Lab Manual for Digital System Design BECE102P

Introduction:

Today's digital integrated circuits are designed to perform highly complex functions. Their design would be very challenging (and nearly impossible) if it were not for the availability of the numerous CAD (computer-aided design) tools used for the following tasks: design entry, simulation, synthesis and optimization, and physical design. Initially, the specifications of the system to be designed are derived from the requirements.

1. Design entry.

Design entry is the first step in the design process using CAD tools. A designer describes the circuit to be implemented using some design entry method provided by the tools. The most common design-entry method is to write a hardware description language program in VHDL(Very High-Speed Integrated Circuit Hardware Description Language) or Verilog, which can be compiled using CAD tools to simulate the design, synthesize, optimize, and generate the circuit implementation. Other methods of design entry include
schematic entry (the circuit is drawn using symbols from a library supported by the tool), and entry using truth tables (where inputs and corresponding outputs are described using a truth table in a text file).

2. Simulation.

Once the design entry is completed, the design is tested for correct functionality using simulation. The CAD tool that performs this job is called a functional simulator. For simulation, the user has to supply to the simulator "test vectors" for the inputs and expected outputs. Simulation is also performed after synthesis, which is the process of translating the design entered (at the design-entry phase) to a physically realizable circuit using logic gates (in the Application-specific integrated circuit(ASIC) design flow) or logic blocks inside a Field Programmable Gate Arrays (FPGA). Post-synthesis simulation involves functional testing as well as tests to ensure that the timing constraints of that the circuit are met.

3. Synthesis.

A synthesis tool is used to translate the design described in a design entry method (usually, a program in VHDL or Verilog) into a physically realizable circuit. The same tool is also used to optimize the circuit.

4. **Place and Route (PAR).**

This is the final step in the design process before actual hardware implementation of a digital integrated circuit (IC). PAR, also known as the physical design phase, is where the gates are placed and interconnected (routing) to complete the circuit. A final timing simulation is performed after PAR to make sure that the circuit meets timing constraints when the parasitic capacitances due to the transistors and interconnecting metal wires are added.

HARDWARE DESCRIPTION LANGUAGES

A hardware description language (HDL) is a programming language used to describe the behavior or structure of digital circuits (ICs). HDLs are also used to stimulate the circuit and check its response. Many HDLs are available, but VHDL and Verilog are by far the most popular. Most CAD tools available in the market support these languages. VHDL stands for "very high-speed integrated-circuit hardware description language." Both VHDL and Verilog are officially endorsed IEEE (Institute of Electrical and Electronics Engineers) standards. Other HDLs include JHDL (Java HDL), and proprietary HDLs such as Cypress Semiconductor Corporation's Active-HDL.

In the 1980s, the rapid advances in IC technology necessitated a need to standardize design practices. In 1983, VHDL was developed under the VHSIC program of the U.S. Department of Defense. It was originally intended to serve as a language to document descriptions of complex digital circuits. It was also used to describe the behavior of digital circuits and could be fed to software tools that were used to simulate a circuit's operation. In 1987, IEEE adopted the VHDL language as standard 1076 (also referred to, as VHDL-87). It was revised in 1993 as the standard VHDL-93. Verilog HDL and a simulator were released by Gateway Design
Automation in 1983. In 1989, Cadence Design Systems acquired Gateway Design Automation.

In 1990, Cadence separated the HDL from its simulator (Verilog-XL) and released the HDL into the public domain. Verilog HDL is guarded by the Open-Verilog International Organization, now part of Accelera Organization. In 1995, IEEE adopted Verilog HDL as standard 1364. Hardware description languages, including VHDL, are used to program PLDand FPGA-based systems. The Altera and Xilinx corporations provide free limited versions (for educational purposes) of CAD software and tools, which can be used to program FPGA-based development boards. The CAD tools include a schematic editor, a VHDL/Verilog editor, compilers, libraries, design simulators, and various utilities tools. VHDL ["VHSIC hardware description language" (VHSIC is "very high-speed integrated circuit")] is one of the two most popular HDLs, the other being Verilog HDL. In this lab course, we restrict to Verilog

Introduction to Verilog

In verilog, the basic unit of hardware is called module. Modules cannot contain the definitions of other module A Module can, however, be instantiated with in another module which allows the creation of a hierarchy in a Verilog description. A Semicolon acts as a Statement terminator. Commenting out code is done as following. A double slash // for a Single line commenting and /* */ for Multi-line comment can be used. Verilog is a Case sensitive language. This section is a brief introduction to hardware design using a Hardware Description Language (HDL). A language describing hardware is quite different from C, Pascal, or other software languages. A computer program is dynamic, i.e., sharing the same resources, allocating resources when needed and not always optimized for maximum speed, optimal memory management, or lowest resource requirements. The main focus is functionality, but it is still not uncommon that software programs can behave quite unexpected. When problems arise, new versions of the programs are distributed by the vendor, usually with a new version number and a higher price tag. The demands on hardware design are high compared to software. Often it is not possible, or at least very tricky, to patch hardware after fabrication. Clearly, the functionality must be correct and in addition how the code is written will affect the size and speed of the resulting hardware. Each mm2 of a chip costs money, lots of money. The amount of logic cells, memory blocks and input/output connections will affect the size of the design and therefore also the manufacturing cost. A software designer using a HDL has to be careful. The degrees of freedom compared with software design have dramatically increased and must be taken into account.

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. Syntactic differences include: required bit-widths for variable declarations, demarcation of procedural blocks (Verilog uses begin/end instead of curly braces {}), and many other minor differences. Verilog requires that variables be given a definite size. In C these sizes are assumed from the 'type' of the variable (for instance an integer type may be 8 bits).

A Verilog design consists of a **hierarchy of modules**. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language is synthesizable. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip- flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bitstream file for an FPGA).

HDL simulators are better than gate level simulators for 2 reasons: portable model development, and the ability to design complicated test benches that react to outputs from the model under test. Finding a model for a unique component for your particular gate level simulator can be a frustrating task; with an HDL language you can always write your own model. Also most gate level simulators are limited to simple waveform based test benches which complicate the testing of bus and microprocessor interface circuits.

- **Verilog** is a great low level language. Structural models are easy to design and Behavioral RTL code is pretty good. The syntax is regular and easy to remember. It is the fastest HDL language to learn and use. However Verilog lacks user defined data types and lacks the interface-object separation of the VHDL's entity- architecture model.

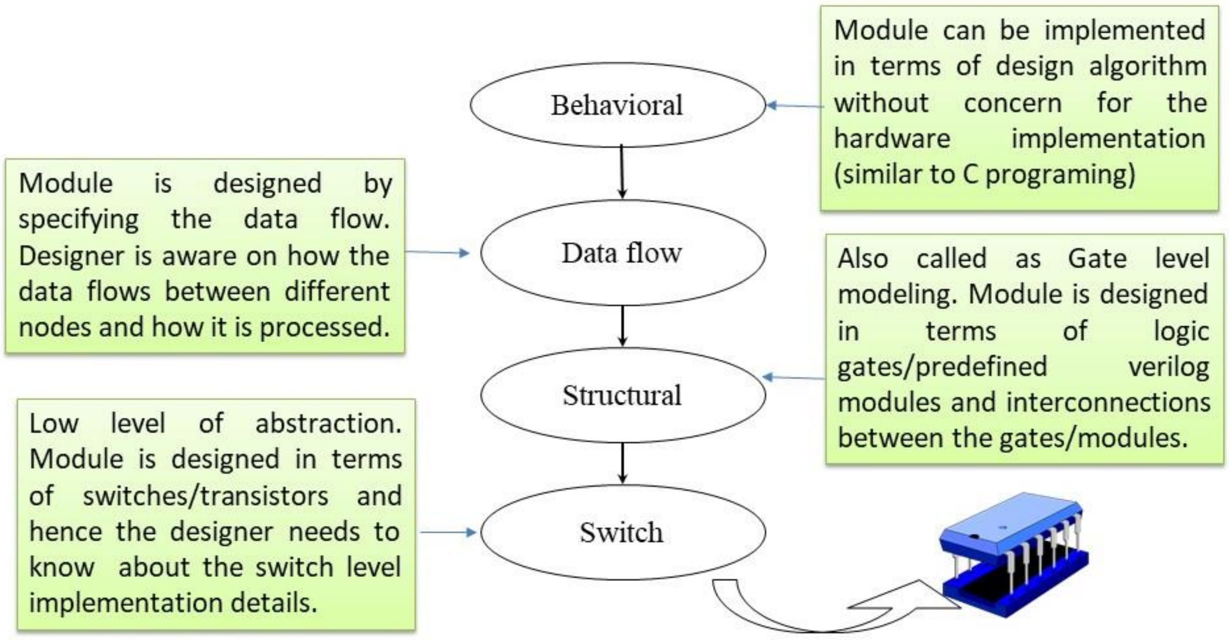
- **VHDL** is good for designing behavioral models and incorporates some of the modern object oriented techniques. Structural models require a lot of code that interferes with the readability of the model.

A Verilog input file consists of the following segments:

1. ***Header:*** module name, list of input and output ports.
2. ***Declarations:*** input and output ports, registers and wires.
3. ***Logic Descriptions:*** equations, state machines and logic functions.
4. ***End:*** endmodule

All your designs for this lab must be specified in the above Verilog input format. Note that the *state diagram* segment does not exist for combinational logic designs.

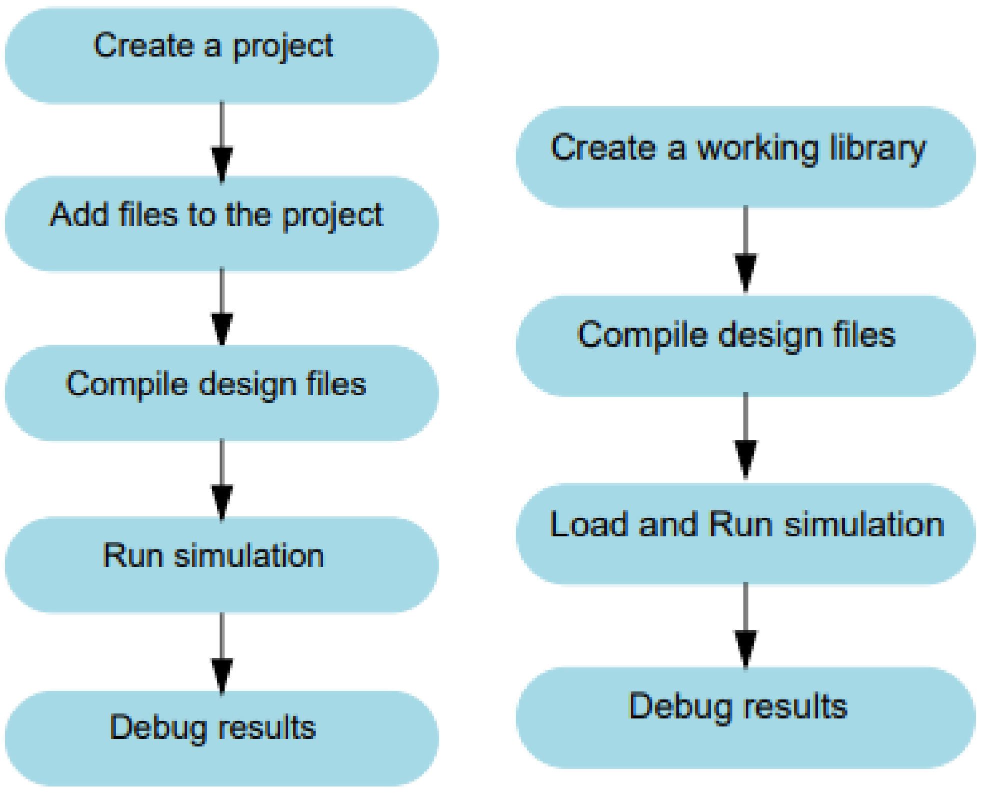
Levels of Abstraction:


Behavioral
Data flow
Structural
Switch
Module can be implemented in terms of design algorithm without concern for the hardware implementation (similar to C programing)
Module is designed by specifying the data flow. Designer is aware on how the data flows between different nodes and how it is processed.
Also called as Gate level modeling. Module is designed in terms of logic gates/predefined verilog modules and interconnections between the gates/modules.
Low level of abstraction. Module is designed in terms of switches/transistors and hence the designer needs to know about the switch level implementation details.

Procedure to Work with Modelsim Altera

ModelSim is a simulation verification and simulation tool by Siemens (previously developed by Mentor Graphics) for designs developed using the hardware description languages such as VHDL, Verilog, System Verilog. ModelSim can be used independently, or in conjunction with Intel Quartus Prime. Simulation is performed using the graphical user interface (GUI), or automatically using scripts Tool: https://tinyurl.com/modelsimtool

Work Flow in Modelsim

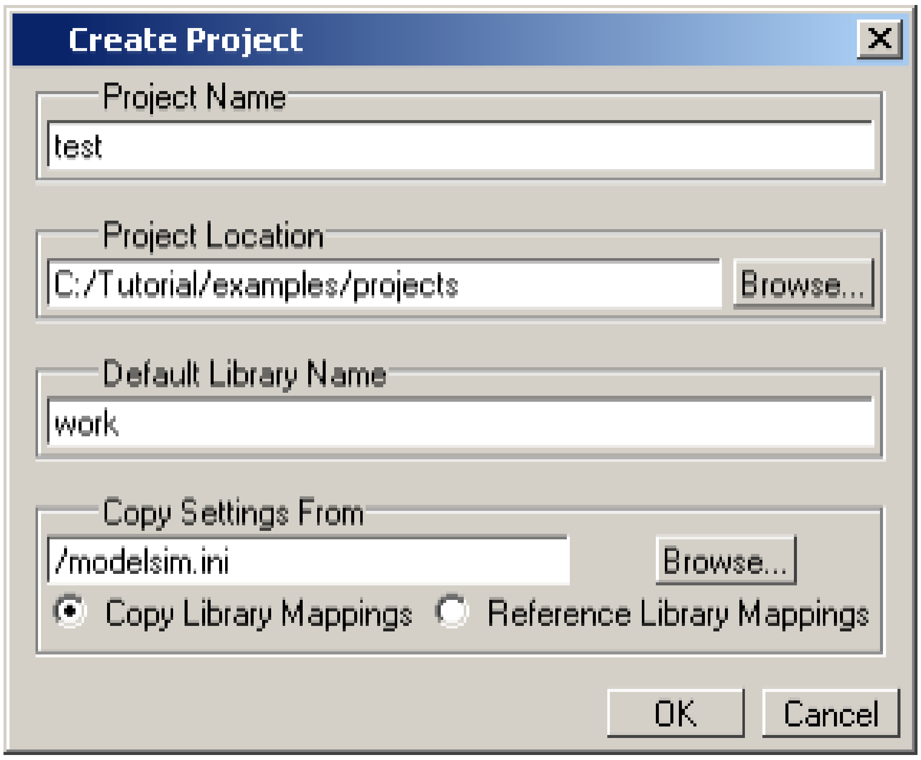


Project Flow
In ModelSim, all designs are compiled into a library. You typically start a new simulation in ModelSim by creating a working library called "work," which is the default library name used by the compiler as the default destination for compiled design units.

• **Compiling Your Design**
After creating the working library, you compile your design units into it. The ModelSim library format is compatible across all supported platforms. You can simulate your design on any platform without having to recompile your design.
• **Loading the Simulator with Your Design and Running the Simulation**
With the design compiled, you load the simulator with your design by invoking the simulator on a top-level module.
Assuming the design loads successfully, the simulation time is set to zero, and you enter a run command to begin simulation.
• **Debugging Your Results**
If you don't get the results you expect, you can use ModelSim's robust debugging environment to track down the cause of the problem.

Create a New Project

Select File > New > Project (Main window) from the menu bar.\
This opens the Create Project dialog where you can enter a Project Name, Project Location (i.e., directory), and Default Library Name


Create Project

Project Name: test

Project Location: C:/Tutorial/examples/projects

Default Library Name: work

Copy Settings From: /modelsim.ini

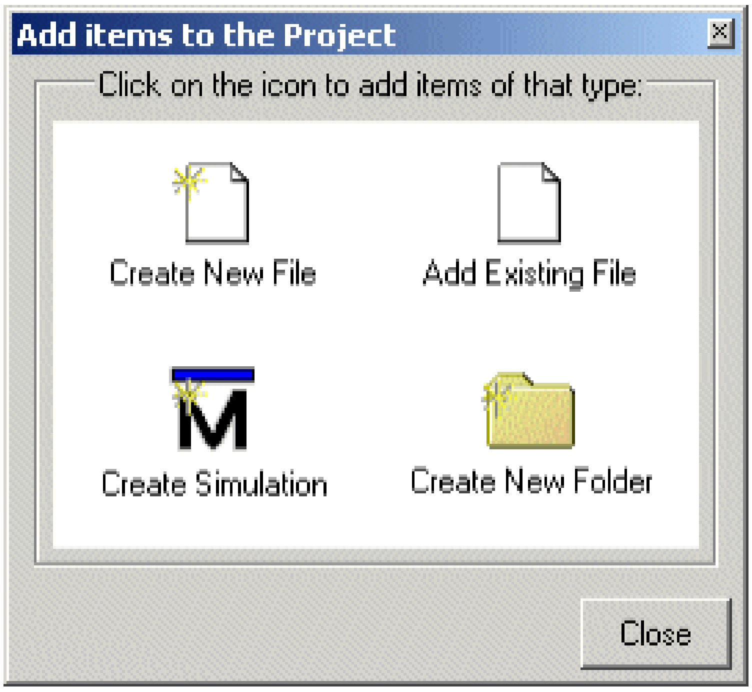
[✓ Copy Library Mappings] Reference Library Mappings

➢ Type the Project Name

- Click the Browse button for the Project Location field to select a directory where the project file will be stored
- Leave the Default Library Name set to work.
- Click OK.

