SLICEL and SLICEM only (no Spartan-6 FPGA SLICEX)
- All slices support carry logic
- More optimized

The common features in the CLB structure simplify design migration from the Spartan-6 and Virtex-6 families to the 7 series devices. The unique floor plan means that location constraints

should be removed before implementing designs originally targeted to earlier FPGAs. The interconnect routing resources are increased in size, quantity, and flexibility relative to the Virtex-6 FPGA family, improving the quality of automatic place and route results.

Device Resources

The CLB resources are scalable across all the 7 series families, providing a common architecture that improves efficiency, IP implementation, and design migration. The number of CLBs and the ratio between CLBs and other device resources differentiates the 7 series families. Migration between the 7 series families does not require any design changes for the CLBs.

Device capacity is often measured in terms of logic cells, which are the logical equivalent of a classic four-input LUT and a flip-flop. The 7 series FPGA CLB six-input LUT, abundant flip-flops and latches, carry logic, and the ability to create distributed RAM or shift registers in the SLICEM, increase the effective capacity. The ratio between the number of logic cells and 6-input LUTs is 1.6:1.

Recommended Design Flow

CLB resources are inferred for generic design logic and do not require instantiation. Good HDL design is sufficient. A few items to note:

- CLB flip-flops have either a set or a reset. The designer must not use both set and reset.
- Flip-flops are abundant. Pipelining should be considered to improve performance.
- Control inputs are shared across a slice or CLB. The number of unique control inputs required for a design should be minimized. Control inputs include clock, clock enable, set/reset, and write enable.
- A 6-input LUT can be used as a 32-bit shift register for efficient implementation.
- A 6-input LUT can be used as a 64 x 1 memory for small storage requirements.
- Dedicated carry logic implements arithmetic functions effectively.

These steps indicate the recommended design flow:

1. Implement the design using preferred methodologies (HDL, IP, etc.).
2. Evaluate utilization reports to determine resources used. Check to make sure arithmetic logic, distributed RAM, and SRL are used, when helpful.
3. Consider flip-flop usage. a. Pipeline for performance b. Use dedicated flip-flops at the outputs of dedicated resources (block RAM, DSP) c. Allow shift registers to use SRL (avoid set/resets)

4. Minimize the use of set/resets.

Pinout Planning

Although the use of most resources affects the resulting device pinout, CLB usage has little effect on pinouts because they are distributed throughout the device. The ASMBL™ architecture provides maximum flexibility with CLBs on both sides of most I/Os.

The best approach is to let the tools choose the I/O locations based on the FPGA requirements. Results can be adjusted if necessary for board layout considerations. The timing constraints should be set so that the tools can choose optimal placement for the design requirements.

Carry logic cascades vertically up a column, so wide arithmetic buses might drive a vertical orientation to other logic, including I/O.

While most 7 series devices are available in flip-chip packages, taking full advantage of the distributed I/O in the ASMBL architecture, the smaller devices are available in wire-bond packages at a lower cost. In these packages, some pins are naturally closer to the I/Os and special resources than others, so pin placement should be done after the internal logic is defined.

Slice Description

Every slice contains:

- Four logic-function generators (or look-up tables)
- Eight storage elements
- Wide-function multiplexers
- Carry logic These elements are used by all slices to provide logic, arithmetic, and ROM functions.

In addition, some slices support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers. Slices that support these additional functions are called SLICEM; others are called SLICEL. SLICEM represents a superset of elements and connections found in all slices. Each CLB can contain two SLICEL or a SLICEL and a SLICEM.

SLICEM and SLICEL

The components discussed after this all exist as pieces within a slice. This fact does *not* mean that the whole is simply the sum of its parts! There are some unique features to the slices themselves that allow an FPGA to expand its functionality.

First, the slices within a CLB are not connected to each other. They are physically oriented in a similar fashion to the above diagram so that they may be connected with the same slice type (SLICEM or SLICEL) within CLBs above or below, creating columns. This allows interconnections between SLICEM or SLICEL in a column to create large scale functions.

The distinguishing feature of the two slice types is the configurability of the SLICEM. SLICEM can be configured so that the look-up tables within it can act as shift registers or as data storage (creating distributed memory on the chip) in addition to its normal logic functionality.

A note on naming: the ‘M’ may be an indication of its ability to act as distributed memory, while the ‘L’ may be an indication of its exclusive logic functionality. This is just speculative but it can be helpful to remember which is which.

