Unit- 3

DIGITAL DESIGN WITH VERILOG HDL: Modules – instances – Data types – Arrays – System tasks – directives – Modules and Ports – Gate-Level Modeling – Dataflow Modeling – Behavioral Modeling - Design of Multiplexers, counters and full adders – Introduction - Hierarchical Modeling concepts – 4-bit ripple carry counter.

DIGITAL DESIGN WITH VERILOG HDL

Overview of Digital design with Verilog HDL

Evolution of CAD,

Digital circuit design has evolved rapidly over the last 25 years. Integrated circuits were then invented where logic gates were placed on a single chip. The first integrated circuit (IC) chips were SS1 (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 MS1 (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. Computer Aided Design (CAD) 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.

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. Logic simulators came into existence to verify the functionality of these circuits before they were fabricated.

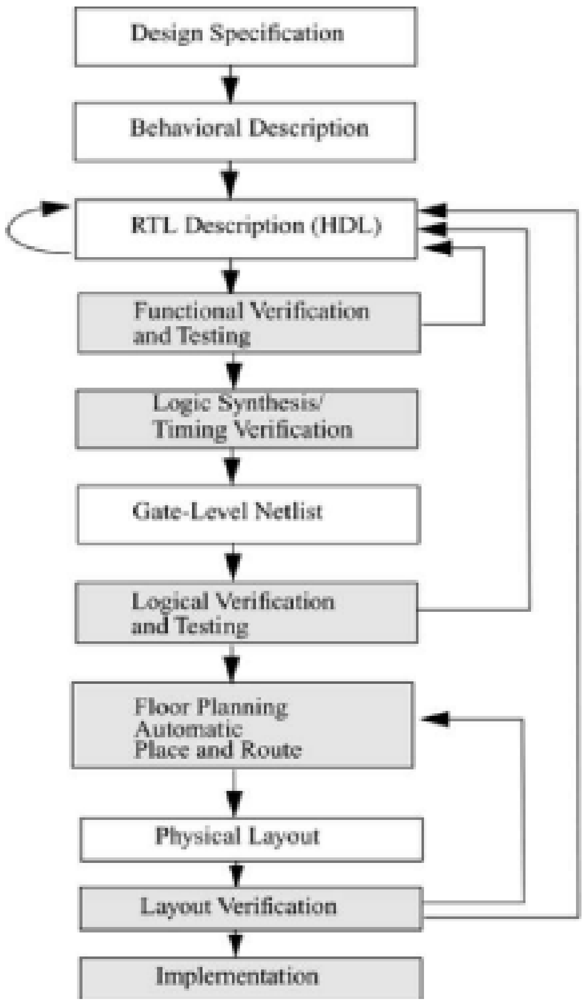
Emergence of HDLs

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. HDLs allowed the designers to model the concurrency of processes found in hardware elements

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

Design Flow

A typical design flow for designing VLSI IC circuits is shown in the following Figure.



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

Design Specification:

Design Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed.

Behavioural description:

A behavioural description is then created to analyse the design in terms of functionality, performance and compliance to standards, and other high-level issues. Behavioural descriptions can be written with HDLs.

RTL Description and Functional verification:

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

Logic synthesis tools convert the RTL description to a gate-level netlist.

A gate level netlist is a description of the circuit in terms of gates and connections between them. 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 chip.

Recently Behavioural synthesis tools are emerging. These tools can create RTL descriptions from a behavioural or algorithmic description of the circuit. As these tools mature, digital circuit design will become similar to high-level computer programming

<u>Need of Verilog HDL:</u>

· 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. If a new technology emerges, designers need not have to redesign their circuit Logic.

· By describing designs in HDLs, functional verification of the design can be done early in the design cycle

· This provides a precise representation of the design.

<u>Trends in HDLs:</u>

1. The current trend in design in HDL is at RTL level, because logic synthesis tools can be created easily.

2. Behavioural synthesis has recently emerged. As these tools improve, designers will be able to design directly in terms of algorithms and the behaviour of the circuit and then use CAD tools to do the translation.

3. For very high speed circuits like microprocessors, the gate level netlist provided by logic synthesis tools is not optimal. In such cases, designers can mix gate-level description directly into the RTL description to achieve optimum results.

4. A trend that is emerging for system-level design is a mixed bottom-up methodology. This is done to reduce development costs and compress design schedules.

Hierarchical Modelling Concepts:

Design methodologies:

There are two basic types of digital design methodologies:

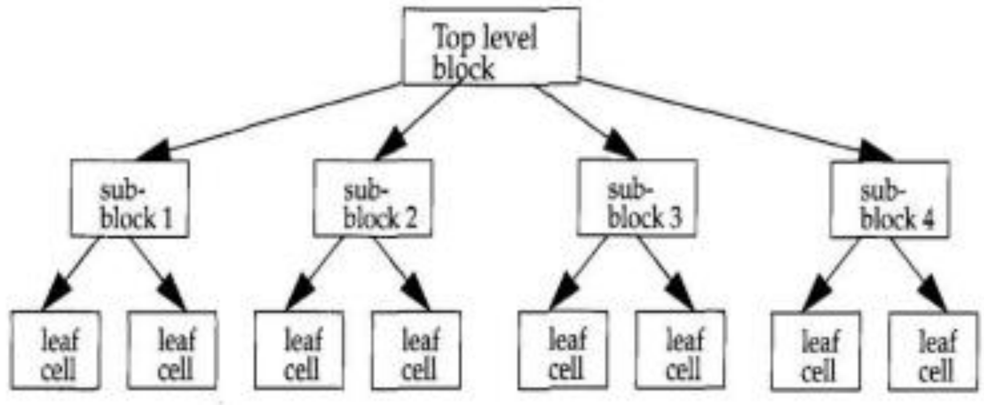
1. Top-down design methodology and

2. Bottom-up design methodology.

Top-Down design methodology:

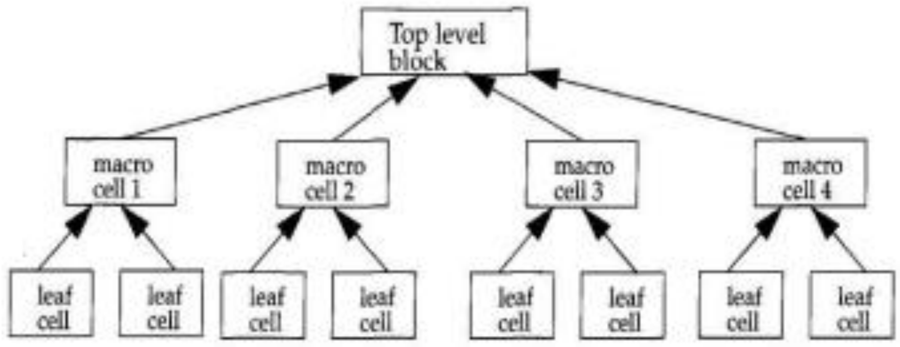
The block diagram of top-down design process is as shown below

In a top-down design methodology, the top-level blocks are defined. The sub-blocks which are necessary to build the top-level block are identified. The sub-blocks are further subdivided to leaf cells, which are the cells that cannot further be divided.



Bottom-up design methodology:

The block diagram of Bottom – up design process is as shown below

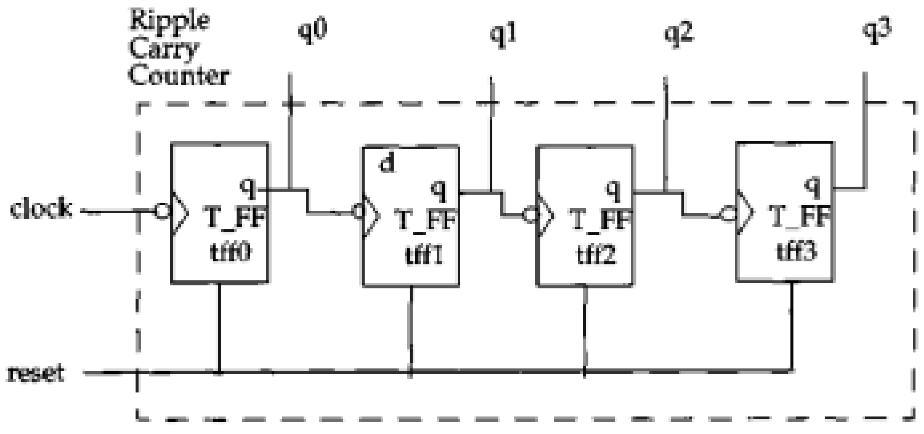


In a bottom-up design methodology, the building blocks that are available are identified. Bigger cells are built, using these building blocks. These cells are then used for higher-level blocks until the top level block in the design is built.

Typically, a combination of top-down and bottom-up flows is used.

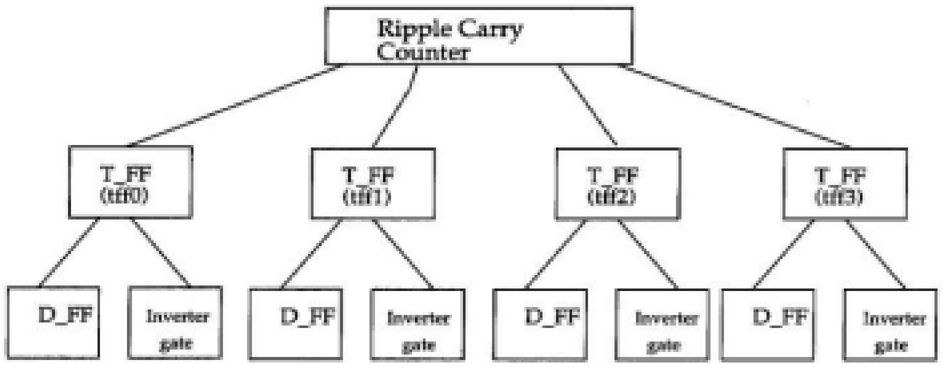
Example:

Consider a negative edge-triggered 4 bit ripple carry counter. The circuit diagram is as shown



below

The ripple carry counter shown in Figure is made up of negative edge triggered toggle flip-flops (T FF). Each of the T-FFs can be made up from negative edge-triggered D-flip-flops (D-FF) and inverters. Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is shown below.



In a top-down design methodology, the functionality of the top-level block i.e., the ripple carry counter is specified. Then, the counter with T-FFs.is implemented which is built with the T-FFs from the D-FF and an additional inverter gate. Thus, bigger blocks are broken into smaller building sub blocks until the blocks cannot be broken further.

The bottom-up methodology flows in the opposite direction. The top-level building blocks are built using small building blocks.

e.g., D-FF is built from **and** and **or** gates, and T-FFs are built using D-FF and inverter gates.

Modules

A module is the basic building block in Verilog. In Verilog, a module is declared by the keyword **module.** A corresponding keyword **endmodule** must appear at the end of the module definition. 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.

The Syntax is

module<module-name> (<module-terminal-list>)

<module internals>

Endmodule

Example:

Consider T- FF. The module for T-FF is as shown below.

module T-FF (q, clock, reset);

.

.

<functionality of T-flip-flop>

endmodule

Internals of each module can be defined at *four* levels of abstraction, depending on the needs of the design. They are

1. Behavioural or algorithmic level

2. Dataflow level

3. Gate level

4. Switch level

Behavioural 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 considering the hardware implementation details.

· It specifies the circuit in terms of its expected behaviour.

· Designing at this level is very similar to C programming.

Dataflow level

· At this level the module is designed by specifying the data flow.

· This design describes how data flows between hardware registers and how the data is processed in the design

· This style is similar to logical equations.

