

UNIT-1

1. Evolution of Computer-Aided Digital Design

Digital circuit design has evolved rapidly over the last 25 years. The earliest digital circuits were designed with vacuum tubes and transistors. Integrated circuits were then invented where logic gates were placed on a single chip. The first integrated circuit (IC) chips were SSI (Small Scale Integration) chips where the gate count was very small. As technologies became sophisticated, designers were able to place circuits with hundreds of gates on a chip. These chips were called MSI (Medium Scale Integration) chips. With the advent of LSI (Large Scale Integration), designers could put thousands of gates on a single chip. At this point, design processes started getting very complicated, and designers felt the need to automate these processes.

Electronic Design Automation (EDA) techniques began to evolve. Chip designers began to use circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 100 transistors. The circuits were still tested on the breadboard, and the layout was done on paper or by hand on a graphic computer terminal.

Computer-Aided Design (CAD) tools refers to back-end tools that perform functions related to place and route, and layout of the chip . The term Computer-Aided Engineering (CAE) tools refers to tools that are used for front-end processes such HDL simulation, logic synthesis, and timing analysis. Designers used the terms CAD and CAE interchangeably. Today, the term Electronic Design Automation is used for both CAD and CAE. For the sake of simplicity, in this book, we will refer to all design tools as EDA tools.

With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 100,000 transistors. Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard. Computer aided techniques became critical for verification and design of VLSI digital circuits. Computer programs to do automatic placement and routing of circuit layouts also became popular. The designers were now building gate-level digital circuits manually on graphic terminals. They would build small building blocks and then derive higher-level blocks from them. This process would continue until they had built the top-level block. Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip.

As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further.

2. Emergence of HDLs

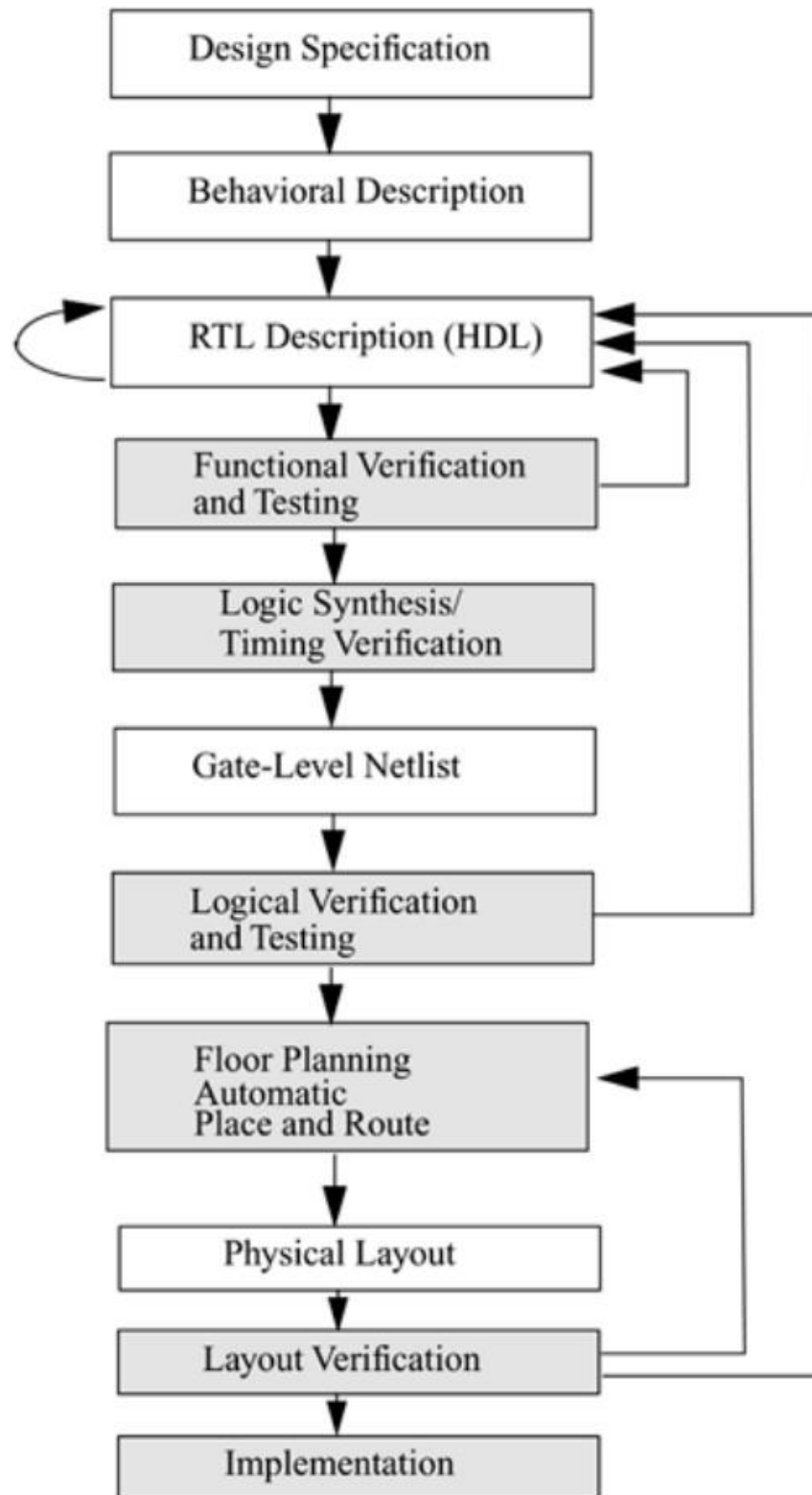
For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allowed the designers to model the concurrency of processes found in hardware elements.

Hardware description languages such as Verilog HDL and VHDL became popular. Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from DARPA. Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers. Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a register transfer level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL Description.

Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.

HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation. Today, Verilog HDL is an accepted IEEE standard. In 1995, the original standard IEEE 1364-1995 was approved. IEEE 1364-2001 is the latest Verilog HDL standard that made significant improvements to the original standard.

3. Typical Design Flow



The design flow is typically used by designers who use HDLs. Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.

In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. A behavioral description is 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.

New EDA tools have emerged to simulate behavioral descriptions of circuits. These tools combine the powerful concepts from HDLs and object oriented languages such as C++. These tools can be used instead of writing behavioral descriptions in Verilog HDL.

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. 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.

Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit. After the RTL description is frozen, EDA tools are available to assist the designer in further processes.

Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit. As these tools mature, digital circuit design will become similar to high-level computer programming. Designers will simply implement the algorithm in an HDL at a very abstract level. EDA tools will help the designer convert the behavioral description to a final IC chip

4.Importance of HDLs

HDLs have many advantages compared to traditional schematic-based design.

Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.

By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.

Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

HDL-based design is here to stay. With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs. No digital circuit designer can afford to ignore HDL-based design.

New tools and languages focused on verification have emerged in the past few years. These languages are better suited for functional verification. However, for logic design, HDLs continue as the preferred choice.

5. Popularity of Verilog HDL

Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features

- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of Vendors.
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

6. Trends in HDLs

The speed and complexity of digital circuits have increased rapidly. Designers have responded by designing at higher levels of abstraction. Designers have to think only in terms of functionality. EDA tools take care of the implementation details. With designer assistance, EDA tools have become sophisticated enough to achieve close-to-optimum implementation.

The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level design. Behavioral synthesis allowed engineers to design directly in terms of algorithms and the behavior of the circuit, and then use EDA tools to do the translation and optimization in each phase of the design. However, behavioral synthesis did not gain widespread acceptance.

Today, RTL design continues to be very popular. Verilog HDL is also being constantly enhanced to meet the needs of new verification methodologies.

Formal verification and assertion checking techniques have emerged. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate-level netlists. However, the need to describe a design in Verilog HDL will not go away. Assertion checkers allow checking to be embedded in the RTL code.

New verification languages have also gained rapid acceptance. These languages combine the parallelism and hardware constructs from HDLs with the object oriented nature of C++. These languages also provide support for automatic stimulus creation, checking, and coverage. However, these languages do not replace Verilog HDL. They simply boost the productivity of the verification process. Verilog HDL is still needed to describe the design.

For very high-speed and timing-critical circuits like microprocessors, the gate-level netlist provided by logic synthesis tools is not optimal. In such cases, designers often mix gate-level description directly into the RTL description to achieve optimum results. EDA tools sometimes prove to be insufficient to achieve the desired results.

Another technique that is used for system-level design is a mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules.

For Example, consider a system that has a CPU, graphics chip, I/O chip, and a system bus. The CPU designers would build the next-generation CPU themselves at an RTL level, but they would use behavioral models for the graphics chip and the I/O chip and would buy a vendor-supplied model for the system bus. Thus, the system-level simulation for the CPU could be up and running very quickly and long before the RTL descriptions for the graphics chip and the I/O chip are completed.

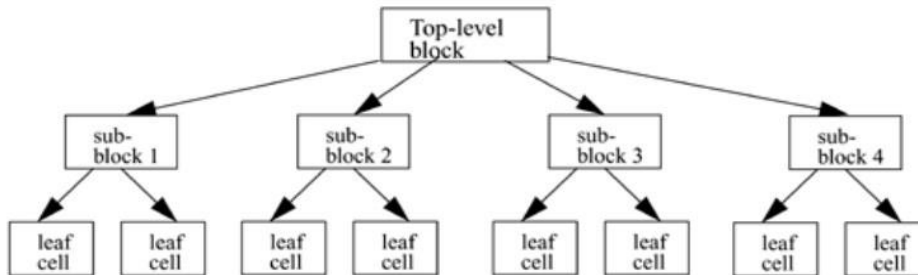
Hierarchical Modeling Concepts:

7. Design Methodologies

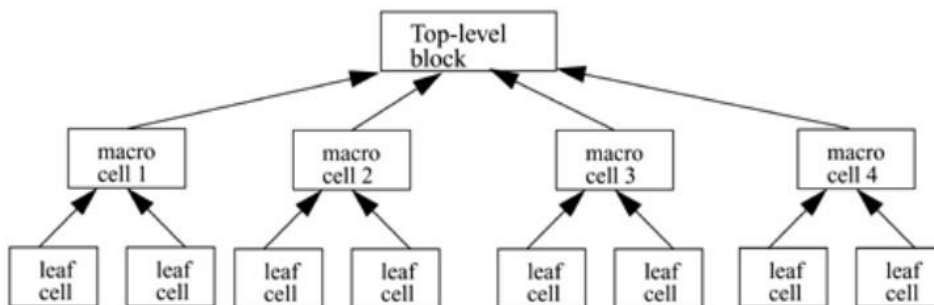
There are two basic types of digital design methodologies: **a top-down design methodology and a bottom-up design methodology.**

Top-down Design Methodology:

In this 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.

**Bottom-up Design Methodology:**

In this 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.



