# Section 1: Verilog

Aim: To learn about Verilog and familiarize oneself with usage of Verilog using Xilinx ISE

**What is Verilog?**

Verilog is one of the two-major Hardware Description Languages (HDL) used by hardware designers in industry and academia. VHDL is the other one. Many feel that Verilog is easier to learn and use than VHDL. VHDL was made an IEEE Standard in 1987, and Verilog in 1995. Verilog is very similar to C-language.

Verilog allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i. e., gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDLs is the simulation of designs before the designer must commit to fabrication.

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of abstraction, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels. Verilog allows hardware designers to express their design with **behavioral constructs**, determining the details of implementation to a later stage of design. An abstract representation helps the designer explore architectural alternatives through **simulations** and to detect design bottlenecks before detailed design begins. Though the behavioral level of Verilog is a high level description of a digital system, it is still a precise notation.

Programs written in Verilog notation can be compiled to get the circuits consisting of logic gates and flip flops. One could then go to the lab and wire up the logical circuits and have a functioning system. And, other tools can “compile” programs in Verilog notation to a description of the integrated circuit masks for **very large scale integration** (VLSI). Therefore, with the proper automated tools, one can create a VLSI description of a design in Verilog and send the VLSI description via electronic mail to a **silicon foundry** and receive the integrated chip. Verilog also allows the designer to specify designs at the logical gate level using **gate constructs** and the transistor level using **switch constructs**.

Our goal in the course is not to create VLSI chips but to use Verilog to precisely describe the *functionality* of *any* digital system. However, a VLSI chip designed using Verilog‟s behavioral constructs will be rather slow and be wasteful of chip area. The lower levels in Verilog allow engineers to optimize the logical circuits and VLSI layouts to maximize speed and minimize area of the VLSI chip.

**Levels of Abstraction in Verilog Programming**:

1) Gate Level Modeling

2) Data Flow Modeling

3) Behavioral Modeling

4) RTL Modeling

For the sake of simplicity, this manual takes you through all the levels of abstraction with a sample code for a 2:1 Multiplexer.

Before looking into the various levels of abstraction, we shall define Modules and Instances.

**Module**

A Module is a basic building block in Verilog. A module provides the necessary functionality to the higher-level block through its port interface (inputs and output) but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design and its functionality.

**Please refer Sec2.3 of the book “Verilog HDL” by Samir Palnitkar.**

**Instance**

A module provides a template from which you can create objects/ instances of the module type. When a module is invoked, Verilog creates a unique object from the template. The process of creation objects from module template is called instantiation and the objects are called instance.

**Please refer Sec2.4 of the book “Verilog HDL” by Samir Palnitkar.**

**Gate Level Modeling**

This is the basic level of modeling in terms of logic gates and the connections between these gates. Most digital design is now done at the gate level or higher levels of abstractions. At gate level, the circuit is described in terms of gates say AND, OR etc. Hardware design at this level is intuitive for a user who is familiar with the basic knowledge of Digital logic Design. This allows the user to see a direct correspondence between the Verilog Description and the Circuit Diagram.

**Example:**



Figure 1. 2:1 Multiplexer

In terms of Logic Gates, Out= OR (AND (S, I1), AND ((NOT (S)), I0))

**Verilog Code:**

module mux2to1\_gate (I0,I1,S,Out);

input I0,I1,S;

output Out;

wire c,d,e;

not n1(e,S); // e=~s

and a1(c,I1,S);

and a2(d,I0,e);

or o1(Out,c,d);

endmodule

For Further reading on Gate Level Modeling refer Chapter 5 of the book “Verilog HDL” by Samir Palnitkar.

**Data Flow Modeling**

The design at this level specifies how the data flows between the hardware registers and how the data is processed. For small circuits the gate level modeling works well as the number of gates is limited. However, in complex designs the designers may have to concentrate on implementing the function than bother about the gates. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of gates using expressions (=), operators like (&,|, ?) etc.. and continuous assignments(the assign statement).

The 2:1 Multiplexer can be written as

F= (S&I1)|(~S&I0);

Or

F = S ? I1 : I0;

**Verilog Code:**

module mux2to1\_df (I0,I1,S,Out);

input I0,I1,S;

output Out;

assign Out = S ? I1 : I0;

endmodule

For Further reading on Data Flow Modeling refer Chapter 6 of the book “Verilog HDL” by Samir Palnitkar.

**Behavioral Modelling**

This is the highest level of abstraction provided by Verilog. The design at this level is similar to an algorithm. This design is very similar to „C‟ programming. A module can be implemented in terms of the desired design algorithm without looking into the hardware details using structured procedures (like *always* and *initial*), conditional statements (like *if* and *else*) and multiway branching (like *case*, *casex* and *casez*).

[Note: THOUGH THE CODING STYLE IS SIMILAR TO C, THE DIFFERENCE IS THAT, THE VERILOG PROGRAM CONSISTS OF MODULES THAT MAY RUN CONCURRENTLY. Remember, that you need to simulate hardware and not software.]

The 2:1 Multiplexer can be written as

If(S= = 1)

Out=I1;

Else

Out=I0;

**Verilog Code:**

module mux2to1\_beh(I0,I1,S,Out);

input I0,I1,S;

output Out;

reg Out;

always@(S or I0 or I1)

if(S==1) Out = I1;

else Out = I0;

endmodule

For Further reading on Behavioral Modeling refer Chapter 7 of the book “Verilog HDL” by Samir Palnitkar.

**RTL Modeling**

The term Register Transfer Level Modeling refers to the Verilog description that uses a combination of both Behavioral and Data Flow constructs that is synthesizable.

For the input and output connections for the ports, please refer Sec 4.2.3 Port Connection Rules.

**Simulation**

**Testbench or Stimulus**

Once a design block is completed, it must be tested for its correctness. The functionality of a design block can be tested by applying stimulus and checking the results. We call such a block Stimulus or TestBench. It is recommended to keep the design and the stimulus blocks separate. Given below is an example to test the 2:1 Multiplexer we have designed so far.