Create a New File

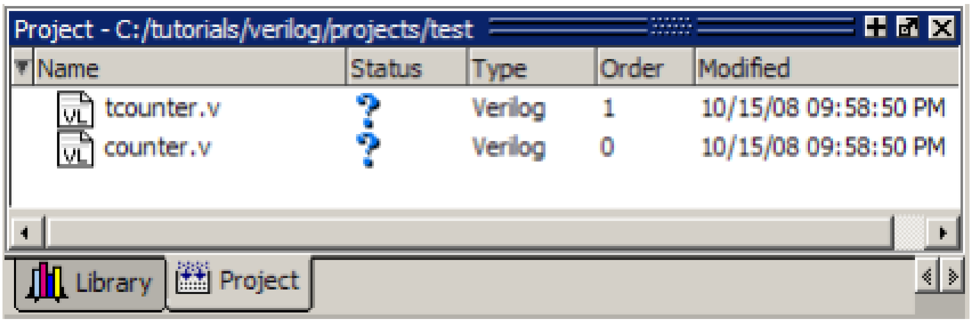
- Once you click OK to accept the new project settings, a blank Project window and the "Add items to the Project" dialog will appear



- Type the File name to be created
- Select Add file as type as **Verilog** from the drop down menu (By default it will be VHDL).
- Click OK.
- The new file will be added to the project. It can be selected and the design can be typed in the editor.

Compilation and Simulation

- New files show the status as ?



Project - C:/tutorials/verilog/projects/test

Name	Status	Type	Order	Modified
tcounter.v	?	Verilog	1	10/15/08 09:58:50 PM
counter.v	?	Verilog	0	10/15/08 09:58:50 PM

- Select the file, right click – compile – Compile selected.
- If compilation is successful, status changes to green colour tick mark.
- Go to work library, select the module to be simulated, right click and select simulate.
- Force the inputs and check the outputs in the wave window

Verilog Identifier

User-provided names for Verilog objects in the descriptions
Legal characters are "a-z", "A-Z", "0-9", "_", and "$"
First character has to be a letter or an "_"
Example: Count, _R2D2, FIVE$

Verilog Keywords

Predefined identifiers to define the language constructs
All keywords are defined in lower case
Cannot be used as identifiers
Example: initial, assign, module, always….

Verilog Ports

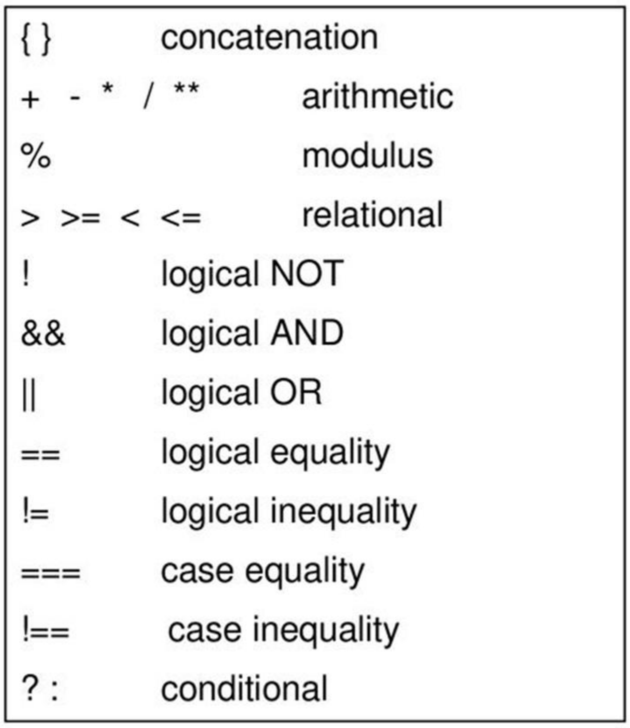
Three types of ports

- Input (keyword - input)
- Output (keyword – output)
- Inout (keyword – inout)

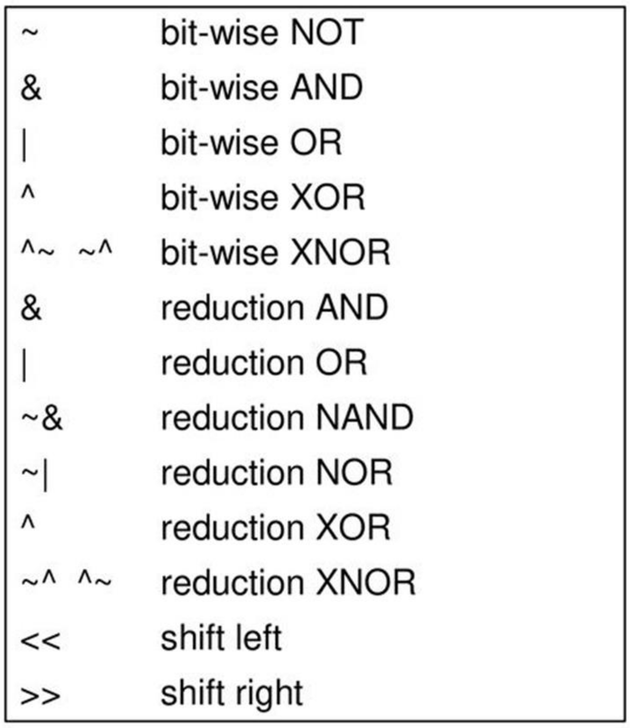
Port declarations example

```
input a;
input a, b;
input [3:0] c, d;
output [4:0] y;
inout x;
```

Verilog Operators



{ }	concatenation
+ - * / **	arithmetic
%	modulus
> >= < <=	relational
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
? :	conditional



~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~ ~^	bit-wise XNOR
&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^ ^~	reduction XNOR
<<	shift left
>>	shift right

Verilog Data Types

Two basic data types

- Nets and
- Registers

Nets represent physical wires in the design.(**wire, tri**)

- Default initial value for a wire is “Z”

Registers represent storage in the Verilog model. (**reg, integer, real, time**)

- Register data type may or may not result in physical registers.
- Registers are manipulated within procedural blocks (***always*** and ***initial***) only.
- Default initial value for a reg is “X”

Verilog Numbers

Constants:

14 ordinary decimal number
-14 2’s complement representation
12’b0000_0100_0110 binary number (“_” is ignored)
12’h046 hexadecimal number with 12 bits

Signal Values:

By default, Values are unsigned
e.g., **C[4:0] = A[3:0] + B[3:0];**
if A = 0110 (6) and B = 1010 (treated as 10 not -6)
C = 10000 not 00000
i.e., B is zero-padded, not sign-extended
wire signed [31:0] x;
Declares a signed (2’s complement) signal array

Expt. 1 **Design and test Verilog implementation for a Half Adder**

Objective :

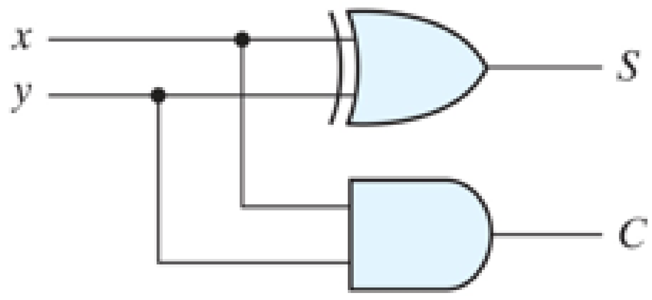
1. Understand what is half adder with truth table
2. Understanding how to create a new project in Modelsim, new Verilog file, compiling and running the codes and seeing the output

Theory:

Half Adder is a combinational arithmetic circuit that adds two binary numbers and produces sum bit (S) and carry bit (C) as the output. It adds 2 single-bit binary numbers.
Implementation in terms of logic gates is as the following.

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table



$$S = x \oplus y$$
$$C = xy$$

Verilog Code

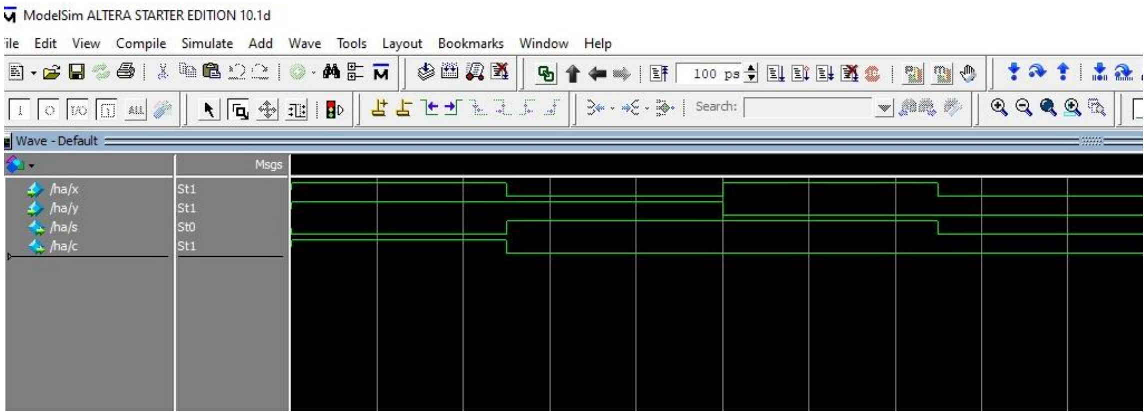
Gate Level Modeling

```
//Half Adder : Structural Verilog Description
module ha_gate_level (x,y,s,c);
  input x,y;
  output s,c;
  xor x1(s, x, y);
  and a1(c, x, y);
endmodule
```

Dataflow Modeling

```
//Half Adder : Dataflow description
module ha_df (x,y,s,c);
  input x,y;
  output s,c;
  assign s = x^y;
  assign c = x & y;
  //assign {c,s} = x + y; //Alternate Method
endmodule
```

Output


ModelSim ALTERA STARTER EDITION 10.1d
ile Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
100 ps
Search:
Wave - Default
Msgs

```
/ha/x  St1
/ha/y  St1
/ha/s  St0
/ha/c  St1
```

After the simulation, the variables can be added to wave and forced to clock signal of varied duty cycle and period. Here 100ps and 200ps period clock signals were forced on input x and y to create various combinations of inputs that spans the entire truth table. Simulate and Run button were clicked further to see the waveforms.The output can be verified.

Conclusion

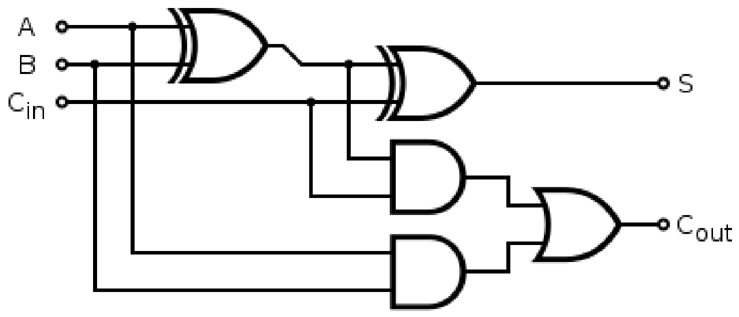
The output is observed as expected validating the design.

Expt. 2 **FULL ADDER AND FULL SUBTRACTOR DESIGN MODELING**

Objective:

1. Undestand what is full adder with truth table
2. Understand full substractor
3. Four bit parallel adder

Theory: A full adder consists of 3 inputs and 2 outputs. Fig 7.1 shows truth table of full adder. Use “assign” keyword to represent design in dataflow style. The output signal expressions can be obtained from the truth table using K-maps.



Logic diagram for 1-bit full adder

Table 5.1 Truth table for 1-bit full adder

Inputs			Outputs	
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This is not practical to perform subtraction only between two single bit binary numbers. Instead binary numbers are always multibits. The subtraction of two binary numbers is performed bit by bit from right (LSB) to left (MSB). During subtraction of same significant bit of minuend and subtrahend, there may be one borrow bit along with difference bit. This borrow bit (either 0 or 1) is to be added to the next higher significant bit of minuend and then next corresponding bit of subtrahend to be subtracted from this. It will continue up to MSB. The combinational logic circuit performs this operation is called full subtractor. Hence, full subs tractor is similar to half subs tractor but inputs in full subs tractor are three instead of two.

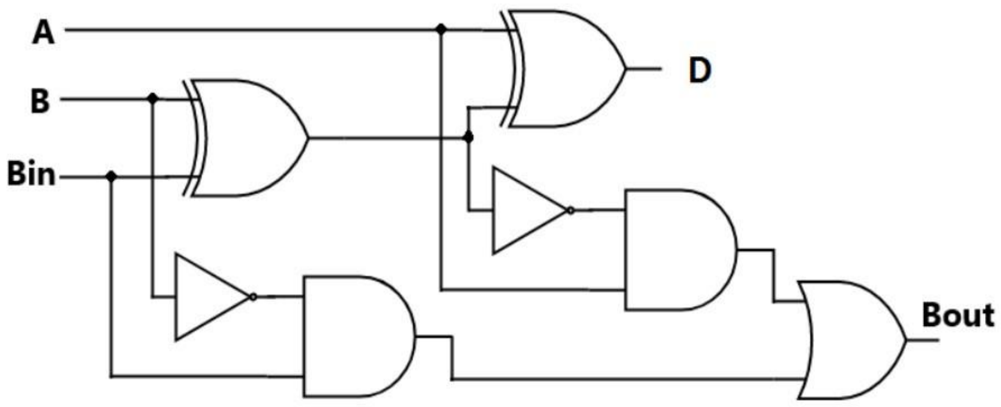
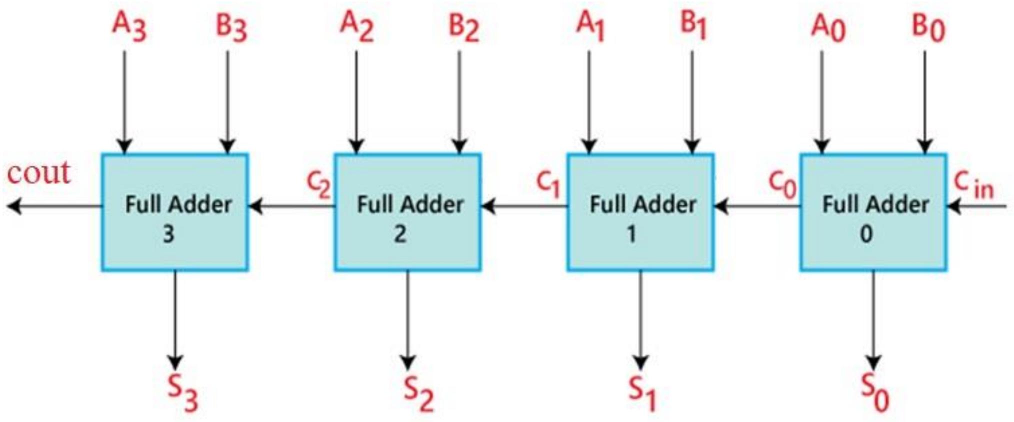
Two inputs are for the minuend and subtrahend bits and third input is for borrowed which comes from previous bits subtraction. The outputs of full adder are similar to that of half adder, these are difference (D) and borrow (b).

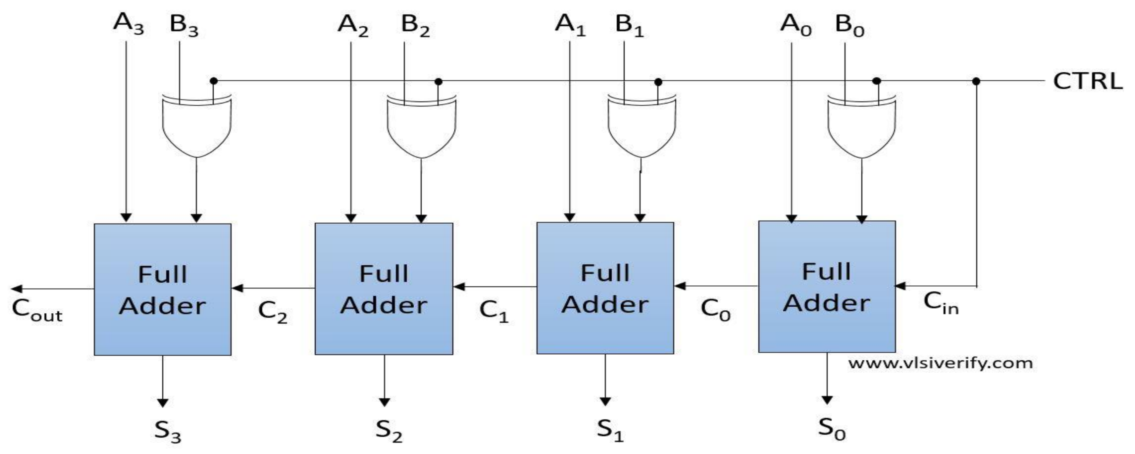
The combination of minuend bit (A), subtrahend bit (B) and input borrow (bi) and their respective differences (D) and output borrows (b) are represented in a truth table

The output signal expressions can be obtained from the truth table using K-maps.