Typically, a combination of top-down and bottom-up flows is used. Design architects define the specifications of the top-level block. Logic designers decide how the design should be structured by breaking up the functionality into blocks and sub-blocks. At the same time, circuit designers are designing optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells.

EX:4-bit Ripple Carry Counter

8.Modules

A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design.

A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

Each module must have a `module_name`, which is the identifier for the module, and a `module_terminal_list`, which describes the input and output terminals of the module.

```
module <module_name> (<module_terminal_list>);  
...  
<module internals>  
...  
...  
endmodule
```

Example:

```
module T_FF (q, clock, reset);  
.  
.  
<functionality of T-flipflop>  
.  
.  
endmodule
```

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The levels are defined below.

1. Behavioral or algorithmic level:

This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

2. Dataflow level:

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

3. Gate level:

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

4. Switch level:

This is the lowest level of abstraction provided by Verilog. A module can be

implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

Verilog allows the designer to mix and match all four levels of abstractions in a design. In the digital design community, the term register transfer level (RTL) is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.

9.Instances:

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template.

Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances.

Example:

The top-level block creates four instances from the T-flipflop (T_FF) template. Each T_FF instantiates a D_FF and an inverter gate. Each instance must be given a unique name. Note that // is used to denote single-line comments.

Module Instantiation

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops.
module ripple_carry_counter(q, clk, reset);
output [3:0] q; //I/O signals and vector declarations
//will be explained later.
input clk, reset; //I/O signals will be explained later.
//Four instances of the module T_FF are created. Each has a
unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);
endmodule
```

Module definitions simply specify how the module will work, its internals, and its interface. Modules must be instantiated for use in the design.

10.Components of a Simulation:

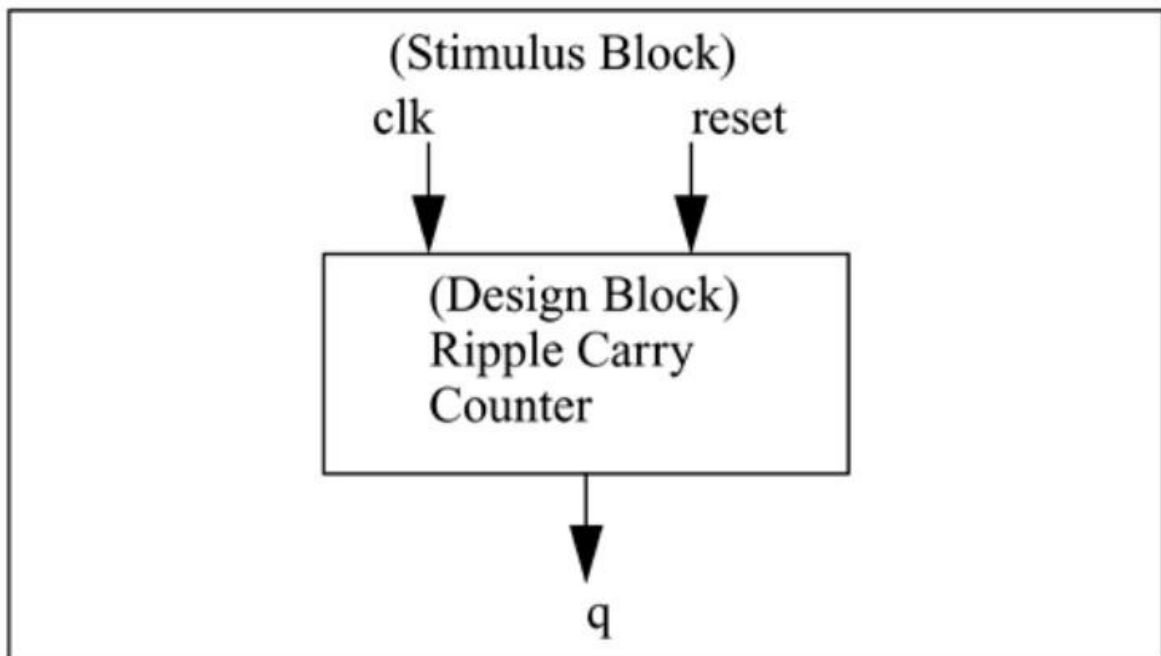
- ❖ Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results.
- ❖ We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate.
- ❖ The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.
- ❖ Two styles of stimulus application are possible.

First Style:

In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block.

In the figure below, the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and displays output signal q.

Stimulus Block Instantiates Design Block:



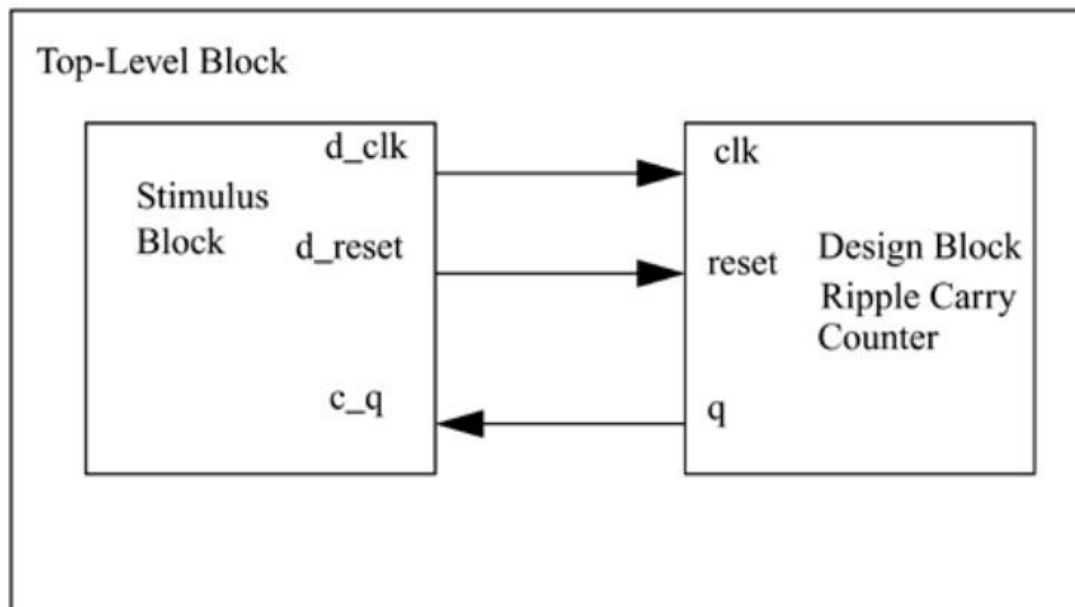
Second Style:

The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface.

The stimulus module drives the signals d_clk and d_reset, which are connected to the signals clk and reset in the design block. It also checks and displays signal c_q, which is connected to the signal q in the design block.

The function of top-level blocks is simply to instantiate the design and stimulus blocks.

Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module:



11. Data Types:

Value Set:

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in the table below.


Value 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

In addition to logic values, strength levels are often used to resolve conflicts between

drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in the table below.

Strength Level:

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signals of unequal strengths are driven on a wire, the stronger signal prevails. For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x. Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.

Nets:

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.

In the figure below, net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.



```
wire a; // Declare net a for the above circuit
```

```
wire b,c; // Declare two wires b,c for the above circuit
wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.
```

Registers:

- ❖ Registers represent data storage elements. Registers retain value until another value is placed onto them.
- ❖ In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver.
- ❖ Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register.
- ❖ 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
initial // this construct will be discussed later
begin
reset = 1'b1; //initialize reset to 1 to reset the digital
circuit.
#100 reset = 1'b0; // after 100 time units reset is
deasserted.
end
```

Vectors:

Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual
address 41 bits
wide
```

Vectors can be declared at [high# : low#] or [low# : high#], but the left number in the squared brackets is always the most significant bit of the vector. In the example shown above, bit 0 is the most significant bit of vector virtual_addr.

12.System Tasks and Compiler Directives:

System Tasks:

- ❖ 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.

Displaying information:

\$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: \$display(p1, p2, p3,....., pn);

p1, p2, p3,...., pn can be quoted strings or variables or expressions. The format of \$display is very similar to printf in C. A \$display inserts a newline at the end of the string by default. A \$display without any arguments produces a newline.

String Format Specifications:

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display real number in scientific or decimal

Monitoring information:

Verilog provides a mechanism to monitor a signal when its value changes. This facility is

provided by the \$monitor task.

Usage: \$monitor(p1,p2,p3,.....,pn);

The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the \$display task is used in the \$monitor task.

Two tasks are used to switch monitoring on and off.

Usage:

```
$monitoron;  
$monitoroff;
```

Stopping and finishing in a simulation:

The task \$stop is provided to stop during a simulation.

Usage: \$stop;

The \$stop task puts the simulation in an interactive mode. The designer can then debug the design from the interactive mode. The \$stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.

The \$finish task terminates the simulation.

Usage: \$finish;

Example : Stop and Finish Tasks

```
// Stop at time 100 in the simulation and examine the results  
// Finish the simulation at time 1000.  
initial // to be explained later. time = 0  
begin  
clock = 0;  
reset = 1;  
#100 $stop; // This will suspend the simulation at time = 100  
#900 $finish; // This will terminate the simulation at time =  
1000  
end
```

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).

Example : `define Directive

```
//define a text macro that defines default word size
```



```
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32
//define an alias. A $stop will be substituted wherever 'S
appears
'define S $stop;
//define a frequently used text string
'define WORD_REG reg [31:0]
// you can then define a 32-bit register as 'WORD_REG reg32;
```

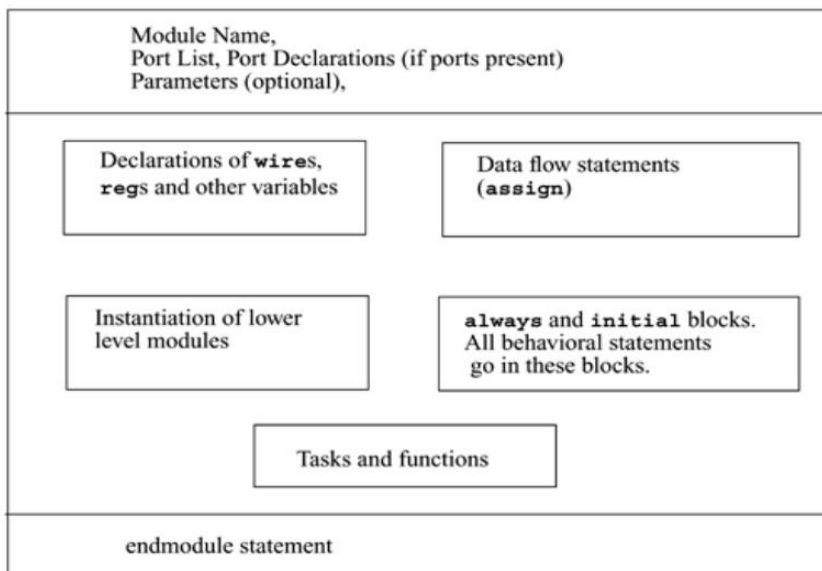
UNIT:2

1.MODULES & PORTS:

Modules:

A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design.

Components of a Verilog Module:

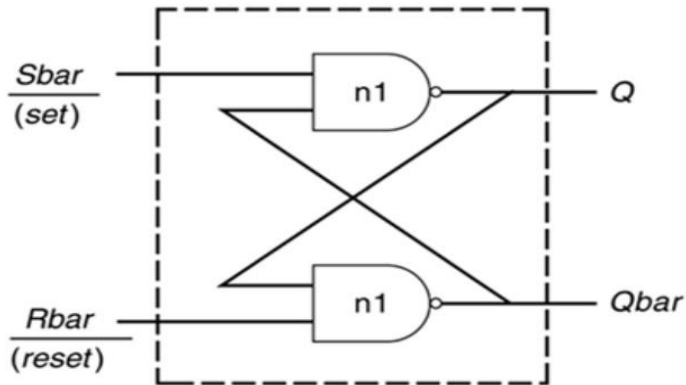


- ❖ A module definition always begins with the keyword module. The module name, port list, port declarations, and optional parameters must come first in a module definition.
- ❖ Port list and port declarations are present only if the module has any ports to interact with the external environment. The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions.
- ❖ These components can be in any order and at any place in the module definition.
- ❖ The endmodule statement must always come last in a module definition.
- ❖ All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

❖ The modules can be defined in any order in the file.

To understand the components of a module shown above, let us consider a simple example of an SR latch:

SR Latch:



The SR latch has S and R as the input ports and Q and Qbar as the output ports.

Example: Components of SR Latch

// This example illustrates the different components of a module

// Module name and port list

// SR_latch module