**Code:**

Same order as in module (Positional mapping)

module testbench;

reg tI0,tI1,tS;

wire tOut;

mux2to1\_gate mux\_gate (tI0,tI1,tS,tOut);

// mux2to1\_gate mux\_gate (.I1(tI1), .I0(tI0), .S(tS), .out(tOut));

initial

begin

$monitor(,$time," a=%b, b=%b, s=%b f=%b",a,b,s,f);

Order not important (Named Mapping)

#0 a=1'b0;b=1'b1;

#2 s=1'b1;

#5 s=1'b0;

#10 a=1'b1;b=1'b0;

#15 s=1'b1;

#20 s=1'b0;

#100 $finish;

end

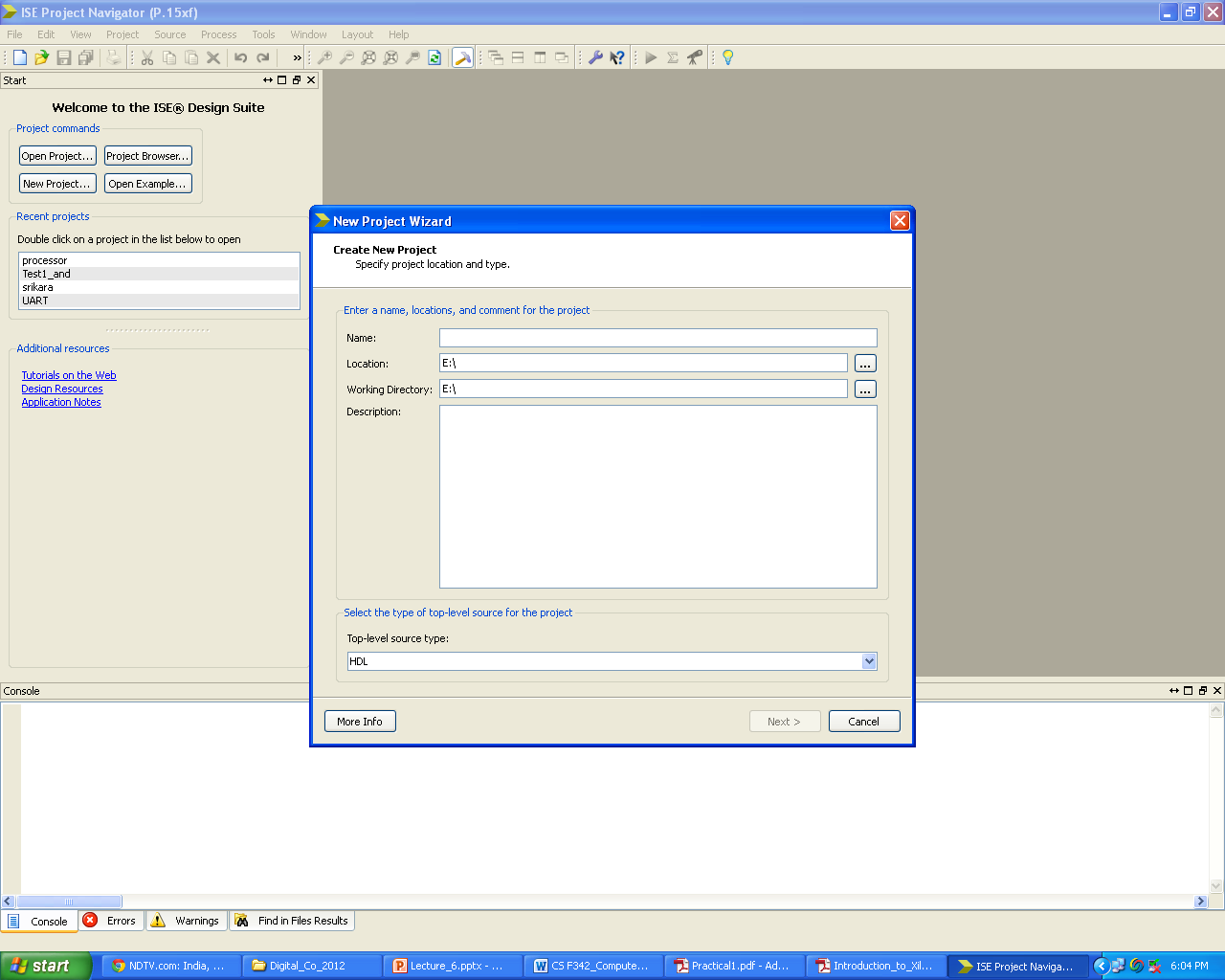
endmodule

**Experiment No 1: How to use Xilinx ISE, Revisiting Verilog**

**1.1 Using Xilinx ISE**

This part of experiment is intended to make the student to use Xilinx ISE for implementing a 4-bit adder using a 1-bit full adder in Verilog. The steps to write a Verilog code using Xilinx ISE are shown below.

1. Open Xilinx ISE Project Navigator (shortcut will be on the Desktop)



1. Give project Name in the space provided. The name of the Project should follow the naming convention as below.

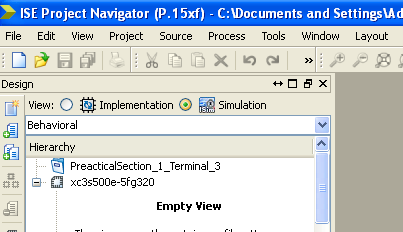
PracticalSection\_[1/2/3]\_Terminal\_[Terminal Number]

e.g. if you are from Practical section 1 (Monday batch) on terminal 4 then the name should be

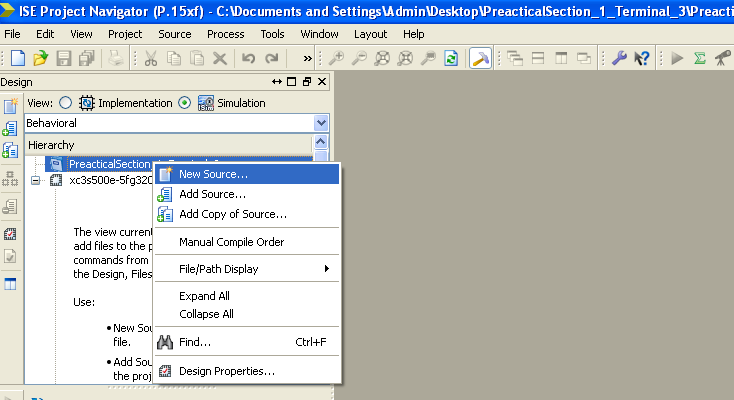
PracticalSection\_1\_Terminal\_4

Change the Location to Desktop. Click **“Next”** and make sure that preferred language entry is Verilog and “**Finish”**. All the Verilog files should be in the same project.