· The specification is comprised of expressions made up of input signals and assigned to outputs. .

Gate level (Structural level)

· The module is implemented in terms of logic gates and interconnections between these gates. · It resembles a schematic drawing with components connected with signals.

· A change in the value of any input signal of a component activates the component. If two or more components are activated concurrently, they will perform their actions concurrently as well.

· Since logic gate is most popular component, Verilog has a predefined set of logic gates known as ***primitives***. Any digital circuit can be built from these primitives.

Switch level

· This is the lowest level of abstraction in 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.

Instances:

A module provides a template from which actual objects are created. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and Input/output interface. The process of creating objects from a module template is called ***instantiation,*** and the objects are called ***instances***

{Explanation:

The difference between these two can be summed up as follows: A module position is a container in which you can assign modules so they appear on the front end. A module instance is a single module, injected directly into a specific place in the page}

Differences between modules and module instances:

Modules are the basic building blocks in Verilog. **Modules are used in a design by instantiation.** An instance of a module has a unique identity and is different from other instances of the same module. Each instance has an independent copy of the internals of the module **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. Such a block is called the ***stimulus* block**. The stimulus block is also commonly called a **test bench**.

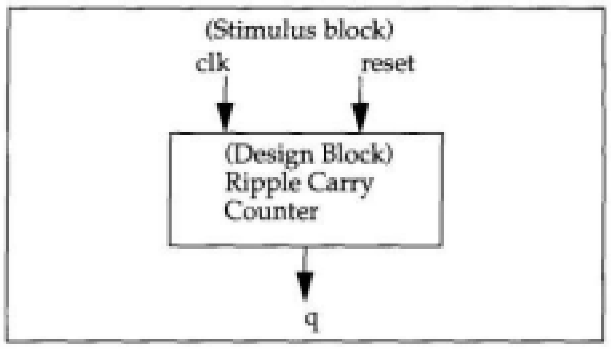
There are two distinct components in a simulation:

· Stimulus block and

· . Design block

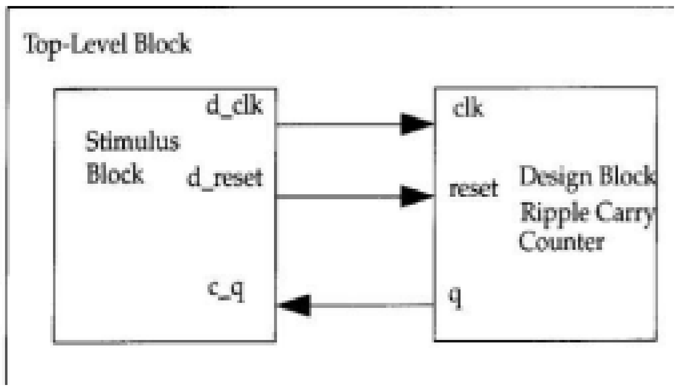
A stimulus block is used to test the design block. The stimulus block is usually the top-level block. There are two styles of stimulus application.

In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block. The block diagram is as shown below.



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

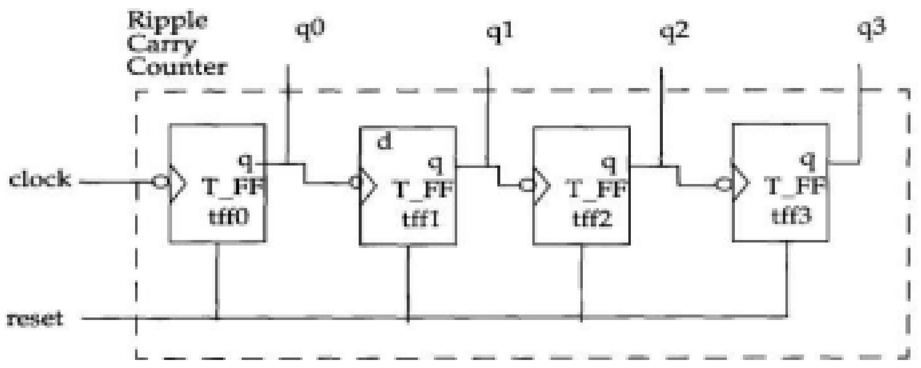
The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface. This style of applying stimulus is shown below.



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 block is simply to instantiate the design and stimulus blocks.
Example:

Consider ripple carry counter. The design block and the stimulus block are defined. The stimulus is applied to the design block and the outputs are monitored. The circuit is as shown below



Design Block

Let us consider top-down design methodology. First, the Verilog description of Ripple carry counter which is the top-level design block is written.

Module ripple-carry-counter(q, clk, reset) ;

output [3:0]q;

input clk, reset

T-FF tff0 (q[0] ,clk, reset);1`234

T-FF tff l (q[l] ,q[0], reset);

T-FF tff 2 (q[2] ,q[l], reset);

T-FF tff 3 (q[3] ,q[2], reset);

endmodule

In the above module, four instances of the module TFF (T-flip-flop) are used. Therefore, the internals of the module T-FF must be defined.

module T-FF (q, clk, reset) ;

output q;

input clk, reset;

wire d;

D-FF dff0 (q, d, clk, reset);

notnl(d, q) ; // not is a Verilog-provided primitive. case sensitive

endmodule

Since T-FF instantiates D-FF, the internals of module D-FF must be defined.

// module D-FF with synchronous reset

module D-FF(q, d, clk, reset) ;

output q;

input d, clk, reset;

reg q;

always @ (posedge, reset or negedgeclk)

if (reset)

q = l'b0;

else

q = *d;*

endmodule

All modules have been defined down to the lowest-level leaf cells in the design methodology. The design block is now complete.

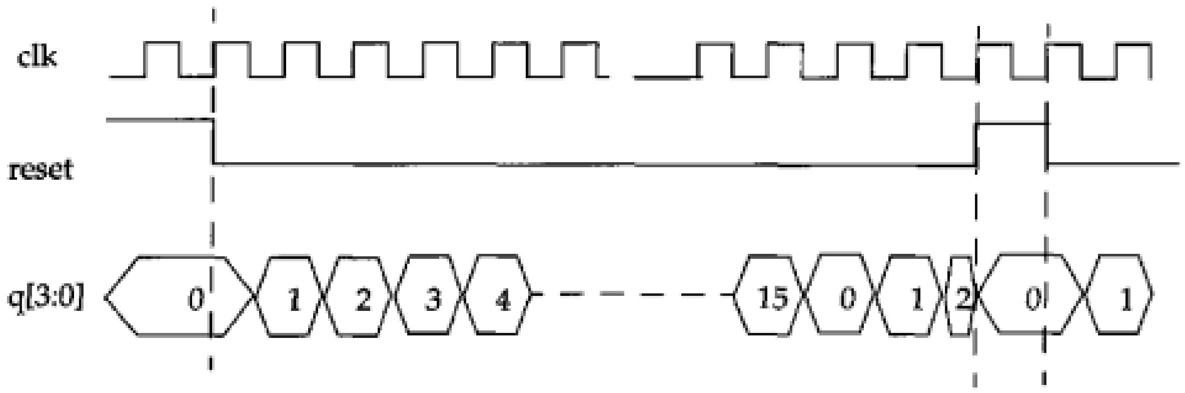
Stimulus Block:

Now the stimulus block is written to check if the ripple carry counter design is functioning correctly. In this case, the signals clk and reset are controlled so that the regular function of the ripple carry counter and the asynchronous reset mechanism are both tested. The wave forms are as shown below.

age9

P

0 age1 P



Now the stimulus block is written to know how the design block is instantiated in the stimulus block that will create the above waveforms

Here the cycle time for clk is 10 units; the *reset* signal stays up from time 0 to 15 and then goes up again from time 195 to 205. Output q counts from 0 to 15.

```
module stimulus;
reg clk;
reg reset;
wire [3:0] q;
// instantiate the design block
ripple-carry-counterrl (q, clk, reset);
// Control the clk signal that drives the design block.Cycle time = l0 initial
clk = l'b0; //set clk to 0
always
#5 clk = ~clk;
//toggle clk every 5 time units
// Control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220 initial
begin
reset = l'bl;
#l5 reset = l'b0;
#l80 reset = l'bl;
#l0 reset = l'b0;
#20 $finish; //terminate the simulation
end
// Monitor the outputs
endmodule
```

3. **Basic concepts:**

The digital circuits can be represented in textual form in Verilog. The code written in textual form is called source code. The source code is created with the combinations of character and words which are called **KEYWORDS** and SYNTAX / SEMANTICS. These are called Lexical tokens. Combination of more than one lexical tokens forms LEXICAL CONVENTIOINS

LEXICAL CONVENTIOINS

Lexical tokens in Verilog HDL are given below.

· White space

· Comment

· Operator

· Number

White space (Blank spaces):

· String · Identifier · Keyword

1 age1 P

White space is **a term used to represent the characters for spaces, tabs, newlines .**White spaces are used in stings which can be incorporated using double quotes (“). The semantics for white spaces in the string is as shown below:

Escape string	**Character produced by escape string**
\b	Blank space
\n	New line character
\t	Tab character
\\	\ character
\”	“ character

Comments:

There are two ways to write comments in Verilog. **A single line comment starts with // and tells Verilog compiler to treat everything after this point to the end of the line as a comment**. A multiple-line comment starts with /* and ends with */ and cannot be nested.

Comments are used to increase the readability of code. Comments are ignored by the simulator. **Example:**

a = b &&c; // This is a one-line comment

/ * This is a multiple line

comment* /

/ * This is / * an illegal Binary operator – which worsl* / comment * /

Operators:

The operation which is to be performed is decided by operators. There are three kinds of operators available in Verilog. They are

1. *Unary operator* – which works on single operands

Eg: a = ~ b; //~ is a unary operator. b is the operand

2. *Binary operator* – which works on two operands

Eg: a = b &&c; // &&is a binary operator. b and c are operands

3. *Ternary operator*– which works on three operands. A question mark and a colon separate the three targets of the operation.

Eg: a = b ?c :d; // ?: is a ternary operator. b, c and d are operands

Number Specification:

There are two types of number specification in Verilog:

· Sized and

· Unsized

Sized numbers:

Sized numbers are represented as *<size>'<base format><number>.*

· *Size* is written only in decimal and specifies the number of bits in the number. · Legal base formats are decimal *('d* or 'D), hexadecimal *('h* or 'H), binary *('b* or 'B) and octal *('*oor *'O).*

· The number is specified as consecutive digits from *0, 1, 2,3,4,5,* 6, *7, 8, 9, a, b, c, d, e, f.* Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification

Example:

4'bllll // This is a 4-bit binary number

l2'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number.

Unsized numbers

Numbers that are specified without a <base *format>*specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits of 32. **Example:**

23456 // 23456 is a 32-bit decimal number by default

'hc3 // c3is a 32-bit hexadecimal number

'o21 // 21 is a 32-bit octal number

X or Z values:

Verilog has two symbols for unknown and high impedance values. An unknown value is denoted by an X. A high impedance value is denoted by z.

Example:

12'h13x // 13x is a 12-bit hex number; 4 least significant bits unknown

6'hx //x is a 6-bit hex number

32'bz //z is a 32-bit high impedance number

Negative numbers

· Negative numbers can be specified by putting a minus sign before the size for a constant number.

· Negative numbers are stored as **compliment two** and the minus sign must be included before the specification of size. [It is illegal to have a minus sign between <base format> and <number>].

Example:

-10'd5 // 10 bit negative number stored as 2's complement of 5.