```
module SR_latch(Q, Qbar, Sbar, Rbar);
```

//Port declarations

```
output Q, Qbar;
```

```
input Sbar, Rbar;
```

// Instantiate lower-level modules

```
nand n1(Q, Sbar, Qbar);
```

```
nand n2(Qbar, Rbar, Q);
```

// endmodule statement

```
Endmodule
```

```
// endmodule statement
```

```
endmodule
```

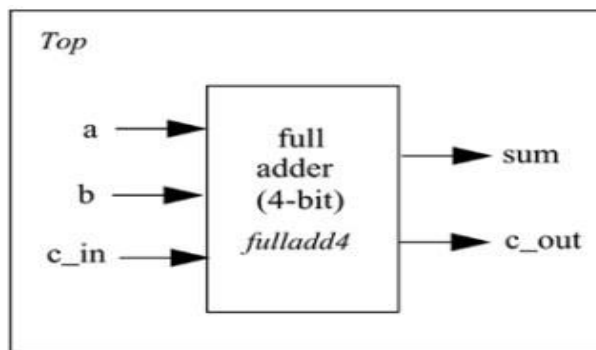
Ports:

- ❖ Ports provide the interface by which a module can communicate with its environment.
- ❖ For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports.
- ❖ The internals of the module are not visible to the environment.
- ❖ This provides a very powerful flexibility to the designer.
- ❖ The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

List of Ports:

- ❖ A module definition contains an optional list of ports.
- ❖ If the module does not exchange any signals with the environment, there are no ports in the list.
- ❖ Consider a 4-bit full adder that is instantiated inside a top-level module Top. The diagram for the input/output ports is shown in the figure below.

I/O Ports for Top and Full



Adder:

- ❖ In the above figure, the module Top is a top-level module.
- ❖ The module fulladder is instantiated below Top. The module fulladder takes input on ports a, b, and c_in and produces an output on ports sum and c_out.
- ❖ Thus, module fulladder performs an addition for its environment.
- ❖ The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment.

Example: List of Ports

```
module fulladder(sum, c_out, a, b, c_in); //Module with a list of ports
```

```
module Top; // No list of ports, top-level module in simulation
```

Port Declaration:

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.

```
module fulladder(sum, c_out, a, b, c_in);
```

```
//Begin port declarations section
```

```
output[3:0] sum;
```

```
output c_cout;
```

```
input [3:0] a, b;
```

```
input c_in;
```

```
//End port declarations section
```

```
...
```

```
<module internals>
```

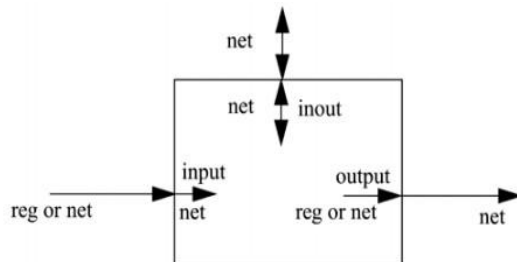
```
...
```

```
endmodule
```

- ❖ All port declarations are implicitly declared as wire in Verilog. Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout. Input or inout ports are normally declared as wires.
- ❖ However, if output ports hold their value, they must be declared as reg.

Port Connection Rules:

- ❖ A port consists of two units, one unit that is internal to the module and another that is external to the module.
- ❖ The internal and external units are connected.
- ❖ There are rules governing port connections when modules are instantiated within other modules.
- ❖ The Verilog simulator complains if any port connection rules are violated.



Inputs:

Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

Outputs:

Internally, output ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

Inouts:

Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to be net.

Width matching: It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

Unconnected ports:

Verilog allows ports to remain unconnected.

For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. We can let a port remain unconnected by instantiating a module as shown below.

```
fulladder fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected
```

Connecting Ports to External Signals:

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed.

- A. Connecting by ordered list
- B. Connecting ports by name

Connecting by ordered list:

Connecting by ordered list is the most intuitive method for most beginners. The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.

Example :Connection by Ordered List

```
module Top;

//Declare connection variables

reg [3:0]A,B;

reg C_IN;

wire [3:0] SUM;

wire C_OUT;

//Instantiate fulladd4, call it fa_ordered.

//Signals are connected to ports in order (by position)

fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
```

```
...  
<stimulus>  
...  
Endmodule  
  
module fulladd4(sum, c_out, a, b, c_in);  
  
output[3:0] sum;  
  
output c_cout;  
  
input [3:0] a, b;  
  
input c_in;  
  
...  
  
<module internals>  
  
...  
  
endmodule
```

Connecting ports by name:

For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position.

Example:

```
// Instantiate module fa_byname and connect signals to ports by name  
  
fulladder fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN),  
  
.a(A),);
```

Note that only those ports that are to be connected to external signals must be specified in port connection by name. Unconnected ports can be dropped. For example, if the

port c_out were to be kept unconnected, the instantiation of fulladder would look as follows.

The port c_out is simply dropped from the port list.

```
// Instantiate module fa_byname and connect signals to ports by name
```

```
fulladder fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

2.GATE LEVEL MODELING:

Gate Types:

- ❖ A logic circuit can be designed by use of logic gates.
- ❖ Verilog supports basic logic gates as predefined primitives.
- ❖ These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.
- ❖ All logic circuits can be designed by using basic gates. There are two classes of basic gates:
 1. and/or gates
 2. buf/not gates.

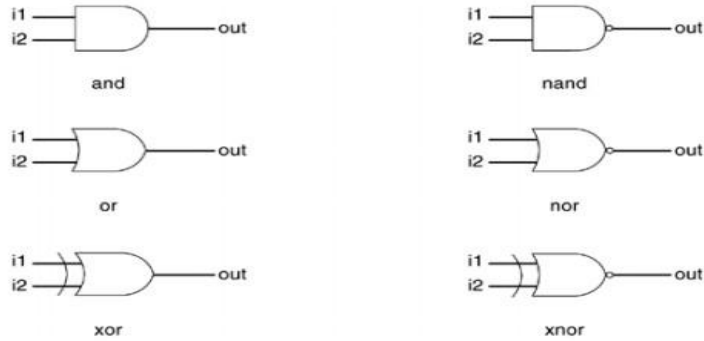
And /Or Gates:

- ❖ And/or gates have one scalar output and multiple scalar inputs.
- ❖ The first terminal in the list of gate terminals is an output and the other terminals are inputs.
- ❖ The output of a gate is evaluated as soon as one of the inputs changes.
- ❖ The and/or gates available in Verilog are shown below.

i.and or xor

ii.nand nor xnor

Basic Gates:



- ❖
- ❖ These gates are instantiated to build logic circuits in Verilog.
- ❖ For all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives.
- ❖ Note that the instance name does not need to be specified for primitives.
- ❖ This lets the designer instantiate hundreds of gates without giving them a name.
- ❖ More than two inputs can be specified in a gate instantiation.
- ❖ Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation.

Example- Gate Instantiation of And/Or Gates:

```
wire OUT, IN1, IN2;
```

```
// basic gate instantiations.
```

```
and a1(OUT, IN1, IN2);
```

```
nand na1(OUT, IN1, IN2);
```

```
or or1(OUT, IN1, IN2);
```

```
nor nor1(OUT, IN1, IN2);
```

```
xor x1(OUT, IN1, IN2);
```

```
xnor nx1(OUT, IN1, IN2);
```

```
// More than two inputs; 3 input nand gate
```

```
nand na1_3inp (OUT, IN1, IN2, IN3);
```

```
// gate instantiation without instance name
```

```
and (OUT, IN1, IN2); // legal gate instantiation
```

Truth Tables for And/Or:

		i1			
		0	1	x	z
i2	and	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

		i1			
		0	1	x	z
i2	nand	0	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

		i1			
		0	1	x	z
i2	or	0	0	1	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		i1			
		0	1	x	z
i2	nor	0	1	0	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

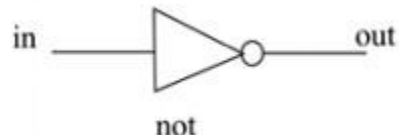
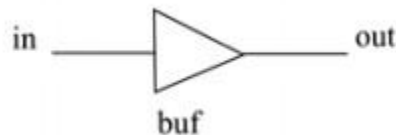
		i1			
		0	1	x	z
i2	xor	0	0	1	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

		i1			
		0	1	x	z
i2	xnor	0	1	0	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

Gates

Buf/Not Gates:

- ❖ Buf/not gates have one scalar input and one or more scalar outputs.
- ❖ The last terminal in the port list is connected to the input.
- ❖ Other terminals are connected to the outputs.
- ❖ Two basic buf/not gate primitives are provided in Verilog.

**Gate Instantiations of Buf/Not Gates:**

```
// basic gate instantiations
```

```

buf b1(OUT1, IN);
not n1(OUT1, IN);
// More than two outputs
buf b1_2out(OUT1, OUT2, IN);
// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation .

```

The truth tables for these gates are very simple.

Truth Tables for Buf/Not Gates:

buf	in	out	not	in	out
	0	0		0	1
	1	1		1	0
	x	x		x	x
	z	x		z	x

Bufif/notif:

Gates with an additional control signal on buf and not gates are also available.