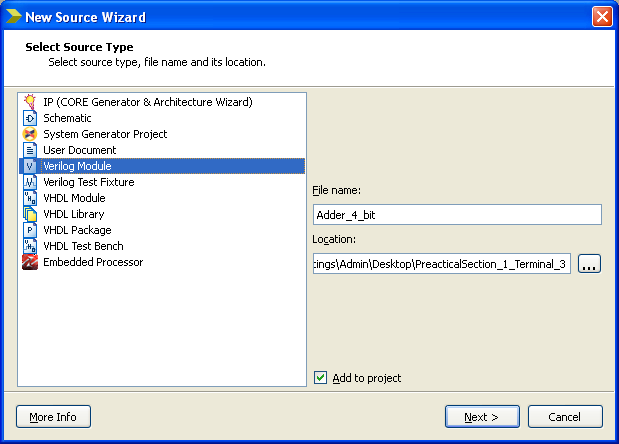
1. Change the design **View:** option to “**Simulation”** mode



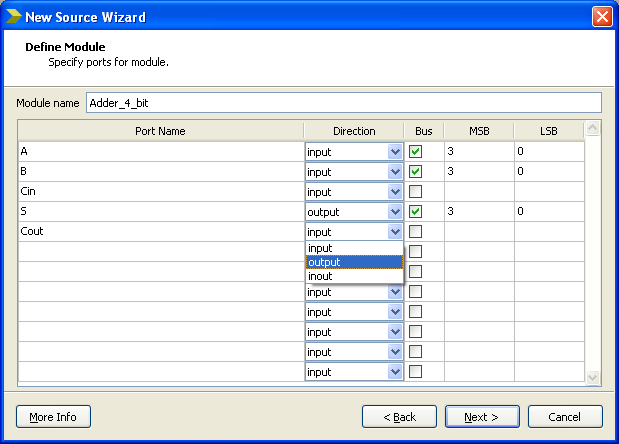
1. To add Verilog files in to the project, Right click on the project name and select **“New Source”**



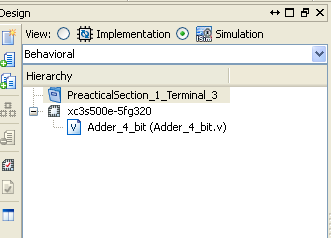
1. In the New source wizard window that appears select source file type as Verilog module and give a **suitable** file name. The file name will act as module name for your Verilog code. In most experiments the module names will be given.



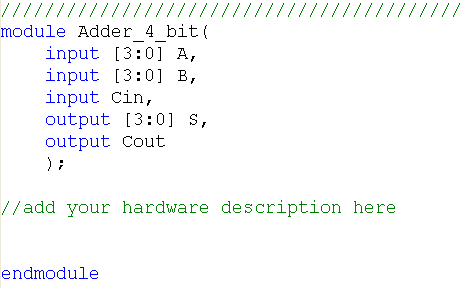
1. Click on Next tab and define the input/output ports along with their sizes.



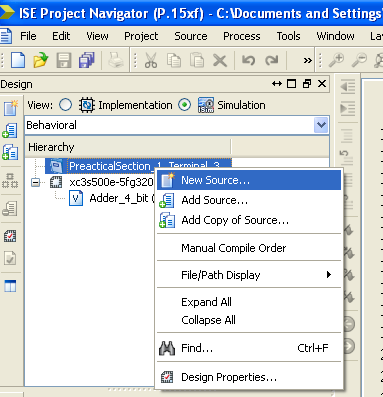
Your Verilog file will be added to the project



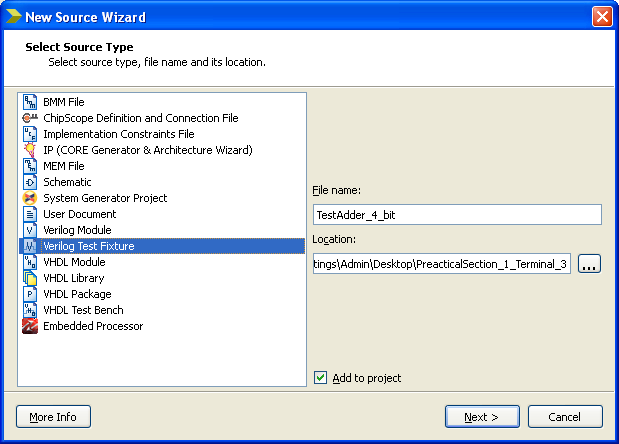
You can start describing your hardware in the main window



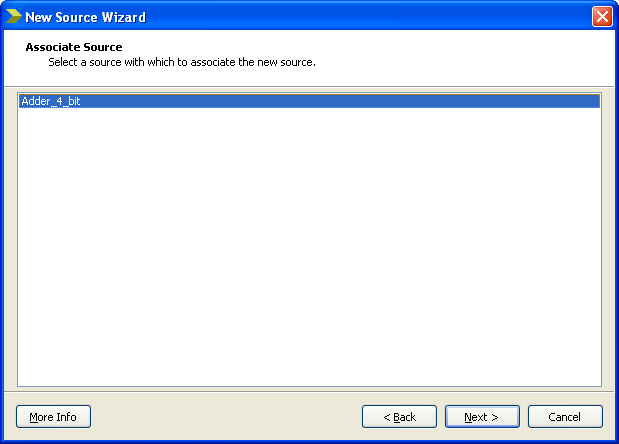
1. Once the Verilog code is complete you have to write the test bench for Verilog module. **Every design module must be tested with the help of testbench.** To create test bench right click on the project name and select “**New source”**.