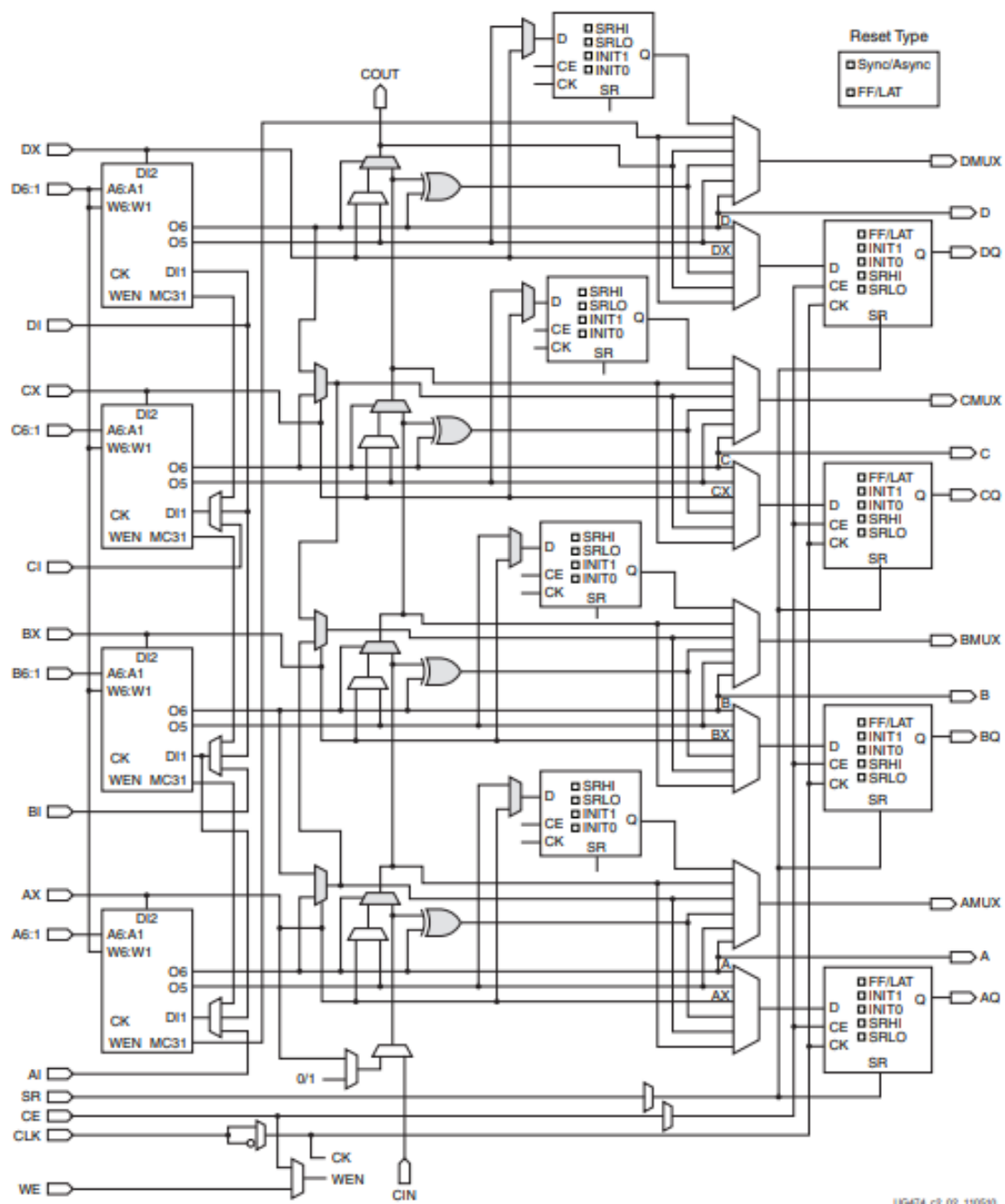


Figure 5.6 : SLICEM

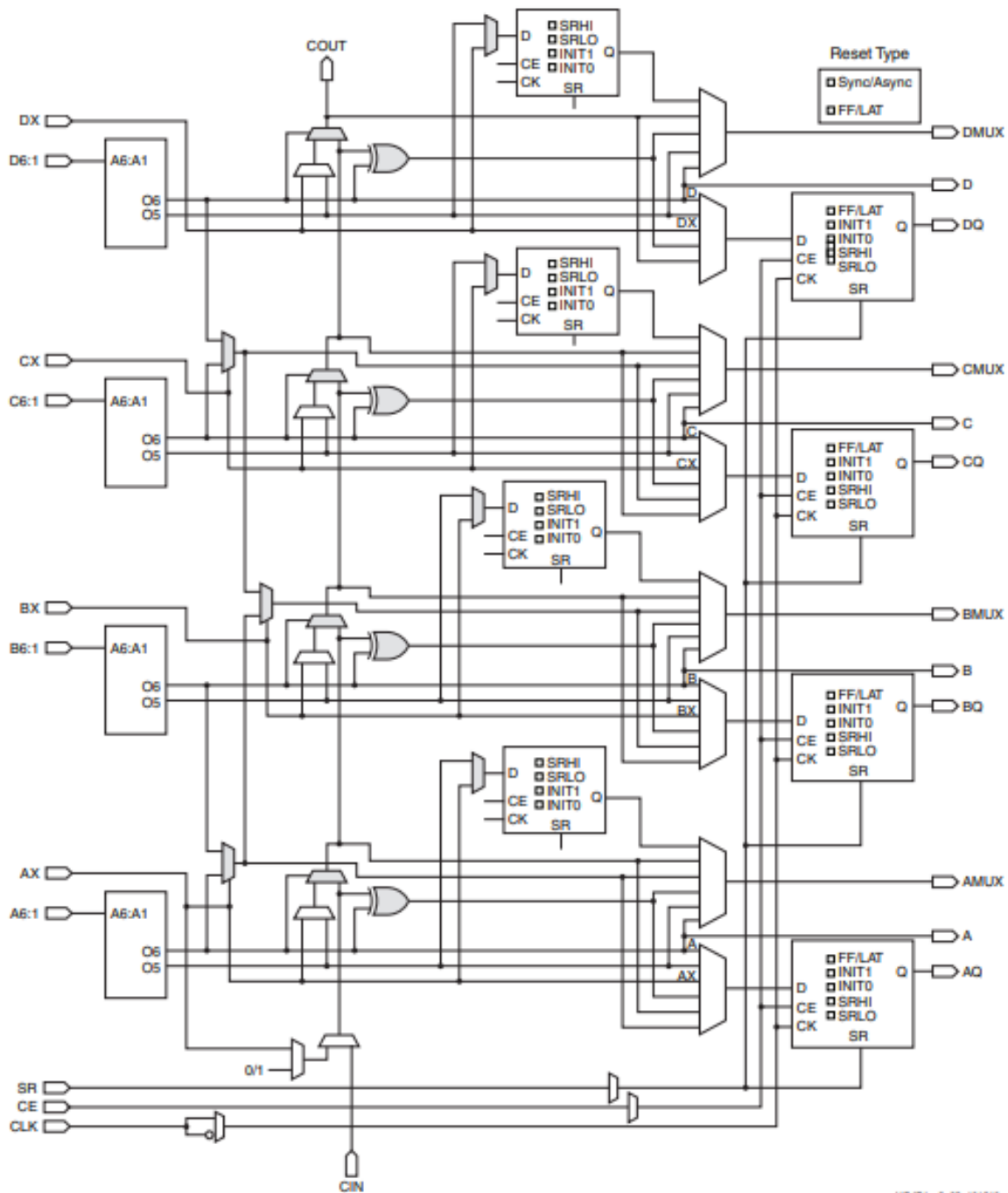


Figure 5.7 : SLICEL

Look-Up Table (LUT)

The function generators in 7 series FPGAs are implemented as six-input look-up tables (LUTs). There are six independent inputs (A inputs - A1 to A6) and two independent outputs (O5 and O6) for each of the four function generators in a slice (A, B, C, and D). The function generators can implement:

- Any arbitrarily defined six-input Boolean function

- Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs
- Two arbitrarily defined Boolean functions of 3 and 2 inputs or less

A six-input function uses:

- A1-A6 inputs
- O6 output
- Two five-input or less functions use:
- A1–A5 inputs
- A6 driven High
- O5 and O6 outputs

The propagation delay through a LUT is independent of the function implemented. Signals from the function generators can:

- Exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5)
- Enter the XOR dedicated gate from an O6 output
- Enter the carry-logic chain from an O5 output
- Enter the select line of the carry-logic multiplexer from O6 output
- Feed the D input of the storage element
- Go to F7AMUX/F7BMUX wide multiplexers from O6 output

In addition to the basic LUTs, slices contain three multiplexers (F7AMUX, F7BMUX, and F8MUX). These multiplexers are used to combine up to four function generators to provide any function of seven or eight inputs in a slice.

- **F7AMUX:** Used to generate seven input functions from LUTs A and B
- **F7BMUX:** Used to generate seven input functions from LUTs C and D
- **F8MUX:** Used to combine all LUTs to generate eight input functions.

Functions with more than eight inputs can be implemented using multiple slices. There are no direct connections between slices to form function generators greater than eight inputs within a CLB.

Storage Elements

There are eight storage elements per slice. Four can be configured as either edge-triggered D-type flip-flops or level-sensitive latches. The D input can be driven directly by a LUT output via AFFMUX, BFFMUX, CFFMUX, or DFFMUX, or by the BYPASS slice inputs bypassing the function generators via AX, BX, CX, or DX input. When configured as a latch, the latch is

transparent when the CLK is Low.

There are four additional storage elements that can only be configured as edge-triggered D-type flip-flops. The D input can be driven by the O5 output of the LUT or the BYPASS slice inputs via AX, BX, CX, or DX input. When the original four storage elements are configured as latches, these four additional storage elements cannot be used.

Programmable Interconnect

In Fig 5.8 , a hierarchy of interconnect resources can be seen. There are long lines that can be used to connect critical CLBs that are physically far from each other on the chip without inducing much delay. These long lines can also be used as buses within the chip.