Inputs			**Outputs**	
A	B	Bin(Borrow in)	Difference	Borrow out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Truth table for 1-bit subtractor adder

	$D = A \oplus B \oplus Bin$ $Bout = A'\,Bin + A'\,B + B\,Bin$   A3 B3 A2 B2 A1 B1 A0 B0 cout Full Adder 3 C2 Full Adder 2 C1 Full Adder 1 C0 Full Adder 0 Cin S3 S2 S1 S0 Four Bit Parallel Adder



4-Bit Adder Subtractor

Procedure:

1. Create a module with required number of variables and mention it’s input/output.
2. Write the description of the full adder in 3 styles.
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained outputwith the required one.

code

Full adder Structural/Gate Level Modeling

```
module fa1(s,cout,a,b,cin);
input a,b,cin;
output s, cout;
wire w1, w2, w3;
xor x1(w1,a,b);
xor x2(s,w1,cin);
and a1(w2,a,b);
and a2(w3,w1,cin);
or o1(cout,w2,w3);
endmodule
```

Full adder Behavioral Modeling

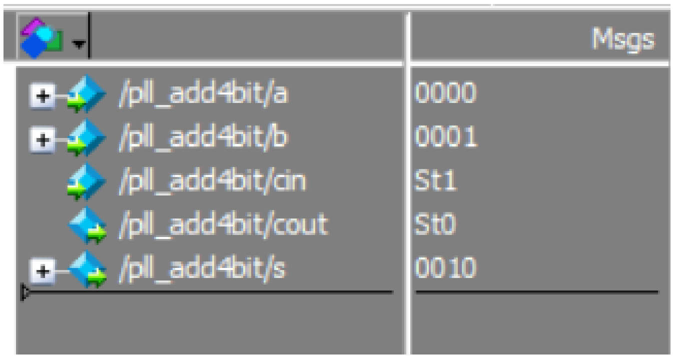
```
module fa(sum,cout,a,b,cin);
input a,b,cin;
  output sum,cout;
  assign {cout,sum}= a+b+cin;
  endmodule
```

Full adder/Substractor with control line

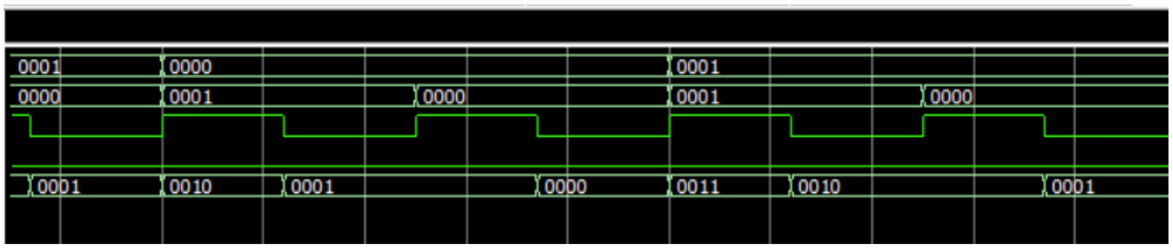
```
module paaddsub(a,b,cntrl,sum,cout);
 input [3:0]a,b;
input cntrl;
output [3:0] sum;
output cout;
wire t1,t2,t3,t4,c0,c1,c2;
xor x1(t1,b[0],cntrl);
xor x2(t2,b[1],cntrl);
xor x3(t3,b[2],cntrl);
xor x4(t4,b[3],cntrl);
fa1 f1(sum[0],c0,a[0],t1,cntrl);
fa1 f2(sum[1],c1,a[1],t2,c0);
fa1 f3(sum[2],c2,a[2],t3,c1);
fa1 f4(sum[3],cout,a[3],t4,c2);
endmodule
```

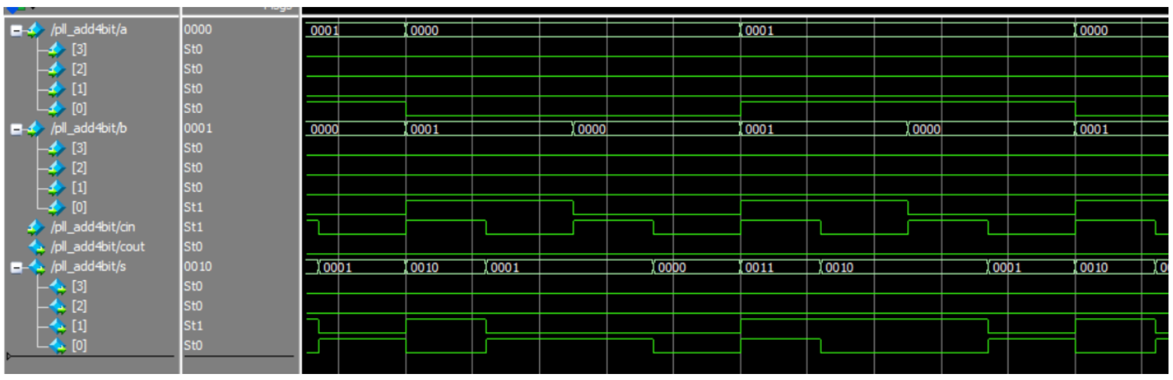
<u>Output</u>

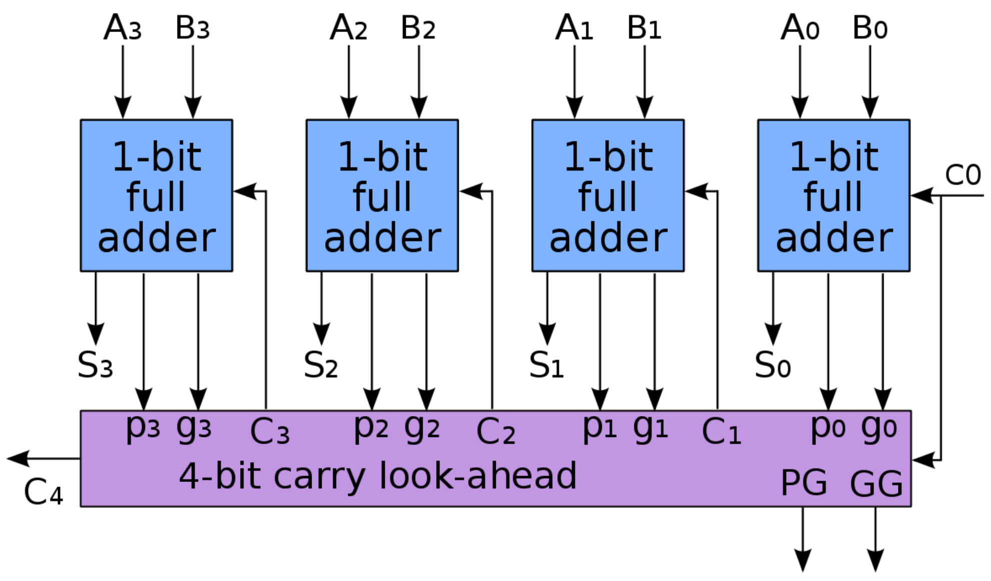
1.



	Msgs
/pll_add4bit/a	0000
/pll_add4bit/b	0001
/pll_add4bit/cin	St1
/pll_add4bit/cout	St0
/pll_add4bit/s	0010



2.	 /pll_add4bit/a 0000 [3] St0 [2] St0 [1] St0 [0] St0 /pll_add4bit/b 0001 [3] St0 [2] St0 [1] St0 [0] St1 /pll_add4bit/cin St1 /pll_add4bit/cout St0 /pll_add4bit/s 0010 [3] St0 [2] St0 [1] St1 [0] St0
	The module takes four inputs: a, b, and cin (the carry-in), and generates two outputs: s (the sum) and cout (the carry-out). The intermediate signals c1, c2, and c3 are used to carry the carry-out from one full adder cell to the next. The full adder cells are instantiated as instances of the full_adder module, with the appropriate inputs and outputs connected.
Conclusion	The output is observed as expected validating the design.

Expt. 3	**4 bit carry look ahead adder**
Objective:	Design and Simulate 4-bit carry look ahead adder
	The adder produce carry propagation delay while performing other arithmetic operations like multiplication and divisions as it uses several additions or subtraction steps. This is a major problem for the adder and hence improving the speed of addition will improve the speed of all other arithmetic operations. Hence reducing the carry propagation delay of adders is of great importance. There are different logic design approaches that have been employed to overcome the carry propagation problem. One widely used approach is to employ a carry look-ahead which solves this problem by calculating the carry signals in advance, based on the input signals. This type of adder circuit is called a carry look-ahead adder.  A3 B3 A2 B2 A1 B1 A0 B0 1-bit full adder C0 S3 S2 S1 S0 p3 g3 C3 p2 g2 C2 p1 g1 C1 p0 g0 4-bit carry look-ahead C4 PG GG The inputs to the module are two 4-bit arrays a and b, and the outputs are a 4-bit array s (the sum) and a scalar cout (the carry-out). The intermediate signals g0, g1, g2, and g3 represent the individual bitwise AND operations of

a and b, and the signals p0, p1, p2, and p3 represent the individual bitwise EX-OR operations of a and b. The final expression for cout computes the carry-out based on the intermediate signals, using the carry lookahead logic. Sum is the ex-or of p and c.

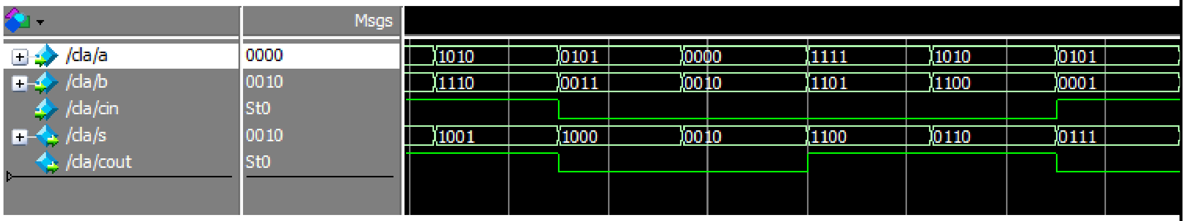
$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

Code

```
module cla(input [3:0] a, input [3:0] b, input cin, output [3:0] s, output cout);
  wire [3:0] g, p, c;
  assign g = a & b;
  assign p = a ^ b;
  assign c[0] = cin;
  assign c[1] = g[0] | (p[0] & c[0]);
  assign c[2] = g[1] | (p[1] & c[1]);
  assign c[3] = g[2] | (p[2] & c[2]);
  assign cout =g[3] | (p[3] & c[3]);
  assign s = p ^ c;
endmodule
```

out put.

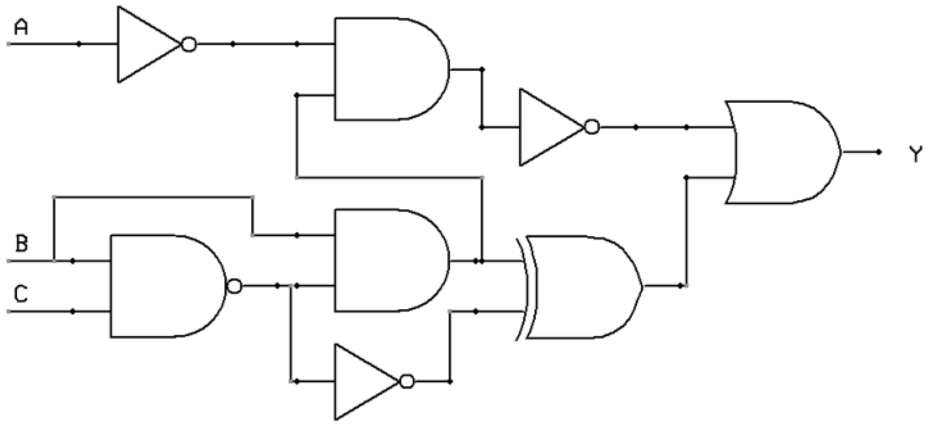


```
            Msgs
/da/a       0000   1010  0101  0000  1111  1010  0101
/da/b       0010   1110  0011  0010  1101  1100  0001
/da/cin     St0
/da/s       0010   1001  1000  0010  1100  0110  0111
/da/cout    St0
```

Conclusion	Output of the carry look ahead adder is verified.
2.	

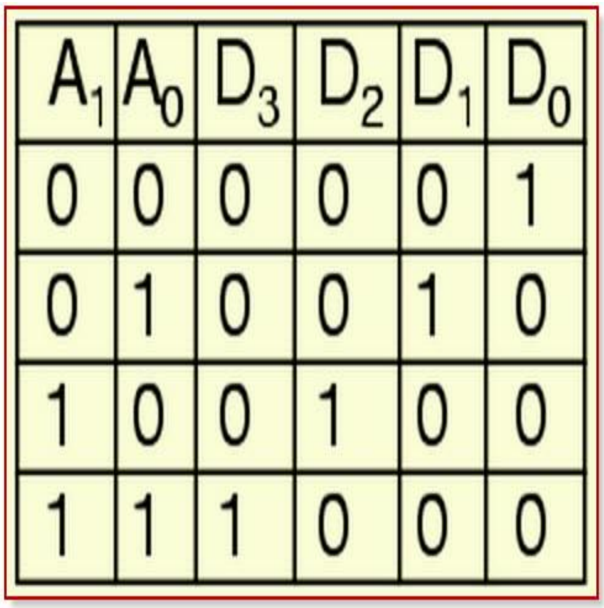
Additional Exercise:

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



Implement the above combinatory circuit by structural Modelling

Expt. 4	**DESIGN OF DECODER AND ENCODER**
Objective :	To design and simulate the HDL code for the following combinational circuits a. 2 to 4 Decoder b. 4 to 2 Encoder
Theory:	**Program logic for Decoder** A decoder is a multiple-input, multiple-output logic circuit which converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code. Each input code word produces a different output code word, i.e., there is one-to-one mapping from input code words into output code words. This one-to-one mapping can be expressed in a truth table. The most common decoder circuit is an n-to-2^n decoder or binary decoder. Sucha decoder has an n-bit binary input code and a 1-out-of-2^n output code. A binary decoder is used when you need to activate exactly one of 2^noutputs based on an n-bit input value. Figure shows the general structure of the 2 to 4 decoder circuit and its truth table.



A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

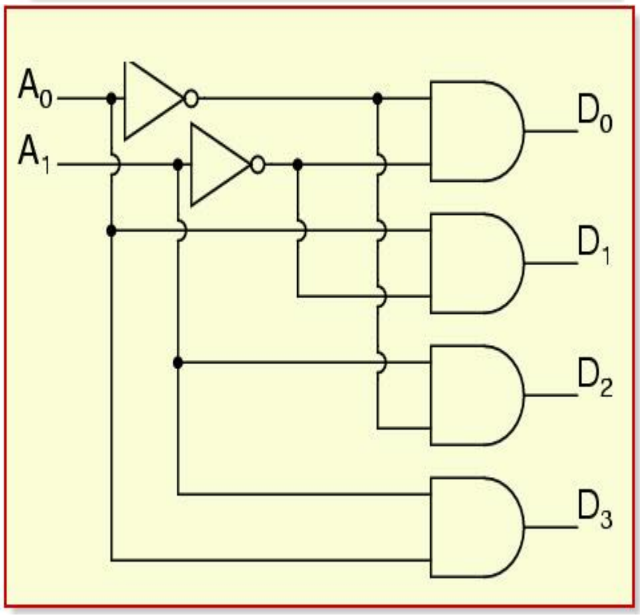
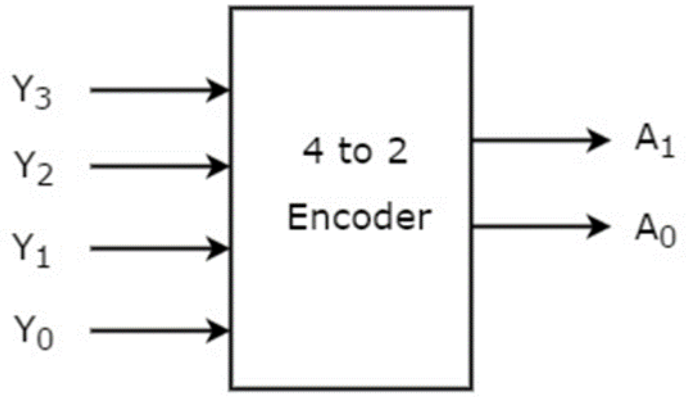
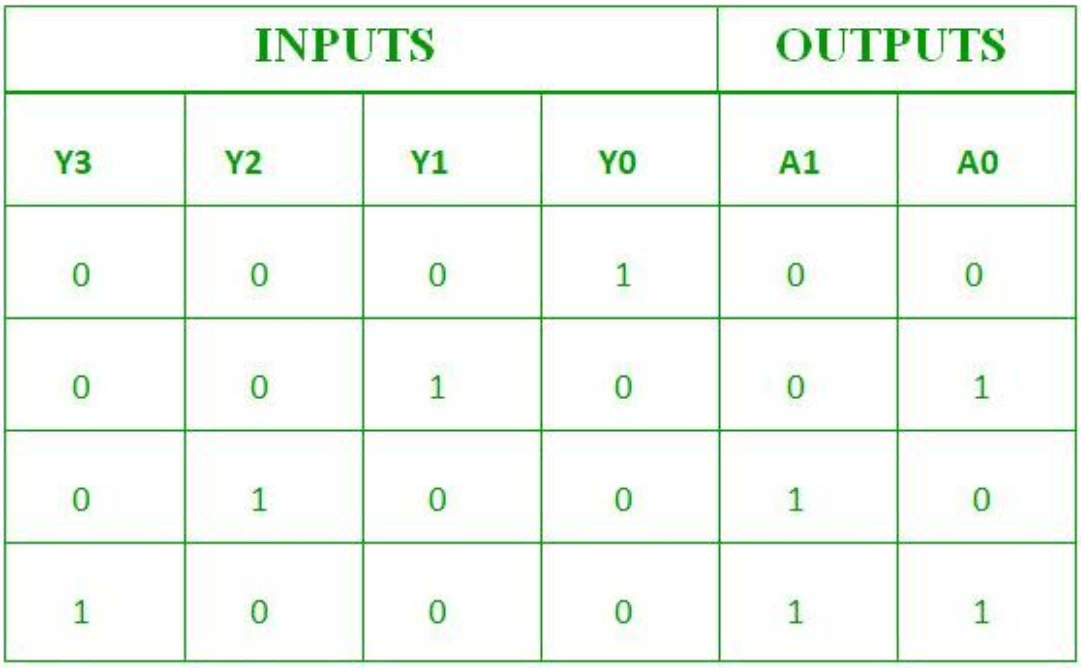


Figure: General Structure of 2 to 4 Decoder and its truth table

a. Program logic for Encoder

An encoder has M input and N output lines. Out of M input lines only one is activated at a time and produces equivalent code on output N lines. If a device output code has fewer bits than the input code has, the device is usually called an encoder. Example Octal-to-Binary take 8 inputs and provides 3 outputs. For an 8-to-3 binary encoder with inputs D0-D7 the logic expressions of the outputs XYZ are obtained by using the Table 3.1.