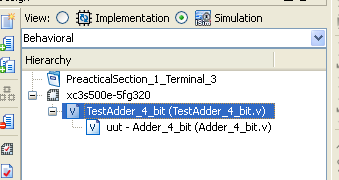
1. In the new source wizard window select Verilog Test Fixture and select appropriate test bench name.



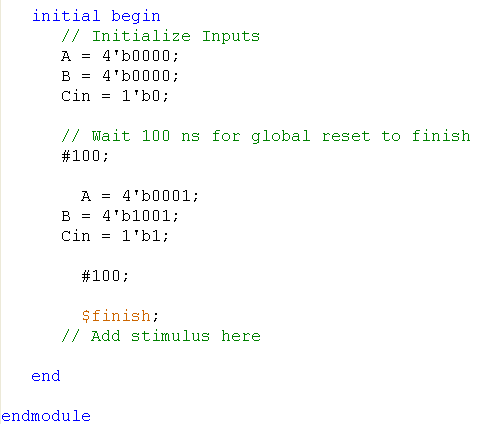
1. Click on **“Next”** and select the design module that you are testing



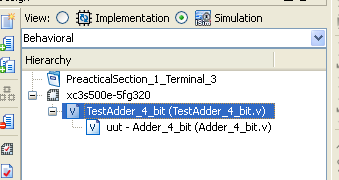
1. Click on **“Next”**  and **“Finish”**
2. Once the Testbench is created the design module file will get listed under the Testbench

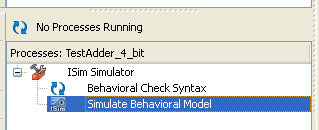


1. Add the stimulus (different input values by specifying some delay between two successive input cases). After the stimulus is given you can use $finish; command to indicate the end of simulation.



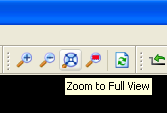
1. To simulate the testbench select the testbench file (in hierarchy window) and double click the “Simulate Behavioral model” in the process window