There are also short lines that are used to connect individual CLBs that are located physically close to each other. Transistors are used to turn on or off connections between different lines. There are also several programmable switch matrices in the FPGA to connect these long and short lines together in specific, flexible combinations.

Three-state buffers are used to connect many CLBs to a long line, creating a bus. Special long lines, called *global clock lines* , are specially designed for low impedance and thus fast propagation times. These are connected to the clock buffers and to each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA, ensuring minimal skew between clock signals arriving at different flip-flops within the chip.

In an ASIC, the majority of the delay comes from the logic in the design, because logic is connected with metal lines that exhibit little delay. In an FGPA, however, most of the delay in the chip comes from the interconnect, because the interconnect – like the logic – is fixed on the chip. In order to connect one CLB to another CLB in a different part of the chip often requires a connection through many transistors and switch matrices, each of which introduces extra delay.

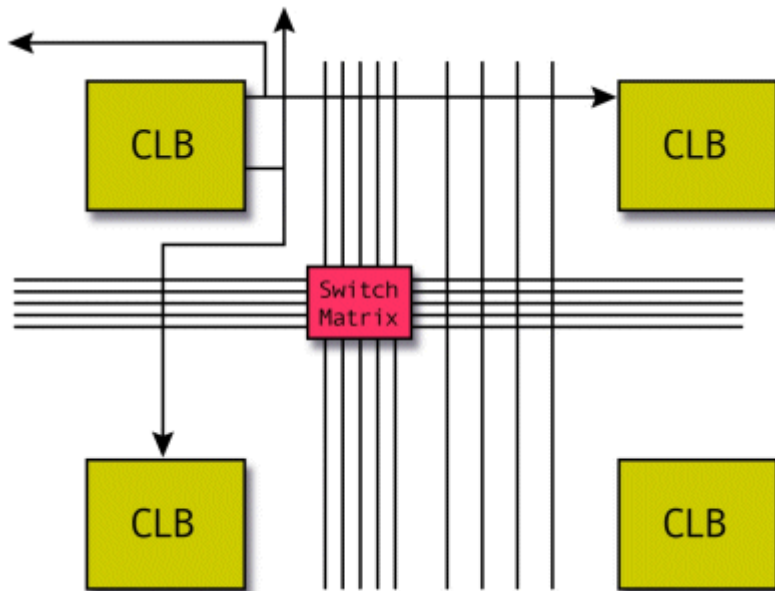


Figure 5.7 : SLICEL

Macros

Create macros using multiple design element primitives. Following are the different types of macros:

- Hard Macro (.nmc)

When we add a hard macro to our design, we are adding an instance of a library hard macro. Our design can contain multiple instances of the same library hard macro, but each hard macro must have a unique name. We can use FPGA Editor to create hard macros using either of the following methods:

- Save a design as a hard macro. For details.
- Create a hard macro. This method is only recommended if we have advanced hand routing skills and knowledge of our targeted architecture.

Note RPM macros are recommended instead of hard macros wherever possible, because hard macros do not allow timing analysis. A hard macro is seen as a "black box" by the Xilinx® timing tools. Timing can be analyzed to the input and output of the hard macro, but we must manually verify the timing paths within the hard macro.

- Relationally Placed Macro (RPM)

RPMs define the spatial relationship of the primitives that comprise the RPM. We can define the relative placements of these primitives to create our own RPMs, using constraints in a UCF file. After create the RPM, we can use FPGA Editor to view the placement of the RPM and to verify that it was created as expected.

Combinational Logic Implemented by Xilinx XC4000 CLB

Any function of up to four variables, plus any second function of up to four unrelated variables, plus any third function of up to three unrelated variables %

Any single function of five variables %

Any function of four variables together with some functions of six variables %

Some functions of up to nine variables.

