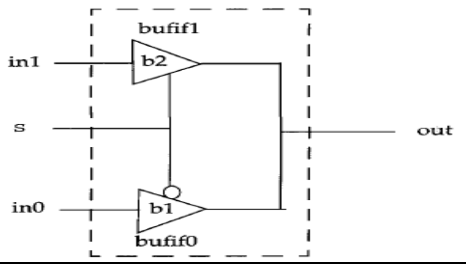
USN | | | | | | | | | |

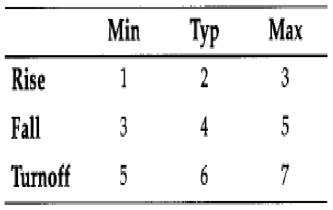
Internal Assessment Scheme –OCT. 2019

Sub:	Verilog HDL					Sub Code:	17EC53	Branch:
Date:	/10/2018	Duration:	90 min's	Max Marks:	50	Sem / Sec:	5th(ECE/TCE)

Answer any FIVE FULL Questions

1)Design a 2-to-1 multiplexer using bufifo and bufifl gates as shown below. The delay specification for gates bl and b2 are shown in table below. Apply stimulus and test the output values.





	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

```
Module mux21(in1,in2,s,out);
Input in1,in2,s;
Output out;
Bufif1(1:3:5,2:4:6,3:5:7) m1(out1,in1,s);
Bufif10(1:3:5,2:4:6,3:5:7) m1(out1,in0,s);
Endmodule
```

```
stimulus
Module mux21tb;
Reg in1,in0,s;
Wire out;
Initial
Begin
In0=0;in1=1;s=0;#100;

In0=1;in1=9;s=0;#100;
End
endmodule
```

2)Explain different operators supported by Verilog HDL.and also discuss the precedence of operators

Arithmetic Operators

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

Binary operators

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

105

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers
```

```
A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
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 +1, 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. Unary + or ? operators have higher precedence than the binary + or ? operators.

```
-4 // Negative 4
+5 // Positive 5
```

Negative numbers are represented as 2's complement internally in Verilog. It is advisable to use negative numbers only of the type integer or real in expressions. Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

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

1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is 01equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.

Relational Operators

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

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.

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.

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

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

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

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

bitwise negation	result
0	1
1	0
x	x

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

Shift Operators

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around. Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

Operator Precedence

Having discussed the operators, it is now important to discuss operator precedence. If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence.

3) (a)Design 4-bit adder using carry look ahead logic using dataflow description.

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;
// compute the p for each stage
assign p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],
```