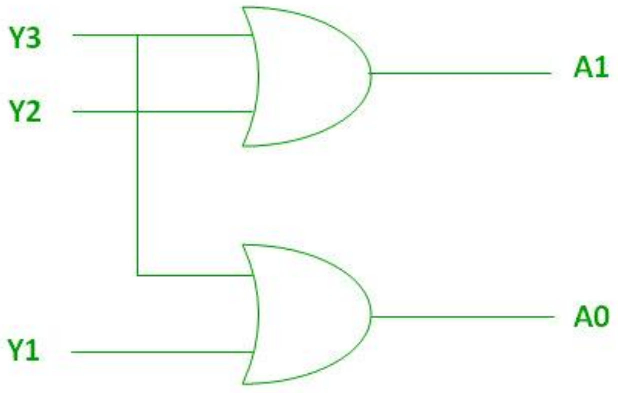




INPUTS				OUTPUTS	
Y3	**Y2**	**Y1**	**Y0**	**A1**	**A0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Logical expression for A1 and A0 :

```
A1 = Y3 + Y2
A0 = Y3 + Y1
```



X = D4 + D5 + D6 + D7Y= D2 + D3 + D6

+ D7 Z = D1 + D3 + D5 + D7

Table 3.1: Truth Table for 8-3 Encoder with D7-D0 inputs

INPUTS								**OUTPUTS**		
D7	**D 6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**X**	**Y**	**Z**
O0	0	0	0	0	0	0	1	0	0	0
n 0	0	0	0	0	0	1	0	0	0	1
e 0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
o 0	0	0	1	0	0	0	0	1	0	0
f 0	0	1	0	0	1	0	0	1	0	1
0	1	0	0	0	0	1	0	1	1	0
O1	0	0	0	0	0	0	1	1	1	1

One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level “1”.

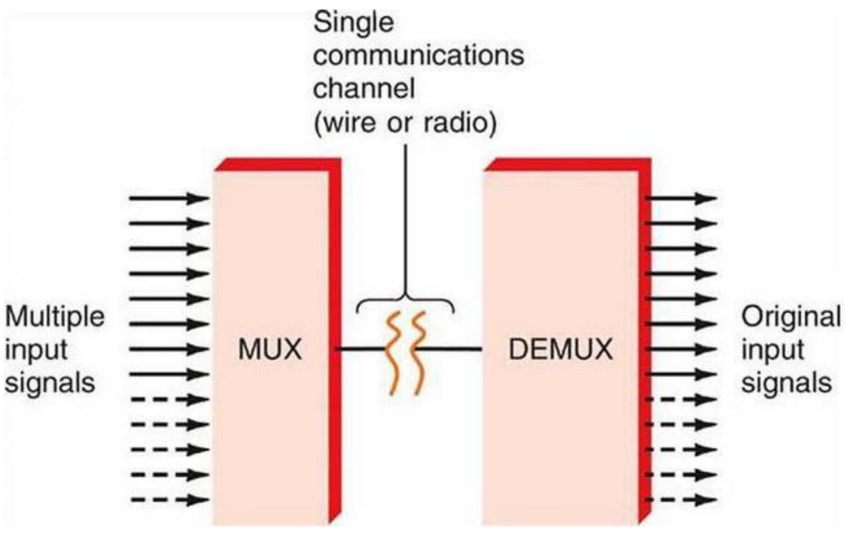
The Priority Encoder solves the problems mentioned above by allocating a priority level to each input. The priority encoders output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. The priority encoder comes in many different forms with an example of an 8- input priority encoder along with its truth table shown above.

	Decoder or encoder can be designed using HDL through its truth table in two ways: one is using gate level modeling and another is by behavioral model
PROCEDURE	1. Create a module with required number of variables and mention it's input/output. 2. Implement the logic for decoder or encoder using behavioral or gate level model. 3. Create another module referred as test bench to verify the functionality. 4. Follow the steps required to simulate the design and compare the obtained outputwith the corresponding truth table.

Code	**Verilog Code Decoder**
	Behavioral Level
	
	module decoder2to4(en,a,b,y);
	 input en,a,b;
	output reg [3:0]y;
	always @(en,a,b)
	Begin
	 if(en==0)
	begin
	if(a==1'b0 & b==1'b0) y=4'b1110;
	else if(a==1'b0 & b==1'b1) y=4'b1101;
	else if(a==1'b1 & b==1'b0) y=4'b1011;
	else if(a==1 & b==1) y=4'b0111;
	else y=4'bxxxx;
	end
	else
	y=4'b1111;
	end
	endmodule
	
	Data Flow Modeling
	
	module decoder24_data(en,a,b,y);
	    input en,a,b;
	    output [3:0]y;
	    wire enb,na,nb;
	    assign enb = ~en;

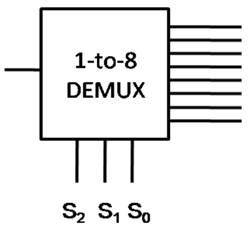
```
    assign na = ~a;
    assign nb = ~b;
    assign y[0] = ~(enb&na&nb);
    assign y[1] = ~(enb&na&b);
    assign y[2] = ~(enb&a&nb);
    assign y[3] = ~(enb&a&b);
endmodule




module encod4to2(d0,d1,d2,d3,q0,q1);
output q0,q1;
input d0,d1,d2,d3;
assign q0 = d1 | d3;
assign q1 = d3 | d2;
endmodule


module encod4to2b(dout,din);
    input [3:0]din;
    output [1:0]dout ;
reg [1:0]dout;
always @ (din)
case (din)
    4'b0001 : dout = 2'b00;
    4'b0010 : dout = 2'b01;
    4'b0100 : dout = 2'b10;
    4'b1000 : dout = 2'b11;
    default : dout = 2'bxx;
endcase
endmodule
```

Addition Design	Design a 4 bit priority encoder with D0 with the hightest priority, then D1, then D2 and D3.

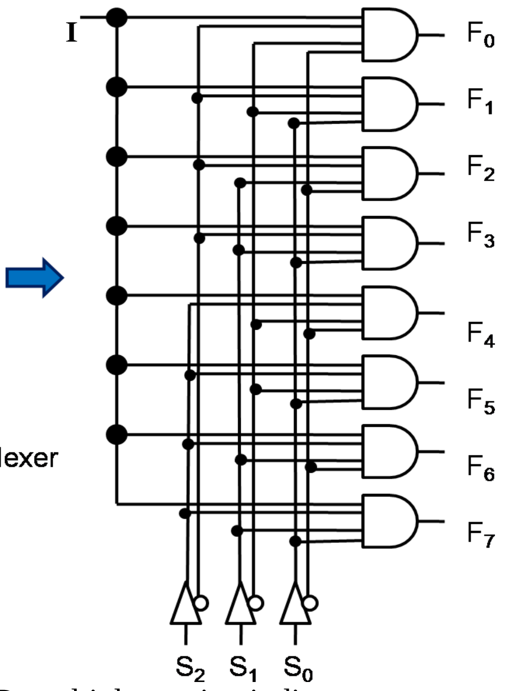
Expt. 5	**DESIGN OF MULTIPLEXER AND DEMULTIPLEXER**
Objective:	To write Verilog codes for an 4X1 multiplexer and 1X4 demultiplexer and verify its functionality.
Theory:	In the large-scale-digital systems, a single line is required to carry on two or more digital signals – and, of course! At a time, one signal can be placed on the one line. But, what is required is a device that will allow us to select; and, the signal we wish to place on a common line, such a circuit is referred to as multiplexer. The function of a multiplexer is to select the input of any 'n' input lines and feed that to one output line. The function of a de-multiplexer is to inverse the function of the multiplexer and the shortcut forms of the multiplexer. The de-multiplexers are mux and demux. Some multiplexers perform both multiplexing and de-multiplexing operations. The main function of the multiplexer is that it combines input signals, allows data compression, and shares a single transmission channel.  Single communications channel (wire or radio) Multiple input signals MUX DEMUX Original input signals Multiplexer and De-multiplexer The output value of a 8x1 multiplexer can be represented using the equation For the combination of selection input, the data line is connected to the output line. The 8x1 multiplexer requires 8 AND gates, one OR gate and 3 selection lines. As an input, the combination of selection inputs are giving to the AND gate with the corresponding input data lines.

In a similar fashion, all the AND gates are given connection. In this 8x1 multiplexer, for any selection line input, one AND gate gives a value of 1 and the remaining all AND gates give 0. And, finally, by using OR gate, all the AND gates are added; and, this will be equal to the selected value.

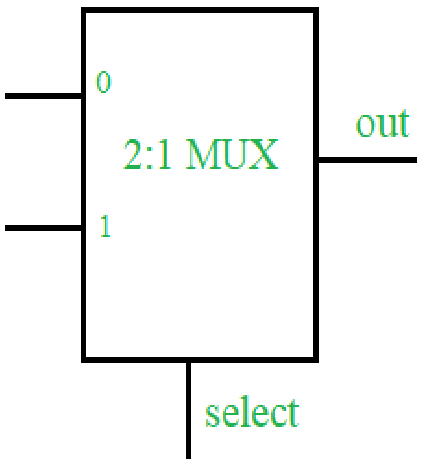
The demultiplexer is also called as data distributors as it requires one input, 3 selected lines and 8 outputs. De-multiplexer takes one single input data line, and then switches it to any one of the output line. 1-to-8 demultiplexer circuit diagram is shown below; it uses 8 AND gates for achieving the operation. The input bit is considered as data D and it is transmitted to the output lines.

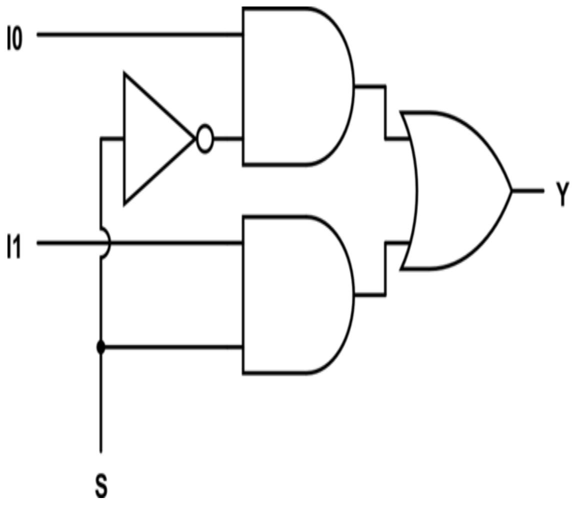


1-to-8 line demultiplexer

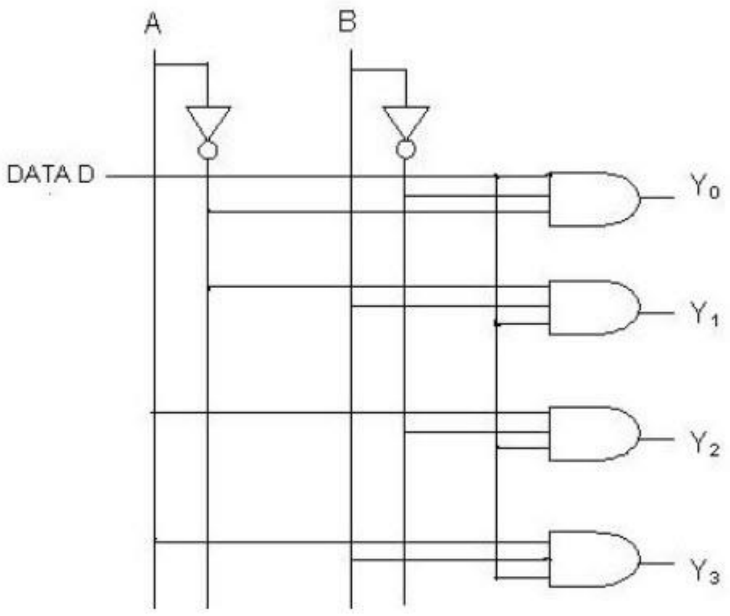


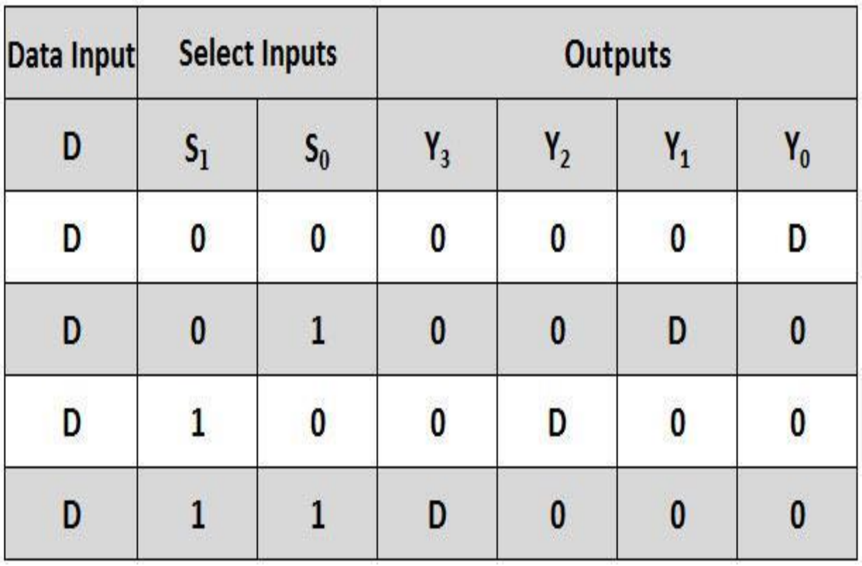
Demultiplexer circuit diagram





Out
I0
I1





Data Input	Select Inputs		Outputs			
D	S_1	S_0	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	D
D	0	1	0	0	D	0
D	1	0	0	D	0	0
D	1	1	D	0	0	0

1. Create a module with required number of variables and mention it's input/output.
2. Write the description of the multiplexer or demultiplexer using data flow model or gate level model
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained output with the corresponding truth table

Gate Level Implementation for 2×1 Multiplexer

```
module mux2to1(y,i0,i1,s);
  output y;
  input i0,i1,s;
  wire w1,w2,sbar;
  and a1(w1,i0,sbar);
  not n1(sbar,s);
  and a2(w2,s,i1);
  or o1(y,w1,w2);
endmodule
```

Data Flow Implementation for 2×1 Multiplexer

```
module m2to1d(i0, i1, s, y);
output y;
input i0, i1, s;
assign y= (~s&i0)|(s&i1);
endmodule
```

Behavioral Implementation for 2×1 Multiplexer

```
module m2to1d(i0, i1, s, y);
output y;
input i0, i1, s;
reg y;
always @(s)
y= (~s&i0)|(s&i1);
endmodule
```

Data Flow Implementation for 4×1 Multiplexer

```
module four_mul (input a, b, c, d, sel, output y);
  assign y = (sel == 0) ? a : (sel == 1) ? b : (sel == 2) ? c : d;
endmodule

module fourmul_dataflow (input a, b, c, d, s0,s1, output y);
  wire w0, w1, w2, w3;
  assign y= (~s1 & ~s0& a) | (~s1 & s0 & b) |
              (~s1 & ~s0& c) | (~s1 & s0 & d) ;
endmodule
```

Behavioral Implementation for 1×4 De-multiplexer

```
module demux1to4(y,a,din);
output reg [3:0] y;
input [1:0] a;
input din;
always @(a,din)
begin
case (a)
2'b00 : begin y[0] = din; y[3:1] = 0; end
2'b01 : begin y[1] = din; y[0] = 0;   end
2'b10 : begin y[2] = din; y[1:0] = 0; end
2'b11 : begin y[3] = din; y[2:0] = 0; end
endcase
end
endmodule
```

Dataflow Behavioral Implementation for 1×4 De-multiplexer

```
module demux1to4d(y,a,din);
output [3:0] y;
input [1:0] a;
input din;
assign y[0] = din & (~a[0]) & (~a[1]);
assign y[1] = din & (~a[1]) & a[0];
assign y[2] = din & a[1] & (~a[0]);
assign y[3] = din & a[1] & a[0];
endmodule
```

The module four_mul takes four input signals a, b, c, d and a 2-bit selection signal sel, and generates a single output signal y.

