USN

CMRIT CMR INSTITUTE OF TECHNOLOGY, BENGALURU. ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 1 – Sept. 2018

Sub:	Verilog HDL				Sub Code:	15EC53	Branch:	ECE
Date:	08/09/2018	Duration:	90 min’s	Max Marks: 50	Sem / Sec:	5th /C and D		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Explain the different levels of Abstraction used for programming in Verilog.	[10]	CO1	L1
2	Illustrate the design of 4-bit ripple carry counter using top down design methodology. Also write the Verilog HDL code for Ripple carry counter including testbench.	[10]	CO1	L3
3	Explain the factors that have made Verilog HDL popular and also explain the advantages of using Verilog HDL.	[10]	CO1	L1
4	Explain the typical design flow for designing VLSI IC with the flow chart.	[10]	CO1	L1
5	Explain the different data types used in Verilog HDL.	[10]	CO1	L1
6	Illustrate the design of 4-bit subtractor using top down design methodology. Also write the Verilog HDL code for 4-bit subtractor and also draw the simulation waveform.	[10]	CO1	L3
7	Explain the following: i) Module, ii) Instance, iii) Stimulus, iv) Compiler directive, v) System Task	[10]	CO1	L1

USN

CMRIT CMR INSTITUTE OF TECHNOLOGY, BENGALURU. ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 1 – Sept. 2018

Sub:					Sub Code:		Branch:	
Date:		Duration:	90 min’s	Max Marks: 50	Sem / Sec:			OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Explain the different levels of Abstraction used for programming in Verilog.	[10]	CO1	L1
2	Illustrate the design of 4-bit ripple carry counter using top down design methodology. Also write the Verilog HDL code for Ripple carry counter including testbench.	[10]	CO1	L3
3	Explain the factors that have made Verilog HDL popular and also explain the advantages of using Verilog HDL.	[10]	CO1	L1
4	Explain the typical design flow for designing VLSI IC with the flow chart.	[10]	CO1	L1
5	Explain the different data types used in Verilog HDL.	[10]	CO1	L1
6	Illustrate the design of 4-bit subtractor using top down design methodology. Also write the Verilog HDL code for 4-bit subtractor and also draw the simulation waveform.	[10]	CO1	L3
7	Explain the following: i) Module, ii) Instance, iii) Stimulus, iv) Compiler directive, v) System Task	[10]	CO1	L1

Q. Explain the different levels of Abstraction used for programming in Verilog.

A. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment. The levels are defined below:

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

```
module andg(c,a,b);
input a, b;
output reg c;

always@(a,b)            //behavioral modelling style uses keyword “always”.
  if (a ==1’b1 && b ==1’b1)
        c = 1’b1;
else
        c = 1’b0;
endmodule
```

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

```
module andg(c,a,b);
input a, b;
output c;
assign c = a & b; //dataflow modelling style uses keyword “assign”.
endmodule
```

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

Ex:

```
module andg(c,a,b);
input a, b;
output c;

and a1(c, a, b); //gatelevel modelling style uses instantiation of primitive
modules.
// here and is primitive module which functions as logical and gate.
```

```
endmodule
```

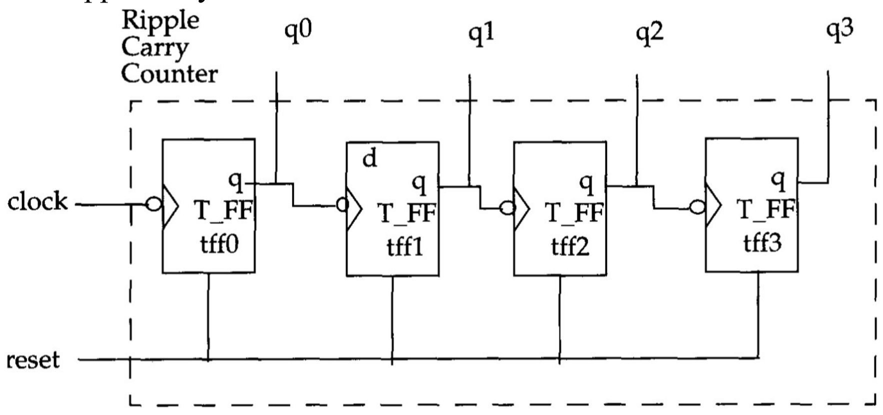
d) Switch level:
This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

```
module notg(b,a);
input a;
output b;

supply1 vdd;
supply0 gnd;
pmos p1 (b, vdd, a);
nmos n1 (b, gnd, a);

endmodule
```