$$F(a, b, c, d, e) = a \cdot F(a=1) + a' \cdot F(a=0)$$

- Both $F(a=1)$ and $F(a=0)$ are four-input functions %

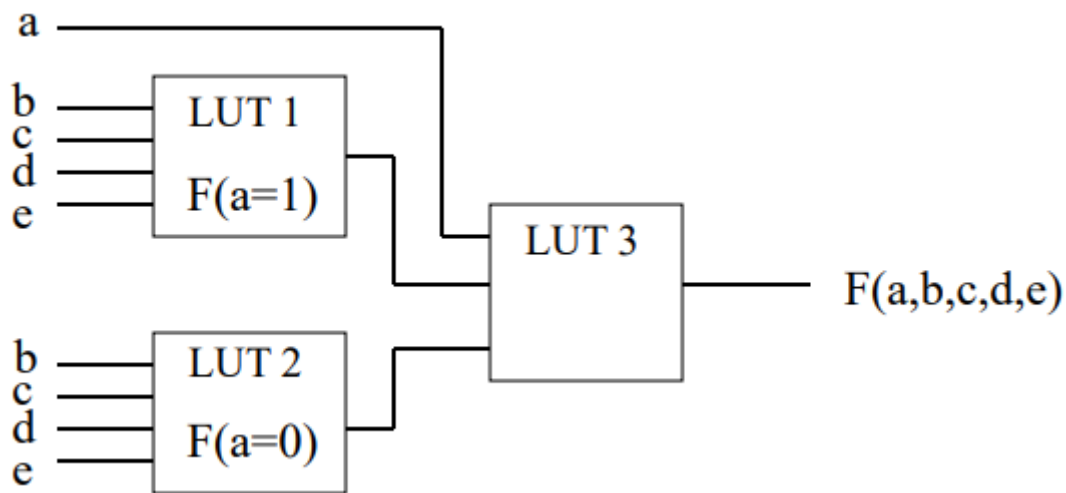


Figure 5.8 : Four variable implementation

Any function of four variables together with some functions of six variables can be implemented by a single CLB

$$F(a, b, c, d, e, f) = a \cdot b \cdot F1 + a \cdot b' \cdot F2 + a' \cdot b \cdot F3 + a' \cdot b' \cdot F4$$

$$F1 = F(a=1, b=1);$$

$$F2 = F(a=1, b=0);$$

$$F3 = F(a=0, b=1);$$

$$F4 = F(a=0, b=0)$$

Condition: Among $F1$ - $F4$, three of them are constant (e.g. $F1=1, F2=F3=0$)