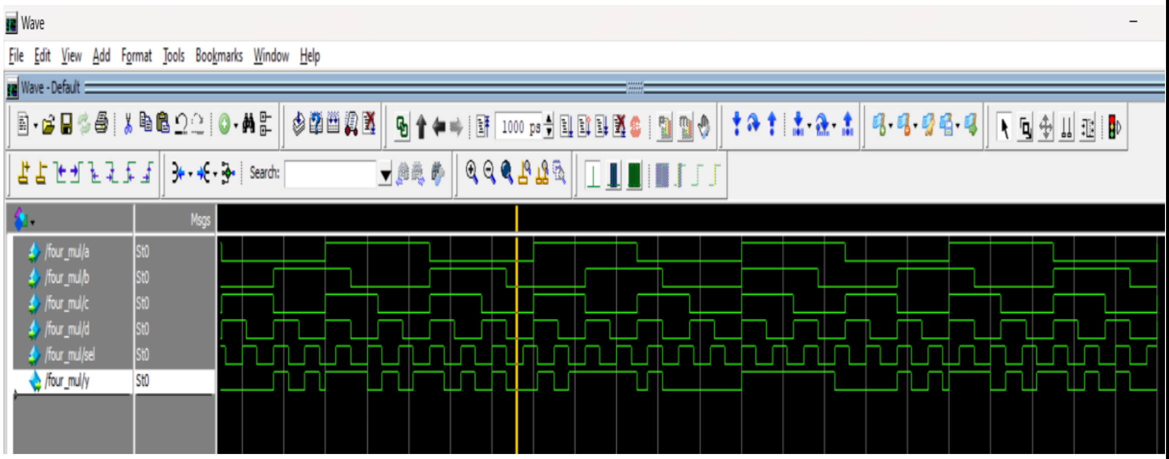
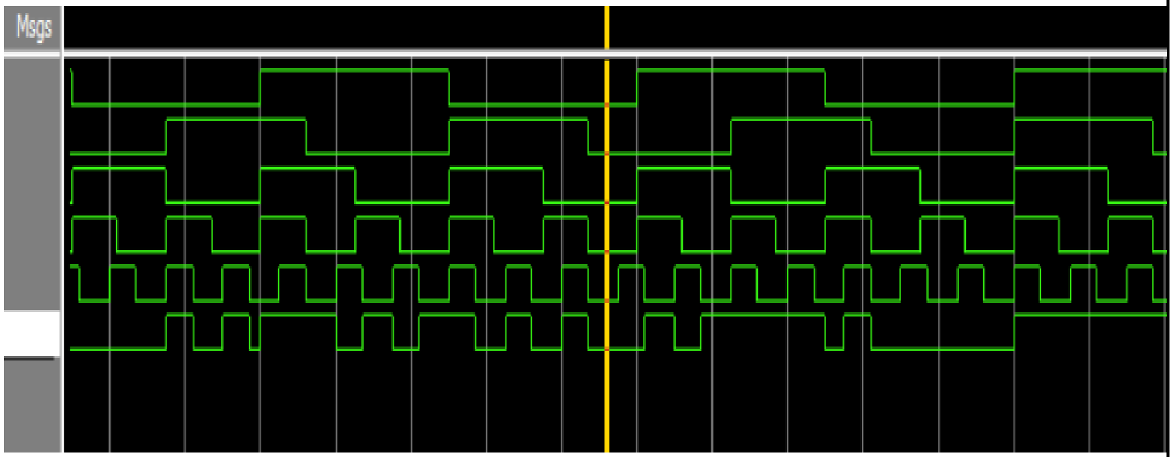
The assign statement assigns the output y to the expression (sel==0)?a:(sel==1)?b:(sel==2)?c:d, which selects one of the four input signals based on the value of sel.

The conditional operator ?: is used to perform the selection. The first condition (sel==0) checks if the selection signal sel is equal to 0, and selects input a if true. If not, it checks the second condition (sel==1) to select input b, and so on.

The expression **(~s1 & ~s0 & a) | (~s1 & s0 & b) | (s1 & ~s0 & c) | (s1 & s0 & d)** works as follows:

(~s1 & ~s0 & a) selects input **a** when both **s1** and **s0** are zero.
(~s1 & s0 & b) selects input **b** when **s1** is zero and is **s0**one.
(s1 & ~s0 & c) selects input **c** when **s1** is one and **s0** is zero.
(s1 & s0 & d) selects input **d** when both **s1** and **s0** are one.
Note that the use of **~** operator before **s1** and **s0** in some cases means the logical NOT operation. Also, this implementation uses a single **assign** statement to describe the MUX logic, which is a typical approach for dataflow modeling.

<u>Output for the multiplexer</u>

1.	 Wave File Edit View Add Format Tools Bookmarks Window Help Wave - Default 1000 ps Search: Msgs /four_mul/a St0 /four_mul/b St0 /four_mul/c St0 /four_mul/d St0 /four_mul/sel St0 /four_mul/y St0 Zoomed View 

Expt. 6	**Verilog Implementation of FOR FLIP FLOPS**
Objective :	To write Verilog codes for SR, JK, D, T flip flops and verify its functionality

Theory:

Each flip-flop stores a single bit of data, which is emitted through the Q output on the output section side. Normally, the value can be controlled via the inputs to the input side. In particular, the value changes when the clock input, marked by a triangle on each flip-flop, rises from 0 to 1 (or otherwise as configured); on this rising edge, the value changes according to the tables below.

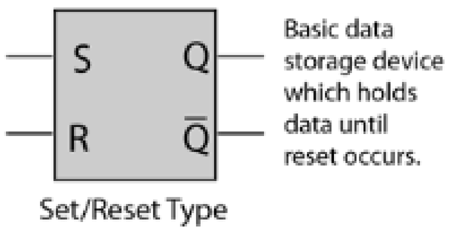
Truth tables of D, T, SR, JK flip flops

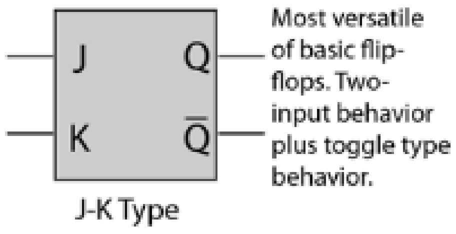
D FF	
D	**Q_n**
1	**1**
0	**0**

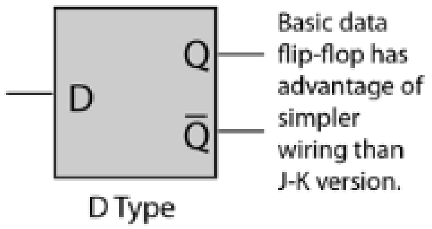
T FF	
T	**Q_n**
0	**Q**
1	Q_n

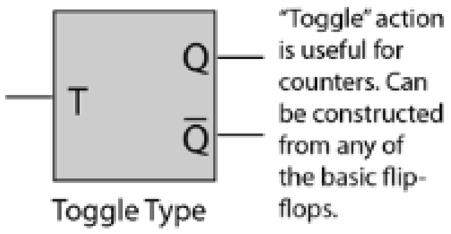
SR FF		
S	**R**	**Q_n**
0	**0**	**Q_n**
0	**1**	**0**
1	**0**	**1**
1	**1**	**?**

JK FF		
J	**K**	**Q_n**
0	**0**	**Q_n**
0	**1**	**0**
1	**0**	**1**
1	**1**	Q_n


S
R
Q
Q̄
Basic data storage device which holds data until reset occurs.
Set/Reset Type


J
K
Q
Q̄
Most versatile of basic flip-flops. Two-input behavior plus toggle type behavior.
J-K Type


D
Q
Q̄
Basic data flip-flop has advantage of simpler wiring than J-K version.
D Type


T
Q
Q̄
"Toggle" action is useful for counters. Can be constructed from any of the basic flip-flops.
Toggle Type

.

Note: Each flip flop above has a clock input which is not shown

Another way of describing the different behavior of the flip-flops is in English text.

D Flip-Flop: When the clock triggers, the value remembered by the flip-flop becomes the value of the D input (Data) at that instant.

T Flip-Flop: When the clock triggers, the value remembered by the flip-flop either toggles or remains the same depending on whether the T input (Toggle) is 1 or 0.

J-K Flip-Flop: When the clock triggers, the value remembered by the flip-flop toggles if the J and K inputs are both 1, remains the same if they are both 0; if they are different, then the value becomes 1 if the J (Jump) input is 1 and 0 if the K (Kill) input is 1.

S-R Flip-Flop: When the clock triggers, the value remembered by the flip-flop remains unchanged if R and S are both 0, becomes 0 if the R input (Reset) is 1, and becomes 1 if the S input (Set) is 1. The behavior in unspecified if both inputs are 1

Method

1. Create a module with required number of variables and mention it’s input/output.
2. Write the description of the flip flops using behavioral model
3. Create another module referred as test bench to verify the functionality.
4. Follow the steps required to simulate the design and compare the obtained outputwith the required one.

Code:

Flip-flop modelling in Verilog HDL

SR Flip Flop:

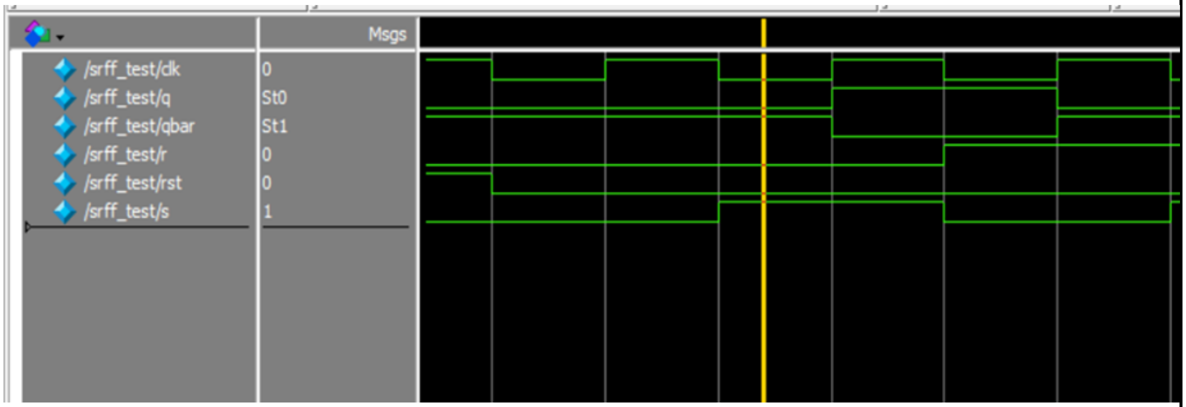
```
module srff (r,s,clk,rst,q,qbar);
input r,s,clk,rst;
output q,qbar;
reg q;
wire qbar;
assign qbar = ~q;
always @ (posedge clk or posedge rst)
begin
if (rst)
   q<=1'b0;
else
begin
case({r,s})
```

```
        2'b00:q<=q;
        2'b01:q<=1'b1;
        2'b10:q<=1'b0;
        2'b11:q<=1'bz;
endcase
end
end
endmodule
```

SR Flip Flop Testbench:

```
module srff_test;
reg r,s,clk,rst;
wire q,qbar;

srff sr1(r,s,clk,rst,q,qbar);

always
#5 clk=~clk;

initial
begin
  r=0;s=0;clk=0;rst=1;
  #10 rst=0;
  #10 r=0;s=1;
  #10 r=1;s=0;
  #10 r=1;s=1;
  #50 $stop;
end
endmodule
```

OUTPUT:



In this implementation, the **r** and **s** inputs are combined into a two-bit input vector **{r, s}** which is used in a **case** statement to control the state transitions of the flip-flop. The output **q** is a reg type which is updated on the rising **edge** of the **clock (posedge clk)** or the rising edge of the asynchronous reset **(posedge rst).**

When **rst** is high, the output **q** is set to 0**.** Otherwise, the case statement determines the new state of **q** based on the values of **r** and **s**. If **{r, s}** is **2'b00**, the output remains unchanged **(q <= q).** If **{r, s}** is **2'b01**, the output is set to **1**. If **{r, s}** is **2'b10**, the output is set to **0**. If **{r, s}** is **2'b11**, the output is set to high-impedance (**1'bz**).

The assign statement for **qbar** computes the complement of q using the bitwise **NOT** operator **(~).** This allows **qbar** to be derived from **q** without any additional logic.

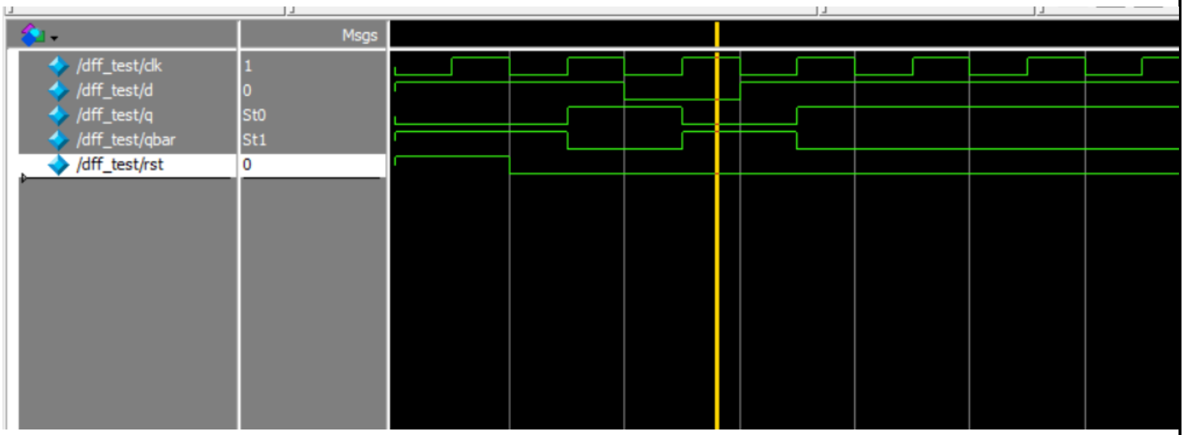
D Flip Flop:

```
module dff(d,clk,rst,q,qbar);
input d,clk,rst;
output q,qbar;
reg q;
wire qbar;

assign qbar = ~q;

always @ (posedge clk or posedge rst)
begin
if (rst)
q<=1'b0;
else
q <= d;
end

endmodule
```

D Flip Flop Testbench:

```
module dff_test;
  reg d,clk,rst;
  wire q,qbar;

dff d1(d,clk,rst,q,qbar);

always
#5 clk=~clk;
```

```
initial
begin
  d=1;clk=0;rst=1;
  #10 rst=0;

  #10 d=0;
  #10 d=1;

  #50 $stop;
end
endmodule
```



In this implementation, the d input is assigned directly to the **q** output on the rising edge of the clock **(posedge clk)**, unless the asynchronous reset **(rst)** input is high, in which case **q** is set to **0**. The **qbar** output is derived from **q** using the bitwise **NOT** operator **(~).**
The testbench initializes **d, clk**, and **rst** to specific values, then toggles d after a delay of 10 time units, and stops the simulation after a delay of 50 time units. The clock signal is generated by toggling it every 5 time units. This testbench will simulate the behavior of the D flip-flop and show the waveforms of the **q** and **qbar** outputs over time.

T Flip Flop:

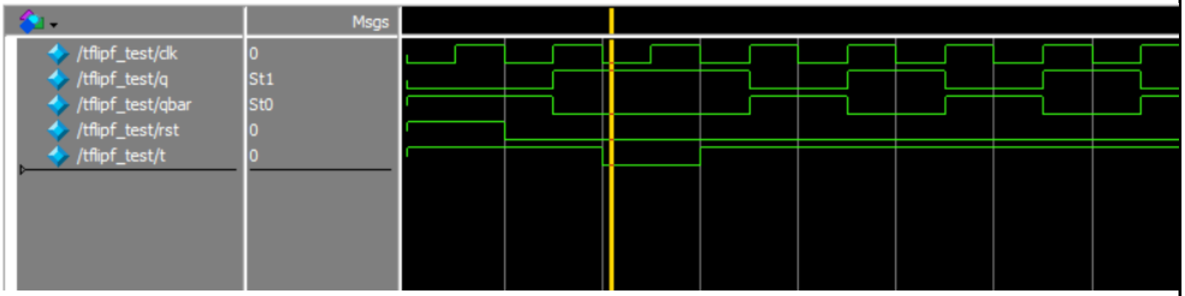
```
module tff (t,clk,rst,q,qbar);
input t,clk,rst;
output q,qbar;
reg q;
wire qbar;
assign qbar = ~q;
always @ (posedge clk or posedge rst)
begin
if (rst)
  q<=1'b0;
```

```
else if (t)
q <= ~q;
else
q <= q;
end
endmodule
```

T Flip Flop Testbench:

```
module tff_test;
  reg t,clk,rst;
  wire q,qbar;

  tff t1(t,clk,rst,q,qbar);

always
#5 clk=~clk;

initial
begin
  t=1;clk=0;rst=1;
  #10 rst=0;

  #10 t=0;
  #10 t=1;

  #50 $stop;
end
endmodule
```

OUTPUT:



```
                 Msgs
/tflipf_test/clk   0
/tflipf_test/q     St1
/tflipf_test/qbar  St0
/tflipf_test/rst   0
/tflipf_test/t     0
```

The Verilog code for the T flip flop module consists of three input ports: **t (toggle), clk (clock), and rst (reset);** and two output ports: **q (Q output) and qbar (not-Q output).** The flip flop is implemented using an always block that triggers on the positive edge of the clock or the positive edge of the reset signal. If the reset signal is **high**, the output (**q**) is set to **0**. If the toggle input (t) is **high**, the output (**q**) is toggled, otherwise the output remains the same. The not-Q output (qbar) is the complement of the Q output (q).