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

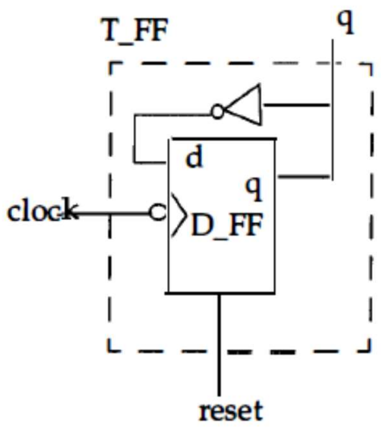
Q. Illustrate the design of 4-bit ripple carry counter using top down design methodology. Also write the Verilog HDL code for Ripple carry counter including testbench.

A. The ripple carry counter construction is as shown below:

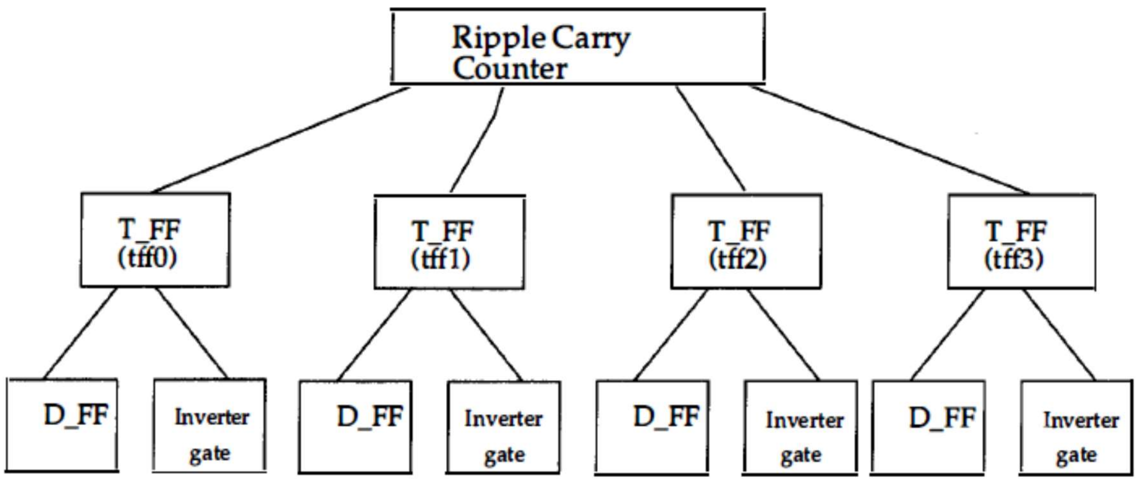


Each of the T-FFs can be made up from negative edge-triggered D-flipflops (D-FF) and inverters.

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



Thus, the ripple carry counter is built in a hierarchical fashion using top down design methodology using the building blocks. The diagram for the design hierarchy is shown in the following figure:



In a top-down design methodology, the functionality of the ripple carry counter is specified, which is the top-level block. Then, the counter with T _FFs is implemented. The T _FFs are built using D _FF and an additional inverter gate. Thus, the bigger blocks are broken into smaller building sub-blocks until further breakup of the blocks is not possible.

```
// Definition of the top-level module called ripple carry counter

module ripple-carry-counter(q, clk, reset);
output [3:01 q;
input clk, reset;
TFF tff0 (q[0] , ~clk , reset);
TFF tffl (q[l] , q[0] , reset);
TFF tff2 (q[2] , q[l] , reset) ;
TFF tff3 (q[3] , q[2] , reset) ;
endmodule



// Definition of the TFF module using D fliflop and inverter
module TFF(q, clk, reset) ;
output q;

input clk, reset;
wire d;

D_FF dff0 (q, d, clk, reset);
not nl(d, q);

endmodule

// Definition of the DFF module
module D_FF(q, d, clk, reset);
output q;
input d, clk, reset;
reg q;

always @(posedge reset or negedge clk)
if (reset)
q <= 1'b0;
else
q <= d;
endmodule



//Stimulus block

module stimulus;
reg clk;
reg reset;
wire[3:0] q;
```

```
ripple_carry_counter r1(q, clk, reset);

initial
clk = 1'b0;

always
#5 clk = ~clk;

initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish; //terminate the simulation
end

initial
$monitor($time, " Output q = %d", q);
endmodule
```

Q. Explain the factors that have made Verilog HDL popular and also explain the advantages of using Verilog HDL.

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

1. Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.

2. Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.

3. Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.

4. All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.

5. The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog.
   Designers can customize a Verilog HDL simulator to their needs with the PLI.

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

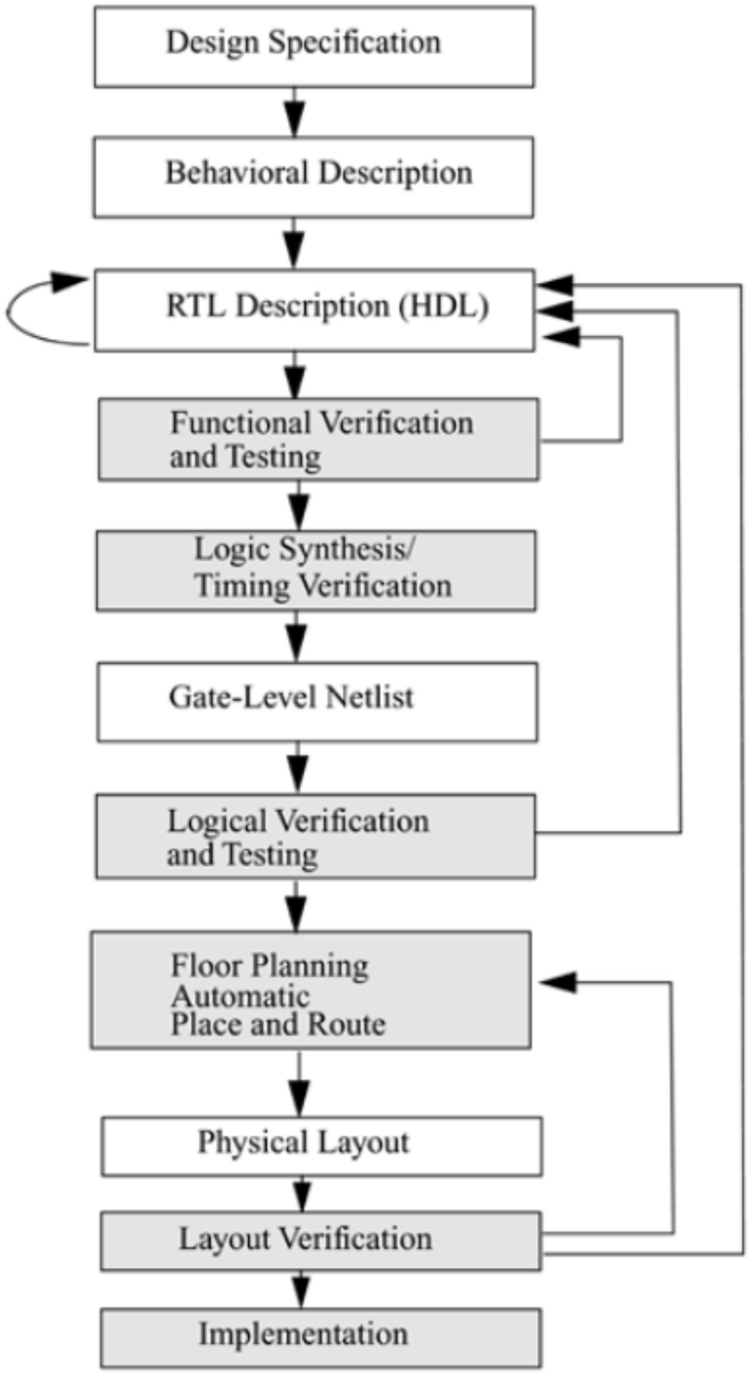
1. Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need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

2. By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point.

3. Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics.

Q. Explain the typical design flow for designing VLSI IC with the flow chart.

A. 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

The specifications are written first in any design. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality,

performance, compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs.
The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of Computer-Aided Design (CAD) tools.

Logic synthesis tools convert the RTL description to a gate-level netlist. A gatelevel netlist is a description of the circuit in terms of gates and connections between them. 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.

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

Behavioral synthesis tools can create RTL descriptions from a behavioral or algorithmic description of the circuit. As these tools mature, digital circuit design will become similar to high-level computer programming. Designers will simply implement the algorithm in an HDL at a very abstract level.