-6' d3 // 6-bit negative number stored as 2's complement of 3

4'd -2 // Illegal specification

Underscore characters and question marks:

An underscore character "-" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.

A question mark "?" is the Verilog HDL alternative for z when used in a number. The ?is used to enhance readability in the casex and casez statements.

Example:

12'b1111_0000_1010// Use of underline characters for readability

4'b10?? // Equivalent of a 4'bl0zz

Strings

· A string is a sequence of characters that are enclosed by double quotes.

· The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines.

Example:

"Hello Verilog World" // is a string

'a / b" // is a string

Keywords and Identifiers:

· Keywords are **special identifiers** reserved to define the language constructs. · Keywords are in lowercase.

Example: module, wire, assign, endmodule etc.,

Identifiers are names given to objects.

· Identifiers are Made up of alphanumeric characters, the underscore (_) and the dollar sign($) · Case sensitive.

· Start with an alphabetic character or an underscore.

· They cannot start with a number or a $ sign

Example:

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

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

Escaped Identifiers

· *Escaped identifiers* begin with the backslash (\ ) character and end with whitespace (space, tab, or newline).

· All characters between backslash and whitespace are processed literally. · Any printable ASCII character can be included in escaped identifiers.

Example:

```
\a+b-c
\**my_name**
```

Data Types:

A data type in Verilog is designed to represent the data storage and transmission. The 10 data types used in Verilog are –

1. Value Set

2. Nets

3. Registers

4. Vectors

5. Numbers –Integer and Real **Value Set**

6. Simulation Time 7. Arrays

8. Memories

9. Parameters

10. Strings

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

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

In addition to logic values, strength levels are often used to resolve which value should appear on a net or gate output.

There are two types of strengths: drive strengths and charge strengths.

The drive strength types are - **supply**, **strong**, **pull**, **weak**, and **highz** strengths

The charge strength types are -**large**, **medium** and **small** strengths

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	Driving	
Large	Storage	
Weak	Driving	

Medium	Storage	
small	Storage	
highz	high impedance	Weakest

When signals combine, their strengths and values shall determine the strength and value of the resulting signal in accordance with the principle.

1. If two signals of unequal strengths are driven on a wire, the stronger signal prevails

Example: If two signals of strength **strongl** and **weak0** drive

2. If two signals of equal strengths are driven on a wire, the result is unknown.

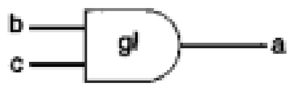
Example:

If two signals of strength **strong l** and **strong 0** conflicts, the result is an **X.**

Nets:

The nets variables represent the physical connection between structural entities. They do not store values. They have the value of their drivers.

Example:



In the Figure net *a* is connected to the output of *and* gate *gl.* Net *a*will continuously assume the value computed at the input of gate *gl,* which is *b* & c

5 $_{age}$1 P

· Nets are declared with the keyword **wire**.

· A wire represents a physical wire in a circuit and is used to connect gates or modules. A wire does not store its value.

· They are one-bit values except in vectors.

· The default value of a net is **Z.**

· ***net*s** not a keyword but represents a class of data types such as w i r e , wand, wor, tri, triand, trior, trireg, etc.

Example:

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

wire b,c; // **Declare two wires b, c for the above circuit**

Registers:

· Registers are data storage elements. They retain value until another value is placed onto them. · *Register* is a variable that can hold a value.

· Unlike a net, a register does not need a driver.

· Verilog registers do not need a clock.

· Values of registers can be changed anytime in a simulation byassigning 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:

reg reset;

initial

begin

reset = l'bl;

#l00 reset = l'b0;

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 (l-bit).

Example:

wire a; // scalar net variable, default

wire [7:0] bus; // 8-bit bus

The left number in the squared brackets is always the most significant bit of the vector.

Integer, Real and Time Register Data Types

Integer

· An integer is a variable data type that stores the value until next assignment. · Integers are declared by the keyword **integer.** The width for an integershould be at least 32 bits. · **Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities**.

Real

· Real number constants and real register data types are declared with the keyword **real**. · They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 X 10^6).

· Real numbers cannot have a range declaration and their default value is 0. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

Time

· Verilog simulation is done with respect to *simulation time*.

· A time variable is declared with the keyword **time**.

· The width for time register data types is at least 64 bits.

Arrays

7 $_{age}$1

· Arrays are used in Verilog for reg, integer, **time** and *vector* register data types. Arrays are not used for real variables.

· Each element of the array can be used as a scalar or vector net.

· **Arrays are accessed by**

· **<array-name> [<subscript>l.**

· Multidimensional arrays are not permitted in Verilog.

· **The difference between a vector and an array is - vector is a single element that is n-bits wide but arrays are multiple elements that are l-bit or n-bits wide.**

Example:

integer count[0:7]; //an array of 8 count variables

Memories

· In digital simulation, register files, RAMS, and ROMs are modeled .

· Memories are modeled in Verilog as one dimensional array of registers. · Each element of the array is known as a word.

· Each word can be one or more bits.

· Depth of memory should be declared by specifying a range following the memory identifier Example: the memory **reg** [5:0]

Parameters

· Constants in Verilog are defined in a module by the keyword **parameter**. · Parameters cannot be used as variables.

· Module instances can be altered by changing the value of a parameter.

Example:

parameter port-id = 5; //Defines a constant port-id

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

Strings

· In **Verilog**, **string** literals are just packed arrays of bits (or a bit-vector).

· Strings are the characters stored in reg data type.

· The width of the register variables must be large enough to hold the string. · Each character in the string takes up 8 bits (1 byte).

· If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros.

· If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string.

Example:

reg [8*18:1] string-value

//Declare a variable that is l8 bytes wide, each 8 bit wide.

System Tasks

Verilog contains the pre-defined system tasks and functions which provide common operations .All system tasks appear in the form ***$<keyword>.*** Operations such as displaying onthe screen, monitoring values, stopping, and finishing are done by system tasks.

Task	Function
To display on screen	**$display**
For monitoring values	**$monitor**
Stopping and finishing simulation	**$time**
Terminate simulation	**$finish**

Compiler Directives

All compiler directives are defined by using the ' <***keyword***>construct. Compiler directives begin with **"`"** an accent grave**.** The compiler directives tell the compiler the method of processing its input. Some of the complier directives are

1. `define

2. `include

The ***'define* directive** is used to define text macros in Verilog. The defined constants or text macros are used in the Verilog code by preceding them with a ' (back tick).

Example:

//define a text macro that defines default word size

//Used as 'WORD-SIZE in the code

'define WORD-SIZE 32

'define WORD-REG reg [31:0]

The ***' include directive*** includes the entire contents of Verilog source file in another Verilog file during compilation. It is used to include header files, which contain global or commonly used definitions

Eg:

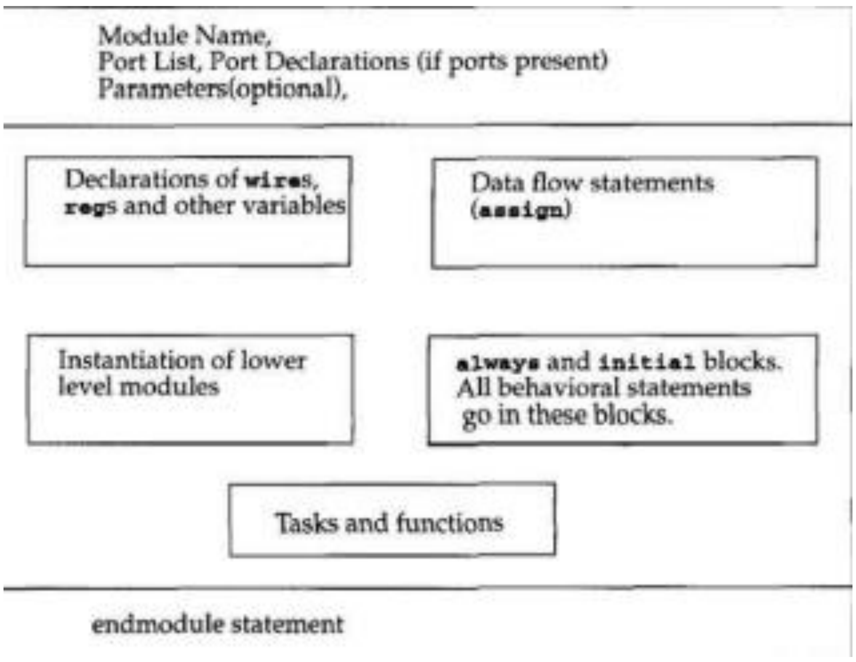
// Include the file header.v, which contains declarations in the

// main verilog file design.v.

'include header.v

4. Modules and Ports

A module is a block of Verilog code that implements certain functionality. The *module name, port list, port declaration,* and optional *parameters* must appear first in a module definition. *Port list* and *port declarations* are present only if the module has a ports to interact with the external environment. The block diagram of components in a Verilog module is as shown below.


Module Name,
Port List, Port Declarations (if ports present)
Parameters(optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

***The five components within a module are –**

1. variable declarations,

2. dataflow statements,

3. instantiation of lower

modules,

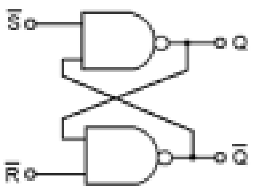
4. behavioral blocks and

5. 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. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

<u>Example:</u>

Consider the S R latch as shown below.



// Module name and port list // SR-latch module

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

The *SR latch* and its stimulus can be modeled as shown in Example

```
module SR-latch (Q, Qbar, Sbar, Rbar) ; //Port declarations
output Q, Qbar;
input Sbar, Rbar;
// Instantiate lower-level modules
nandnl (Q, Sbar, Qbar) ;
nand n2 ( Qbar, Rbar, Q) ;
// endmodule statement
endmodule
```

The S R latch stimulus can be modeled as shown

```
// Module name and port list(No ports are there)
// Stimulus module
module Top;
// Declarations of wire, req, and other variables
wire q, qbar;
reg set, reset;
// Instantiate lower-level modules
// In this case, instantiate SR-latch
// Feed inverted set and reset signals to the SR latch
SR-latch ml(q, qbar, -set, -reset);
// Behavioral block, initial
initial
begin
```

```
$monitor($time, " set = %b, reset= %b, q= %b\nU,set,reset,q);
set= 0; reset = 0;
#5 reset = 1;
#5 reset = 0;
#5 set = 1;
end
// endmodule statement
endmodule
```

[Notice the following characteristics about the modules defined above.

In the SR latch definition above , notice that all components described in Figure need not be present in a module. We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial).

However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.

Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.]

Ports:

Port is an essential component of the **Verilog** module. **Ports** are used to communicate for a module with the external world through input and output. Ports define the interface of a Verilog module to the outside world.

Example:

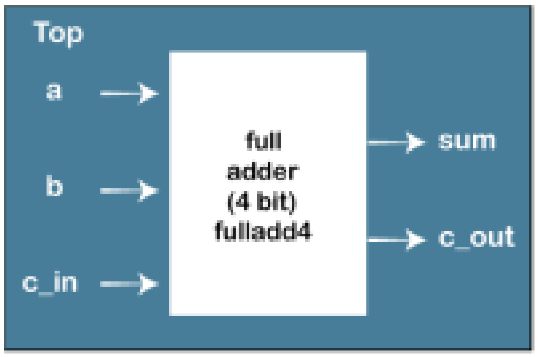
The input output pins of IC are its ports

List of Ports

A module definition contains an optional list of ports.

Example:

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 Figure.