The Verilog code for the testbench initializes the inputs (t, clk, rst) and instantiates the T flip flop module (tff). The testbench toggles the clock signal every 5 time units using an always block, and sets the initial values for the inputs (t, clk, rst). After 10 time units, the reset signal is deasserted (set to 0). Then the toggle input (t) is set to 0 after 10 time units and then set to 1 after another 10 time units. Finally, the simulation is stopped after 50 time units using the $stop system task.
This testbench is verifying the functionality of the T flip flop module by setting its inputs to various values and observing its outputs. It is a good practice to write testbenches to verify the functionality of digital circuits before using them in a larger design.

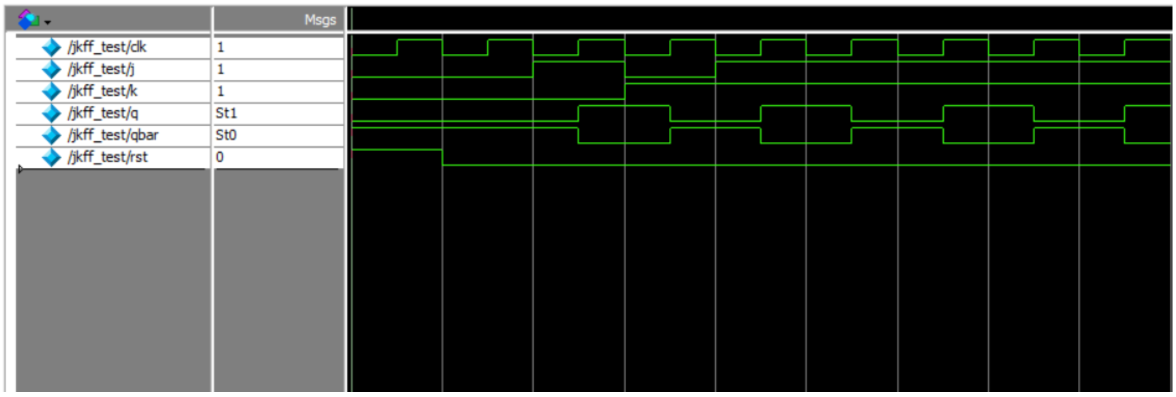
JK Flip Flop:

```
module jkff (j,k,clk,rst,q,qbar);
input j,k,clk,rst;
output q,qbar;
reg q;
wire qbar;
assign qbar = ~q;
always @ (posedge clk or posedge rst)
begin
if (rst)
   q<=1'b0;
else
begin
case({j,k})
      2'b00:q<=q;
      2'b01:q<=1'b0;
      2'b10:q<=1'b1;
      2'b11:q<=~q;
endcase
end
end
endmodule
```

JK Flip Flop Testbench:

```
module jkff_test;
reg j,k,clk,rst;
wire q,qbar;

jkff jk(j,k,clk,rst,q,qbar);

always
#5 clk=~clk;
```

```
initial
begin
  j=0;k=0;clk=0;rst=1;
  #10 rst=0;
  #10 k=0;j=1;
  #10 k=1;j=0;
  #10 k=1;j=1;
  #50 $stop;
end
endmodule
```



The Verilog code for the JK flip flop module consists of four input ports: **j (J input), k (K input), clk (clock), and rst (reset);** and two output ports: **q (Q output)** and **qbar (not-Q output).** The flip flop is implemented using an always block that triggers on the positive edge of the clock or the positive edge of the reset signal. If the reset signal is high, the output (**q**) is set to **0**. Otherwise, the output (**q**) is updated based on the inputs J and K, using a case statement. If **J and K** are both **0**, the output remains the same. If J is 1 and K is 0, the output is set to **1**. If **J** is **0** and **K** is **1**, the output is set to **0**. If **J** and **K** are both **1**, the output is **toggled**. The not-Q output (**qbar**) is the complement of the Q output (**q**).

The Verilog code for the testbench initializes the inputs **(j, k, clk, rst)** and instantiates the JK flip flop module (jkff). The testbench toggles the clock signal every 5 time units using an always block, and sets the initial values for the inputs (j, k, clk, rst). After 10 time units, the reset signal is deasserted (set to 0). Then, the inputs (j, k) are set to various values to test the functionality of the flip flop. Finally, the simulation is stopped after 50 time units using the $stop system task.

Expt. 7 Shift Registers

Objective:

Theory:

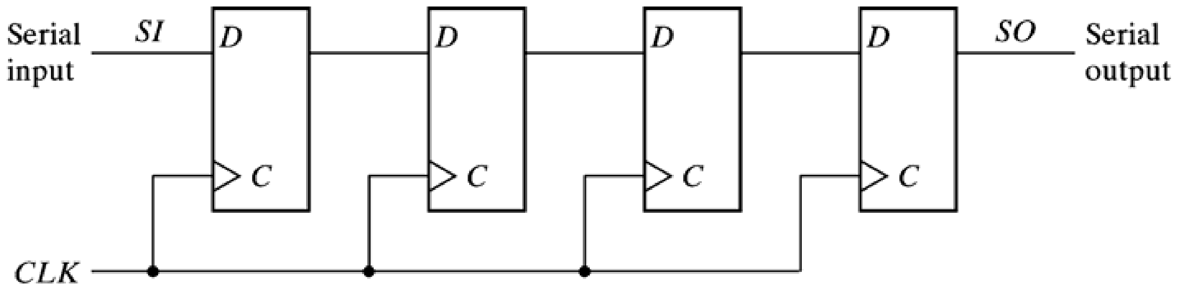


Fig. 6-3 4-Bit Shift Register

Code:

Shift Registers

1) SISO

```
module siso(so,si,clk,rst);
output so;
input si,clk,rst;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b0;
else
      q<={si,q[3:1]};
end
assign so=q[0];
endmodule
```

SISO Testbench

```
module siso_test;
  wire so;
  reg si,clk,rst;

siso s1(so,si,clk,rst);

always
#5 clk=~clk;

initial
begin
  si=0; clk=0;rst=1;
    #10 rst=0;
    #10 si=1;
    #10 si=0;
    #10 si=0;
    #10 si=1;
    #10 si=1;
    #50 $stop;
  end
```

```
endmodule
```

2) SIPO

```
module sipo(q,si,clk,rst);
output [3:0]q;
input si,clk,rst;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b0;
else
      q<={si,q[3:1]};
end
endmodule
```

SIPO Testbench

```
module sipo_test;
  wire [3:0]q;
  reg si,clk,rst;

  sipo s1(q,si,clk,rst);

  always
#5 clk=~clk;

initial
begin
si=0; clk=0;rst=1;
  #10 rst=0;
  #10 si=1;
  #10 si=0;
  #10 si=0;
  #10 si=1;
  #10 si=1;
  #50 $stop;
end
endmodule
```

3) PIPO

```
  module pipo(q,in,load,clk,rst);
output [3:0]q;
input load,clk,rst;
input [3:0]in;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b0;
```

```
else if(load)
      q<=in;
else
      q<=q;
end
endmodule
```

PIPO Testbench

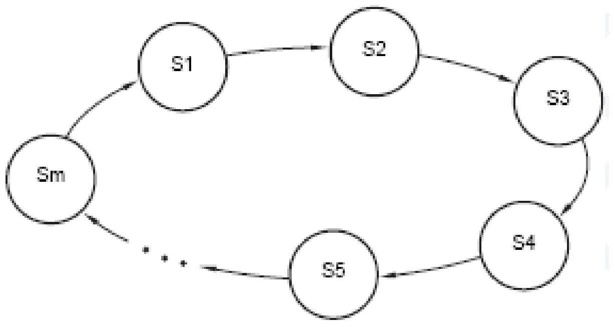
```
module pipo_test;
  wire [3:0]q;
reg load,clk,rst;
reg [3:0]in;

pipo p1(q,in,load,clk,rst);

always
#5 clk=~clk;

initial
begin
   load=1;clk=0;rst=1;
   in=4'b1001;
   #10 rst=0;
   #10 load=0;
   #40 load=1;in=4'b0011;
   #10 load=0;

   #50 $stop;
 end
 endmodule
```

4)PISO

```
module piso(so,in,load,clk,rst);
output so;
input load,clk,rst;
input [3:0]in;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b0;
else if(load)
      q<=in;
else
      q<={1'b0,q[3:1]};
end
assign so=q[0];
endmodule
```

PISO Testbench

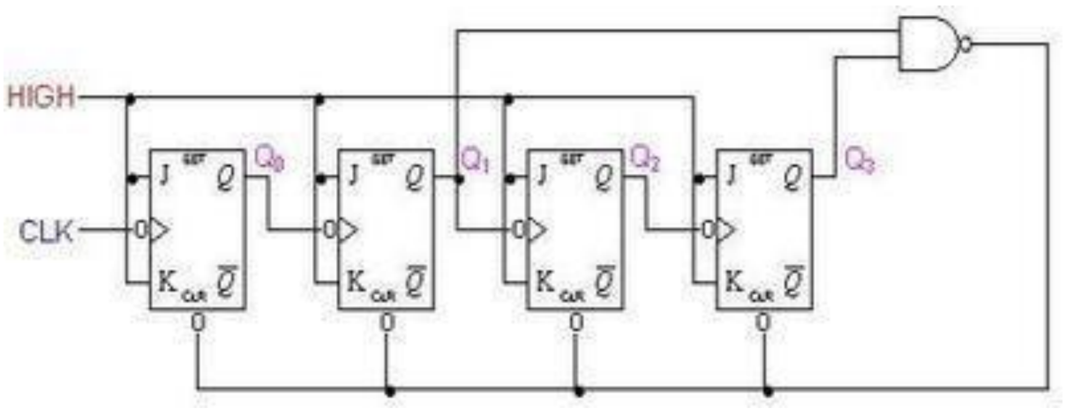
```
module piso_test;
  wire so;
reg load,clk,rst;
reg [3:0]in;

piso p1(so,in,load,clk,rst);

always
#5 clk=~clk;

initial
begin
   load=1;clk=0;rst=1;
   in=4'b1001;
   #10 rst=0;
   #10 load=0;
   #40 load=1;in=4'b0011;
   #10 load=0;

   #50 $stop;
 end
 endmodule
```

Expt. 8	**DESIGN OF COUNTERS**
Objective	Design and Implement Up, Down, Ring, Johnson, Mod N counters
Theory:	Counter is a sequential circuit. A digital circuit which is used for counting pulses is known as counter. Counter is the widest application of flip-flops. It is a group of flip- flops with a clock signal applied. Counters are of two types. • Asynchronous or ripple counters. • Synchronous counters. Asynchronous counters are called as ripple counters, the first flip-flop is clocked by the external clock pulse and then each successive flip-flop is clocked by the output of the preceding flip-flop. The term asynchronous refers to events that do not have a fixed time relationship with each other. An asynchronous counter is one in which the flip-flops within the counter do not change states at exactly the same time because they do not have a common clock pulse In synchronous counters, the clock inputs of all the flip-flops are connected together and are triggered by the input pulses. Thus, all the flip-flops change state simultaneously (in parallel). A counter is a register capable of counting the number of clock pulses arriving at its clock input. Count represents the number clock pulses arrived. A specified sequence of states appears as the counter output. The name counter is generally used for clocked sequential circuit whose state diagram contains a single cycle. The modulus of a counter is the number of states in the cycle. A counter with m states is called a modulo-m counter or divide-by-m counter. 

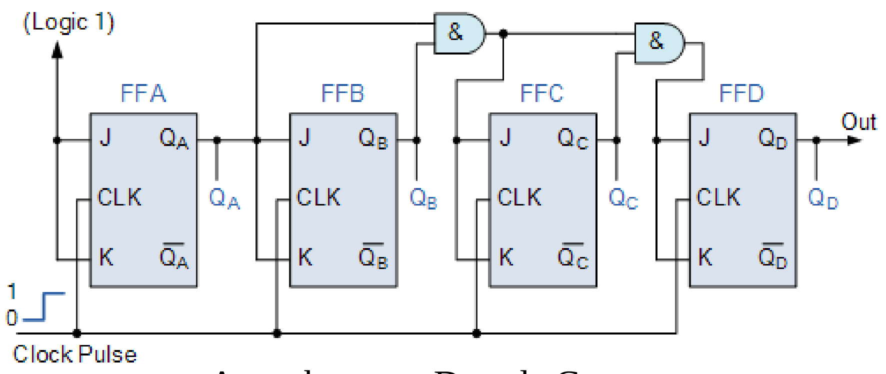
General structure of a counter's state diagram – a single cycle

Asynchronous Decade Counters

The modulus is the number of unique states through which the counter will sequence. The maximum possible number of states of a counter is $2^n$where n is the number of flip-flops. Counters can be designed to have a number of states in their sequence that is less than the maximum of 2^n. This type of sequence is called a truncated sequence. One common modulus for counters with truncated sequences is 10 (Modules10). A decade counter with a count sequence of zero (0000) through 9 (1001) is a BCD decade counter because its 10-state sequence produces the BCD code. To obtain a truncated sequence, it is necessary to force the counter to recycle before going through all of its possible states. A decade counter requires 4flip-flops. One way to make the counter recycle after the count of 9(1001) is to decode count 10 (1010) with a NAND gateand connect the output of the NAND gate to the clear (CLR) inputs of the flip-flops, as shown in Figure below.



Synchronous Decade Counters



Asynchronous Decade Counter

It can be seen from Figure , that the external clock pulses (pulses to be counted) are fed directly to each of the J-K flip-flops in the counter chain and that both the J and K inputs are all tied together in toggle mode, but only in the first flip-flop, flip- flop FFA (LSB) are they connected HIGH, logic “1” allowing the flip-flop to toggle on every clock pulse. Then the synchronous counter follows a predetermined sequence of states in response to the common clock signal, advancing one state for each pulse.

The J and K inputs of flip-flop FFB are connected directly to the output QA of flip- flop FFA, but the J and K inputs of flip-flops FFC and FFD are driven from separate AND gates which are also supplied with signals from the input and output of the previous stage. These additional AND gates generate the required logic for the JK inputs of the next stage.

If we enable each JK flip-flop to toggle based on whether or not all preceding flip- flop outputs (Q) are “HIGH” we can obtain the same counting sequence as with the asynchronous circuit but without the ripple effect, since each flip-flop in this circuit will be clocked at exactly the same time.

Up counter

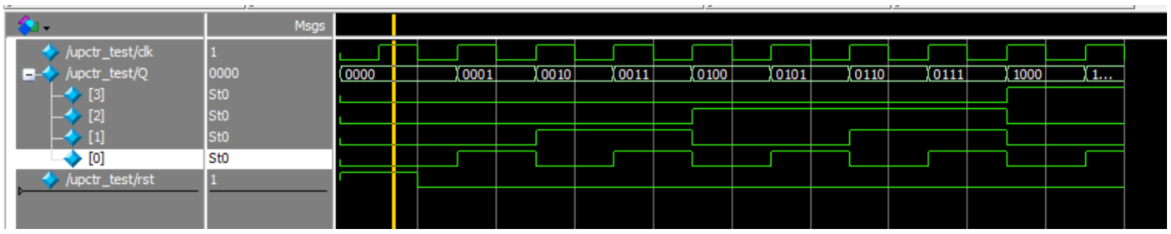
```
module up_counter(Q,clk,rst);
output [3:0]Q;
input clk,rst;
reg [3:0]Q;

always @ (posedge clk or posedge rst)
```

```
begin
if (rst)
Q <= 4'b0;
else
Q <= Q+1;
end
endmodule
```

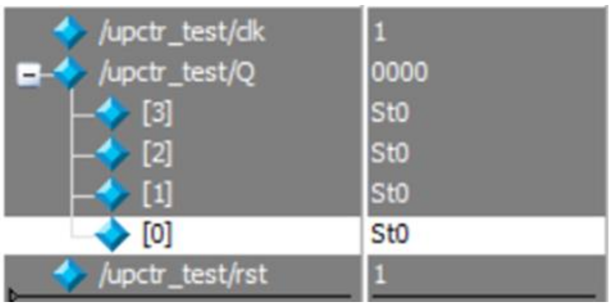
UP COUNTER(TEST BENCH):

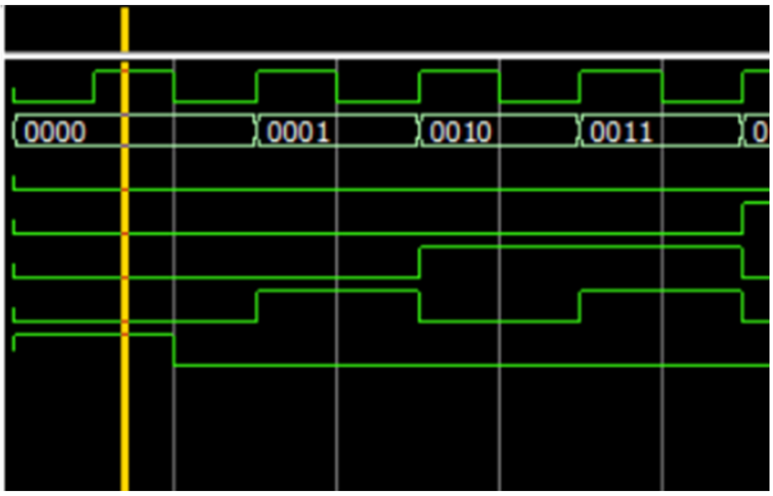
```
module up_counter_test;
wire [3:0]Q;
reg clk,rst;
up_counter c1(Q,clk,rst);
always
#5 clk=~clk;

initial
begin
  clk=0;rst=1;
  #10 rst=0;

  #200 $stop;
end
endmodule
```

OUTPUT:



Zoomed View:





This code implements an up counter using Verilog HDL. The up_counter module contains an output port Q that is 4 bits wide and an input port clk and rst. The Q output port will hold the current count value, and clk is the clock signal that will be used to increment the counter. The rst input port is used to reset the counter to zero.