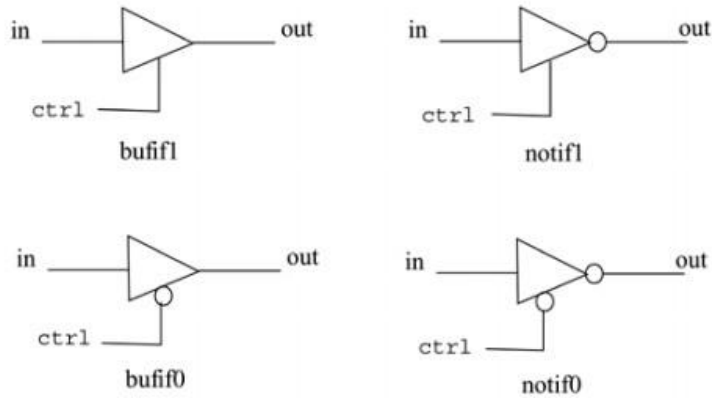
```

bufif1      notif1
bufif0      notif0

```

These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted. Symbols for bufif/notif are shown in the figure.

Gates Bufif and Notif:



Truth Tables for Bufif/Notif Gates:

		ctrl			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

		ctrl			
		0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
		0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Gate Instantiations of Bufif/Notif Gates:

```
//Instantiation of bufif gates.
```

```
bufif1 b1 (out, in, ctrl);
```

```
bufif0 b0 (out, in, ctrl);
```

```
//Instantiation of notif gates
```

```
notif1 n1 (out, in, ctrl);
```

```
notif0 n0 (out, in, ctrl);
```

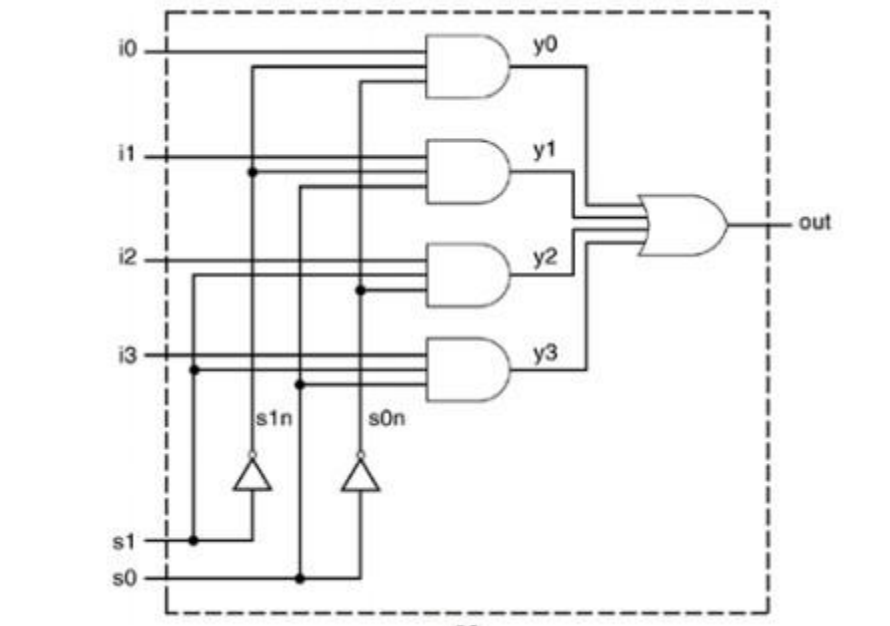
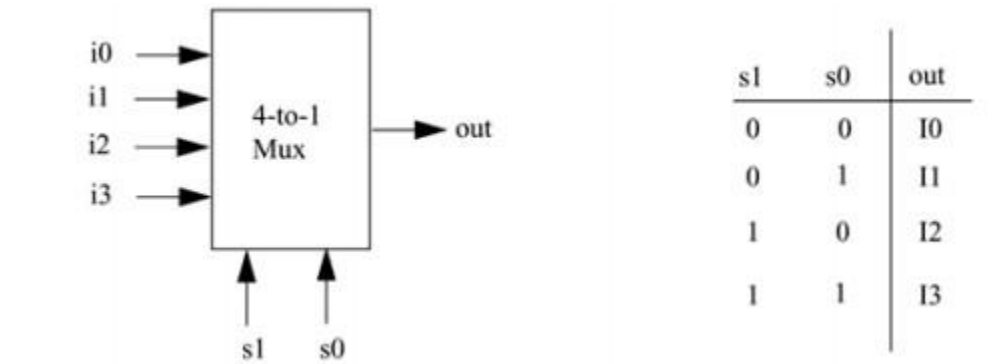
Gate-level multiplexer:

- ❖ Multiplexers serve a useful purpose in logic design.
- ❖ They can connect two or more sources to a single destination.
- ❖ They can also be used to implement boolean functions.

- ❖ We will assume for this example that signals s1 and s0 do not get the value x or z.
- ❖ The I/O diagram and the truth table for the multiplexer are shown in the figure.
- ❖ The I/O diagram will be useful in setting up the port list for the multiplexer.

4-to-1 Multiplexer

Logic Diagram for Multiplexer



3. Gate Delays:

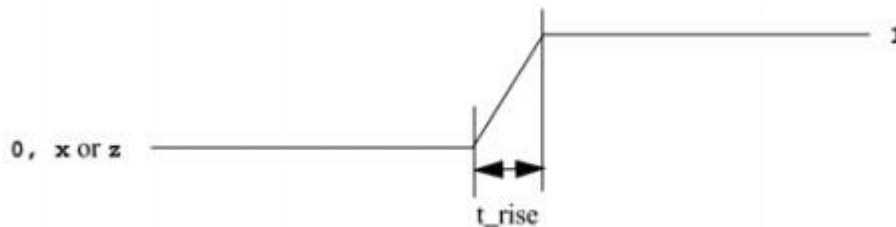
- ❖ In real circuits, logic gates have delays associated with them.
- ❖ Gate delays allow the Verilog user to specify delays through the logic circuits.
- ❖ Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays:

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

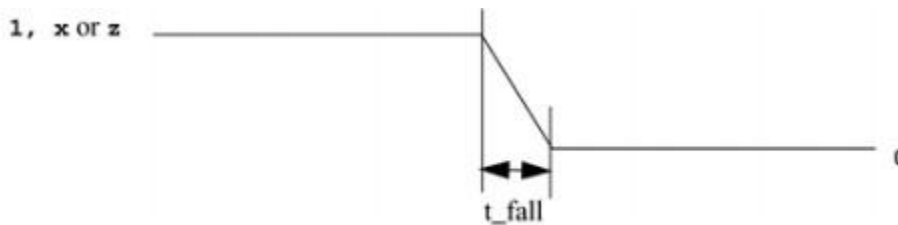
i.Rise delay:

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



ii.Fall delay:

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



iii.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.
- ❖ Three types of delay specifications are allowed.
- ❖ If only one delay is specified, this value is used for all transitions.
- ❖ If two delays are specified, they refer to the rise and fall delay values.
- ❖ The turn-off delay is the minimum of the two delays.
- ❖ If all three delays are specified, they refer to rise, fall, and turn-off delay values.
- ❖ If no delays are specified, the default value is zero.

Types of Delay Specification:

```
// Delay of delay_time for all transitions
```

```
and #(delay_time) a1(out, i1, i2);  
// Rise and Fall Delay Specification.  
and #(rise_val, fall_val) a2(out, i1, i2);  
// Rise, Fall, and Turn-off Delay Specification  
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions  
and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6  
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4,  
Turn-off  
= 5
```

Min/Typ/Max Values:

- ❖ Verilog provides an additional level of control for each type of delay mentioned above.
- ❖ For each type of delay rise, fall, and turn-off three values, min, typ, and max, can be specified.
- ❖ Any one value can be chosen at the start of the simulation.
- ❖ Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.

Min value:

The min value is the minimum delay value that the designer expects the gate to have.

Typ val:

The typ value is the typical delay value that the designer expects the gate to have.

Max value:

The max value is the maximum delay value that the designer expects the gate to have.

Min, typ, or max values can be chosen at Verilog run time. Method of choosing a min/typ/max value may vary for different simulators or operating systems.

Example of Min. Max. and Typical Delay Values:

```
// One delay

// if +mindelays, delay= 4

// if +typdelays, delay= 5

// if +maxdelays, delay= 6

and #(4:5:6) a1(out, i1, i2);

// Two delays

// if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)

// if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)

// if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)

and #(3:4:5, 5:6:7) a2(out, i1, i2);

// Three delays

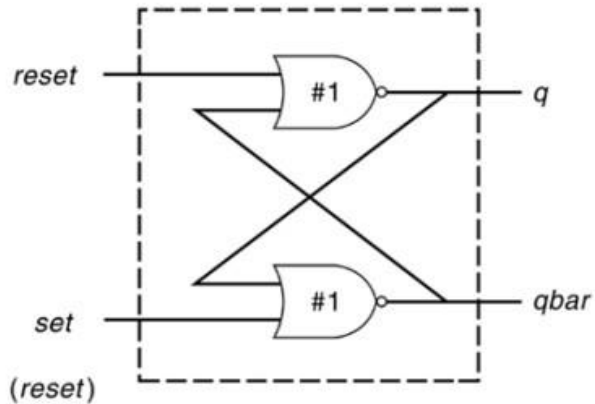
// if +mindelays, rise= 2 fall= 3 turn-off = 4

// if +typdelays, rise= 3 fall= 4 turn-off = 5

// if +maxdelays, rise= 4 fall= 5 turn-off = 6

and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

Example:The logic diagram for an RS latch with delay is shown below.



set	reset	q_{n+1}
0	0	q_n
0	1	0
1	0	1
1	1	?

4.Dataflow Modeling:

- ❖ 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.
- ❖ 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 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
```

In this the drive strength is optional and can be specified in terms of strength levels.

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.

```
// Continuous assign. out is a net.  
  
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
```

5.Delays in Dataflow Modeling:

- ❖ 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.
- ❖ They are:

i.regular assignment delay

ii.implicit continuous assignment delay

iii. net declaration delay.

Regular Assignment Delay:

- ❖ 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.

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

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;
```

6.Expressions, Operators, and Operands:

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions:

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators
```

```
a ^ b
```

```
addr1[20:17] + addr2[20:17]
```

```
in1 | in2
```

Operands:

Operands can be constants, integers, real numbers, nets, registers, times, bit-select (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
```

Operators:

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```
d1 && d2 // && is an operator on operands d1 and d2
```

```
!a[0] // ! is an operator on operand a[0]
```

```
B >> 1 // >> is an operator on operands B and 1
```

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.

Operator Types and Symbols:

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 negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
		bitwise or	two
	^	bitwise xor	two
	~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
		reduction or	one
	~	reduction nor	one
	^	reduction xor	one
~ or ~^	reduction xnor	one	
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

1.Arithmetic Operators:

There are two types of arithmetic operators: binary and unary.

Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

A * B // Multiply A and B.

D / E // Divide D by E.

A + B // Add A and B.