Although CAD tools are available to automate the processes and cut design cycle times, the designer is still the person who controls how the tool will perform. CAD tools are also susceptible to the "GIGO : Garbage In Garbage Out" phenomenon. If used improperly, CAD tools will lead to inefficient designs.

Q. Explain the different data types used in Verilog HDL.

A. The data types used in Verilog is discussed as follows

1. Value Set:
   Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in the following table:

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

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits.

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

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

2) Nets:
Nets represent connections between hardware elements.

Ex: wire a;
wire b,c;
wire d = 1'b0;

3) Registers:
Registers represent data storage elements. Registers retain value until another value is placed onto them.
Ex: reg reset;
reg signed [63:0] m; // 64 bit signed value

4) Vectors:
Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).
Ex: wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.

5) Integer:
An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits. Registers declared as data type ‘reg’ store values as unsigned quantities, whereas integers store values as signed quantities.
Ex:
integer counter; // general purpose variable used as a counter.
initial
counter = -1;

6) Real:
Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation or in scientific notation.

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

7) Time:

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits. The system function $time is invoked to get the current simulation time.
Ex:
time save_sim_time; // Define a time variable save_sim_time
initial
save_sim_time = $time;

8) 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.
Ex:
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
```

8) Memories:

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.

Ex:
```
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.
```

9) Parameters:
Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables.
Ex:
```