The always block uses the posedge clock signal as a trigger and increments the counter by 1 when the clock signal goes from low to high. If the rst input signal goes high, the counter is reset to zero.

The up_counter_test module is used to simulate and test the up_counter module. It creates a clock signal using the always block and connects it to the clk input of the up_counter module. It also creates a reset signal that is initially set to high and then goes low after 10 simulation time units.

The simulation runs for 200 simulation time units and then stops. During the simulation, the clock signal is toggled every 5 simulation time units. This will cause the up counter to increment its count by one every time the clock signal goes from low to high, except when the reset signal is high, which will cause the counter to reset to zero. The final count value will be stored in the Q output port of the up_counter module, which can be used to verify that the counter is working correctly.

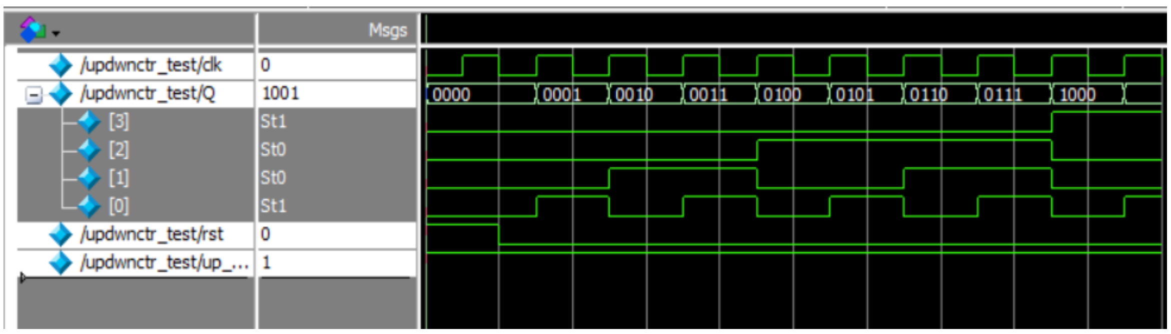
Up/down counter:

```
module up_down(Q,up_down,clk,rst);
output [3:0]Q;
input up_down,clk,rst;
reg [3:0]Q;
always @ (posedge clk or posedge rst)
begin
if (rst)
  Q <= 4'b0;
else if(up_down)
  Q <= Q+1;
else
  Q <= Q-1;
end
endmodule
```

up/down (TEST BENCH):

```
 module up_down_test;
  wire [3:0]Q;
  reg up_down,clk,rst;

  up_down ud1(Q,up_down,clk,rst);

  always
#5 clk=~clk;

initial
begin
  clk=0;up_down=1;rst=1;
  #10 rst=0;

  #100 up_down=0;

  #200 $stop;
end
endmodule
```

OUTPUT:



```
                    Msgs
/updwnctr_test/clk  0
/updwnctr_test/Q    1001
  [3]               St1
  [2]               St0
  [1]               St0
  [0]               St1
/updwnctr_test/rst  0
/updwnctr_test/up_... 1
```

This code implements an **up/down** counter using Verilog HDL. The **up_down** module contains an output port Q that is 4 bits wide and three input ports: **up_down, clk, and rst.** The **up_down** input port is used to control the direction of the counter, with a value of 1 indicating an up count and 0 indicating a down count. The **clk** input port is the clock signal that will be used to increment or decrement the counter, and rst is the reset signal used to reset the counter to zero.

The always block uses the **posedge** clock signal as a trigger and checks the **up_down** input port to determine whether to increment or decrement the counter when the clock signal goes from low to high. If the **rst** input signal goes high, the counter is reset to **zero**.

The **up_down_test** module is used to simulate and test the **up_down** module. It creates a clock signal using the always block and connects it to the **clk** input of the **up_down** module. It also creates a reset signal that is initially set to high and then goes low after 10 simulation time units.

The **up_down** input signal is initially set to 1, indicating an up count, and then set to 0 after 100 simulation time units, indicating a down count.

The simulation runs for 200 simulation time units and then stops. During the simulation, the clock signal is toggled every 5 simulation time units. This will cause the up/down counter to increment or decrement its count by one every time the clock signal goes from low to high, depending on the value of the up_down input signal. The final count value will be stored in the Q output port of the up_down module, which can be used to verify that the counter is working correctly.

Mod N Counter (N=10)

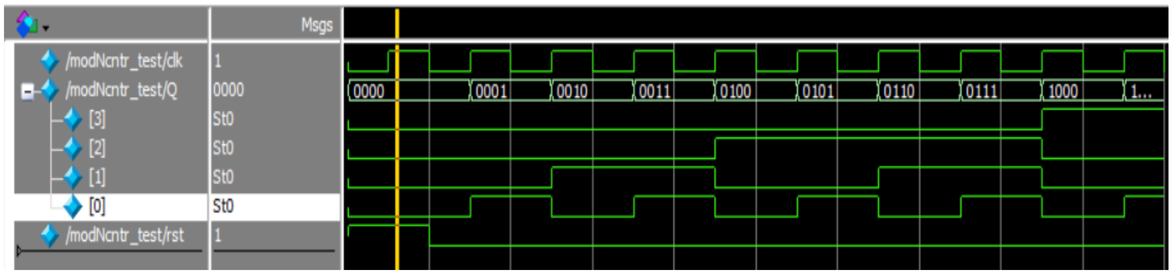
CODE:

```
module modN_counter(Q,clk,rst);
output [3:0]Q;
input clk,rst;
reg [3:0]Q;
always @ (posedge clk or posedge rst)
begin
if (rst)
Q <= 4'b0;
else if (Q == 9)
Q <= 4'b0;
else
Q <= Q+1;
end
endmodule
```

Mod N Counter (TEST BENCH) :

```
module modN_counter_test;
  wire [3:0]Q;
reg clk,rst;
modN_counter mc1(Q,clk,rst);
always
#5 clk=~clk;
initial
begin
  clk=0;rst=1;
  #10 rst=0;
  #200 $stop;
end
endmodule
```

OUTPUT:



It defines a module called modN_counter with inputs clk and rst, and an output Q which is a 4-bit binary value representing the current count of the counter. The counter increments by 1 on each clock cycle, except when it reaches the maximum count value (9), at which point it resets to 0.

The testbench module modN_counter_test instantiates the modN_counter module and connects its inputs and outputs. It also generates a clock signal (clk) and a reset signal (rst) to test the counter's functionality. The clock signal alternates between 0 and 1 every 5 time units, and the reset signal is initially set to 1 and then set to 0 after 10 time units. The simulation runs for 200 time units before stopping.
Overall, this code provides a basic implementation of a ModN counter with N=10, and a testbench to verify its correct operation. However, it does not provide any outputs or tests to verify the counter's behavior, so it may require further development and testing to ensure that it meets the desired requirements.

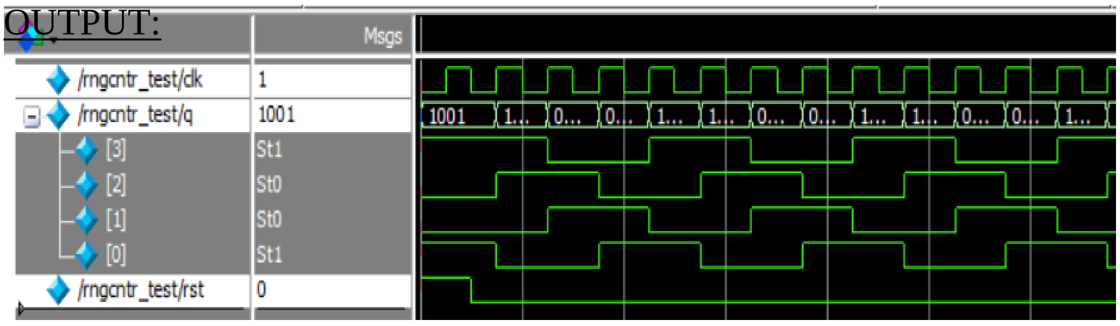
Ring counter:

```
module ring_counter(q,clk,rst);
output [3:0]q;
input clk,rst;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b1001;
else
      q<={q[0],q[3:1]};
end
endmodule
```

Ring counter (TEST BENCH):

```
module ring_counter_test;
wire [3:0]q;
reg clk,rst;
ring_counter r1(q,clk,rst);
always
#5 clk=~clk;
initial
begin
  clk=0;rst=1;
  #10 rst=0;

  #200 $stop;
end
endmodule
```

OUTPUT:



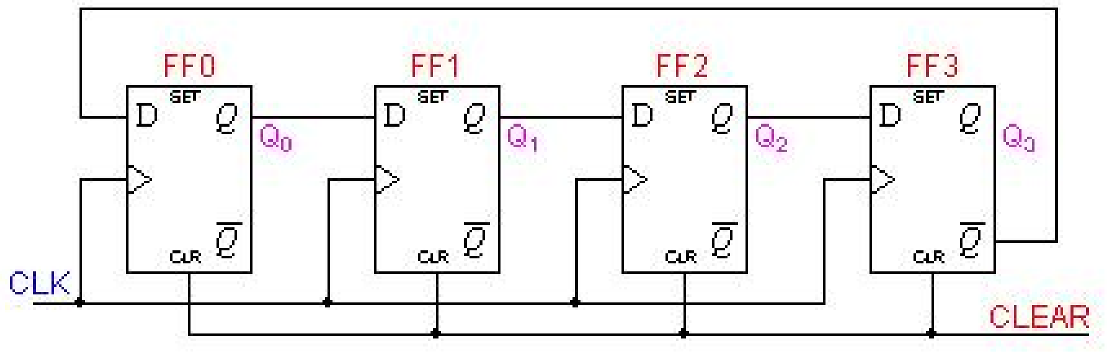
A Ring Counter is a type of counter where the output of one flip-flop is connected to the input of the next flip-flop in a circular manner. This creates a circular shift of the output bits, with only one bit being set at any given time.

The module ring_counter takes two inputs, clk and rst, and outputs a 4-bit value q representing the current state of the counter. The counter initializes to the binary value 1001 (decimal 9), and then shifts the value of q one bit to the left on each clock cycle, using the syntax {q[0],q[3:1]} to shift the values of q[3:1] to the left by one position, and set the value of q[0] to the previous value of q[3]. When the reset input is asserted (rst=1), the counter resets to its initial state.

The testbench ring_counter_test instantiates the ring_counter module and connects its inputs and outputs. It generates a clock signal (clk) and a reset signal (rst) to test the counter's functionality. The clock signal alternates between 0 and 1 every 5 time units, and the reset signal is initially set to 1 and then set to 0 after 10 time units. The simulation runs for 200 time units before stopping.

Overall, this code provides a basic implementation of a Ring Counter, and a testbench to verify its correct operation. However, it does not provide any outputs or tests to verify the counter's behavior, so it may require further development and testing to ensure that it meets the desired requirements.

Johnson counter



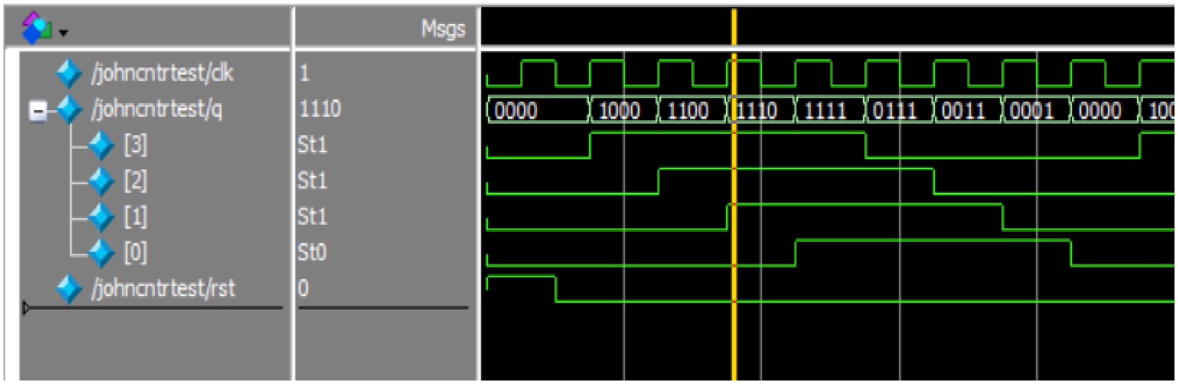
CODE:

```
module johnson_counter(q,clk,rst);
output [3:0]q;
input clk,rst;
reg [3:0]q;
always @ (posedge clk or posedge rst)
begin
if(rst)
      q<=4'b0000;
else
      q<={~q[0],q[3:1]};
end
endmodule
```

Johnson counter (test bench):

```
module johnson_counter_test;
wire [3:0]q;
reg clk,rst;
johnson_counter j1(q,clk,rst);
always
#5 clk=~clk;
initial
begin
  clk=0;rst=1;
  #10 rst=0;
    #200 $stop;
end
endmodule
```

Output:



A Johnson Counter is a type of shift register that can generate a sequence of all possible bit patterns of a given length.
The module johnson_counter takes two inputs, clk and rst, and outputs a 4-bit value q representing the current state of the counter. The counter initializes to the binary value 0000, and then shifts the value of q one bit to the left on each clock cycle, using the syntax {~q[0],q[3:1]} to shift the values of q[3:1] to the left by one position, and set the value of q[0] to the logical complement of its previous value. When the reset input is asserted (rst=1), the counter resets to its initial state.

The testbench johnson_counter_test instantiates the johnson_counter module and connects its inputs and outputs. It generates a clock signal (clk) and a reset signal (rst) to test the counter's functionality. The clock signal alternates between 0 and 1 every 5 time units, and the reset signal is initially set to 1 and then set to 0 after 10 time units. The simulation runs for 200 time units before stopping.

Overall, this code provides a basic implementation of a Johnson Counter, and a testbench to verify its correct operation. However, it does not provide any outputs or tests to verify the counter's behavior, so it may require further development and testing to ensure that it meets the desired requirements.

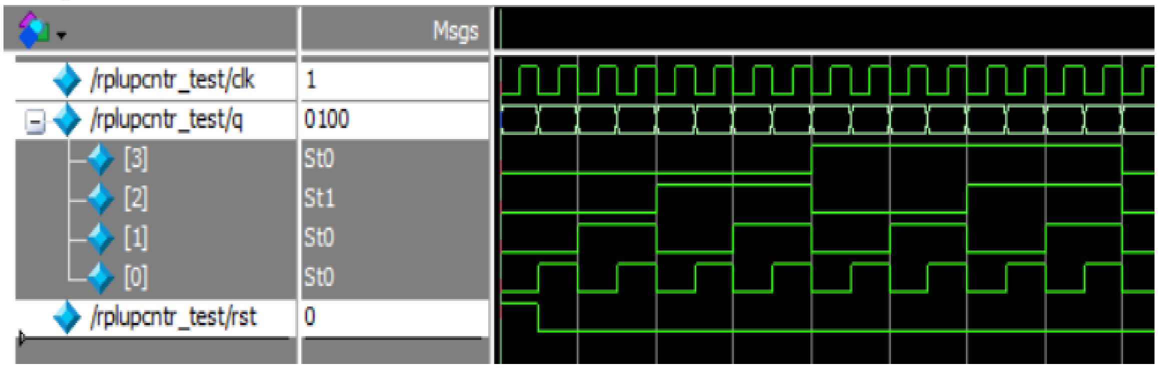
Ripple up counter

```
module ripple_up_counter(q,clk,rst);
  output [3:0]q;
  input clk,rst;
  wire [3:0] q,qbar;
  wire clk,rst;
  tff t0(1'b1,~clk,rst,q[0],qbar[0]);
  tff t1(1'b1,~q[0],rst,q[1],qbar[1]);
  tff t2(1'b1,~q[1],rst,q[2],qbar[2]);
  tff t3(1'b1,~q[2],rst,q[3],qbar[3]);
endmodule
```

Ripple up counter (test bench):

```
module ripple_up_test;
wire [3:0]q;
reg clk,rst;
ripple_up_counter r1(q,clk,rst);
always
#5 clk=~clk;
initial
begin
  clk=0;rst=1;
  #10 rst=0;

  #200 $stop;
end
endmodule
```

Output:



Verilog implementation of a 4-bit ripple up counter and its testbench. The counter is composed of T flip-flops (TFFs) connected in series to form a binary counter that increments on the rising edge of the clock signal.

The input signals to the counter module are clk (the clock signal) and rst (the reset signal), and the output signal is q (the 4-bit binary count).

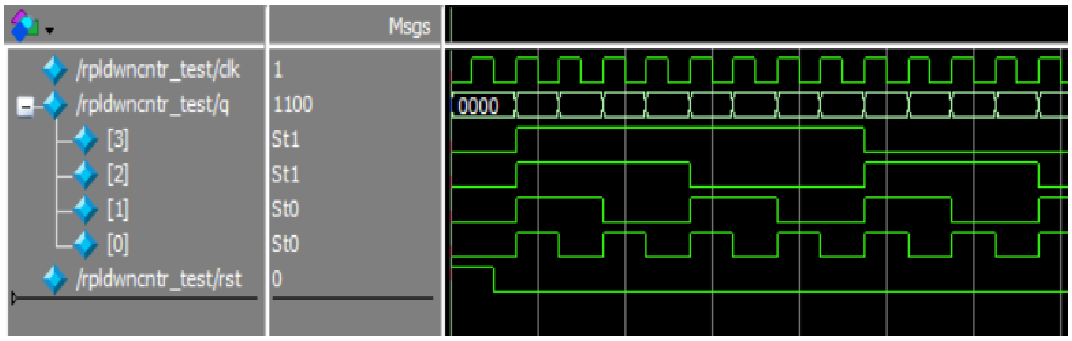
The testbench initializes the input signals to the counter module and toggles the clk signal every 5 simulation time units. The reset signal is held high for the first 10 simulation time units to ensure the counter is in a known state before starting the count. The simulation is stopped after 200 simulation time units.