1. Once the simulation is done select the “Default.wcfg” tab in the ISim window to see the wave forms



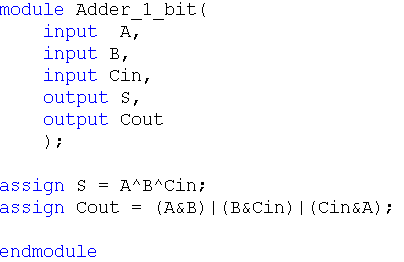


1. For better view of the waveforms select “Zoom to Fullview” option in Isim window.

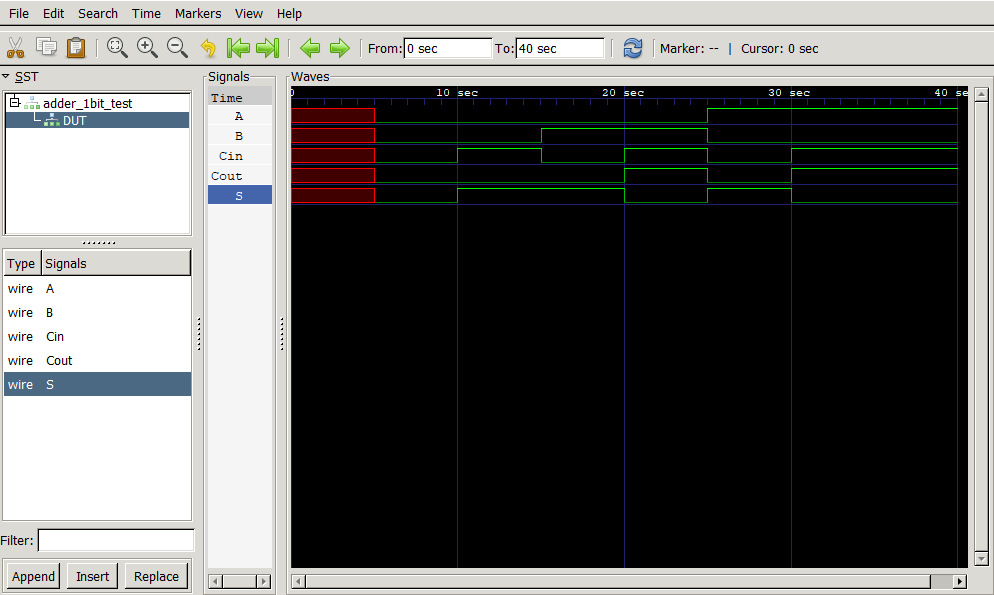
**1.2 Implementation of 4-bit Adder**

**Exercise 1.1: Implement 1-bit Full adder using Gate-level and Data-flow modelling. Test 1-bit adder.**

**Code for 1-bit adder using Date-flow modelling: Adder\_1\_bit.v**

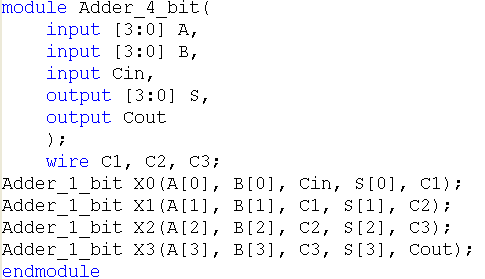


**Paste the wave from images showing the simulation for 1-bit adder.**

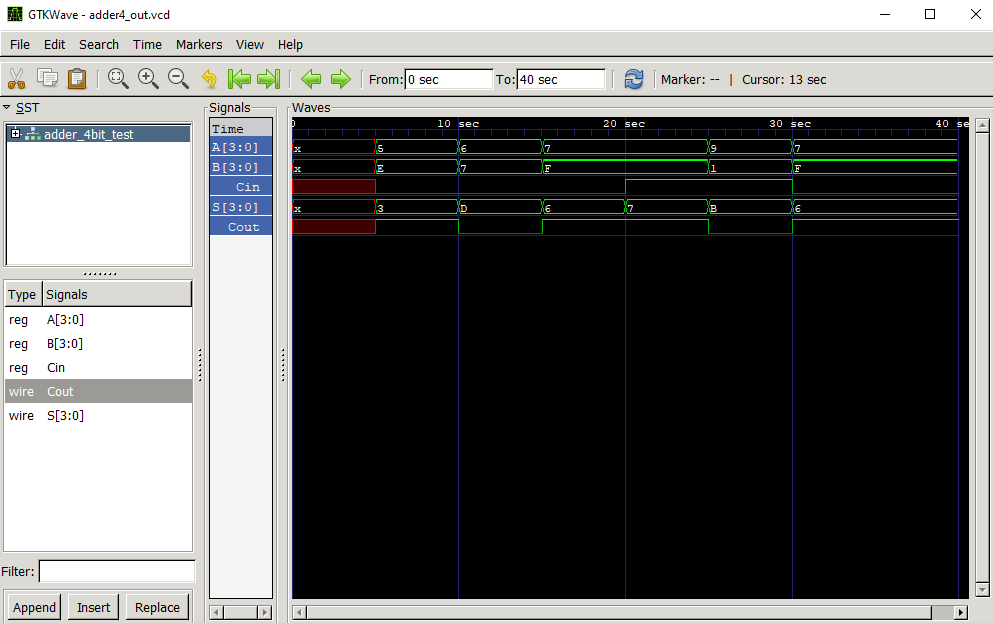
Answer:  ****

**Exercise 1.2: Implement 4-bit adder by instantiating the 1-bit module. Test 4-bit adder**

**Code for 4-bit adder using 1-bit adders: Adder\_4\_bit.v**

****

**Paste the wave from images showing the simulation for 4-bit adder.**

Answer:  ****

**What does ^ symbol indicate?**

Answer: XOR

**What is the significance of keyword wire?**

Answer: It creates a net type variable which must be constantly driven to hold a value, much like a real-life wire.

**What do X0, X1, X2 and X3 indicate?**

Answer: Names of the instantiated 1 bit adders

**What is the basic difference between gate-level and data-flow modeling?**

Answer: Gate level modelling lets us model circuits by instantiating primitive gates. Data flow modelling lets us describe the circuit according to data flow between registers and the gates are then synthesized

* 1. **Decoder and Multiplexer:**

Decoders and multiplexers are the basic combinational blocks which can be used in many applications. The decoder typically has n inputs and 2n outputs. A multiplexer has 2n inputs one output and n select lines. In this part of the experiment you must implement a 2: 4 decoder and 4:1 Multiplexer. The block diagram and basic implementation using logic gates are shown below.

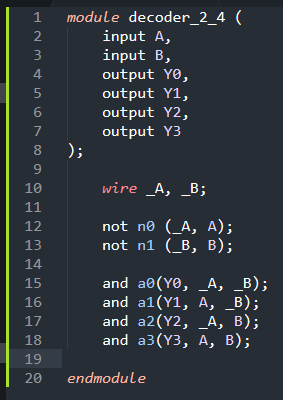


**Write the truth table for 2:4 decoder.**

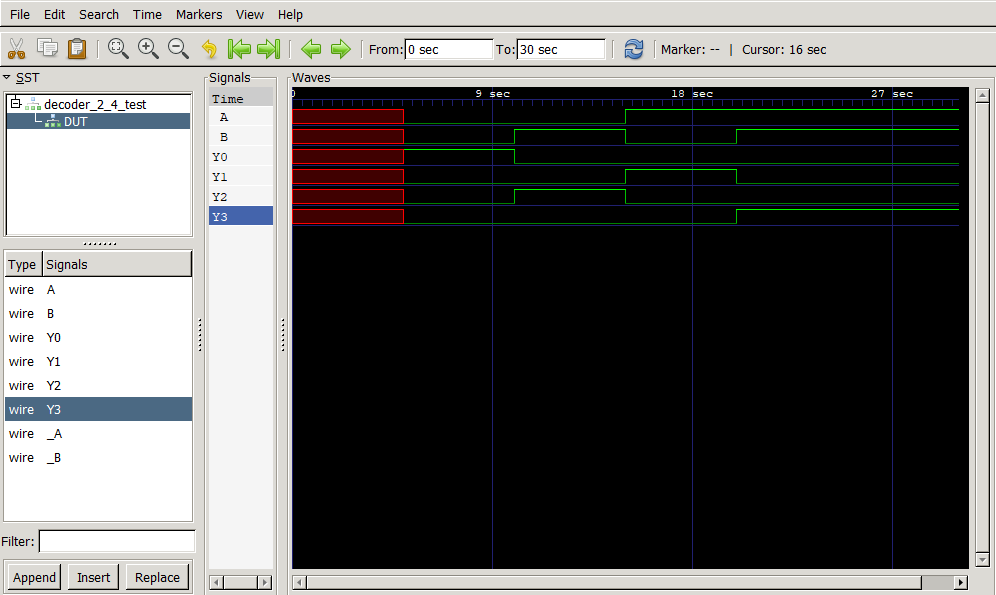
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| B | A | Y0 | Y1 | Y2 | Y3 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

# Exercise 1.3. Implement 2:4 decoder using gate-level modeling. In this design you require two inverters and four AND gates. Test the 2:4 decoder.

**Paste the image of the Verilog code used to implement 2:4 decoder using gate-level Modelling.**

Answer:  ****

**Paste the wave from images showing the simulation for 2:4 decoder using gate-level modelling.**

Answer: ****

**Did you get any errors during the simulation of the above Verilog code? If yes then How did you solve the error?**

Answer: Nope

**Did you get any errors during the simulation of the testbench? If yes then How did you solve the error?**

Answer: Yes, I had accidentally take AB instead of BA as inputs. Switched them around to solve it.