```
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
```

```
// compute the g for each stage
assign g0 = a[0] & b[0],
g1 = a[1] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];
// compute the carry for each stage
// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in,
sum[1] = p1 ^ c1,
sum[2] = p2 ^ c2,
sum[3] = p3 ^ c3;
// Assign carry output
assign c_out = c4;
endmodule
```

(b) Explain the primitive gates supported by Verilog HDL?.

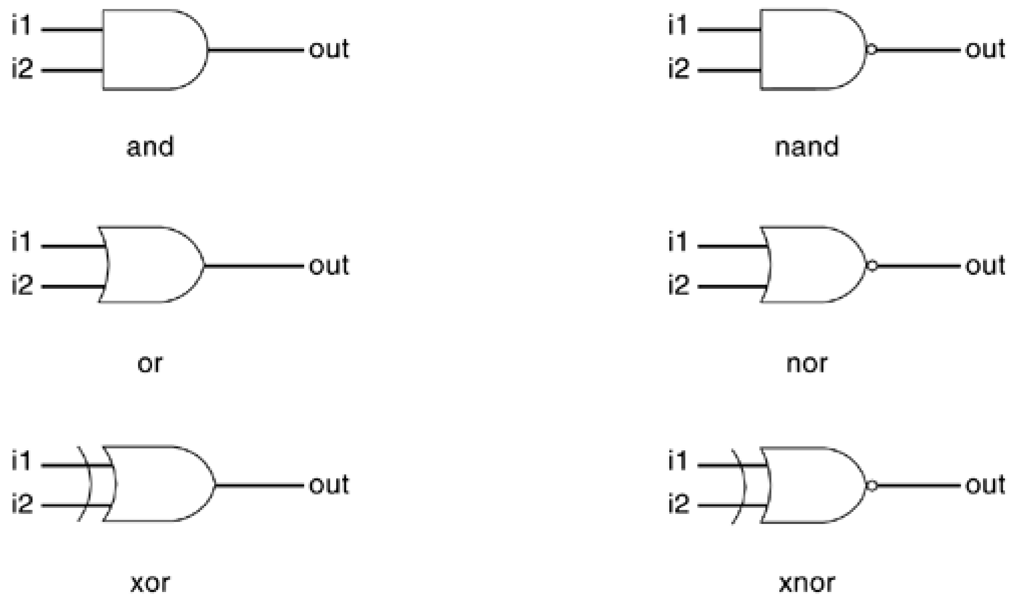
Gate Types

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

5.1.1 And/Or Gates

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

and or xor
nand nor xnor

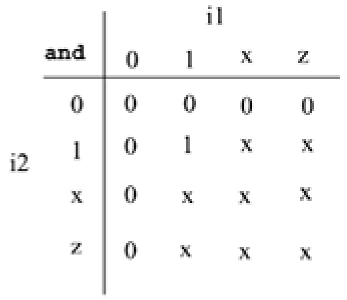


These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below

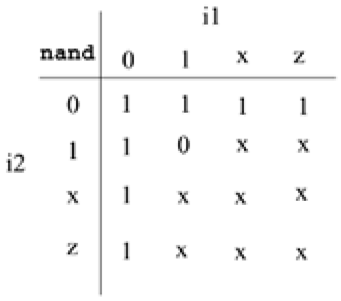
Gate Instantiation of And/Or Gates