I/O Ports for Top and Full Adder

The **module fulladd4** takes input on ports **a, b, and c-in** and produces an output on ports **sum and c-out.** Thus, module fulladd4 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. Thus, it does not have a list of ports.

The module names and port lists for both module declarations in Verilog are as shown below.
Example:

module fulladd4(sum, c-out, a, b, c-in); //module with a list ofports **module Top;** // No list of ports, top-level module in simulation

<u>Port Declaration</u>

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

Example:

Input list of ports/terminals ; Input s,r; Input[7:0] s,r;

Output list of ports/terminals ;Output y,z; Output [3:0] y,z

Inout list of ports/terminals ;Inout a,b;Inpit[15:0] a,b;

Example1 : 4 bit full adder

module fulladd4(sum, c-out, a, b, c-in);

//Begin port declarations section

output [3 : 0] sum; // 4 bit full adder hence the vector [3: 0] which is 4 bit

output c-out;

input [3:0] a, b;

input c-in;

.....

//End port declarations section

Endmodule

All ports declarations are implicitly declared as **w i r e** in Verilog. Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.

However, if output ports hold their value, they must be declared as **reg.**

Example2 : D Flip- flop

module DFF(q, d, clk, reset) ;

output q;

reg q; // **Output port q holds value; therefore it is declared as reg.**

input d, clk, reset;

...

...

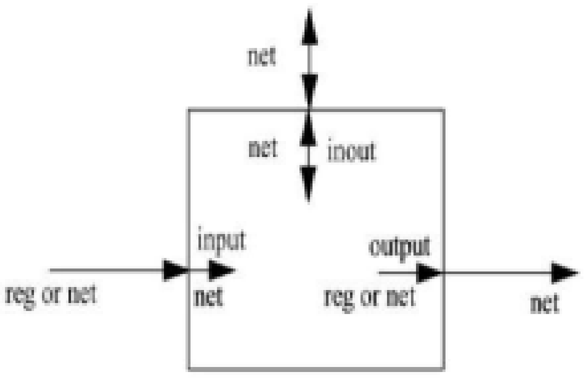
endmodule

All port declarations can be done in module declaration as shown below.

module full_add4**(output** reg[3:0] sum; **output** reg cout; **input** [3:0] a,b; **input** cin); **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 block diagram of the rules that are governing port connections when modules are instantiated within other modules are summarized below.



Outputs

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***.

Internally, outputs ports can be of the type **reg or *net.*** Externally, outputs must always be connected to a ***net.***

Inouts

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

Width matching

It is legal to connect internal and external items of different sizes when making inter-module port connections. A warning is issued when the widths do not match.

Unconnected ports

Verilog allows ports to remain unconnected. A port can remain unconnected by instantiating a module as shown below.

fulladd4 fa0 (SUM, , A, B, C-IN); // Output port c-out is unconnected.

Example of illegal port connection

module Top;

//Declare connection variables

reg [3:O]A,B;

reg C-IN;

reg [3:0] SUM; // this is the output. The output should be always net or wire. //Hence illegal

wire C-OUT;

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. They are

· Connecting by ordered **list**

· Connecting ports **by** name

Connecting by ordered list

In this method 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:

Consider 4 bit full adder with Top module. The Verilog code is

```
module Top;
//Declare connection variables reg [3:O]A,B;
reg C-IN;
wire [3:0] SUM;
wire C-OUT;
.....
<stimulus>
....
Endmodule
```

```
module fulladd4 (sum, c-out, a, b, c-in) ; output [3 : 01 sum;
output c-out;
input [3:0] a, b;
input c-in;
...
<module internals>
...
endmodule
```

consider the module ***fulladd4*** To connect signals in module Top by ordered list, the external signals SUM, C-OUT, ***A,*** B, and *CJN* appear in exactly the same order as the ports sum, c-out, a, b, and c-in in module definition of ***fulladd4.***

<u>Connecting ports by name</u>

Verilog provides the capability to connect external signals to ports by the port names.
Example: consider Full – adder 4

module fulladd4 (sum, c-out, a, b, c-in) ;

output *[3* : 01 **sum;**

output c-cout;

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

input c-in;

...

<module internals>

...

endmodule

module Top;

//Declare connection variables

reg [3:O]A,B;

reg C-IN;

wire [3:0] SUM;

wire C-OUT;

.....

<stimulus>

....

Endmodule

fulladd4 fa1 (.c-out(C-OUT), .sum(SUM), .b(B), .c-in(C-IN), .a(A) ,);

<u>Hierarchical Names</u>

Every module instance, signal, or variable is defined with an identifier. Hierarchical name referencing allows to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by **dots (" .").** The advantages of mentioning the hierarchical name is any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier.

The top-level module is called the root module because it is not instantiate. To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier anywhere.

Example:

Consider S R latch. The design hierarchy is shown in Figure

5 _{age}2

n1

(nand) .

Stimulus

(Root level)

m1

(SR_latch)

n2

(nand)

q, qbar

set, reset

(variables)

Q, Qbar

S,R

(signals)

For this simulation, stimulus is the top level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates **ml**, which is a module of type SR-latch. The module **m1** instantiates nand gates nl and n2. Q, Qbar, S, and Rare port signals in instance ml. Hierarchical name referencing assigns a unique name to each identifier

Example:

stimulus

stimulus.qbar stimulus.reset stimulus.ml.Q stimulus.ml.S stimulus.nl

stimulus.q

stimulus.set

stimu1us.ml.Qbar stimu1us.ml.R stimulus.n2

Each identifier in the design is uniquely specified by its hierarchical path name

Stimulus

(Root level)

q, qbar

m1

(SR_latch)

n1

(nand)

n2

(nand)

Q, Qbar S,R

(signals)

set, reset (variables)

Behavioral Modeling

In behavioural modelling the digital design is carried out in terms of algorithm and its performance. **Structured Procedures**

Structured procedures provide a means of modeling blocks of procedural statements. There are two structured procedure statements in Verilog: ***always*** **and** ***initial***

initial Statement

All statements inside an ***initial*** statement constitute an ***initial*** block. An ***initial*** block starts at time 0, executes exactly once during a simulation and then does not execute again. If there are multiple ***initial*** blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks

Example:

module stimulus;

reg x,y, **a,b, m;**

initial

m = l'b0 //single statement; does not need to be grouped

initial

begin

#5 c //multiple statements; need to be grouped

#25 b = l'b0;

end

initial

begin

#l0 ***X*** = l'b0;

#25 y = l'bl;

end

initial

#50 $finish; endmodule

Time	**Statement executed**
0	m = l'b0 ;
5	a = l'b0 ;
10	x = l'b0
30	b = l'b0;
35	y = l'bl;
50	$finish;

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

Example:

Consider a clock generator module that toggles the clock signal every half cycle **module clock-gen;**

reg clock

//Initialize clock at time zero

initial

clock = l'b0;

//Toggle clock every half-cycle (time period = 20)

always

#l0 clock = ~clock;

initial

#l000 $finish;

endmodule

In the above Example the **always** statement starts at time 0 and executes the statement *clock* = *~clock* every 10 time units. Notice that the initialization of *clock* has to be done inside a separate **initial** statement. Also, the simulation must be halted inside an **initial** statement. If there is no **$ stop** or **$finish** statement to halt the simulation, the clock generator will run forever.

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. The syntax for the simplest form of procedural assignment is **<assignment> : : = < 1value> = <expression>**

There are two types of procedural assignment statements:

blocking and nonblocking.

Blocking assignments

Verilog supports blocking and non-blocking assignments statements within the always block with their different behaviours.

✓ Blocking assignment statements are executed in the order they are specified in a sequential block.

✓ A blocking assignment will not block execution of statements that follow in a parallel block. ✓ The ‘ = ‘ operator is used to specify the Blocking assignments .

Example:

reg x, y, *z;*

reg [15: 0] reg-a, reg-b;

integer count;

initial

begin

x=0; y=l; z = **1; //Scalar assignments**

count = **0; //Assignment to integer variables**

reg-a = **16'b0; reg-b** = **reg-a; //initialize vectors**

#l5 reg-a[2] = **lrbl; //Bit select assignment with delay**

#l0 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**

✓ In the above example the statement y = 1 is executed only after ***x*** = *0* is executed. ✓ The behaviour in a particular block is sequential in a begin-end block

✓ The statement count = count + 1 is executed last.

The simulation times at which the statements are executed are as follows:

✓ All statements ***x*** = *0* through reg-b = reg-a are executed at time 0

✓ Statement regal21 = *0* at time = 15

✓ Statement reg-b[15:13] = **{x,** y, *z)* at time = 25

✓ Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Nonblocking Assignments

Non-blocking assignment statements are allowed to be scheduled without blocking the execution of the following statements and is specified by a (<=) symbol.

Example:

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 reg-a[2] **<=** #l5 l'bl; //Bit select assignment with delay

reg-b[15: 13] **<=** #l0 **{x,** y, z***);*** //Assign result of concatenation

//to part select of a vector

count <= count + 1; //Assignment to an integer (increment)

end

The statements **x** = 0 through reg-b = reg-a are executed sequentially at time 0. The three nonblocking assignments are processed at the same simulation time. *1.* **reg-a[2] = 0** is scheduled to execute after 15 units (i.e., time = 15)

2. **reg-b[15:13]** = **{x**, y, z} is scheduled to execute after 10 time units (i.e., time = 10) 3. **count = count + 1** is scheduled to be executed without any delay (i.e., time = 0)

<u>Application of nonblocking assignments</u>

They are used as a method to model several concurrent data transfers that take place after a common event.

Consider the following example where three concurrent data transfers take place at the positive edge of clock.

<u>Example:</u>

always @(posedge clock)

begin

regl <= **#l** i n l ;

reg2 <= @(negedge clock) in2 **A** in3;

reg3 <= #l regl; //The old value of regl

end

At each positive edge of clock, the following sequence takes place for the nonblocking assignments.

1. A read operation is performed on each right-hand-side variable, inl, in2, in3 and regl, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.

2. The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg l after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit. **3.** The write operations are executed at the scheduled time steps. The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values. For example, note that reg3 is assigned the old value

Thus, the final values of regl, reg2, and reg3 are not dependent on the order in which the assignments are processed.

To understand the read and write operations further, consider the following example, which is intended to swap the values of registers a and b at each positive edge of clock, using two concurrent **always** blocks

Example:

//Illustration 1: Two concurrent *always* blocks with blocking statements always @(posedge clock)

a = b;

always @(posedge clock)

b = a;

//Illustration 2: Two concurrent *always* blocks with nonblocking statements always @(posedge clock)

a <= b;

always @(posedge clock)

b <= a;

In the above example, in *illustration* 1, there is a race condition when blocking statements are used. Either *a* = b would be executed before b = *a,* or vice versa, depending on the simulator implementation. Thus, values of registers *a* and b will not be swapped. Instead, both registers will get the same value (previous value of *a* or *b),* based on the Verilog simulator implementation.

However, nonblocking statements used in *illustration* 2 eliminate the race condition. At the positive edge of clock, the values of all right-hand-side variables are *"read,"* and the right-hand-side expressions are evaluated and stored in *temporary* variables. During the write operation, the values stored in the temporary variables are assigned to the left-hand-side variables. Separating the read and write operations ensures that the values of registers *a* and b are swapped correctly, regardless of the order in which the write operations are performed.

Conditional Statements:

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords i f and **else** are used for conditional statements.

There are three types of conditional statements.

1. if statement

2. if – else statement

3. if-else- if statement.