**For behavioral implementation of 2:4 decoder if Y0,Y1,Y2,Y3 are not declared as reg and are only declared as output, What error will you get?**

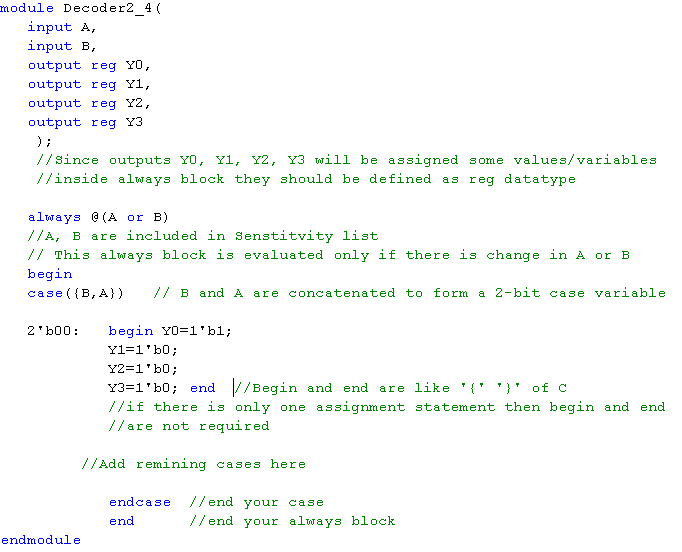
Answer: “Not a valid l-value”

# Behavioral coding:

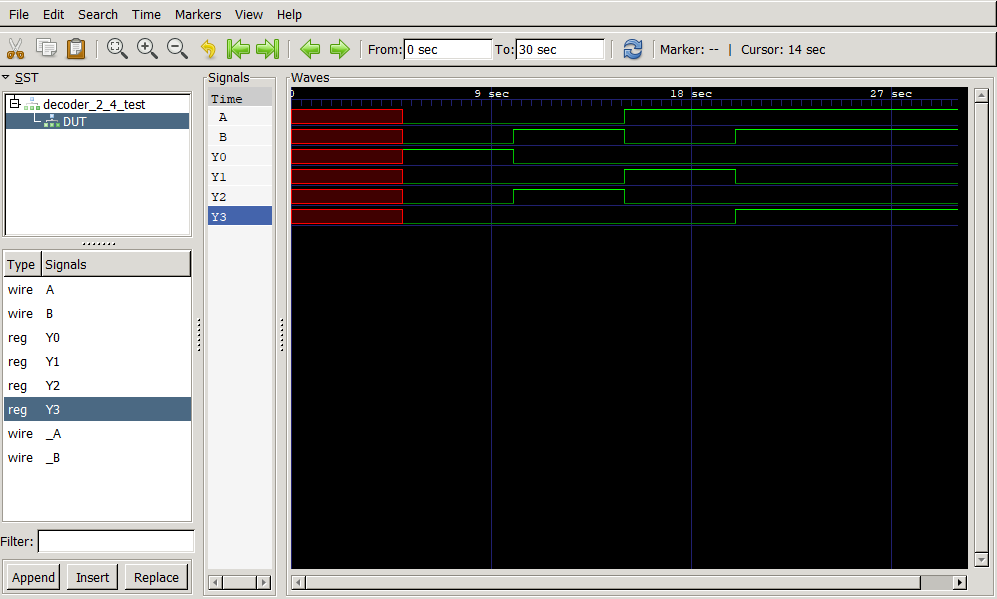
Behavioral modeling is style of modeling where the hardware designs are described in algorithmically. The behavioral design style uses two constructs ***always*** and ***initial***. The ***initial*** construct is used only in test benches. The ***always*** block will contain an event control expression also called sensitivity list and procedural assignment statements. The target output of the procedural assignment statements must be of ***reg*** data type. The **reg** data type retains its value until new value is assigned.

# Exercise 1.4. Implement 2:4 decoder using Behavioral modeling. Test the 2:4 decoder.

The partial code for 2:4 decoder is shown below:

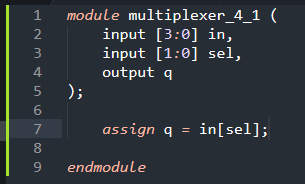


**Paste the wave from images showing the simulation for a 2:4 decoder implemented using behavioral modeling.**

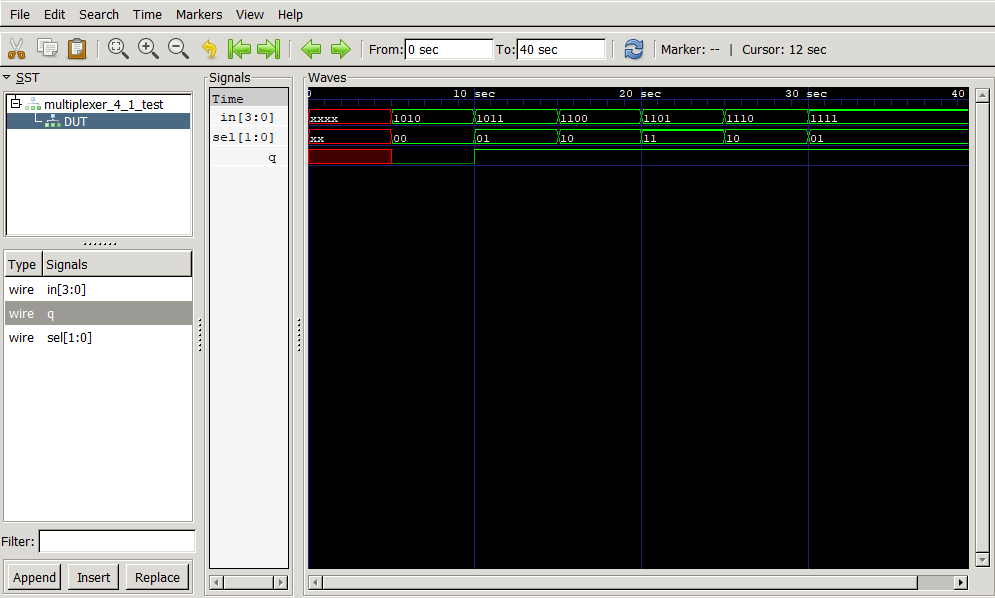
Answer:  ****

# Exercise 1.5. Implement 4:1 multiplexer using Behavioral modeling. Test the 4:1. Multiplexer.

**Paste the image of the Verilog code used to implement 4:1 Multiplexer using behavioral Modelling.**

Answer: ****

**Paste the wave from images showing the simulation for a 4:1multiplexer implemented using behavioral modeling.**

Answer: 