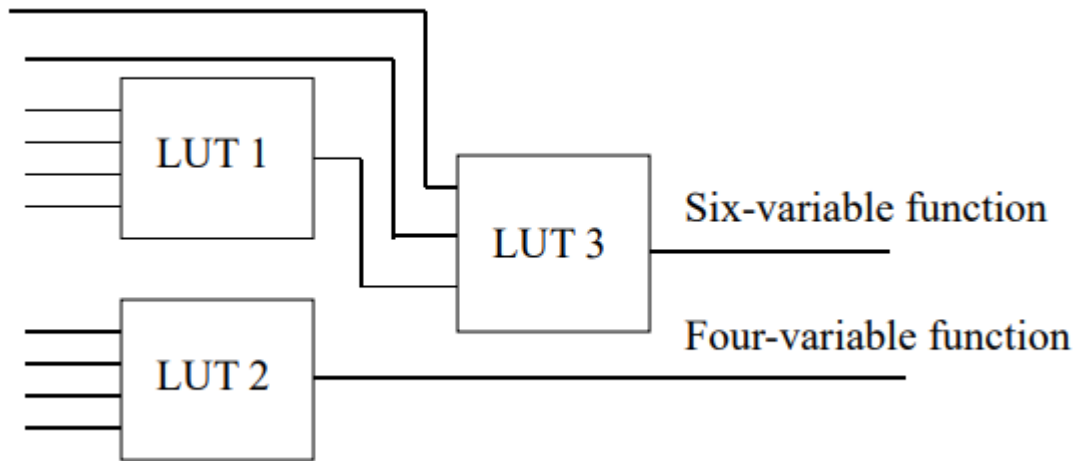


Figure 5.9 : Four variable CLB implementation

Decoding Circuits

2-to-4 Decoding circuit

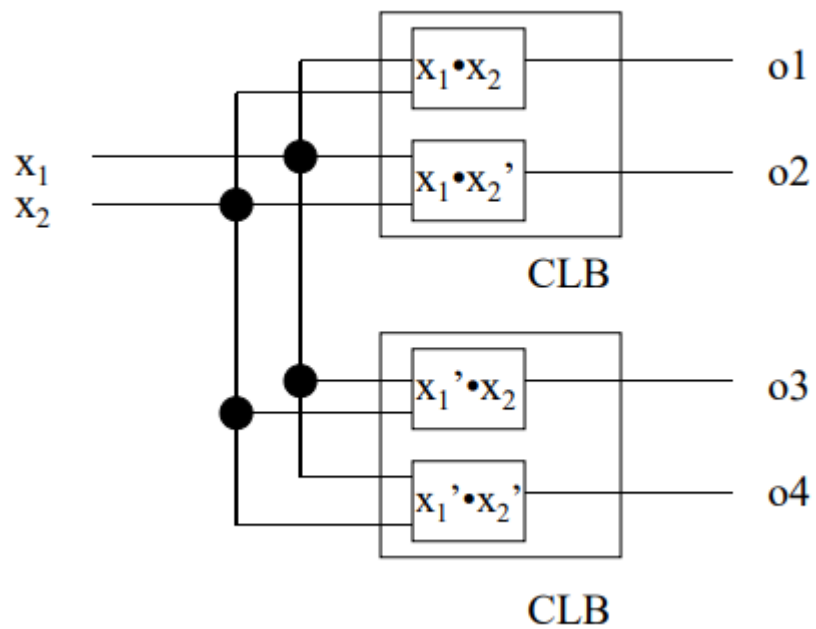


Figure 5.10 : 2 to 4 decoding circuit

10-to-1024 Decoding circuit

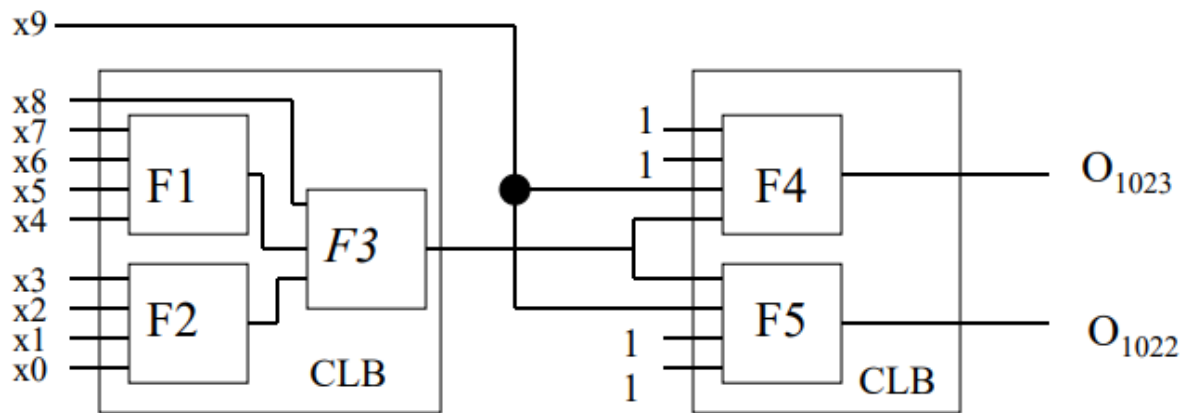


Figure 5.11 : 10 to 1024 decoding circuit

$$F1 = x4 \cdot x5 \cdot x6 \cdot x7$$

$$F2 = x0 \cdot x1 \cdot x2 \cdot x3$$

$$F3 = x8 \cdot F1 \cdot F2$$

$$F4 = x9 \cdot F3$$

$$F5 = x9' \cdot F3$$

Disadvantages

- It needs 1024 CLBs; expensive to implement.
- It is a two level implementation, resulting large delay.

Dedicated Decoding Circuits in Xilinx FPGAs

Four dedicated programmable decoding circuits are included in Xilinx FPGAs. %

The number of decoder inputs ranges from 42 to 132 for different devices. %

The decoding circuits use wired-AND gate structures (like the AND plane in PAL).

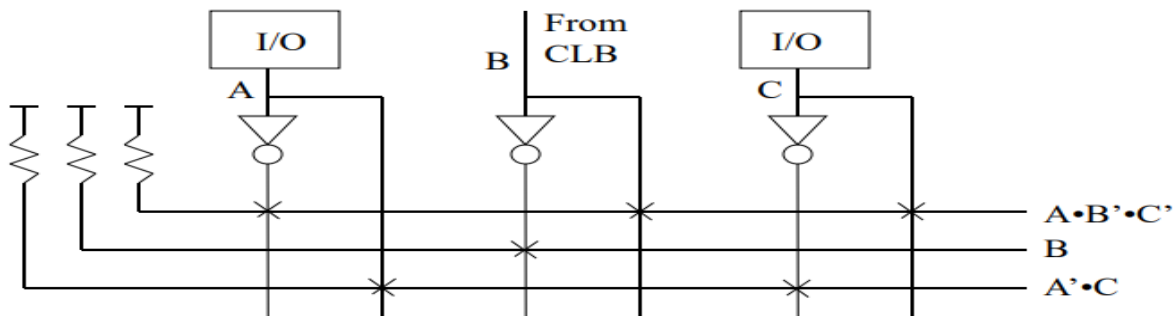


Figure 5.12 : Dedicated decoding circuit

FPGA Implementation of Sequential Logic

Sequential Circuit: the circuit outputs depend on not only the current values of inputs but also previous input values.