1. In if statement - Statement executes or does not execute.

Syntax:

if (<expression>) true-statement ;

Example:

If (!lock) buffer = data;

If (enab1e) out = in;

2. In if – else statement -either true-statement or false-statement is evaluated **Syntax :**

if (<expression>) true-statement ; else false-statement ;

Example:

```
if (reset)
begin
 dff <= 0;
end
else
begin
 dff <= din;
end
```

3. In if-else- if statement - Choice of multiple statements. Only one is executed. **Syntax:**

if (<expressionl>) true-statement1 ;

else if (<expression2>) true-statement2 ;

else if (cexpression3>) true-statement3 ;

else default-statement ;

Example:

```
if (alu-control = = 0)
l y=x + z;
else if(a1u-control = = 1)
y=x - z;
else if (a1u-control = = 2)
y=x*z;
else
$display ("Invalid ALU control signal");
```

Multiway Branching

The nested **if-else-if** can become unwieldy if there are too many alternatives. **A** shortcut to achieve the same result is to use the **case** statement.

The keywords **case, endcase,** and **default** are used in the *case* statement

Syntax:

```
case ( expression)
alternativel: statement l;
alternative2: statement 2;
alternative3: statement 3;
...
...
Default: default-statement;
endcase
```

1. Each of *statement 1, statement2 ..., default-statement* can be a single statement or a block of multiple statements.

2. The expression is compared to the alternatives in the order they are written. 3. For the first alternative that matches, the corresponding statement or block is executed. 4. If none of the alternatives match, the *default-statement* is executed.

Example:

```
reg [1:0] alu-control;
...
...
case (alu-control)
2'd0 : y = x + z;
2'dl : y = x - z;
2'd2 : y = x* z;
default : $display("Inva1id ALU control signal");
endcase
```

4 : 1 Multiplexer

```
module mux4-to-l (out, i0, il, i2, i3, sl, s0);
output out;
input i0, il, i2, i3;
input sl, S0;
reg out;
```

```
always @(sl or S0 or i0 or il or i2 or i3)
case ({sl, s0))
2'd0 : out = i0;
2'dl : out = il;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

<u>1 : 4 Demultiplexer</u>

```
module demultiplexerl-to-4 (out0, outl, out2, out3, in, sl, s0);
output out0, outl, out2, out3;
reg out0, outl, out2, out3;
input in;
input sl, s0;
ilways @(sl or s0 or in)
case ({sl, s0))
2 'b00 :
begin: out0 = in; outl = l'bz; out2 = l'bz; out3 = l'bz; end
2'b0l : begin out0 = l'bz; outl = in; out2 = l'bz; out3 = l'bz; end
2'bl0 : begin out0 = l'bz; outl = l'bz; out2 = in; out3 = l'bz; end
2'bll : begin out0 = l'bz; outl = l'bz; out2 = l'bz; out3 = in; end
2'bx0, 2'bxl, 2'bxz, 2'bxx, 2'bOx, 2'blx, 2'bzx :
begin
out0 = l'bx; outl = l'bx; out2 = l'bx; out3 = l'bx;
end
2'bz0, 2'bzl, 2'bzz, 2'bOz, 2'blz :
begin
ou0 = l'bz; outl = l'bz; out2 = l'bz; out3 = l'bz;
end
```

default: $display("Unspecified control signals");

end case

endmodule

Account for unknown signals on select.

- If any select signal is then outputs are x.
- If any select signal is z, outputs are z.
- If one is x and the other is z, x gets higher priority.

casex, casez Keywords

There are two variations of the **case** statement. They are denoted by keywords, **casex** and **casez.**

casez treats all **z** values in the case alternatives or the case expression as don't cares. All bit positions with **z** can also represented by ? in that position.

casex treats all **x** and *z* values in the case item or the case expression as don't cares.

The use of **casex** and **casez** allows comparison of only non-X or **-z** positions in the case expression and the case alternatives.

Example:

reg [3 : 0] encoding;

integer state;

casex (encoding) //logic value X represents a don't care bit

4'blxxx : next-state = 3;

4'bxlxx : next-state = 2;

4'bxxlx : next-state = 1;

4'bxxxl : next-state = 0;

default : next-state = 0;

endcase

Thus, an input encoding = 4'bl0xz would cause next-state = **3** to be executed. **Loops**

There are four types of looping statements

- while
- for
- repeat and
- Forever

<u>While Loop</u>

The keyword **while** is used to specify this loop. The ***while*** loop executes until the ***while expression*** **becomes false.** If multiple statements are to be executed in the loop, they must be grouped typically using keywords **begin** and **end.**

<u>Example:</u>

<u>Example 1: Increment count from 0 to 127. Exit at count 128.</u>

```
integer count;

initial

begin

count = 0;

while (count < 128) //Execute loop till count is 127. Exit at count 128

begin

 $display ( "Count = %d" , count) ;

count = count + 1;

end

end
```

<u>Example 2: Find the first bit with a value 1 in flag</u>

```
'define TRUE l'bl';

'define FALSE l'b0;

reg [15:0] flag;

integer i //integer to keep count

reg continue;

initial

begin

flag = 16'b 0010~0000~0000~0000;

i = 0;

continue = 'TRUE;

while((i < 16) && continue ) //Multiple conditions using operators. begin

if ( flag [i] )

begin

$display("Encountered a TRUE bit at element number %d", i);
```

continue = 'FALSE;

end

i=i+l;

end

end

For Loop

The keyword **for** is used to specify this loop. The **for** loop contains three parts: · An initial condition

· A check to see if the terminating condition is true

· A procedural assignment to change value of the control variable **Example:**

integer count;

initial

for (count=O; count < 128; count = count + 1)

$display("Count = %do, count);

for loops can also be used to initialize an array or memory.

Example:

'define MAX-STATES 32

integer state [O: 'MAX-STATES-l]; //Integer array state with elements 0: 31 integer i;

initial

begin

for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0 state[il = 0;

for(i = 1; i < 32; **i** = i + 2) //initialize all odd locations with 1 state [i l = 1;

end

Repeat Loop

The keyword **repeat** is used for this loop. The **repeat** construct executes the loop a fixed number of times. A **repeat** construct cannot be used to loop on a general logical expression. A **repeat c**onstruct must contain a number, which can be a constant, a variable or a signal value. **Example:**

//Example1 : incrernent and display count from 0 to 127

integer count;

initial

begin

```
count = 0;
repeat (128)
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

// Example 2 : Data buffer module

```
//After it receives a data-start signal. Reads data for next 8 cycles.
parameter cycles = 8;
input data-start;
input [15:01 data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock)
begin
if(data-start) //data start signal is true
begin
i = 0;
repeat(cyc1es) //Store data at the posedge of next 8 clock cycles
begin
@(posedge clock) buffer[ i ] = data; //waits till next posedge to latch data
i=i+l;
end
end
end
endmodule
```

Forever loop

The keyword **forever** is used to express this loop. The loop does not contain any expression and executes forever until the **$finish** task is encountered. The loop is equivalent to a **while** loop with an expression that always evaluates to true, A forever loop can be exited by use of the **disable** statement.

<u>//Example 1: Clock generation</u>

//Use forever loop instead of always block

reg clock;

initial

begin

clock = l'bO;

forever #l0 clock = -clock; //Clock with period of 20 unit

end

<u>//Example</u> <u>2</u>: **<u>Synchronize two register values at every positive edge of c</u>lock reg clock;**

reg x y;

initial

forever @(posedge clock) x = y;

<u>Dataflow Modeling</u>

Dataflow modeling describes hardware in terms of the flow of data from input to output.

<u>Continuous Assignment</u>

The continuous assignment statement is the main construct of dataflow modeling and is used to assign value to the net. It starts with the keyword **assign**.

General syntax is:

<continuous-assign> : : = assign <drive-strength>?<delay>? <list-of-assignments>;

Continuous assignments have the following characteristics.

· Continuous assignments are always active. That is the LHS net value changes as soon as the value of any operand in the RHS changes.

· The LHS of an assignment should be either scalar or vector nets or a concatenation of both. Registers are not applicable on the LHS.

· The RHS of the assignment can be register, net, or function calls of scalar or vector type. · Delays can be specified.

<u>Examples of Continuous Assignment</u>

1. // Continuous assign. out is a net. **il** and **i2** are nets

assign out = il & i2;

2. // Continuous assign for vector nets. *addr* is a 16-bit vector net

assign addr[l5:0] = addrl_bits[l5:0] ^ addr2_bits[l5:0];

3. // Concatenation. Left-hand side is a concatenation of a scalar

assign {c-out, sum[3:0]) = a[3:0] + b[3:01 + c-in;

<u>Implicit Continuous Assignment</u>

Instead of declaring a net and then writing a continuous assignment on the net, a continuous assignment can be placed on a net when it is declared.

Example:

//Regular continuous assignment

wire out;

assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment

wire out = in1 & in2;

<u>Delays</u>

Three ways of specifying delays in continuous assignment statements are

1. Regular assignment delay,

2. implicit continuous assignment delay, and

3. net declaration delay.

<u>Regular Assignment Delay</u>

A delay value in the continuous assignment statement is assigned first. The delay value is specified after the keyword **assign.**

Example:

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

· Any change in values of *in1* or *in2* will result in a delay of 10 time units before recomputation of the expression *in1* & *in2,* and the result will be assigned to *out.*

· If *in1* or *in2* changes value again before 10 time units when the result propagates to *out,* the values of *in1* and *in2* at the time of recomputation are considered.

This property is called **inertial delay.** An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

In1

In2

out

Time

10 $^{20\ 30}$ $_{60}$ 70 $_{80}$ 85

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.

Example:

wire #l0 out = in1 & in2;

//same as

wire out;

assign #l0 out = in1 & in2;

Expressions, Operators, and Operands:

Expressions:

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

· a&b

· addr1[20:17] + addr2[20:17]

· in1 | in2

Operands:

Operands can be any one of the data types. They can be *constants, integers, real numbers, nets, registers, times, bit-select etc.,*

Example:

1. integer count, final-count;

final-count = count + 1

2. real a, b, c;

c = a - b; etc.,

Operators

Verilog has Ten different types of operators. They are

1. Arithmetic
2. Logical
3. Relational
4. Equality

5. Bitwise logical

1. Arithmetic Operators

There are two types of arithmetic operators: · binary and

· Unary.

6. Reduction 7. Shift operators 8. Concatenation 9. Replication 10. Conditional

Binary arithmetic operators are

1. Multiply (*),
2. divide (/),
3. add (+),
4. subtract (-) and
5. modulus (%).

Example:

If A = 4'b00ll; B = 4'b0l00; D = 6; E = 4; determine

a) A * B (Ans: 4'bll00)

b) D / E (Ans: Evaluates to 1)

c) A + B (Ans: 4'b0lll)

d) B – A (Ans: 4'b000l)

If any operand bit has a value **x,** then the result of the entire expression is **x. Example:**

in1 = 4'bl0lx;

in2 = 4'bl0l0;

sum = in1 + in2; // sum will be evaluated to the value 4'bx

Modulus operators produce the *remainder* from the division of two numbers. Example:

13 % 3 // Evaluates to 1

16 % 4 // Evaluates to 0

-7 % 2 // Evaluates to -1, takes sign of the first operand

7 % -2 // Evaluates to +l, takes sign of the first operand

Unary operators

The operators + and - can also work as *unary* operators. They are used to specify the positive or negative sign of the operand

Example:

-4 // Negative 4

+5 // Positive 5

2. <u>Logical Operators</u>

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

Logical operators follow the following conditions –

➢ Logical operators always evaluate to a l-bit value, 0 (false), **1** (true), or ***X***

➢ 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).