```
wire OUT, IN1, IN2;
// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);
// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation
```

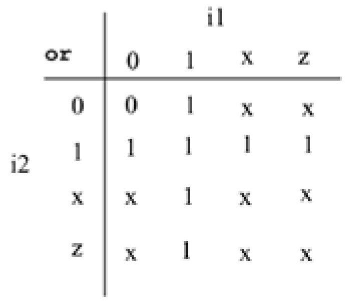
Truth Tables for And/Or



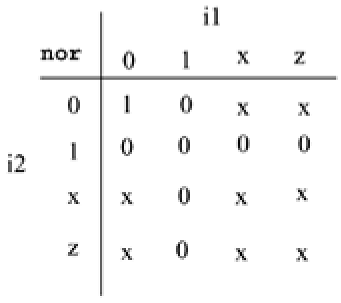
		i1			
	and	0	1	x	z
	0	0	0	0	0
i2	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x



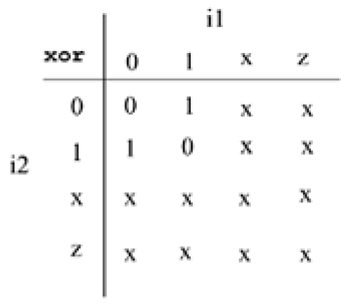
		i1			
	nand	0	1	x	z
	0	1	1	1	1
i2	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x



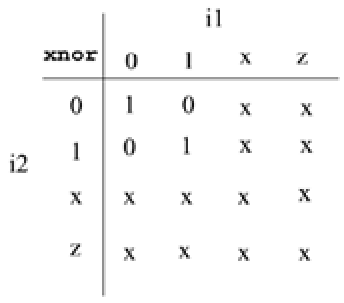
		i1			
	or	0	1	x	z
	0	0	1	x	x
i2	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x



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



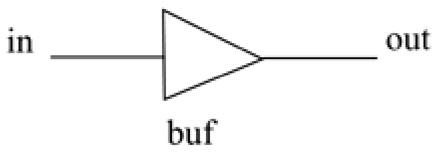
		i1			
	xor	0	1	x	z
	0	0	1	x	x
i2	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

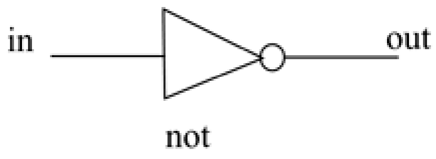


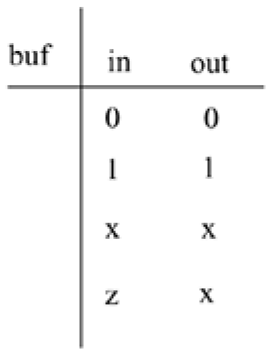
		i1			
	xnor	0	1	x	z
	0	1	0	x	x
i2	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

Buf/Not Gates

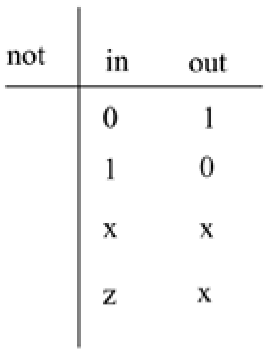
Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output.







buf	in	out
	0	0
	1	1
	x	x
	z	x



not	in	out
	0	1
	1	0
	x	x
	z	x

4) Explain system tasks with Verilog programs.

System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form $<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks. We will discuss only the most useful system tasks. Other tasks are listed in Verilog manuals provided by your simulator vendor or in the IEEE Standard Verilog Hardware Description Language specification.

Displaying information

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

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

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

String Format Specifications

Format	**Display**
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

Any one example

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World
//Display value of current simulation time 230
$display($time);
-- 230
//Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c
//Display value of port_id 5 in binary
reg [4:0] port_id;
```

```
$display("ID of the port is %b", port_id);
-- ID of the port is 00101
//Display x characters
//Display value of 4-bit bus 10xx (signal contention) in binary
reg [3:0] bus;
$display("Bus value is %b", bus);
-- Bus value is 10xx
//Display the hierarchical name of instance p1 instantiated under
//the highest-level module called top. No argument is required. This
//is a useful feature)
$display("This string is displayed from %m level of hierarchy");
-- This string is displayed from top.p1 level of hierarchy
```

5) Explain arrays, parameter and memories data types with examples in Verilog.

Arrays

Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types. Multi-dimensional arrays can also be declared with any number of dimensions. Arrays of nets can also be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net. Arrays are accessed by <array_name>[<subscript>]. For multi-dimensional arrays, indexes need to be provided for each dimension.

```
integer count[0:7]; // An array of 8 count variables
reg bool[31:0]; // Array of 32 one-bit boolean register variables
time chk_point[1:100]; // Array of 100 time checkpoint variables
reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits
wide
integer matrix[4:0][0:255]; // Two dimensional array of integers
reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array
wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire
wire w_array1[7:0][5:0]; // Declare an array of single bit wires
count[5] = 0; // Reset 5th element of array of count variables
chk_point[100] = 0; // Reset 100th time check point value
port_id[3] = 0; // Reset 3rd element (a 5-bit value) of port_id array.
matrix[1][0] = 33559; // Set value of element indexed by [1][0] to
33559
array_4d[0][0][0][0][15:0] = 0; //Clear bits 15:0 of the register
//accessed by indices [0][0][0][0]
port_id = 0; // Illegal syntax - Attempt to write the entire array
```

Memories

In digital simulation, one often needs to model register files, RAMs, and ROMs. Memories are modeled in Verilog simply as a one-dimensional array of registers. Each element of the array is known as an element or word and is addressed by a single array index. Each word can be one or more bits. It is important to differentiate between n 1-bit registers and one n-bit register. A particular word in memory is obtained by using the address as a memory array subscript.

```
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)
membyte[511] // Fetches 1 byte word whose address is 511.
```