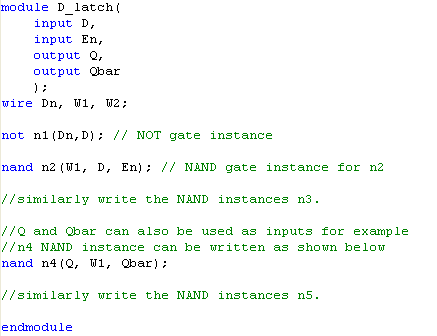
**D-Latch and D-Flip-Flop**

A D-latch circuit with active high enable (i.e. D-latch is positive level sensitive) is shown in the figure below.



**Exercise 1.6 Write Verilog code for gate-level design of D-latch using primitive gates. In this design you require four 2-input NAND gates and one inverter.**

**Partial Verilog code D-latch**

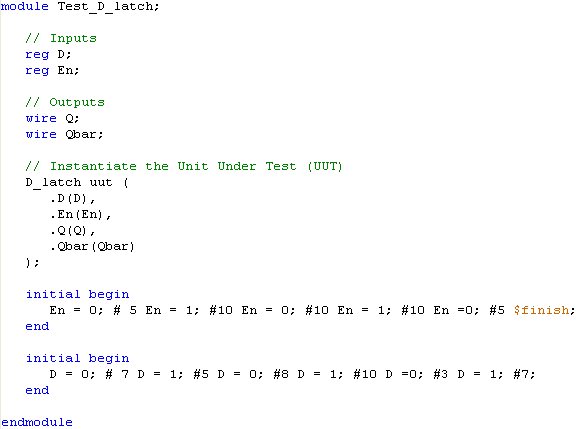
****

**Testing of D-latch**

For testing the latches and flip-flops a particular waveform might have to be generated as inputs. This section explains about generation of specific waveforms for driving inputs. For the above example of D-latch let us assume the following waveforms have to be given as inputs D and En.



The simplest way to generate two different waveforms in the test bench is to have two separate initial blocks. In the wave forms above En is 0 for first 5 time units and then is logic 1 for next 10 time units and so on. The test bench for the D-latch with above input patterns is shown below. Check the output for the above test pattern.

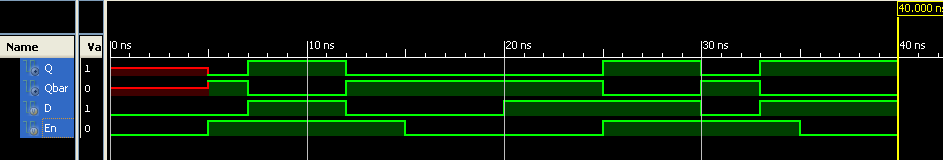


Generates D

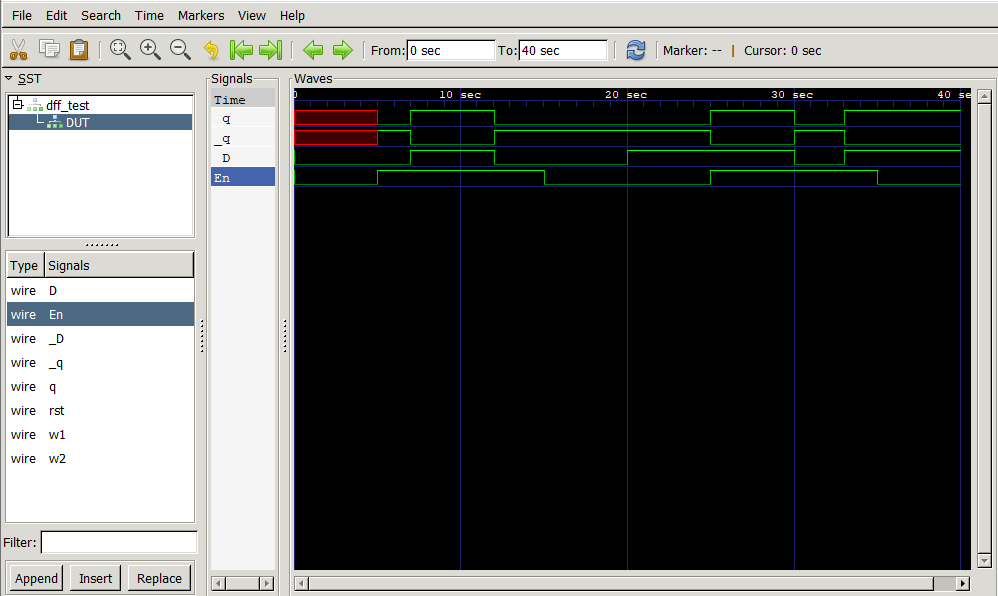
Generates En

// End the simulation after 40 time units

**Expected Output**

****

**Paste the wavefrom images showing the simulation for a D latch for given test bench.**

Answer: ****

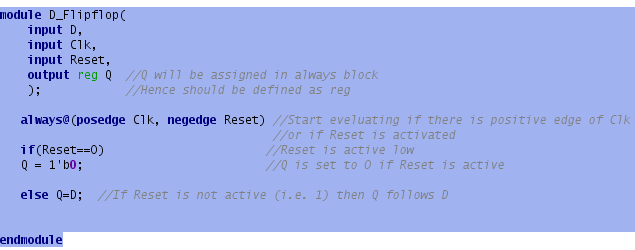
**Exercise 1.7 : Implement a D Flip-flop using behavioral modeling. Test the D flip-flop with given test pattern.**

**Design and Implementation of D Flip-flop:**

Write Verilog code for D flip-flop using behavioral modeling.