Example:

I. A = 4’b0011; B = 4’b0000;

a) A && B // Evaluates to 0.

b) A || B // Evaluates to 1.

c) ! A / / Evaluates to 0.

d) !B// Evaluates to 1

II. A = 2'b0x; B = 2'bl0;

A && B // Evaluates to X.

III. (a = = 2) && (b = = 3) // Evaluates to 1 if both a = = 2 and b = = **3** are true. // Evaluates to 0 if either is false.

3. <u>Relational Operators</u>

Relational operators are -

➢ greater-than **(>**),

➢ less-than **(<)**

➢ greater-than-or-equal-to **(>=)** and

➢ less-than-or-equal-to **(<=)**

<u>Example:</u>

If A=4, B=3

X = 4'b1010, Y = 4'b1101, Z = 4'blxxx then evaluate

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

b) A > B // Evaluates to a logical 1

c) Y >= X // Evaluates to a logical 1

d) Y < Z // Evaluates to an X

4. <u>Equality Operators</u>

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. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.

Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if **X** or **z** in a or b	**0, 1, X**
a ! = b	a not equal to b, result unknown if **X** or **z** in a or b	**0, 1, X**
a = = = b	a equal to b, including **X** and **z**	**0, 1**
a!== b	a not equal to b, including **X** and **z**	**0, 1**

Problems:

A = 4 , B = 3

X = 4’b1010 ; Y = 4’b1011;

Z = 4’b1xxz ; M = 4’b1xxz ; N = 4’b1xxx;

A == **B** // **Results in logical** 0

X != **Y** // **Results in logical** 1

X == **Z** // **Results in X**

Z === **M //~esults in logical l (all bits match, including X and** z)

z === N **//~esults in logical** 0 **(least significant bit does not match)**

M !== **N** // **Results in logical** 1

5. <u>Bitwise Operators</u>

Bitwise operators are

- ***negation*** **(~),**
- ***and(&),***

➢ ***or* (|)**

➢ **xor *(^)***

➢ ***xnor (^ ~ , ~^)).***

Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit extended with zeros to match the length of the longer operand. **Example:**

X = 4' b1010; Y = 4' b1101; Z = 4' b10x1;

~X // Negation. Result is 4'b0l0l

X & Y // Bitwise and. Result is 4'bl000

X | Y // Bitwise or. Result is 4'bllll

X ^ Y // Bitwise xor. Result is 4'b0lll

X ^~ Y // Bitwise xnor. Result is 4'bl000

X & Z // Result is 4'bl0x0

6. <u>Reduction Operators</u>

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 l-bit result.

Example:

X = 4' b1010

&X //Equivalent to 1 **& 0** & **1** & **0. Results in l'b0**

|X //equivalent to 1 | 0 **| 1 | 0. Results in l'bl**

^ X //Equivalent to 1 ^ **0 A 1** ^ **0. Results in l'b0**

7. <u>Shift Operators</u>

Shift operators are

➢ *right shift* (>>) and

➢ *left shift* (<<).

These operators shift a vector operand to the right or the left by a specified number of bits

Example:

X = 4’b 1100

Y = X >> l; //Y is 4'b0ll0.Shift right 1 bit.0 is filled in MSB position

Y = X << 1; //Y is 4'bl000.Shift left 1 bit.0 filled in LSB position.

Y = X << 2; //Y is 4'b0000.Shift left 2 bits.

8. Concatenation Operator

The *concatenation* operator ({,) provides a mechanism to append multiple operands. The operands must be sized. Concatenations are expressed as operands within braces, with commas separating the operands.

Example:

If A = l'bl, B = 2'b00, C = 2'b10, D = 3'bll0

Then what is

Y = {B , C} // Result Y is 4'b00l0

Y = {A , B , C , D , 3'b00l) // Result Y is 11'b10010110001

Y = {A , B[0], C[1] 1} // Result Y is 3'bl0l

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 (**{ }**). **Example:**

Let reg A;

reg [1:0] B, C;

reg [2:0] D;

A = l'bl; B = 2'b00; C = 2'bl0; D = 3'bll0;

Y = { 4{A) } // Result Y is 4'bllll

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

Y = { 4{A} , 2{B} , C } // Result Y is 10'b1111000010

10. Conditional Operator

The conditional operator(? :) takes three operands.

Usage: condition-expr ? true-expr : false-expr ;

The condition expression (condition-expr) is first evaluated.

➢ If the result is true (logical **1),**then the true-expr is evaluated.

➢ If the result is false (logical **0),** then the false-expr is evaluated.

➢ If the result is **X** (ambiguous), then both true-expr and false-expr are evaluated and their results are compared, bit by bit, to return for each bit position an **X** if the bits are different and the value of the bits if they are the same

Conditional operations can be nested. Each *true-expr* or *false-expr* can itself be a conditional operation.

Example:

Consider 4 – to – 1 multiplexer.

Let ***(A==3)*** and control are the two select signals of 4-to-1 multiplexer with n, m, y, *x* as the inputs and out as the output signal.

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

Example:

Write the Verilog code for 4-to-1 Multiplexer, Using Conditional Operators module multiplexer4-to-l (out, i0, il, i2, i3, sl, s0);

output out;

input i0, il, i2, i3;

input sl, s0;

assign out = sl ? (S0 ? i3 : i2) : (SO ? il : i0);

endmodule ;

11. **<u>Precedence of operators :</u>**

<u>Summary of operators in Verilog:</u>

Examples:

<u>Write Verilog code for 4-to-1 Multiplexer using data flow modelling</u>

<u>Method 1: using logic equation</u>

S1 S0

S1n ^{S0n}y0

i0

y1

i1

y2

i2

y3

D3

module mux4-to-l (out, i0, il, i2, i3, sl, s0);

output out;

input i0, il, i2, i3;

input sl, s0;

OUT

assign out = (~sl & ~s0 & i0) | (~sl & s0 & il) | (sl & ~s0 & i2) | (sl & s0 & i3) ;
endmodule.

<u>Method 2: Using Conditional Operators</u>

module multiplexer4-to-l (out, i0, il, i2, i3, sl, s0);

output out;

input i0, il, i2, i3;

input sl, s0;

assign out = sl ? (S0 ? i3 : i2) : (SO ? il : i0);

endmodule

<u>Write Verilog code for 4 Bit Full Adder using data flow modelling</u>

<u>Method 1: Using Dataflow Operators</u>

module fulladd4 (sum, c-out, x, y, c-in);

output [3:0] sum;

output c-out;

input [3: 0] x, y;

input c-in;

assign {c-out, sum) = x + y + c-in;

endmodule

<u>Method 2: full adder with carry look ahead</u>

module fulladd4(sum, c-out, a, b, c-in);

output [3:0] sum;

output c-out;

```
input [3:0] a,b;
input c-in;
wire p0,g0, pl, g1, , p2, g2, , p3, g3;
wire c4, c3, c2, c1;
assign p0 = a[0] ^ b[0],
p1= a[1] ^ b[1],
p2 = a[2] ^ b[2],
p3= a[3] ^ b[3];
assign g0 = a[0] & b[0],
g1 = a[l] & b[1],
g2 = a[2] & b[2],
 g3 = a[3] & b[3];
assign c1 = g0 | (p0 & c-in);
c2 = g1 | (p1 & g0) | (p1 & p0 & c-in);
c3 = g2 | (p2 & g1) | (p2 & p0 & c-in) | (p2 & p1 & p0 & c-in);
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p0 & c-in) | (p3 & p2 & p1 &   p0 & c-in);
assign sum[0] = p0 ^ c-in;
sum[1] = p1 ^ c1;
sum[2] = p2 ^ c2;
sum[3] = p3 ^ c3;
assign c-out = c4;
endmodule
```

UNIT 2:

Gate-Level Modelling

In gate level modelling the circuit is described in terms of gates (e.g., and, nand). **Gate Types**

A logic circuit can be designed by use of logic gates. In Verilog the basic logic gates are defined as **predefined primitives**. There are two types of basic gates:

· ***And / or*** **gates and**

· ***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 inbuilt gate primitives are

and or

not

nand nor xor

xnor

The logic symbol and truth table of these gates are as shown below: **and gate: nand gate**

I1	**I2**				
	and	**0**	**1**	**x**	**z**
	0	**0**	**0**	**0**	**0**
	1	**0**	**1**	**x**	**x**
	x	**0**	**x**	**x**	**x**
	z	**0**	**x**	**x**	**x**

I1	**I2**				
	nand	**0**	**1**	**x**	**z**
	0	**1**	**1**	**1**	**1**
	1	**1**	**0**	**x**	**x**
	x	**1**	**x**	**x**	**x**
	z	**1**	**x**	**x**	**x**

or gate nor gate

I1	**I2**				
	or	**0**	**1**	**x**	**z**
	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

I1	**I2**				
	nor	0	1	x	z
	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

xor gate xnor gate

I1	**I2**				
	xor	**0**	**1**	**x**	**z**
	0	**0**	**1**	**x**	**x**
	1	**1**	**0**	**x**	**x**
	x	**x**	**x**	**x**	**x**
	z	**x**	**x**	**x**	**x**

I1	I2				
	xnor	0	1	x	z
	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In this example, for all instances, *OUT* is connected to the output out, and *IN1* and IN2 are connected to the two inputs **il** and i2 of the gate primitives.

wire OUT, IN1, IN2;

// basic gate instantiations.

and al (OUT, IN1, IN2);

nand nal (OUT, IN1, IN2) ;

or orl (OUT, IN1, IN2);

nor nor1 (OUT, IN1, IN2) ;

xor xl (OUT, IN1, IN2) ;

xnor nxl (OUT, IN1, IN2) ;

The instance name does not need to be specified for primitives

Example:

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

<u>Buf / Not Gates</u>

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 –

· buf

· not

The symbols and truth tables are as shown below

buf	**in**	**out**
pg. 2	0	0
	1	1
	x	x
	z	z

not	**in**	**out**
	0	1
	1	0
	x	x
	z	x

bufif / notif

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

Bufif 1 bufif0

notif 1 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 below

Examples of instantiation of *bufif* and *notif* gates.

//Instantiation of bufif gates.

bufifl bl (out, in, ctrl) ;

bufif0 b0 (out, in, ctrl) ;

//Instantiation of notif gates

notifl nl (out, in, ctrl) ;

notif0 no (out, in, ctrl) ;

Example 1: Multiplexer 4: 1

<u>The Verilog code for 4 : 1 multiplexer is as shown below .</u>

The 1/O diagram and the truth table for the multiplexer are shown

The logic diagram for the multiplexer is shown below. **S1 S0**

S1n ^{S0n}y0

i0

y1

i1

y2

i2

y3

D3

The Verilog description for the multiplexer is shown below;

module mux4-to-l (out, i0, il, i2, i3, sl, S0); output out;

input i0, il, i2, i3;

input sl, S0;

wire s1n, son;

wire y0, yl, y2, y3;

not (sln, sl) ;

not (s0n, S0);

and (y0, i0, sln, s0n);

and (yl, il, sln, s0);

and (y2, i2, sl, s0n);

and (y3, i3, sl, S0);

or (out, y0, yl, y2, y3);

end module

This multiplexer can be tested with the stimulus as shown below.