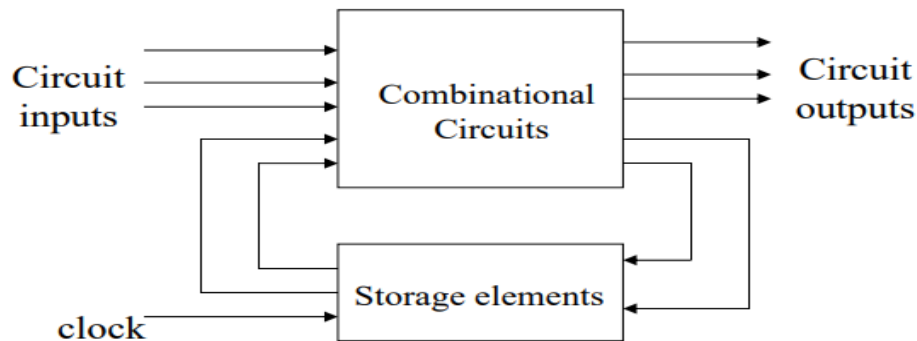


Figure 5.13 : Sequential Logic

Storage Elements in Xilinx CLB

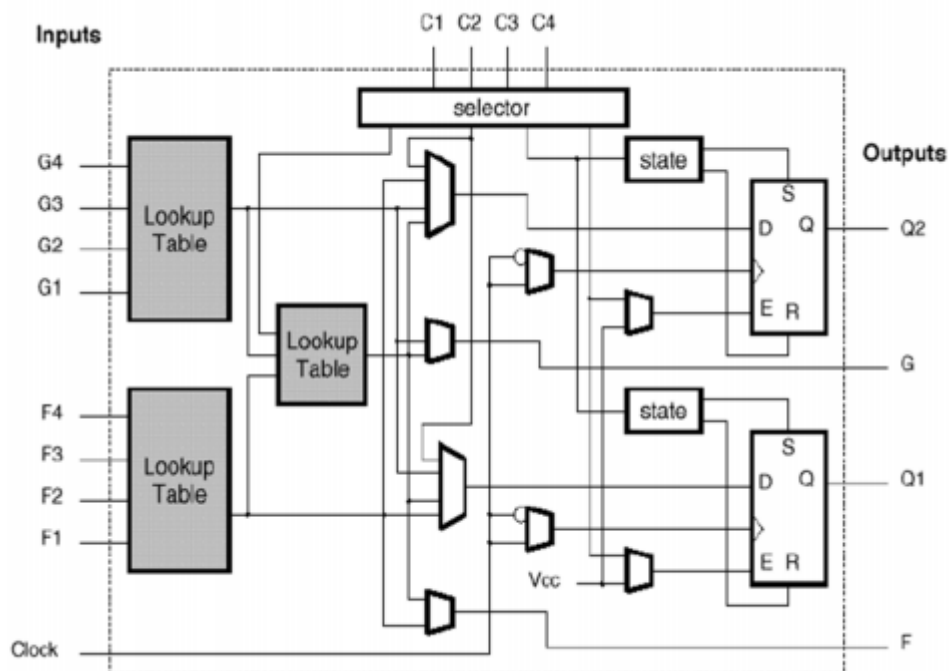


Figure 5.14 : Storage Elements in Xilinx CLB

Each CLB contains two edge-triggered D flip-flops. They can be configured as positive-edge-triggered or negative-edge-triggered. %

Each D flip-flop has clock enable signal E, which is active high. %

Each D flip-flop can be set or reset by SR signal. A global reset or reset signal is also available for set or reset all D flip-flops once the device is powered up.

FPGA Implementation of Finite State Machines

Example of Finite State Machine

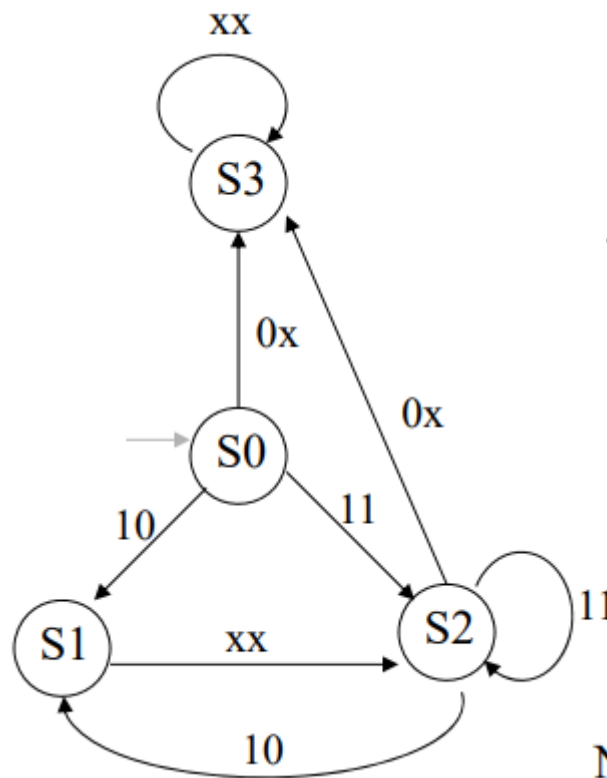


Figure 5.15 : State transition diagram

Current States	Inputs: xy			Outputs				
	0x	10	11	a	b	c	d	e
S0	S3	S1	S2	0	0	1	1	1
S1	S2	S2	S2	0	1	0	1	1
S2	S3	S1	S2	1	0	0	1	0
S3	S3	S3	S3	0	0	0	0	0