B - A // Subtract A from B.

Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand. Unary + or - operators have higher precedence than the binary + or - operators.

-4 // Negative 4


```
+5 // Positive 5
```

2.Logical Operators:

Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators & and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.

```
// Logical operations
```

```
A = 3; B = 0;
```

```
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
```

```
A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
```

```
!A// Evaluates to 0. Equivalent to not(logical-1)
```

```
!B// Evaluates to 1. Equivalent to not(logical-0)
```

3.Relational Operators:

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

```
// A = 4, B = 3
```

```
// X = 7, Y =6, Z =8
```

```
A <= B // Evaluates to a logical 0
```

A > B // Evaluates to a logical 1

Y >= X // Evaluates to a logical 0

Y < Z // Evaluates to a logical 1

4.Equality Operators:

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==). When used in an expression, equality operators return logical value 1 if true, 0 if false.

// X = 4'b1010, Y = 4'b1101

// Z = 4'b1xxx, M = 4'b1xxxz, N = 4'b1xxxx

A == B // Results in logical 0

X != Y // Results in logical 1

X == Z // Results in x

Z === M // Results in logical 1 (all bits match, including x and z)

Z === N // Results in logical 0 (least significant bit does not match)

M !== N // Results in logical 1

5.Bitwise Operators:

Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands

Truth Tables for Bitwise Operators

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

6.Reduction Operators:

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.

```
// X = 4'b1010
&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

7.Shift Operators:

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<).

```
// X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled MSB position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled LSB position.
```

8.Concatenation Operator:

The concatenation operator ({ , }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

9.Replication Operator:

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;
```

```

reg [1:0] B, C;

reg [2:0] D;

A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111

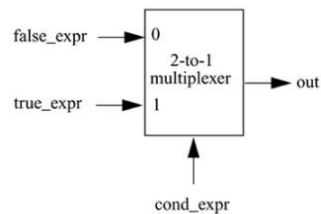
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000

Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010

```

10.Conditional Operator:The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;



example: assign out = (A == 3) ? (control ? x : y) : (control ? m : n) ;

Operator Precedence:

Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ , ~	
Logical	&& 	
Conditional	?:	Lowest precedence

UNIT - III

Behavioral Modeling: -

- Designers need to be evaluating the trade off various architectures & algorithms before they in hardware.
- Behavioral modeling represents the circuit at a very high level of abstraction.
- Behavioral verilog constructs are similar to c language constructs in many ways

Structured procedures: -

- There are two types in verilog structured procedures & they are always and initial.
- In this activity flow in verilog run in paralle rather than in sequence.
- Each always & initial statement represents a separate activity flow in verilog & activity flow starts at simulation time 0.
- The statement always & initial cannot be nested.

Initial statements: -

- In initial block starts at time 0, executes exactly once during a simulation.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- Each block finishes execution independently of other blocks.
- Multiple behavioral statements must be grown typically using the keywords Begin & Send.
- If there is only one behavioral statement, grouping is not necessary& similar to begin-end blocks in Pascal programming.


```

Module stimulus:
reg x, y, a, b, m;
  Initial
M=/b0//single statement; does not need to be grouped begin
#5 a=ib/; multiple statements, cannot b grouped
#15 b=in0;
End
  Initial
Begin
#10x=1'b0;
#25y=1'b1;
End
  Initial
#50 finish;
End module.

```

The initial blocks are typically used for initialization, monitoring, wave forms & other processes that must be executed only once during the entire simulation run.

Combined variable Declaration & Initialization:

 Variables can be initialized when they are declared

```
// the clock variable is defined first reg clock;

// the value of clock is set to 0

Initial clock=0;

//Instead of the above method, clock variable
//can be initialized at the time of declaration
//This is allowed only for variables declared
//at module level.
reg clock = 0;
```

Combined Port/Data Declaration and Initialization:

The combined port/data declaration can also be combined

```
Module adder (output reg [7:0] sum = 0,
  //Initialize 8 bit output
Output reg co = 0,
  //Initialize 1 bit output co
Input [7:0] a, b,
Input ci
);
--
--
End module
```

Always Statement

All behavioral statements inside an always statement constitute an always block. The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit.

```
module clock_gen (output reg clock);
//Initialize clock at time zero
initial
clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
#10 clock = ~clock;

initial

#1000 $finish;
End module.
```

Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables.

- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

Assignment ::= variable_lvalue = [delay_or_event_control]

The left-hand side of a procedural assignment <lvalue> can be one of the following:

- A reg, integer, real, or time register variable or a memory element
- A bit select of these variables (e.g., addr[0])
- A part select of these variables (e.g., addr[31:16])

Blocking Assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block.
- The = operator is used to specify blocking assignments.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
```

//All behavioral statements must be inside an initial or always block

initial

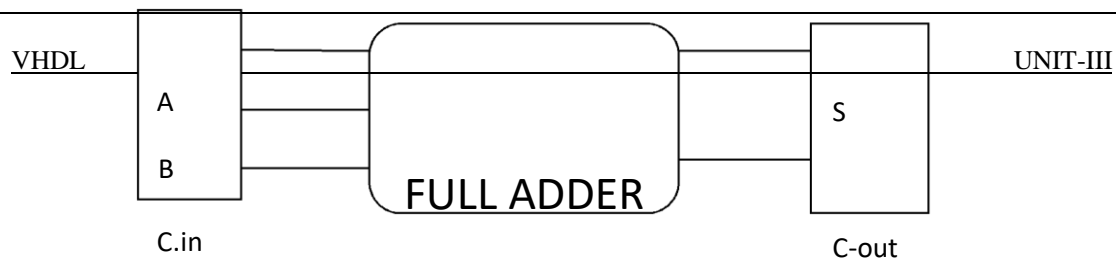
begin

```
x = 0; y = 1; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1; //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation
to
// part select of a vector
count = count + 1; //Assignment to an integer (increment)
End
```

- Statement reg_a[2] = 1 at time = 15
- Statement reg_b[15:13] = {x, y, z} at time = 25

Non blocking Statements:

- A <= operator is used to specify non blocking assignments.
- It is same symbol as relational operator, less_than_equal_to.



```

Entity full adder IN
Port (a, b, cin, IN BIT, s, c-out, out Bit);
End full adder;

Begin
S<=a XOR b XOR cin;
Cont<= (a AND b) OR (a AND c-in) OR (b AND cin);
End

```

Timing Controls

There are 3-types of timing controls.

- 1) Delay based Timing Control.
- 2) Event based Timing Control.
- 3) Level Based timing control.

Delay- Based timing control:

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

```
delay3::= # delay_value | # (delay_value [ , delay_value [ , delay_value ] ] )
```

It is 3-types of delay control procedural assignments.

Regular Delay control:

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment.

```

//define register variables
Reg x, y, q
Initial
Begin
X=0; //no delay control
#10 y=1; //delay control with a number
                Delay execution of //y=1 by 10 units
# (4:5:6) q=0; // min, typical, (y, max delay values).

```


Intra- assignment delay control:

- It is possible to assign a delay to the right of the assignment operator.

```
//define register variables
reg x, y, z;
//intra assignment delays
initial
begin
x = 0; z = 0;
y = #5 x + z; //Take value of x and z at the time=0, evaluate
//x + z and then wait 5 time units to assign value
//to y.
```

Zero Delay control:

Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.

```
initial
begin
x = 0;
y = 0;
end
initial
begin
#0 x = 1; //zero delay control
#0 y = 1;
End
```

Event-Based Timing Control:

- An event is the change in the value on a register or a net.
- There are four types of event-based timing control: regular event control, named event control, event OR control, and 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 positive transition.

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d;
```

Named Event control:

Verilog provides the capability to declare an event and then trigger and recognize.

```
Event_received_data; //define an event called received_data.
```

Event OR control:

This is expressed as an OR of events or signals.

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

Level-Sensitive Timing Control:

Verilog also allows level-sensitive timing control ability to wait for a certain condition to be true before a statement is executed.

```
always
wait (count_enable) #20 count = count + 1;
```

Conditional statements:-

These statements are used for making decision upon certain conditions.

It is executed by keywords if and else.

```
// Type 1 Conditional statement, no else statements
//statement executes (or) does not execute if (<expression>) true-statements;
//Type 2 conditional statement, one else statements
//Either true-statement (or) false statement is evaluate is (<expression>)
true-statement; else false-statement.
//Type 3 Conditional statement nested if-else-if
//choice of multiple statements only one is executed if (<expression>) one-
statement-1;
Else if (<expression>) true-statement2;
Else if (<expression>) true-statement-2;
Else default-statement;
```

Example:-

```
//Type 1 statements
If () buffer =data;
If (enable) out=in;
//Type 2 statements
If (number_queued <Max_Q_Depth)
Data_queue= number_queued
Number_queued = number_queued+1;
End
```

Multi-way branching:-

The nested if-else-if become unwieldy if there are too many alternatives.

To achieve the same result is to use case statement.

Case statement:-

The keywords case, end case & default are used in case statement.

Case (<expression>)

Alternative1 = statement 1;

Alternative2 = statement 2;

Alternative3 = statement 3;

Default= default statement;

End case

Each statement can be single or block of multiple statements.

A block of multiple statements must be granted by keywords and end.

Example: - uxl mux with case statements.

```
Module mux uxl (out, i0, i1, i2, i3, s1, s0);
```

```
//port declaration from I/O diagram
```

```
Output out;
```

```
Input i0, i1, i2, i3;
```

```
Input s1, s0;
```

```
Reg out
```

```
Always @ [s1 (or) s0 (or) i0 (or) i1 (or) i2 (or) i3]
```

```
Case1 {s1, s0} //switch based on concatenation of control signals
```

```
2'd0: out=i0;
```

```
2'd1: out=i1;
```

```
2'd2: out=i2;
```

```
2'd3: out=i3;
```

```
Default: {display ("Email ID control signals")
```

```
End case
```

```
End module
```

There are two case statements they are denoted by caseX & caseZ

CaseX: - It treats all x values in case alternatives (or) case expressions as don't cares.

CaseZ: - It treats all x & z values in case item (or) case expression as don't cares.

Loops

There are four types of looping statements in VHDL - while, for, repeat & forever.

All looping statements can appear only inside in initial (or) always block loops may contain delay expressions.