Stimulus for Multiplexer

OUT

// Define the stimulus module (no ports)

module stimulus;

// Declare variables to be connected to inputs

reg IN0, IN1, IN2, IN3;

reg S1, S0;

// Declare output wire

wire OUTPUT;

// Instantiate the multiplexer

mux4-to-1 mymux (OUTPUT, INO, IN1, IN2, IN3, S1, SO) ; // define the stimulus module (no ports)

// Stimulate the inputs

initial

begin

// set input lines

IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;

#l $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n , IN0, IN1, IN2, IN3); // choose IN0

S1 = 0; S0 = 0;

#l $display("Sl = %b, S0= %b, OUTPUT = %b/n, S1, S0, OUTPUT); // choose IN1

S1 = 0; S0 = 1;

#l $display ("Sl = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT); // choose IN2

S1 = l; S0 = 0;

#l $display("Sl = %b, S0 = %b, OUTPUT = %b \n , S1, S0, OUTPUT); // choose IN3

S1 = l; SO = 1;

#l $display("Sl = %b, S0 = %b, OUTPUT = %b \n , S1, S0, OUTPUT); End

endmodule

The output of the simulation is displayed as shown below

IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0

Example 2: **1-bit full adder:**

The logic diagram for a l-bit full adder is shown below:

The Verilog description is given below:

```
module fulladd(sum, c-out, a, b, c-in) ;
output sum, c-out;
input a, b, c-in;
wire S0, C0, C1;
xor (S0, a, b);
and (C0, a, b);
xor (sum, S0, c-in) ;
and (C1, S0, c-in);
or (c-out, C0, Cl);
endmodule
```

Example 3: 4-bit full adder:

A0, A1, A2 and A3 are instances of the module full add. The Verilog code for the above circuit is as shown below.

```
module fulladd4(sum, c-out, x, y, c-in);
// I/O port declarations
output [3:01 sum;
output c-out;
input [3 : 01 a, b;
input c-in;
// Internal nets
wire cl, c2, c3;
// Instantiate four l-bit full adders.
fulladd A0 (sum[0l, cl, x[0], y[0], c-in);
fulladd A1 (sum[l], c2, x[1] , y[l] cl);
fulladd A2 (surn[2], c3, x[2], y[2], c2);
fulladd A3 (sum[3], c-out, x[3], b[3], c3);
endmodule
```

The design is checked by applying stimulus.

```
// Define the stimulus (top level module)
module stimulus;
reg [3:0] X, Y;
reg C-IN;
wire [3:0] SUM;
wire C-OUT;
// Instantiate the 4-bit full adder. call it FA1-4
fulladd4 FA1_4(SUM, C-OUT, X, Y, C-IN) ;
// Setu~ the monitorins for the sisnal values
initial
begin
$monitor($time," A= %b, B=%b, C-IN= %b, --- C-OUT= %b, SUM= %b\n’, A, B, C-IN, C-
OUT, SUM);
end
// Stimulate inputs
initial
begin
A = 4'd0; B = 4'd0; C-IN = l'b0;
#5 A = 4’d3, B = 4’d4;
#5 A = 4’d2, B = 4’d5;
#5 A = 4’d9, B = 4’d9;
```

```
#5 A = 4’d10, B = 4’d15;
A = 4'd10; B = 4'd5; C-IN = l'b1;
end
endmodule
```

The output of the simulation is shown below.

```
0 A= 0000, B=0000, C-IN= 0, --- C-OUT= 0, SUM= 0000
```

5 A= 0011, B=0100, C-IN= 0, --- C-OUT= 0, SUM= 0111

10 A= 0010, B=0101, C-IN= 0, --- C-OUT= 0, SUM= 0111 15 A= 1001, B=1001, C-IN= 0, --- C-OUT= 1, SUM= 0010 20 A= 1010, B=llll, C-IN= 0, --- C-OUT= 1, SUM= 1001

25 A= 1010, B=0101, C-IN= *l,* C-OUT= 1, SUM= 0000

Gate Delays

In real circuits, logic gates have delays associated with them. There are three types of delays from the inputs to the output of a primitive gate. · **Rise delay**

· **Fall delay**

· **Turn-off delay**

Rise delay

The time taken for the output of a gate to change from some value to 1 is called a rise delay.

Fall delay

The time taken for the output of a gate to change from some value to 0 is called a fall delay

Turn-off delay:

The time taken for the output of a gate to change from some value to high impedance (z) is called turn-off delay.

➢ 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.

Example:

Delay of delay-time for all transitions,

Syntax:

and #(delay-time) al (out, il, i2);

and #(5) al (out, il, i2); //Delay of 5 for all transitions

➢ If two delays are specified, they refer to the rise and fall delay values.

Example:

Rise and Fall Delay Specification.

Syntax

and # (rise-val, fall-val) a2 (out, il, i2)

and #(4,6) a2(out, il, i2); // Rise = 4, Fall = 6;

➢ If a1l three delays are specified, they refer to rise, fall, and turn-off delay values.
Example:

// Rise, Fall, and Turn-off Delay Specification

Syntax

Bufif0 #(rise-val, fall-val, turnoff-val) bl (out, in, control);

buf if0 # (3,4,5)b l (out,i n, control); // ~ise= 3, Fall = 4, Turn-off = 5

<u>Min / Typ / Max Values</u>

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.

<u>Min value</u>

The min value is the minimum 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. If no option is specified, the typical delay value is the default.

Examples of min, typ, and max value specification is given below.

// One delay

and #(4:5:6) al(out, il, i2);

// if +mindelays, delay= 4

// if +typdelays, delay= 5

// if +maxdelays, delay= 6

// Two delays

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

// 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)

// Three delays

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

// 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

Delay Example

Consider the equation

out = (a . b) + c

The logic diagram for the above equation is as shown below. The gate-level implementation is shown below.

module D (out, a, b, c);

output out;

input a,b,c;

wire e;

and # (5) a1 (e, a, b) ; //Delay of 5 on gate a1

or #(4) ol(out, e,c); //Delay of 4 on gate 01

endmodule

This module is tested by the stimulus file shown below

module stimulus

reg A, B, C;

wire OUT;

D dl(OUT, A, B, C);

// Stimulate the inputs. Finish the simulation at 40 time units

initial

begin

A= l'bO; B= l'bO; C= l'bO; #l0 A= l'bl; B= l'bl; C= l'bl; #l0 A= l'bl; B= l'bO; C= l'bO; #20 $finish;

end

endmodule

Operators in C

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. The C operators can be classified into the following categories

1. Arithmetic operators

2. Relational operators

3. Logical operators

4. Assignment operators

5. Increment and decrement operators

6. Bitwise operators

7. Special operators

1. Arithmetic operators

These operators are used for numerical calculations (or) to perform arithmetic operations. There are two types of mathematical operators: unary and binary. Unary operators perform an action with a single operand. Binary operators perform actions with two operands.

When both the operands in a single

arithmetic expression are integers, the

expression is called an integer expression.

Example:

If a = 14 and b= 4

a + b = 18;

a - b = 14;

a * b = 56;

a / b = 3;

a % b = 2;

Real arithmetic

An arithmetic operation involving only real operands is called real arithmetic. Since floating point values are rounded to the number of significant digits permissible, the final value of the correct result.

Example:

If x , y and z are floats, then

x = 6.0/7.0 = 0.857143

y = 1.0/3.0 = 0.333333

z = -2.0/3.0 = - 0.66667

Mixed mode arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number.

Example:

15/10.0 = 10. 5 where as

15/10 = 1.

2. Relational operators:

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	**Meaning of Operator**	**Example**
= =	Equal to	5 = = 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

3. Logical operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. These are used to combine 2 (or) more expressions logically. They are

logical **AND (&&)**

logical **OR (||)** and

logical **NOT (!)**

Operator	**Meaning**	**Example**
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.

||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c= =5) equals to 0.

4. Assignment operators

An assignment operator is used for assigning a value to a variable. When two quantities are compared depending on their relation the values are assigned.

Operator	**Example**	**Same as**
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example:

If a = 5, then

c = a; // c is 5

c += a; // c is 10

c -= a; // c is 5

c *= a; // c is 25

c /= a; // c is 5

c %= a; // c = 0

5. Increment and decrement operators

C programming has two operators

$\boxed{++}$

· increment and

$\boxed{--}$

· decrement to change the value of an operand by 1.

Increment $\underline{++}$ increases the value by 1 whereas decrement $\underline{--}$ decreases the value by 1. These two

operators are unary operators, meaning they only operate on a single operand.

Increment operator:

There are two types −

· pre increment

· post increment

If we place the increment operator before the operand, then it is **pre-increment**. Later on, the value is first incremented and next operation is performed on it.

Example:

z = ++a; // a= a+1

z = a

If a = 10, then z = ++ a gives

z= 11

a=11

If we place the increment operator after the operand, then it is **post increment** and the value is incremented after the operation is performed.

Example:

z = a++; // z=a

a= a+1

If a = 10, then z = a+ + gives

z= 10

a=11

Decrement operator:

It is used to decrement the values of a variable by 1.

The two types are −

· pre decrement

· post decrement

If the decrement operator is placed before the operand, then it is called pre decrement. Here, the value is first decremented and then, operation is performed on it.

Example:

z = - - a; // a= a-1

z= a

If a = 10, then z = - - a gives

z= 9

a= 9

If the decrement operator is placed after the operand, then it is called post decrement. Here, the value is decremented after the operation is performed.

Example:

z = a--; // z=a

a= a-1

If a = 10, then z = a - - gives

z= 10

a= 9.

6. Conditional operator (? :)

It is also called ternary operator.

The syntax is as follows −

exp1? exp2: exp3

The operator **?:** works as follows: exp1 is evaluated first. If it is true, then the exp2 is evaluated and becomes the value of the expression.

If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.
Example:

a = 10;

b = 15;

x = (a > b)? a: b;

If **(a > b)**

x = a;

else

x = b;

7. <u>Bitwise operators:</u>

Bitwise operators operate on bits.

Example: A = 60 = 0011 1100 and

B = 13 = 0000 1101

<u>Bitwise AND(&)</u>

Bitwise AND Operator copies a bit to the result if it exists

in both oerands.

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left Shift
>>	Right shift
~	One's Complement

<u>Bitwise XOR (^).</u>

Binary XOR Operator copies the bit if it is set in one operand but not both.

(A ^ B) = 49, i.e., 0011 0001

<u>One's Complement (~):</u>

Binary One's Complement Operator is unary and has the effect of 'flipping' bits. Meaning, all the 0s become 1s and vice-versa.

(~A) = ~(60), i.e,. -0111101

Left Shift (<<)

Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.

A << 2 = 240 i.e., 1111 0000

Right shift(>>)

Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

A >> 2 = 15 i.e., 0000 1111.

8. Special operators:

Some of the special operators are **Comma** operator, **Sizeof** operator, **pointer operator (& and *)** and **member selection** operator (. and ->).

Comma Operator:

The Comma operator is used to link the related expressions together. It evaluates first operand and then discards the result of the same, then the second operand is evaluated and result of same is returned. The comma linked list of expressions are evaluated left to right and the value of right most expression is the value of combined expression.

Example:

value = (x =10, y = 5, x+y);

First assigns the value 10 to x, then 5 to y and finally assigns 15 to **value.**

Sizeof operator:

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc). **Example:**

sizeof(a),where a is integer, will return 4.

sizeof(b), where b is float, will return 4.

sizeof(c), where c is double, will return 8.

sizeof(d), where d is integer, will return 1.

Arithmetic expressions:

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

Types of Expressions in C

· Arithmetic expressions.

· Relational expressions.

· Logical expressions.

· Conditional expressions.

An arithmetic expression in c is a combination of variables, constants, and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions.