Ripple down counter

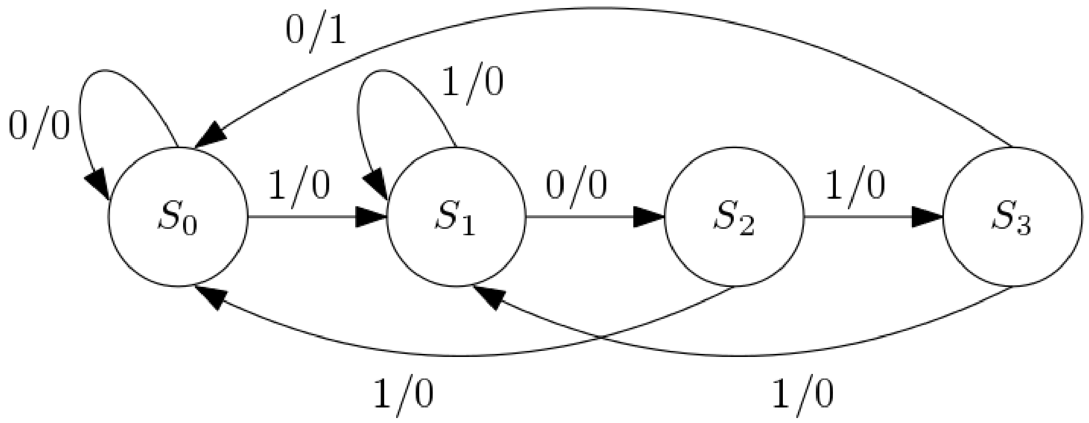
```
module ripple_down_counter(q,clk,rst);
  output [3:0]q;
  input clk,rst;
  wire [3:0] q,qbar;
  wire clk,rst;

  tff t0(1'b1,clk,rst,q[0],qbar[0]);
  tff t1(1'b1,q[0],rst,q[1],qbar[1]);
  tff t2(1'b1,q[1],rst,q[2],qbar[2]);
  tff t3(1'b1,q[2],rst,q[3],qbar[3]);
  endmodule
```

Ripple down counter (test bench):

```
module ripple_down_test;
wire [3:0]q;
reg clk,rst;
ripple_down_counter r1(q,clk,rst);
always
#5 clk=~clk;
initial
begin
  clk=0;rst=1;
  #10 rst=0;

  #200 $stop;
end
endmodule
```

	<u>Output:</u>  Msgs /rpldwncntr_test/clk 1 /rpldwncntr_test/q 1100 [3] St1 [2] St1 [1] St0 [0] St0 /rpldwncntr_test/rst 0 0000 Verilog implementation of a 4-bit ripple down counter and its testbench. The counter is composed of T flip-flops (TFFs) connected in series to form a binary counter that decrements on the rising edge of the clock signal. The input signals to the counter module are clk (the clock signal) and rst (the reset signal), and the output signal is q (the 4-bit binary count). The testbench initializes the input signals to the counter module and toggles the clk signal every 5 simulation time units. The reset signal is held high for the first 10 simulation time units to ensure the counter is in a known state before starting the count. The simulation is stopped after 200 simulation time units.
Procedur e	1. Create a module with required number of variables and mention it's input/output. 2. Write the description of the counter to count required number of states and tosatisfy its conditions. 3. Create another module referred as test bench to verify the functionality. 4. Follow the steps required to simulate the design and compare the obtained outputwith the required one

Expt. 9	**Sequence Detection by Mealy Machine**
Objective	To perform the design flow to generate state machines in Verilog code to detect the given sequence of bits.
Theory	As an illustrative example a sequence detector for bit sequence '1010' is described. Every clock-cycle a value will be sampled, if the sequence '1010' is detected a '1' will be produced at the output for 1 clock-cycle. There are two methods to design state machines, first is Mealy and second is Moore style. We will discuss Mealy Machine in this exercise and Moore in the next. **Sequence Detector FSM** Sequence detector is good example to describe FSMs. It produces a pulse output whenever it detects a predefined sequence. In this tutorial we have considered a 4-bit sequence "1010". The first step of an FSM design is to draw the state diagram. The sequence detector with **overlapping** sequence. For example, consider the input sequence as "11010101011", the output y in with overlapping style will be "00001010100". Following is the behavior description of the sequencer for a Mealy style implementation and the state diagram is shown in figure below: Mealy State Machine for Detecting a Sequence of '1011'



- When in initial state (S0) the machine gets the input of '1' it jumps to the next state with the output equal to '0'. If the input is '0' it stays in the same state.
- When in 2nd state (S1) the machine gets an input of '0' it jumps to the 3rd state with the output equal to '0'. If it gets an input of '1' it stays in the same state.
- When in the 3rd state (S2) the machine gets an input of '1' it jumps to the 4th state with the output equal to '0'. If the input received is '0' it goes back to the initial state.
- When in the 3rd state (S3) the machine gets an input of '0' it jumps back to the 2ndstate, with the output equal to '1'. If the input received is '0' it goes back to the 3rd state.

After designing the state machines, the models have to be transformed into Verilog code describing the architecture.

Mealy FSM code for a sequence detector that detects the pattern **"0101"** is given below. Here's a breakdown of the code:

- The module **seq_mealy** defines the inputs and outputs of the FSM.
- The output y is the output of the FSM, which is set to **1** when the pattern is detected.
- The input **i** is the input signal to the FSM that the pattern is detected from.
- The input **clk** is the clock signal for the FSM.
- The input **rst** is the reset signal for the FSM.

- The **reg y** is the register for the output signal of the FSM.
- The **reg [1:0] cs,ns** are the registers for the current state and next state of the FSM.

The parameter statements define the states of the FSM as binary values.

Code:

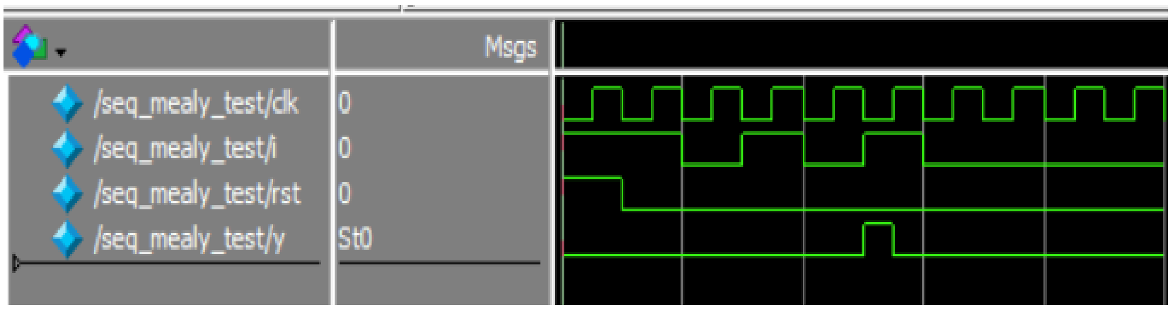
Mealy FSM Verilog Code

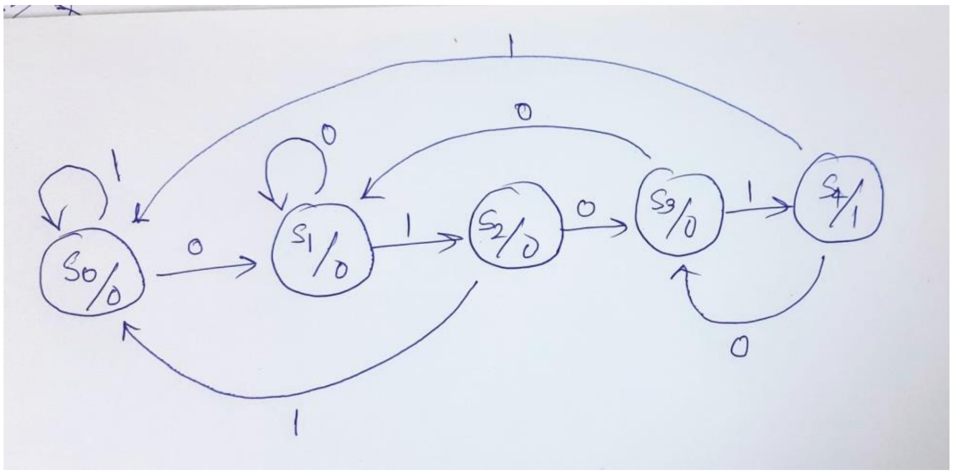
```
//Mealy FSM code for 1010 overlapping sequence detector
module seq_mealy(y,i,clk,rst);
  output y;
  input i,clk,rst;

  reg y;
  reg [1:0] cs,ns;

  //state encoding
  parameter s0=2'b00;
  parameter s1=2'b01;
  parameter s2=2'b10;
  parameter s3=2'b11;

  //next state logic -combinational logic
  always @(cs or i)
  begin
    y=1'b0;
    case(cs)
      s0:begin
        if(i)
          ns =s1;
        else
          ns =s0;
        end
      s1:begin
        if(i)
          ns =s1;
        else
          ns =s2;
        end
      s2:begin
        if(i)
          ns =s3;
        else
        ns =s0;
      end
      s3:begin
      if(i)
        ns =s1;
      else begin
        ns =s2;
        y =1'b1;
      end
      end
    endcase
  end

  //current state logic-sequential logic
  always @(posedge clk or posedge rst)
  begin
    if(rst)
      cs <=s0;
    else
      cs<=ns;
    end

  endmodule
```

Mealy FSM Testbench

```
module seq_mealy_test;
  wire y;
  reg i,clk,rst;

  seq_mealy f1(y,i,clk,rst);

  always
  #5 clk =~clk;


initial

begin

  i=1;clk=0;rst=1;

  #10 rst=0;

 #10 i=0;
 #10 i=1;
 #10 i=0;
 #10 i=1;
  #10 i=0;
```

	#50 $stop;
	
	end
	
	endmodule

	 Msgs /seq_mealy_test/clk 0 /seq_mealy_test/i 0 /seq_mealy_test/rst 0 /seq_mealy_test/y St0
	The **always @(cs or i)** block defines the state transitions of the FSM based on the current state (**cs**) and the input signal (**i**). • In state **s0**, if the input signal is high (**i**), the FSM stays in state **s0**, otherwise it transitions to state **s1**. • In state **s1**, if the input signal is high (**i**), the FSM transitions to state **s2**, otherwise it stays in state **s1**. • In state **s2**, if the input signal is high (**i**), the FSM transitions to state **s0**, otherwise it transitions to state **s3**. • In state **s3**, if the input signal is low (**i**), the FSM transitions to state **s1**. If the input signal is high (**i**), the FSM transitions to state **s2** and sets the output signal y to high. • The **always @(posedge clk or posedge rst)** block is the clocked process that updates the current state of the FSM. When the reset signal is high **(rst)**, the FSM is initialized to state **s0**, otherwise the current state is updated to the next state (**ns**) at the positive edge of the clock signal.
Conclusion	Output goes to logic 1 when it detects 1010 sub-sequence

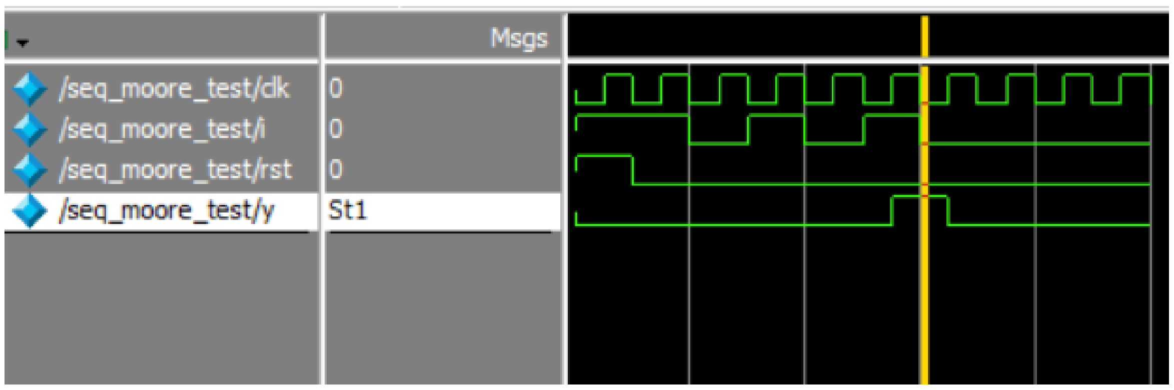
Expt. 10	**Sequence Detection by Moore Machine**
Objective :	1. Design and implement 0101 sequence detector using Moore Machine
Procedure:	Following is the behavior description of the sequencer for a Moore style implementation and the state diagram is shown in figure:  Moore State Machine for Detecting a Sequence of '0101' • In initial state (S0) the output of the detector is '0'. When machine gets the input of '0' it jumps to the next state. If the input is '1' it stays in the same state. • In 2nd state (S1) the output of the detector is '0'. When machine gets an input of '1'it jumps to the 3rd state. If it gets an input of '0' it stays in the same state. • In the 3rd state (S2) the output of the detector is '0'. When machine gets an input of'0' it jumps to the 4th state. If the input received is '1' it goes back to the initial state. • In the 4th state (S3) the output of the detector is '0'. When machine gets an input of'1' it jumps to the 5th state. If the input received is '0' it goes back to the 2nd state. • In the 5th state the output of the detector is '1'. When machine gets an input of '0'it jumps to the 4th state, otherwise it jumps to the initial state.

Code:	**Moore FSM Verilog Code**

```
module seq_moore(y,i,clk,rst);
  output y;
  input i,clk,rst;

  reg y;
  reg [2:0] cs,ns;

  //state encoding
  parameter s0=3'b000;
  parameter s1=3'b001;
  parameter s2=3'b010;
  parameter s3=3'b011;
  parameter s4=3'b100;

  //next state logic-combinational
  always @(cs or i)
  begin
    case(cs)
      s0:if(i)
         ns=s1;
       else
         ns=s0;
      s1:if(i)
         ns=s1;
       else
         ns=s2;
      s2:if(i)
         ns=s3;
       else
         ns=s0;
      s3:if(i)
         ns=s1;
       else
         ns=s4;
      s4:if(i)
         ns=s3;
       else
         ns=s0;
       endcase
     end

  //current state logic -sequential logic
  always @(posedge clk or posedge rst)
    begin
    if(rst)
      cs <=s0;
    else
      cs<=ns;
    end

//output logic -combinational logic
always @(cs)
begin
  if (cs==s4)
    y=1'b1;
  else
    y=1'b0;
  end

endmodule
```

Moore FSM Testbench

```
module seq_moore_test;
  wire y;
  reg i,clk,rst;
  seq_moore f1(y,i,clk,rst);
  always
  #5 clk =~clk;

initial
begin

  i=1;clk=0;rst=1;

  #10 rst=0;

 #10 i=0;

   #10 i=1;

    #10 i=0;

     #10 i=1;

      #10 i=0;

      #50 $stop;

      end

      endmodule
```

1,.	This is a Moore FSM code for a sequence detector that detects the pattern **"0101"** using Verilog. Here's a breakdown of the code: • The **module seq_moore** defines the inputs and outputs of the FSM. • The output **y** is the output of the FSM, which is set to **1** when the pattern is detected. • The input **i** is the input signal to the FSM that the pattern is detected from. • The input **clk** is the clock signal for the FSM. • The input **rst** is the reset signal for the FSM. • The **reg y** is the register for the output signal of the FSM. • The **reg [2:0] cs,ns** are the registers for the current state and next state of the FSM. The parameter statements define the states of the FSM as binary values.
Output	 Msgs /seq_moore_test/clk 0 /seq_moore_test/i 0 /seq_moore_test/rst 0 /seq_moore_test/y St1
	The **always @(cs or i) block** defines the state transitions of the FSM based on the current state (**cs**) and the input signal (**i**). • In state **s0**, if the input signal is **low (!i)**, the FSM transitions to state **s1**, otherwise it stays in state **s0**. • In state **s1**, if the input signal is **low (!i),** the FSM stays in state s1, otherwise it transitions to state **s2**. • In state **s2**, if the input signal is **low (!i),** the FSM transitions to state **s3**, otherwise it transitions to state **s0**. • In state **s3**, if the input signal is **low (!i),** the FSM transitions to state **s1**, otherwise it transitions to state **s4**. • In state **s4**, if the input signal is **low (!i),** the FSM transitions to state **s3**, otherwise it transitions to state **s0**.

- The **always @(posedge clk or posedge rst) block** is the clocked process that updates the current state of the FSM. When the reset signal is high (**rst**), the FSM is initialized to state **s0**, otherwise the current state is updated to the next state (**ns**) at the positive edge of the clock signal.
- ➢ The **always @(cs)** block is the combinatorial process that updates the output signal of the FSM based on the current state. If the current state is **s4**, the output signal **y** is set to **high**, otherwise it is set to **low**.

.

Conclusion:

It returns output as 1 when it detects the sub-sequence 0101.