Figure 5.16 : State Table

State Encoding

Binary encoding: minimum number of D flip-flops

	Q1	Q0
S0 :	0	0
S1 :	0	1
S2 :	1	0
S3 :	1	1

It needs two D flip-flops

Implementation Using Binary Encoding

Excitation table

Inputs		Current States		Next States		Outputs				
x	y	Q1	Q0	D1	D0	a	b	c	d	e
0	x	0	0	1	1	0	0	1	1	1
0	x	0	1	1	0	0	1	0	1	1
0	x	1	0	1	1	1	0	0	1	0
0	x	1	1	1	1	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	1
1	0	0	1	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	0	1	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1	1
1	1	0	1	1	0	0	1	0	1	1
1	1	1	0	1	0	1	0	0	1	0
1	1	1	1	1	1	0	0	0	0	0

Implementation Using Binary Encoding

Combinational functions needed to be implemented

$$D1 = x' + y + Q0 \quad (F1)$$

$$D0 = Q1 \cdot Q0 + y' \cdot Q0' + x' \cdot Q0' \quad (F2)$$

$$a = Q1 \cdot Q0' \quad (F3)$$

$$b = Q1' \cdot Q0 \quad (F4)$$

$$c = Q1' \cdot Q0' \quad (F5)$$

$$d = Q0' + Q1' \quad (F6)$$

$$e = Q1' \quad (F7)$$

Implementation Using Binary Encoding

FPGA implementation

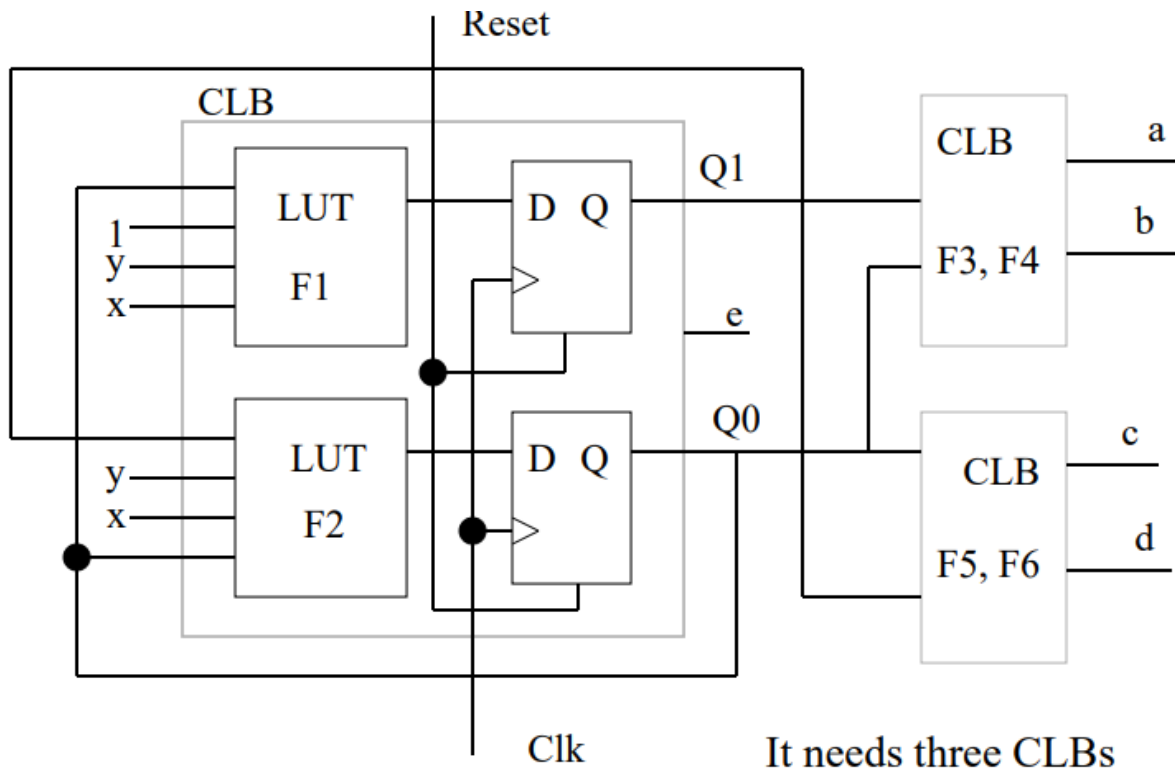


Figure 5.17 : FPGA implementation

TEXT / REFERENCE BOOKS

1. J.Bhaskar, "A VHDL Primer", Prentice Hall of India Limited. 3rd edition 2004
2. Stephen 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.
7. Stuart Sutherland, "RTL Modeling With System Verilog for Simulation and Synthesis: Using System Verilog for ASIC and FPGA Design", 1st Edition, Sutherland HDL, Inc., 2017.

- 8. Simon Monk, “Programming FPGAs: Getting Started with Verilog”, 1st Edition, Tata McGraw Hill, 2016.**
- 9. User Guide – “7 Series FPGAs Configurable Logic Block” - (WWW.XILINX.COM)**