Parameters

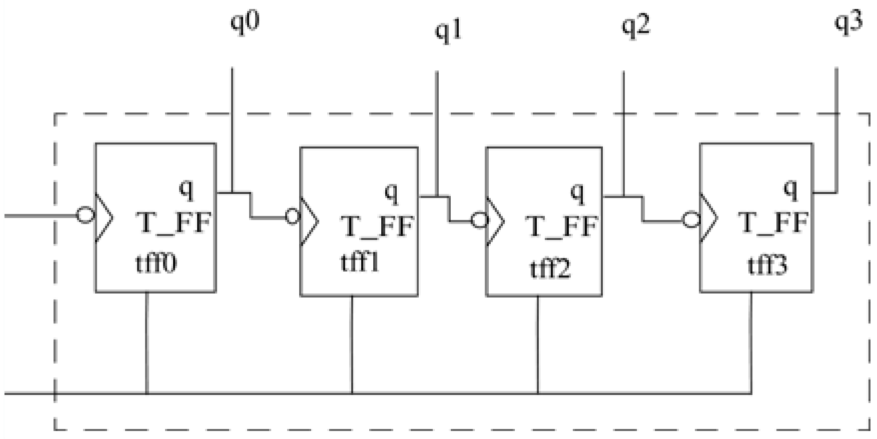
Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types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

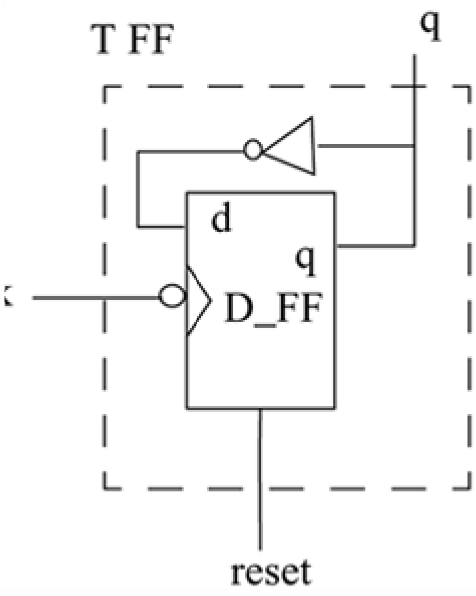
```
parameter port_id = 5; // Defines a constant port_id
parameter cache_line_width = 256; // Constant defines width of cache
line
parameter signed [15:0] WIDTH; // Fixed sign and range for parameter
// WIDTH
```

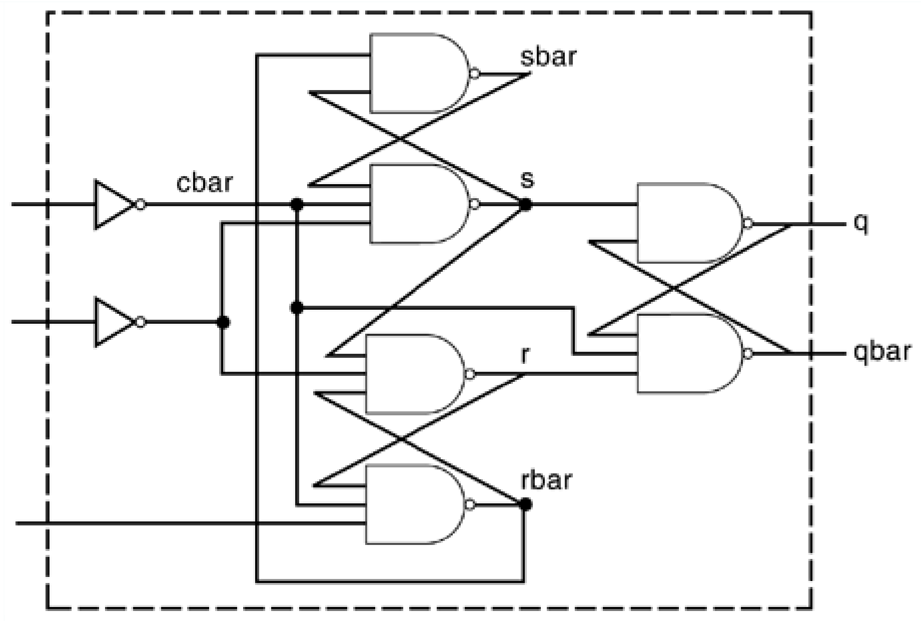
Module definitions may be written in terms of parameters. Hardcoded numbers should be avoided. Parameters values can be changed at module instantiation or by using the defparam statement, Useful Modeling Techniques. Thus, the use of parameters makes the module definition flexible. Module behavior can be altered simply by changing the value of a parameter.