While Loops:-

The while loop executes until while expression is not true

If while expression not true the loop not executed at all

```
//Increment count from 0 to 127 exits at count 128.
```

```
//Display count variable
```

```
Increment out
```

```
Initial
```

```
Begin
```

```
Count=0;
```

```
While (count<128)
```

```
Begin
```

```
Display ("count=%d", count);
```

```
Count=count+1;
```

```
End.
```

For loop

The keyword for loop contains 3 parts

1. Initial count.
2. Check to see terminating condition is true.
3. Procedural assignment to change value of control variable.

```
Integer count;
```

```
Initial
```

```
For (cont=0; cont<128; count= count+1)
```

```
Display ("count=%d", count);
```

Repeat:-

- The repeat construct execute the loop a fixed no. of times.

```
//Increment & display cont from 0 t 127
```

```
Initial Begin
```

```
Count=0;
```

```
Repeat (128)
```

```
Begin
```

```
Display ("count=%d", count);
```

```
Count=count+1;
```

```
End
```

```
End
```

Forever Loop

- Use forever loop instead at always block
- The loop executes forever until & finish task is encountered.

```
Reg clock
```

Initial

Begin

Clock=1'b0

Forever# to clock = clock;

//clock with a period 20count;

Tasks & Functions: -

- A designer is frequently required to implement the same functionality at many places in a behavioral design.
- Most programming language provide procedures (or) subroutines to accomplish this.
- Virlog provide task and functions to break up large re designs in to smaller pieces.
- Tasks have input, output & input arguments functions have input arguments. Thus values can be passed into and out from task & functions.
- Considering the analog of FORTAN tasks are similar to subroutine & functions are similar to function.

Differences between Tasks & Functions

<i>Functions</i>	<i>Tasks</i>
1. Function can enable another function but not another task.	1. A task can enable other tasks and functions.
2. Functions always execute in simulation time.	2. Task execute in non-zero simulation time.
3. Functions don't contain delay event, timing control statements	3. Tasks may contain delay event (or) timing control statements.
4. Functions must have at least one input argument. They can have more than one input.	4. Tasks may have zero (or) more arguments of type- input, output (or) i/o
5. Functions always return a single value.	5. Tasks do not return with a value.

Tasks: -

- They are declared with keyword task & end task
 1. They are delay timing (or) event control.
 2. The procedure has zero (or) more than one o/p argument.
 3. The procedures have no input argument.

Tasks declaration: -

Task declaration:: =

Task (automatic) task – identifier;

```
{Task- item-declaration}
```

```
Statement
```

```
end task
```

```
{task (automatic)task_identifier/task_port_list;
```

```
{Block-item-declaration}
```

```
Statement
```

```
End task
```

I/O declaration use keywords input, output, in out based on type of argument declared.

Input & output Arguments

- Input & Output arguments in tasks consider a task called Bitwise-operator which denotes on bitwise- AND, OR, X-OR of two 16-bit numbers & parameter delay are also used in task.

```
//Define a module called operation that computing task bitwise-operation
```

```
Module operation;
```

```
-
```

```
-
```

```
-
```

```
Parameter delay=10;
```

```
reg [15:0] A, B;
```

```
reg [15:0] AB- AND, AB- OR, AB- XOR;
```

```
always @ (A (or) B)// whenever A (or) B changes in values.
```

```
begin
```

```
//invoke task bitwise operator provides 2 input arguments A, B & 3 output arguments
```

```
Bitwise- operator (AB-AND, AB-OR, AB-XOR, A, B);
```

```
End
```

```
//define task bitwise-operator
```

```
task bitwise-operator;
```

Output [15:0] ab-and, ab-or, ab-xor;

Input [15:0] a,b;

Begin

#delay ab-and= a&b;

ab-or= a'b;

ab-xor=a^b;

end

end task

-

-

-

end module

Functions: -

- They are declared with keyword function & end function.
- They don't allow delay, timing (or) event control.
- There are no outputs or input arguments.
- There are no blocking assignments.

Function declaration: -

Function- declaration

Function(automatic) (signed) (range (or) type)

Function-identifier;

Function-item-declaration {function-item-declaration}

Function-statement

end function

- Function are very similar to function in fortan
Example – left/right – shifter

// Define a module that contain a function shift


```
Module shift;
-
-
- Left/right
shifter

Define left-shifter 1'b0
Define right- shifter 1'b1
reg [31:0] adder, left-adder, right-adder;
reg control;

//compute the right and left shifted values whenever a new address value appears always @ (adder)
Begin
Left-adder= shift (adder, left-shift);
right-adder= shift (adder, right-shift);
end
-
-
-

//define shift function, the output is 32-bit value
Function (31:0) shift;
Input (31:0) address;
Input control;
Begin
Shift= (control==(left shift)? (address<<1): (address>>));
end
end function
```

-

-

-

End module.

UNIT:4 SWITCH LEVEL MODELING

- ❖ Verilog provides various constructs to model switch-level circuits.
- ❖ Digital circuits at MOS-transistor level are described using these elements.
- ❖ Array of instances can be defined for switches.

1.MOS Switches:

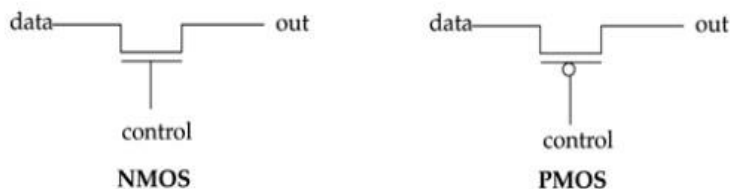
Two types of MOS switches can be defined with the keywords `nmos` and `pmos`.

//MOS switch keywords

`nmos` `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 the figure.

NMOS and PMOS Switches:



Instantiation of NMOS and PMOS Switches:

```
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 an nmos switch; no instance name
pmos (out, data, control); //instantiate a pmos switch; no instance name
```

- ❖ The value of the out signal is determined from the values of data and control signals.
- ❖ Logic tables for out are shown in the 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.

Logic Tables for NMOS and PMOS:

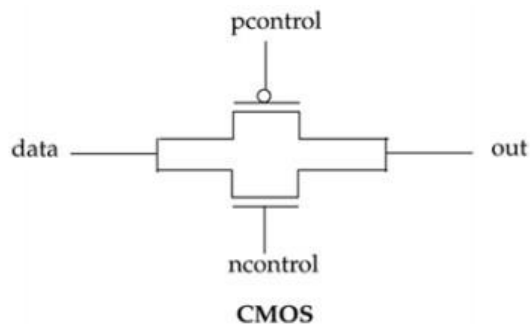
nmos	control				pmos	control			
	0	1	x	z		0	1	x	z
0	z	0	L	L	0	0	z	L	L
data 1	z	1	H	H	data 1	1	z	H	H
x	z	x	x	x	x	x	z	x	x
z	z	z	z	z	z	z	z	z	z

Thus, the nmos switch conducts when its control signal is 1. If the control signal is 0, the output assumes a high impedance value. Similarly, a pmos switch conducts if the control signal is 0.

2.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 the figure below.

CMOS Switch:**Instantiation of CMOS Switch:**

```
cmos c1(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.
- ❖ If ncontrol signal is 0 and pcontrol is 1, the output of the switch is high impedance value.
- ❖ The cmos gate is essentially a combination of two gates: one nmos and one pmos. Thus the cmos instantiation shown above is equivalent to the following:

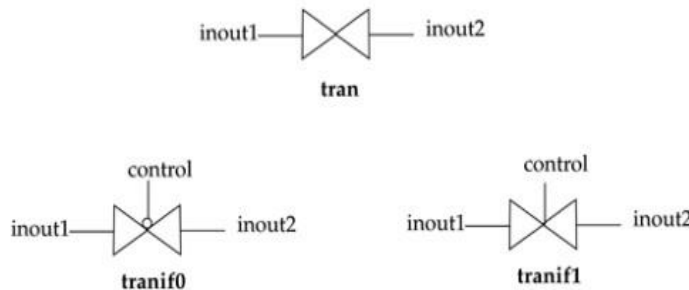
```
nmos (out, data, ncontrol); //instantiate a nmos
switch pmos (out, data, pcontrol); //instantiate a
pmos switch
```

Since a cmos switch is derived from nmos and pmos switches, it is possible to derive the output value from the table, given values of data, ncontrol, and pcontrol signals.

3.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.

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.

Example: Instantiation of Bidirectional Switches:

```
tran t1(inout1, inout2); //instance name t1 is optional
tranif0 (inout1, inout2, control); //instance name is not specified
tranif1 (inout1, inout2, control); //instance name is not specified
```

Bidirectional switches are typically used to provide isolation between buses or signals.

4.POWER AND GROUND:

- ❖ The power (Vdd, logic 1) and Ground (Vss, logic 0) sources are needed when transistor level circuits are designed.
- ❖ Power and ground sources are defined with keywords supply1 and supply0.
- ❖ Sources of type supply1 are equivalent to Vdd in circuits and place a logical 1 on a net.
- ❖ Sources of the type supply0 are equivalent to ground or Vss and place a logical 0 on a net.

- ❖ Both supply1 and supply0 place logical 1 and 0 continuously on nets throughout the simulation.
- ❖ Sources supply1 and supply0 are shown below.

```
supply1 vdd;
supply0 gnd;
assign a = vdd; //Connect a to vdd
assign b = gnd; //Connect b to gnd
```

5.RESISTIVE SWITCHES:

- ❖ MOS, CMOS, and bidirectional switches discussed before can be modeled as corresponding resistive devices.
- ❖ Resistive switches have higher source-to-drain impedance than regular switches and reduce the strength of signals passing through them.
- ❖ Resistive switches are declared with keywords that have an "r" prefixed to the corresponding keyword for the regular switch.
- ❖ Resistive switches have the same syntax as regular switches.