**Verilog code D Flip-flop**

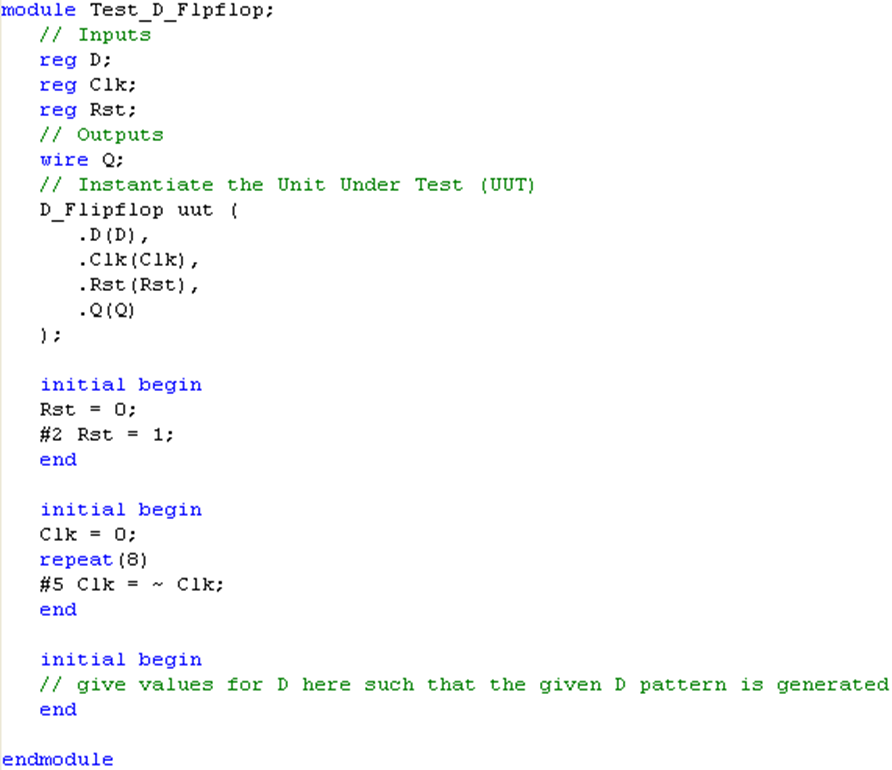
****

**Testing of D Flip-flop**

D Flip-flops need clock as one of the inputs. This section explains the generation of clock and other input patterns for D-flip-flop. For the above example of D Flip-flop let us assume the following waveforms have to be given as inputs D, Clk and Rst.

**Important:** Flip-flop based circuits should be initially reset. In the above example Rst is made 0 for 3 time units so that the flip-flop initializes to logic ‘0’. After 3 time units the Rst is given logic 1 to check the normal operation of D Flip-flop.

Here we have to generate three different waveforms. The simplest way of doing that is to have three different initial blocks in the test bench. The partial code for test bench is shown below.

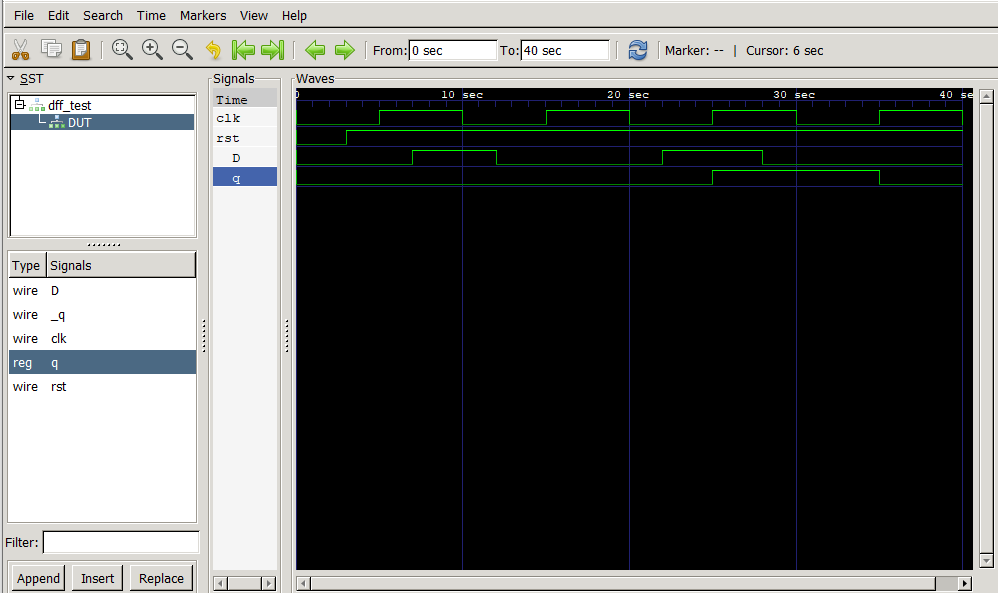


// generate D and End simulation after 40 time units

// Generates four Clock cycles with time period of 10 units

// Generates Rst

**Paste the wavefrom images showing the simulation for a D flip-flop for given test bench.**

Answer:  ****

**Can you construct a D flip-flop using D latches? If yes then what is the name given to such a flip-flop**

Answer: Yes, it’s called a master-slave flipflop

**In the test bench for D latch and D flip-flop multiple initial blocks are used. Will the multiple initial blocks execute sequentially or in parallel?**

Answer: In parallel

**What is the function of Reset in D flip-flop?**

Answer: It sets the output to 0, to initialize the flop

**What is the function of keyword ‘repeat’ that is used in test bench of D Flip-flop?**

Answer:It repeats the line(s) after it the specified amount of times.

**List the concepts you learnt from this experiment (Conclusions/Observations)**

Answer: I was able to revise a bunch of Verilog keywords and basic testbenching.