Example:

Algebraic Expression	**C Expression**
a x b – c	a * b – c
(m + n)(x + y)	(m + n) * (x + y)
ab / c	(a*b) / c
$3x^2 + 2x + 1$	(3 * x * x) + (2 * x) + 1
x / y + c	x / y + c

Precedence of Arithmetic Operators

Operators	**Operations**	**Order Precedence**
( )	Parentheses	Evaluated first.
*** / %**	Multiplication, Division, Remainder	Evaluated second
+ –	Addition, Subtraction	Evaluated third.
=	Assignment	Evaluated last

Example:

Consider following statements:

x = a – b / 3 + c * 2 – 1;

Assume a = 10, b = 12, c = 2, the statement becomes

x = 10 – 12 / 3 + 2 * 2 – 1 is evaluated is as follows

Step 1 : x = 10 - 4 + 2 * 2 -1 (Division is Evaluated 12 / 3 = 4)

Step 2 : x = 10 - 4 + 4 -1 (Multiplication is Evaluated 2 * 2 = 4)

Step 3: x = 6 + 4 -1 (Subtraction is Evaluated 10 - 4 = 6)

Step 4: x = 10 -1 (Addition is Evaluated 6 + 4 = 10)

Step 5: x = 9 (Subtraction is Evaluated 10 - 1 = 9 and Assign 9 to x)

<u>INPUT-OUTPUT</u>

<u>FORMATTED INPUTS:</u>

Formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user. These types of I/O functions can help to display the output to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char, and many more.

These functions are called formatted I/O functions because we can use format specifiers in these functions and hence, we can format these functions according to our needs.

The Formatted input/output functions are -

scanf() – which is used to read one or multiple inputs from the user at the console. **printf() –** which is used to display one or multiple values in the output to the user at the console.

Formatted input refers to the data that has been arranged in a particular format. To read data in from standard input (keyboard), we call the **scanf** function. The basic form of a call to scanf is:

Inputting integer numbers:

scanf(*control_string*, *list_of_variable_addresses*);

· The control string specifies the field format in which the data is to be entered. It contains · format specifiers having conversion character %

· a data type character and

· An optional number specifying the field width.

· The list of variable addresses specify the address of the locations where the data is stored The field specification for reading a number is as follows:

Scanf("%d%d", &num1,&num2);

Data line is

31426 , 50

Then num1 = 31426 and num2 = 50 is assigned.

Any input field may be slipped by specifying * in the place of the filed width. **Example:**

Scanf("%d%*d %d", &num1,&num2);

Data line

123 456 789

The output is

num1 = 123

456 is skipped (because of *)

Num2 = 789.

<u>Inputting Real Numbers</u>

scanf reads **real numbers using the specification %f**

Example:

Scanf("%f%f%f", &x1,&y, &z);

With the input data

475.89 43.21 E-1 678

Will assign the value 475.89 = x , 4.321 = y and 678.0 = z

A number may be skipped using **%*f** specification

<u>Inputting Character Strings</u>

The **'%s'** is used as a format specifier for the string in c language.

Example:

char color[20];

scanf("%s", color);

<u>Reading mixed data types</u>

It is possible to use one scanf statement to input a data line containing mixed mode data. It should be ensured that the input data items match the control specifications in order and type. **Example:**

scanf("%d %c %f %s", &count, &code, &ratio, name);

will read the data

15 p 4.575 coffee

<u>Formatted output</u>

printf function is used for printing captions and numerical results. The output should be easily understandable and easy-to-use form.

The basic format of a **printf** function is:

printf (*control_string*, *list_of_expressions*);

The control string consists of three types of items:

· Characters that will be printed on the screen as they appear · Format specifier that define the output format for display of each item. · Escape sequence characters such as \ **n,**\ **t** and \ **b**

Example:

printf(“programming in C”);

printf(“ ”);

printf(“\n”);

printf(“%d” ,x);

printf(“a = %f\n b = %f , a,b);

printf(“sum =%d” ,1234);

printf(“ \n \n”);

BASIC STRUCTURE OF C PROGRAM:

Documentation Section

This section consists of the description of the program, the name of the program, and the creation date and time of the program. It is specified at the start of the program in the form of comments. Documentation can be represented as:

// description, name of the program, programmer name, date, time etc.

Anything written as comments will be treated as documentation of the program and this will not interfere with the given code. Basically, it gives an overview to the reader of the program.

Link Section

The link section provides instructions to the compiler to link functions from the system library.

Example:

#include<stdio.h>

#include<math.h>

Definition section:

The definition section defines all symbolic constants. Whenever this name is encountered by the compiler, it is replaced by the actual piece of defined code.

Example:

PI = 3.14

Global Declaration Section

There are some variables that are used in more than one function. Such variables are called global variables.

Example:

intnum = 18;

int a=7;

Main Function Section

Every C-programs must have the **main()** function. Each main function contains 2 parts. A **declaration part and an Execution part**. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces. All statements in the declaration and executable parts end with semicolon.

Example:

```
int main(void)

{

int a=10;

printf(" %d", a);

}
```

Sub Program Section

The subprogram section contains all the user defined functions that are called in the main function.

Example:

```
int add(int a, int b)

{

returna+b;

}
```

****

Character set

The character set refers to **a set of all the valid characters that we can use in the source program for forming words, expressions, and numbers**. The characters in C are grouped into following categories:

C tokens:

Types of characters:

1. Letters

2. Digits

3. Special characters and 4. White spaces

Tokens in C language are the smallest elements or the building blocks used to construct a C program.

Types of Tokens

Tokens in C language can be classified as:

1. Keywords

2. Identifiers

3. Constants

Keywords and identifiers:

4. Special Characters 5. Strings

6. Operators

Every C word is classified as either a keyword or identifier. ***Keywords*** have fixed meanings. They serve as the basic building blocks for programming statements. There are 32 built-in keywords.

Identifiers refer to the names of variables, functions and arrays. These are user defined names and consist of sequence of letters and digits with letter as first character. The underscore character can also be used in identifiers.

Constants:

Constants are the fixed values that do not change during the execution of the program. The classification of constants is as shown below.

Numeric Constant:

A ***numeric constant*** consists of numerals. Numeric constants are again divided into two types: · **Integer constant**

· **Real constant**

a) Integer constant:

An integer constant refers to the sequence of digits. There are three types of integers –

1. Decimal
2. octal
3. hexadecimal

Decimal integers consist of set of digits 0 to 9. The + or - sign is optional.

Example:

123, -321, 0 , 654321 , + 78 etc.

Embedded characters, commas and non digit characters are not permitted between digits.

Example:

15 750 , 20, 000 , $ 420 etc., are illegal numbers.

An octal integer constant consists of any combination of digits from 0 to 7with a leading zero.

Example:

037 , 0 , 0435 , 0551 etc.,

The hexadecimal integer constant consists of digits from 0 to 9 and letters from A to F. here the sequence digits must begin with 0x or 0X

Example:

0X2 , 0x9F , 0Xbcd, 0X

b) Real Constant:

Integers are inadequate to represent the quantities that vary continuously. Hence these are represented numbers containing factional part. Such numbers are called Real or Floating point constants.

Example:

215. , 0.95, - 7.1, + 0.5

They can also be represented in exponential form

Example:

0.65 e4 , 12 e-2, 1.5 e+5, 3.18E3 , -1.2E-1

Character constant:

A single character constant contains a single character enclosed within a single quotes. These character constants are translated into ASCII code.

Example:

'5' , 'X', 'a'

String constants:

A string **constant**is a sequence of characters enclosed within double quotes. the characters may be letters, numbers, special characters, blank space etc.,

Example:

"Hello", "1987", "Well Done", " 5 + 3"

Backslash character constants:

A backslash character (\ ) is used to introduce an escape sequence, which allows a visual representation of certain nongraphic characters

Variables:

A variable is an identifier that is used to store different values at different times during the execution of the program

Example:

Average, height, Total, Counter_1,

class_strength etc.,

The following conditions hold good for a variable

a) Variable name must begin with a letter

b) It must begin with a digit

c) Variable names can be any length

d) It should not be a keyword

e) Upper case and lower case letters are different

f) White spaces are not allowed.

DATA TYPES

The data type indicates the type of data stored in the variables. The data types are classified into

- Primary (fundamental) data type
- Derived data type
- User defined data type

<u>Primary (fundamental) data type</u>

The C language has 3 basic (primary or primitive) data types, they are:

The primary data types are of five types –

1. Integer**(int)**
2. Character**(char)**
3. Floating point**(float)**
4. Double precision floating point**(double)**
5. Void**(void)**

<u>Integer types:</u>

Integers are whole numbers with a range of values supported by the word size of the machine. The word size is 16 or 32 bits. For 16 bit word length, the size of the integer value range is -32768 to +32767 (-2^{15} to $+2^{15}$).

In order to provide more range of numbers and storage space, the integer types has been classified into **shortint, int and longint** in both **signed and unsigned forms**

Type	**Size(bits)**	**Range**
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32768 to 32767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2147483648 to 2147483647
unsigned long int	32	0 to 4294967295

float	32	3.4E-38 TO 3.4E+38
double	64	1.7E-308 TO 1.7E+308
long double	80	3.4E-4932 TO 1.1E+4932

For example, **short int** represents small integer values and requires half the amount of storage as a regular **int** number uses.

Floating point types

Floating point numbers are declared by the key word **float.** If more accuracy is required in representing a number, the **type double** can be used to define a number. A double **data type** number uses 64 bits giving a precision of 64 bits.

Declaration of variables:

The variable names designed must be declared to the complier. The declaration of variables does – · That it informs the compiler the name of the variable and

· It specifies what type of data the variable will hold.

There are two types of variable declaration namely –

· Primary type declaration and

· User defined type declaration

Assignment statement:

Values can be assigned to variables using the assignment operator = as follows

variable_name = constant;

Example:

initial_value = 0;

final value = 100;

balance = 75.84;

yes = ‘x’;

DECISION MAKING AND BRANCHING

In programming the order of execution of instructions may have to be changed depending on certain conditions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain instructions accordingly.

Some of the decision making statements are -

1. **if** statement

2. **switch** statement

3. **conditional operator** statement

4. **goto** statement

Decision making with *if* statement:

It is basically a two-way decision statement used in conjunction with an expression. It is of the form

if(test expression)

Example:

***if (room is dark)**

put on lights

***if(code is 1)**

Person is male

There are different forms of ***if*** statement which is implemented depending on the complexity of the conditions to be tested. They are –

1. simple **if** statement

2. **if …..else** statement

3. **nested if …..else** statement

4. **else if** ladder

Simple *if* statement:

The general form of simple if statement is:

```
if (test expression) {
statement block; }
statement x;
```

Example:

```
if (code = = 1)
{
salary = salary + 500; }
printf("%d",salary);
```

The statement block may be a single statement or a group of statements. If the test expression is true, the statement block will be executed otherwise the statement block will be skipped and the execution will jump to the statement –x.

The if…..else statement:

The **if... else** statement is an extension of the simple if statement **Syntax is:**

```
if (test expression)  {
statement block; }
else
{
statement block; }
statement-x;
```

Example:

```
if (code = = 1)
{
boy= boy + 1;
}
else
{
girl = girl + 1;
}
```

Nesting of if …..else statements

When a series of conditions are to be checked, we may have to use more than one if... else statement in the nested form.

The logic of the execution is as shown above. If condition 1 is false, the statement 3 will be executed; otherwise it continues to perform the second test. If the condition 2 is true, the statement 1 will be evaluated; otherwise the statement 2 will be evaluated and then the control is transferred to the statement –x.

Example:

```
if( a>b)
{
 if(a>c)
 printf(“ a is greater”);
 else
 printf(“ c is greater”);
 }
else
 {
 if(b>c)
```