```
rnmos rmos //resistive nmos and pmos switches
rcmos //resistive cmos switch
rtran rtranif0 rtranif1 //resistive bidirectional switches.
```

- ❖ There are two main differences between regular switches and resistive switches: their source-to-drain impedances and the way they pass signal strengths.
- ❖ Strength Modeling and Advanced Net Definitions, for strength levels in Verilog.
- Resistive devices have a high source-to-drain impedance. Regular switches have a low source-to-drain impedance.
- 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 strength supply, the output is of strong strength.

Strength Reduction by Resistive

Switches: Input Strength - Output

Strength

Supply	-	pull
strong	-	pull
pull	-	weak
weak	-	medium
medium	-	small
Small	-	small
high	-	high

6.Delay Specification on Switches:

MOS and CMOS switches:

- ❖ Delays can be specified for signals that pass through these switch-level elements.
- ❖ Delays are optional and appear immediately after the keyword for the switch.
- ❖ Rise, Fall, and Turn-off Delays. Zero, one, two, or three delays can be specified for switches according to the table.

Delay Specification on MOS and CMOS Switches

Switch Element	Delay Specification	Examples
pmos, nmos, rpmos, rnmos	Zero (no delay)	pmos p1(out, data, control);
	One (same delay on all transitions)	pmos #(1) p1(out, data, control);
	Two (rise, fall)	nmos #(1, 2) p2(out, data, control);
	Three (rise, fall, turnoff)	nmos #(1, 3, 2) p2(out, data, control);
cmos, rcmos	Zero, one, two, or three delays (same as above)	cmos #(5) c2(out, data, nctrl, pctrl);
		cmos #(1,2) c1(out, data, nctrl, pctrl);

Bidirectional pass switches:

- ❖ Delay specification is interpreted slightly differently for bidirectional pass switches.
- ❖ These switches do not delay signals passing through them. Instead, they have turn-on and turn-off delays while switching.
- ❖ Zero, one, or two delays can be specified for bidirectional switches, as shown in the table below.

Delay Specification for Bidirectional Switches:

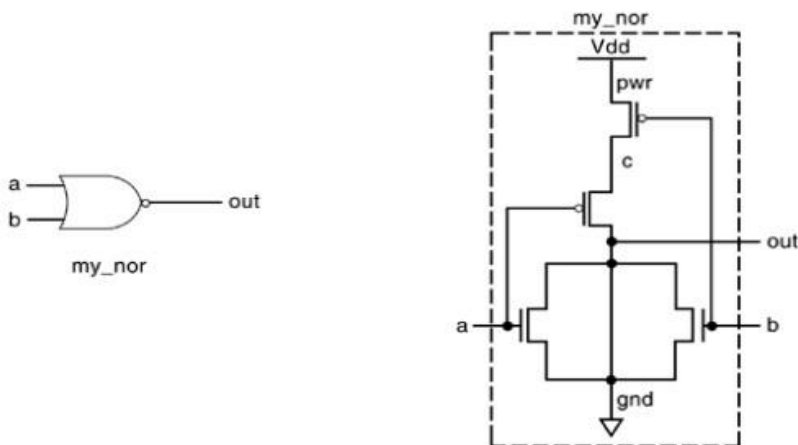
Switch Element	Delay Specification	Examples
tran, rtran	No delay specification allowed	
tranif1, rtranif1 tranif0, rtranif0	Zero (no delay)	rtranif0 rt1(inout1, inout2, control);
	One (both turn-on and turn-off)	tranif0 #(3) T(inout1, inout2, control);
	Two (turn-on, turn-off)	tranif1 #(1,2) t1(inout1, inout2, control);

7.EXAMPLES:

i.CMOS Nor Gate

Though Verilog has a nor gate primitive, let us design our own nor gate,using CMOS switches. The gate and the switch-level circuit diagram for the nor gate are shown in the figure below.

Gate and Switch Diagram for Nor Gate:



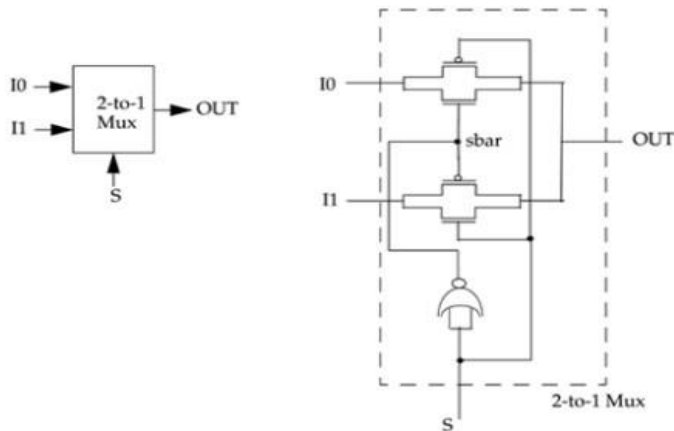
Example:Switch-Level Verilog for Nor Gate

```
//Define our own nor gate, my_nor
module my_nor(out, a, b);
output out;
input a, b;
//internal wires
wire c;
//set up power and ground lines
supply1 pwr; //pwr is connected to Vdd (power supply)
supply0 gnd ; //gnd is connected to Vss(ground)
//instantiate pmos switches
pmos (c, pwr, b);
pmos (out, c, a);
//instantiate nmos switches
nmos (out, gnd, a);
nmos (out, gnd, b);
endmodule
```

ii. 2-to-1 Multiplexer:

A 2-to-1 multiplexer can be defined with CMOS switches. CMOS Nor Gate to implement the not function.

2-to-1 Multiplexer. Using Switches:

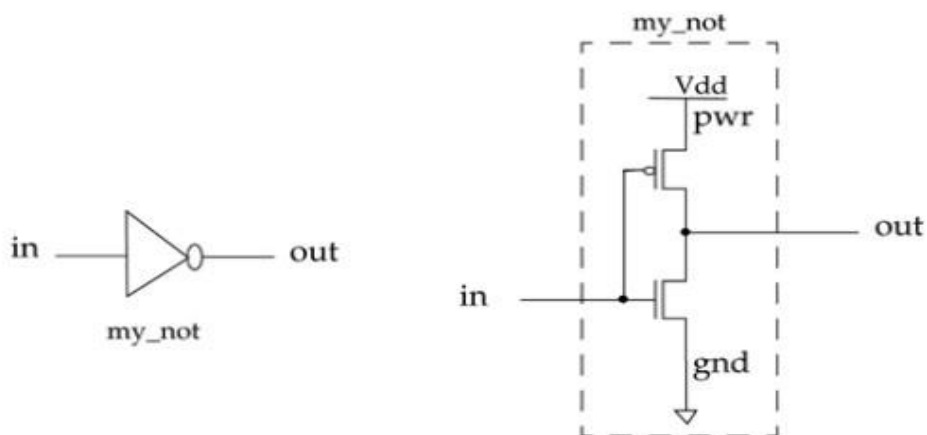


The 2-to-1 multiplexer passes the input I0 to output OUT if $S = 0$ and passes I1 to OUT if $S = 1$.

Switch-Level Verilog Description of 2-to-1 Multiplexer:

```
//Define a 2-to-1 multiplexer using switches
module my_mux (out, s, i0, i1);
output out;
input s, i0, i1;
//internal wire
wire sbar; //complement of s
//create the complement of s; use my_nor defined previously.
my_nor nt(sbar, s, s); //equivalent to a not gate
//instantiate cmos switches
cmos (out, i0, sbar, s);
cmos (out, i1, s, sbar);
endmodule
```

iii. CMOS Inverter:



Example:CMOS Inverter:

```
//Define an inverter using MOS switches
module my_not(out, in);
output out;
input in;
//declare power and ground
supply1 pwr;
supply0 gnd;
//instantiate nmos and pmos switches
pmos (out, pwr, in);
nmos (out, gnd, in);
endmodule
```

Now, the CMOS latch can be defined using the CMOS switches and my_not inverters.

UNIT -V

LOGIC SYNTHESIS WITH VERILOG HDL

- It is forefront of digital design technology tools which have's cut design cycle time significantly.
- It is the process of converting a high level description into an optimized gate level representation.
- It uses a standard cell library and certain design constraints.
- A standard cell library also known as Technology library.
- Standard cell library contains simple cells such as basic logic gates like AND, OR, & NOR or macro cells such as adders, muxes & special flipflops.
- Logic synthesis always existed even in the days of schematic gate level design. But, it was done inside the designer mind.
- The designer constraints are timing, area, testability and power.
- The partition of a design into high level blocks, drawing them on a piece of paper or a computer terminal and describing the functionality of circuit is done by the designer.
- Finally, each block would be implemented on a hand-drawn schematic, using the cells available in standard library.
- The last step of the process is the design flow an its required several time consuming design iteratives before an optimized gate level representation that meets all design constraints like shown in fig(a).

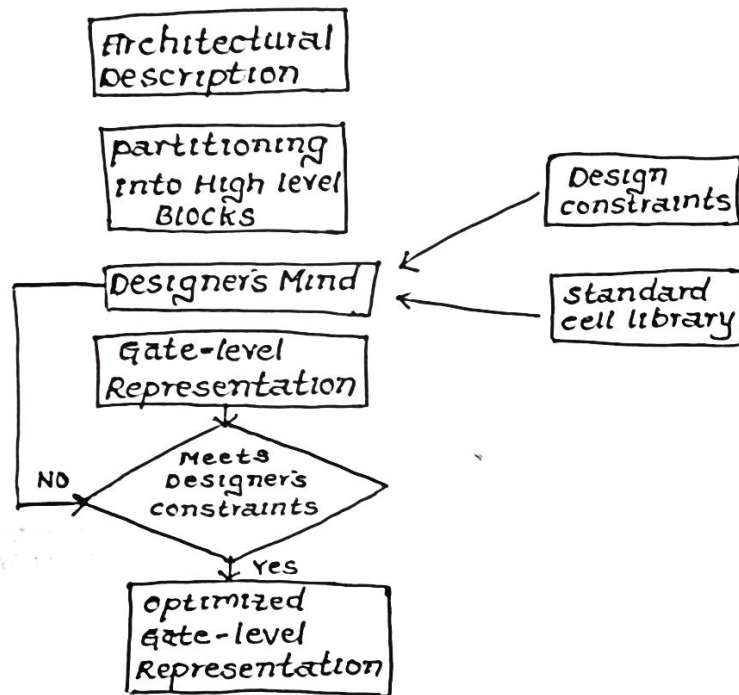
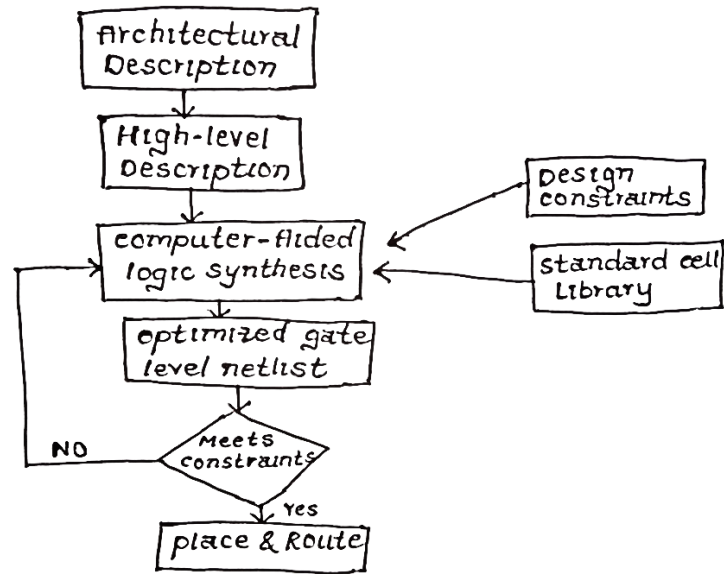


FIG (A)

- The advent of computer aided logic synthesis tools has automated process of converting high level description to logic gates.
- The designers concentrate on architectural trade-offs, high level description of design, accurate design constraints & optimization of cells in standard cell library.
- Finally fed to computer aided logic synthesis tool, which performs several iteratives internally & generates the optimized gate level description.
- Logic synthesis tool has significantly reduced time for conversion from high level design representation to gates.
- It allows all the designers to spend more time on designing at a high level representation because less time required for converting design to gates.



IMPACT OF LOGIC SYNTHESIS

Logic synthesis has revolutionized digital design industry by significantly improving productivity and reducing design cycle time.

- 1) For large designs, manual conversion has prone to human error and small gate missed somewhere could mean redesign of entire blocks.
- 2) The designer could never be sure the design constraints were going to met until the gate level implementation was completed & tested.
- 3) The design cycle was dominated by the time taken to convert a high- level design into gates.
- 4) If gate level did not meet requirements, the turn-around time for redesign of blocks was very high.
- 5) What-if scenarios were hard to verify.
- 6) Each designer would implement design block differently.

- 7) If a bug was found in final gate level design, this would sometimes requires the redesign of thousands of gates.
- 8) Timing, area and power dissipation in library cells are specific fabricated technology and company changed the IC fabrication vendor after the gate level design complete, this would mean the Redesign of entire circuit
- 9) Reuse of design is not possible.

Automated logic synthesis tools addressed the problems ;

- High level design is less prone to human error because designs are described at a higher level of abstraction.
- High level design is done without significant concern about design constraints.
- Conversion from high level design to gates is fast (months, days, hours).
- Turn around time for redesign blocks is shorter because changes are required only at Register transfer level.
- What-if scenarios are easy to verify.
- Logic synthesis tool optimize the design as a whole.
- If a bug is found in gate level design, the designer goes back and changes the high level description to eliminate the bug.
- Logic synthesis tools allow technology independence design.
- Reuse of design is possible.

VERILOG HDL SYNTHESIS

- * Logic synthesis, designs are currently written in HDL at a Register transfer level (RTL).
- * RTL is used for HDL description style and utilizes combination of data flow & behavioral constructs.

VERILOG CONSTRUCTS ;

In general any construct is used to design a cycle-by-cycle RTL description to the logic synthesis tool.

CONSTRUCT TYPE	KEYWORD	NOTES
Ports	Input, output, inout	
Parameters	Parameter	
Signals & variables	Wire, Register, Transistor	Vectors are allowed
Functions and tasks	Function, Task	Timing constructs ignored
Procedure	Always, if , then, else, case, casex, casez	Initial isn't supported
Data flow	assign	Delay information is ignored

The initial construct is not supported by logic synthesis tool.

VERILOG OPERATORS

- All operators in Verilog are allowed for logic synthesis.
- Except two instructions i.e case equality (***) and case inequality (!**) aren't allowed because they are related to X.
- 'X' does not have much meaning in logic synthesis.

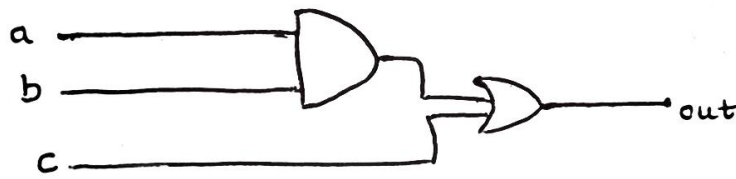
OPERATOR TYPE	OPERATOR SYMBOL	OPERATION PERFORMED
Arithmetic operator	*, /, +, -, % + -	Mul, div, add, sub Modules Unary plus Unary minus
Logical operator	 && 	Logical negation Logical and Logical or
Relational operator	> < >= <=	Greater than Less than Greater than or equal Less than or equal
Equality operator	== !=	Equality Inequality
Shift operator	>> << >>> <<<	Right shift Left shift Arithmetic right shift Arithmetic left shift
Concatenation operator	{ }	Concatenation
conditional operator	? :	conditional

INTERPRETATION OF A FEW VERILOG CONSTRUCTS

1.Assign statements ;

- The assign construct is most fundamental construct used to describe combinational logic at an RTL level.
- A logic expression that uses assign statement.

assign out=(a&b) | C;



Example ; if a conditional operator ? is used, a multiplex circuit is inferred.

2.If-else statements ;

Single if-else statements translate to multiplexers when the control signal is the variable in the if class.

If(s)

out=i1;

else

out=i0;

3.case statements ;

The case statement also can be used to inter multiplexes.

Case(s)

1¹b0; out=i0;

1¹b1; out=i1;

End case

4.for loops ;

- For loops can be used to build cascaded combination logic.
- For example, the following for loop builds a 8-bit full adder.

C= C_in

For(io, i<=7, i++);

For{c, sum(i)} = a{i}+b{l}+c; //builds a 8-bit ripple adder

C_out= C;

5.Always statements ;

- The always statements can be used to inter sequence & combinational logic.
- For sequential logic the always statement must be controlled by the change in the value of a clock signal clk.

Always @(posedge clk)

q<=d;

always @(clk or d)

if (clk)

q<=d

SYNTHESIS DESIGN FLOW

- ✦ The synthesis design flow is from RTL description to optimized gate level description.

RTL to Gates ;

- ✦ The designer must first understand the flow from the high level RTL description to a gate level netlist.

diagram

RTL description ;

The designers spend time in functional verification to ensure that the RTL description functions correctly.

Translation ;

VHDL

The RTL description is converted by logic synthesis tool to an unoptimized intermediate, internal representation & process known as Translation.

The Translator understand the basic primitives and operators in Verilog RTL description.

Unoptimized intermediate representation ;

The design is represented internally by the logic synthesis tool in terms of internal data structure.

Logic optimization ;

Boolean logic optimization techniques are used to remove redundant logic & yields of n optimized internal representation of the design.

Technology mapping & optimization ;

It is independent to specific target technology.

The synthesis tool takes the internal representation & implement the representation in gates using the cells provided in technology library.

The design is mapped to the desired target technology.

Technology library ;

The technology library contains library cells provided by ABC Inc.

The term standard cell library & technology library are interchangeable.

To build a technology library, library cells can be basic gates, macro cells, ALUS such as adders, multiplexers & specific flipflops.

The library cells are the basic building blocks that used for IC fabrication by ABC Inc.

Physical layout of library cells are done first and area of each cell is completed from the cell layout.

Modelling techniques are used to estimate timing & power characteristics of each library cell called Cell Characterization.

Each cell is describes as

- (1) Functionality of the cell
- (2) Area of the cell layout
- (3) Timing information about the cell
- (4) Power information about the cell

Design constraints ;

Timing : The circuit must meet certain timing requirements.

A internal static timing analyzer checks timing.

Area : The area of the final layout must not exceed a limit.

Power : The power dissipation in the circuit must not exceed
Threshold.

On the top of design constraints operating environment factors such as input and output delays, drives, strength, loads will affect the optimization of target technology.

Operating environment factors must be input to the logic synthesis tool to ensure that circuits are optimized for the required operating environment.

Optimized gate level representation ;

After the technology mapping is completed, an optimized gate level netlist described in terms of larger components is produced.

There are 3 points for synthesis flow

1. For high speed clks like microprocessors, vendor technologies may yield non-optimal results.

VHDL

2. Translation logic optimization & technology mapping are done internally in the logic synthesis tool and not visible to designer.
3. For submicron designs, interconnect delays are becoming a dominating factor in the over all delay.

Example ;-

Magnitude comparator checks if number is $>$, $<$, $=$ IC chip.

In RTL description that the designer doesn't have to worry about the target technology at this point.

RTL for magnitude comparator

```
//module magnitude comparator
Module.mag_comp(A_gt_B, A_lt_B, A_eq_B, a, b);
//comparision output
Output A_gt_B, A_lt-B, a_eq_B;
//4-bits numbers input
Input(3:0) A, B;
assign A_gt_B = (A>B);
assign A_lt_B = (A<B);
assign A_eq_B = (A==B);
end module
```

VERIFICATION OF GATE LEVEL NETLIST

- ❖ The optimized gate level netlist produced by logic synthesis tool must be verified for functionality.

VHDL

- ❖ Synthesis tool not always be able to meet both timing and area requirements. If they are, stringent verification can be done on gate level netlist.

Functional verification ;

Identical stimulus is run with original RTL & synthesized gate level description of design.

The output is compared to find mismatches module stimulus.

```

reg(3:0) A, B;
wire A_gt_B, A_ls_B, A_eq_B;
// instantiate the magnitude comparator
mag_comp MG(A_gt_B, A_ls_B, A_eq_B, A< B);
initial
$monitrr($ have, " A=%b, B%b;
           A_gt_B=%b,
           A_ls_B, A_eq_B = %b", A, B,
A_gt_B, A_ls_B, A_eq_B);
//stimulate the magnitude comparator.
Initial begin
A=41b1010; B=41b1001;
#10 A=41 b1110; B=411111;
#10 A=41 b0000; B=41b0000;
#10 A=41 b1000; B=41b1100;
#10 A=41 b0110; B=41b1110;
#10 A=41 b1110; B=41b1110;

```

VHDL

End

End module

The same stimulus is applied to both RTL description & synthesized gate level description.

Th gate level description in terms of library cells VNAN, VAND etc...

Verilog simulators don't understand meaning of these cells.

```
//simulate library abc-100.v;
```

```
Module VAND(out, in0, in1);
```

```
Input in0;
```

```
Input in1;
```

```
Output out;
```

```
(in0 => out)=(0.260:0.513:0.955)
```

```
(in1 => out)=0.260:0.513:0.955)
```

```
End specify
```

```
End module
```

MODELLING TIPS FOR LOGIC SYNTHESIS

The Verilog RTL design style used by designer effects the final gate level netlist produced by logic synthesis.

Logic synthesis can be produce efficient (or) inefficient gate level netlist based on style of RTL description.

Verilog coding style ;

- ⤴ The style of Verilog description greatly effects the final design.
- ⤴ It is important to consider actual hardware implementation issues.

Significant names for signals & variables ;

- ⤴ Names of signals and variables should be meaningful so that code becomes self-commented and readable.

Avoid mixing positive and negative triggered flipflops ;

- ⤴ Mixing of the positive and negative edge triggered flipflops may introduce inverters and buffers into clock tree.

Floor planning constraints ;

- ⤴ Minimize the total chip area.
- ⤴ Make routing phase easy (routable),
- ⤴ Improve the performance by reading signal delays.

Placement of component ;

- ⤴ It is the process of placing standard cells in the rows creating at floor planning stage. The goal is to minimize total area & interconnects cost. The quality of routing is determine by placement.

Logic synthesis ;

- ⤴ As technology advances, the number of gates on a chip increases.
- ⤴ High circuits are impractical to design manually. Today's IC 's have 10m & above transistors.
- ⤴ Huge reduction time is required to create circuits.

- ▲ Logic synthesis are execute very fast when compared to schematic capture , it reduces the ASIC design cycle.

Definition ;

The process of passing, translating, optimizing and mapping RTL code into a specified standard cell library, the logic synthesis input is

- ▲ Timing libraries in library format.
- ▲ RTL in the Verilog language or HDL.
- ▲ Timing constraints are delays , clock etc.

Output ;

The out put is in gate level netlist in Verilog language or other HDL.

Minimize area ; In terms of cell constraints & cell size.

Minimize power ; In terms of switching activity in individual gates and de-active clock blocks.

In terms of leakage power ;-

Logic synthesis = Translation + Mapping + Optimization

Compile strategy ;

In this we can use 3 types of compilation strategies for the hierarchical designs.

1. Top-bottom compile.
2. Bottom-top compile
3. Mixed compile