6)Design 4-bit ripple carry counter using dataflow description. Also write the lus block to verify the same.

ipple Carry Counter







```
module counter(Q , clock, clear);
// I/O ports
output [3:0] Q;
input clock, clear;
// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);
endmodule
```

Verilog Code for Edge-Triggered D-flipflop

```
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);
// Inputs and outputs
output q,qbar;
input d, clk, clear;
// Internal variables
wire s, sbar, r, rbar,cbar;
// dataflow statements
//Create a complement of signal clear
```

120

```
assign cbar = ~clear;
// Input latches; A latch is level sensitive. An edge-sensitive
// flip-flop is implemented by using 3 SR latches.
assign sbar = ~(rbar & s),
s = ~(sbar & cbar & ~clk),
r = ~(rbar & ~clk & s),
rbar = ~(r & cbar & d);
// Output latch
assign q = ~(s & qbar),
qbar = ~(q & r & cbar);
dmodule
module stimulus;
// Declare variables for stimulating input
reg CLOCK, CLEAR;
wire [3:0] Q;
initial
$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);
// Instantiate the design block counter
counter c1(Q, CLOCK, CLEAR);
```

```
// Stimulate the Clear Signal
initial
begin
CLEAR = 1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
end
// Set up the clock to toggle every 10 time units
initial
begin
CLOCK = 1'b0;
forever #10 CLOCK = ~CLOCK;
end
// Finish the simulation at time 400
initial
begin
#400 $finish;
                endmodule
```

7) a)Write a note on lexical conventions used in verilog.

Or

(a)Write a note on following lexical conventions used in verilog.
i)operators ii) Identifiers and keywords iii)Strings

Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

3.1.1 Whitespace

Blank spaces (\b) , tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

Comments

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

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

Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
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 ('o or '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.

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 that is simulator- and machine-specific (must be at least 32).

```
23456 // This is a 32-bit decimal number by default
'hc3 // This is a 32-bit hexadecimal number
'o21 // This is a 32-bit octal number
```

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

```
12'h13x // This is a 12-bit hex number; 4 least significant bits
unknown
6'hx // This is a 6-bit hex number
32'bz // This 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. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

```
-6'd3 // 8-bit negative number stored as 2's complement of 3
-6'sd3 // Used for performing signed integer math
4'd-2 // Illegal specification
```

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. Strings are treated as a sequence of one-byte ASCII values.

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

Identifiers and Keywords

Keywords are special identifiers reserved to define the language constructs. Keywords are in lowercase. A list of all keywords in Verilog is contained in Appendix C, List of Keywords, System Tasks, and Compiler Directives.
Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($). Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore. They cannot start with a digit or a $ sign (The $ sign as the first character is reserved for system tasks, which are explained later in the book).

```
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. Neither the backslash nor the terminating whitespace is considered to be a part of the identifier.

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

(b)Write a Verilog dataflow level of abstraction for 4 to 1 multiplexer using conditional operator.

```
Modulemultiplexer4_1(din,sel,dout)

output                    dout                     ;
input                                      [3:0]din;
input                                      [1:0]sel;


 assign   dout   =   (sel==2'b00)   ?   din[3]
    (sel==2'b01)          ?         din[2]
    (sel==2'b10)          ?         din[1]
    din[0];

endmodule
```