parameter port_id = 5; // Defines a constant port_id
parameter cache_line_width = 256;
```

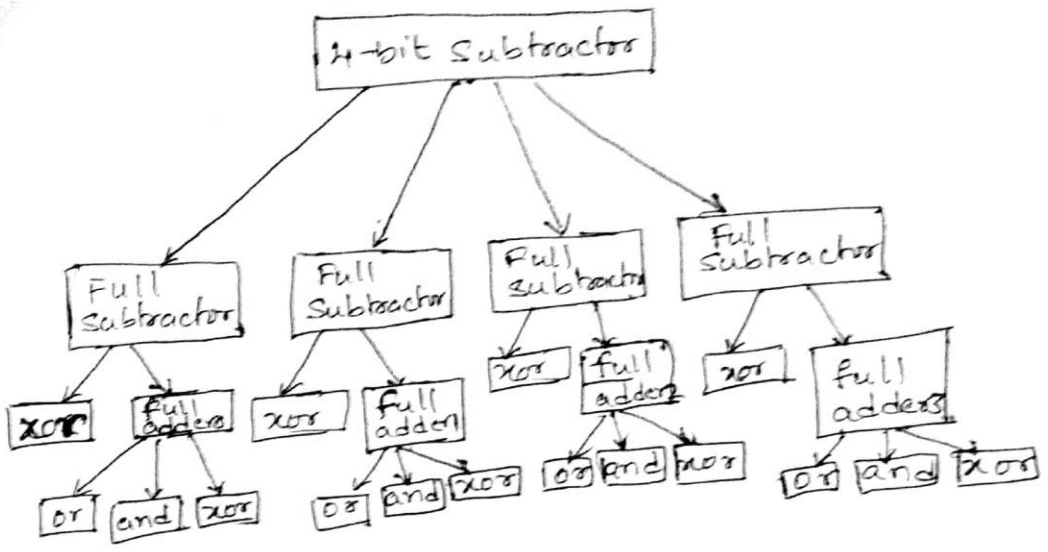
10) Strings:
Strings can be stored in reg. The width of the register variables must be large enough to hold the string. Each character in the string takes up 8 bits (1 byte).
Ex:
```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide
initial
string_value = "Hello Verilog World";
```
Special characters serve a special purpose in displaying strings, such as newline, tabs, and displaying argument values. Special characters can be displayed in strings only when they are preceded by escape characters, as shown

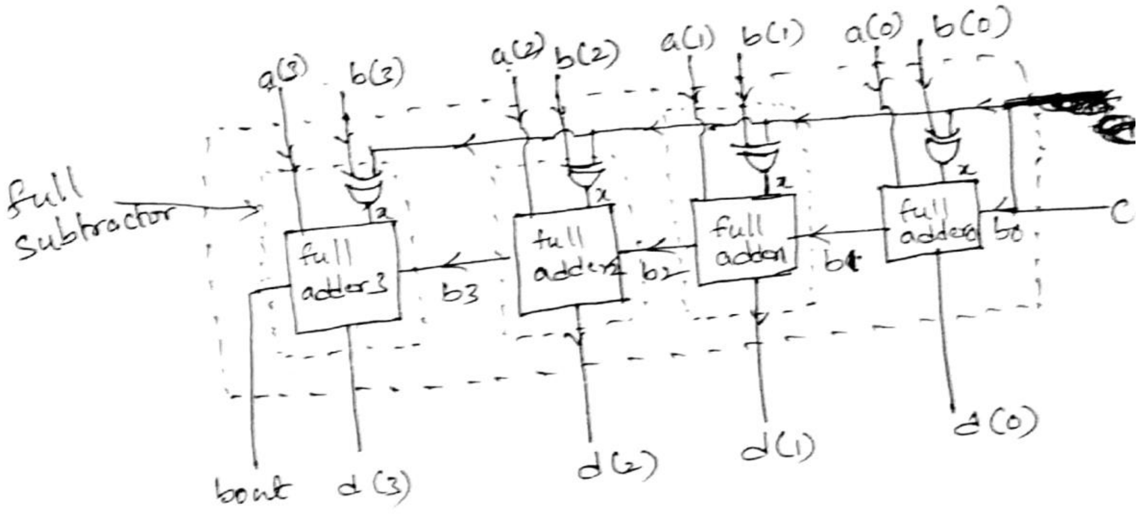
Escaped Characters	Character Displayed
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	Character written in 1?3 octal digits

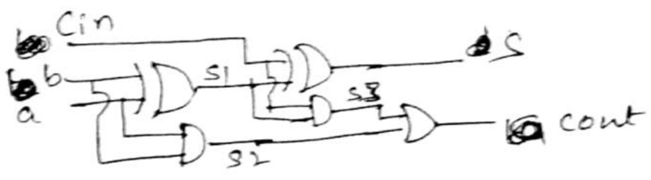
Q. Illustrate the design of 4-bit subtractor using top down design methodology. Also write the Verilog HDL code for 4-bit subtractor and also draw the simulation waveform.

A.

Top down hierarchical flow.





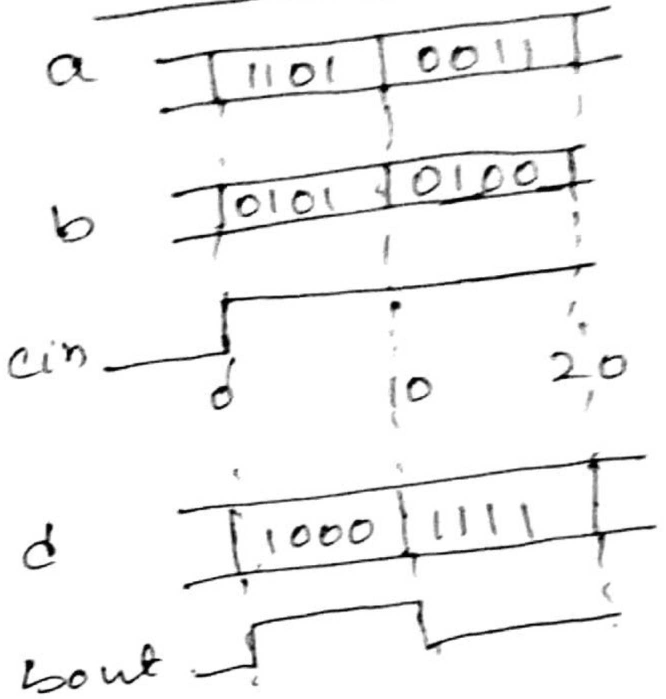


full adder module

```
// Toplevel module foo 4-bit subtoactor.
      module subtoact4bit(bout, d , a, b, cin);
            input cin;
            input [3:0] a, b;
            output [3:0] d;
             output bout;
             wire b0, b1, b2, b3;
            assign b0 = cin;
         FS  fs0 (b1, d[0], a[0], b[0], b0, cin);
         FS  fs1 (b2, d[1], a[1], b[1], b1, cin);
         FS  fs2 (b3, d[2], a[2], b[2], b2, cin);
         FS  fs3 (bout, d[3], a[3], b[3], b3, cin);
         endmodule
    // module definition foo full subtoactor
         module FS (bout, d, a, b, bin, cin);
           input  a, b, bin, cin;
           output  d, bout;
             wire x;
          xor   xg(x, b, cin);
          fa    fa0 (bout, d, a, x, bin);
           endmodule
```

```
// definition of full adder module

module fa (cout, s, a, b, cin);
    input a, b, cin;
    output s, cout;
    wire  s1, s2, s3;

    xor   x1(s1, a, b);
    xor   x2(s, s1, cin);
    and   a1 (s2, a, b);
    and   a2(s3, s1, cin);
     or   o1 (cout, s2, s3);
    endmodule.

//stimulus block
    module stimulus;
     ~~[illegible]~~, reg cin; reg [3:0] a, b;
     wire d, bout;
     subtract4bit uut (bout, d, a, b, cin);
      initial.
       begin
        cin=1'b1; ~~[illegible]~~
             a=4'b1101;  b=4'b0101;
        #10 a =4'b 0011; b= 4'b 0100;
         #10 ~~[illegible]~~ $finish;
        end
     initial $monitor ($time, "a=%d, b=%d, d=%d,
                          bout=%d", a, b, d, bout);
        endmodule.
```

Simulation waveform:



Q. Explain the following:
i) Module, ii) Instance, iii) Stimulus, iv) Compiler directive, v) System Task

A. i) Module:

A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation.

Ex:

```
module ripple-carry-counter(q, clk, reset);
output [3:01 q;
input clk, reset;
…………………
…………………
//statements
…………………
…………………

endmodule
```

ii) Instance:

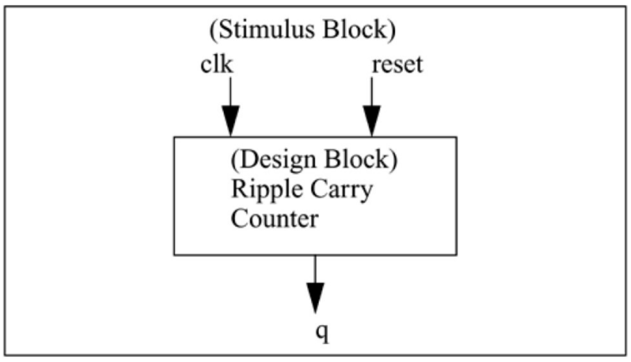
A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and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

Ex:

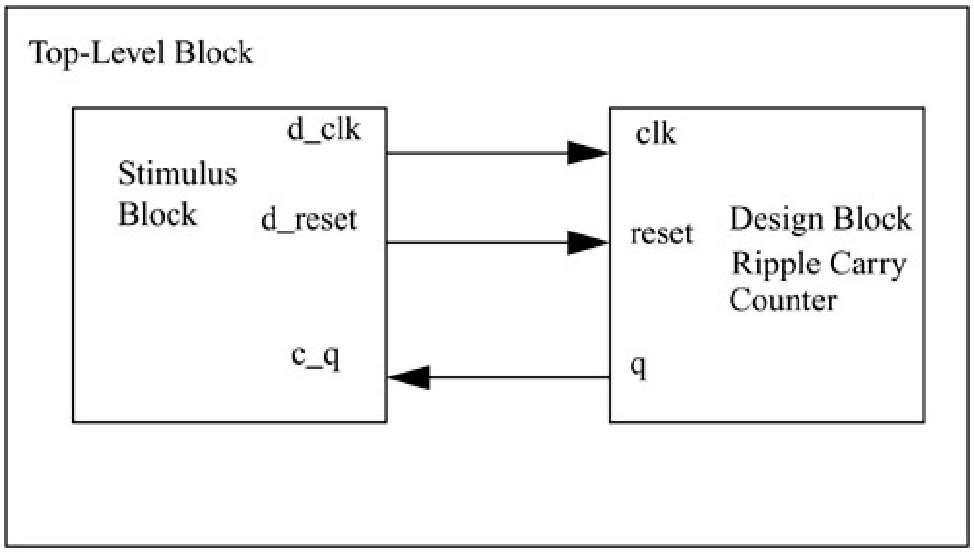
```
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;

//Following statements are the instances of T_FF
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

iii) Stimulus:

The functionality of the design block can be tested by applying stimulus and checking results. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.
Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block.



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.



iv) Compiler directive

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

Ex:

**`define** `WORD_SIZE 32`

The `define directive is used to define text macros in Verilog. The Verilog compiler substitutes the text of the macro wherever it encounters a `<macro_name>.

`include

The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation.

```
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

v) System Task

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.

Ex:

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

$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. p1, p2, p3,..., pn can be quoted strings or variables or expressions.

$monitor
Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the $monitor task.

$monitor(p1,p2,p3,....,pn);
The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the $display task is used in the $monitor task. $monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes. Unlike $display, $monitor needs to be invoked only once.