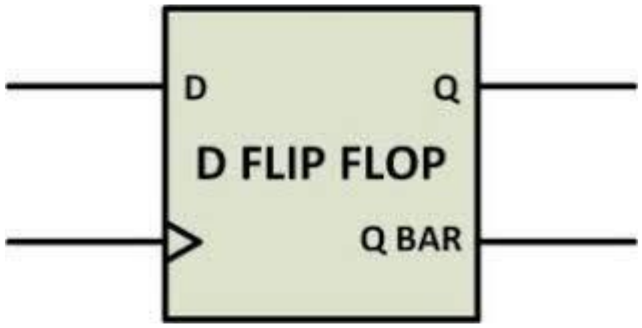
EARTHPEOPLE Technology

Verilog Programming Guide

INTRODUCTION

Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**. A hardware description language is a language used to describe a digital system: for example, a network switch, a microprocessor or a memory or a simple flip-flop. This just means that, by using a HDL, one can describe any (digital) hardware at any level.



```
// D flip-flop Code
module d_ff ( d, clk, q, q_bar);
input d ,clk;
output q, q_bar;
wire d ,clk;
reg q, q_bar;

always @ (posedge clk)
begin
  q <= d;
  q_bar <= !d;
```

```
end

endmodule
```

You could download file d_ff.v here

One can describe a simple Flip flop as that in the above figure, as well as a complicated design having 1 million gates. Verilog is one of the HDL languages available in the industry for hardware designing. It allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level. Verilog allows hardware designers to express their designs with behavioral constructs, deferring the details of implementation to a later stage in the final design.

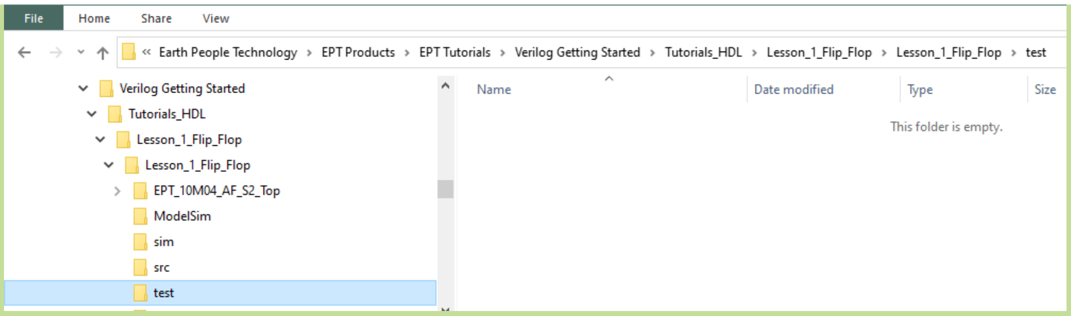
This Verilog Guide is designed to get the user familiar with simulating user code, testbenches, and test modules. We will use ModelSim software. It is free to download and provides an extensive set of tools for simulating designs. So, lets get started…

///////////////////////////////////Lesson #1///////////////////////////////////////

In this first lesson, let's set up the ModelSim environment and simulate a simple flip flop.

First, set up your file system to allow easy access to updating and modifying your files. There are many ways to set up your file system, the following is the one that works best for me.

Go to the xx_Project_xx_DVD-> Verilog Getting Started->Tutorials_HDL. Copy the Lesson 1 Folder over to our local drive from the DVD. In this folder you will notice a top level project folder.

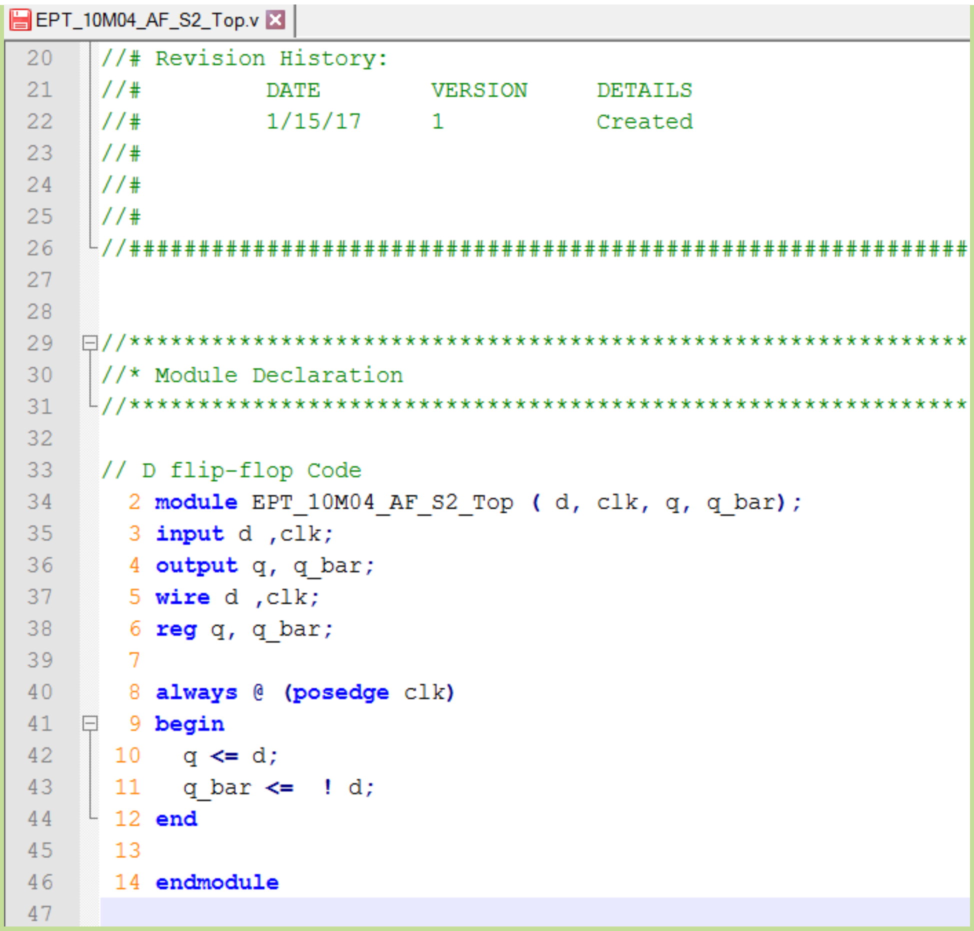


The project level folder is called "Lesson_1_Flip_Flop". Under this project level folder are various revisions of the project (The DVD will only include one revision, the user is expected to save further revisions of the project files). Multiple revisions allow you to keep copies of previously working projects and refer to them later when the currently edited source code no longer compiles.

Under the this top level folder, the project is organized with the following folders:

- **EPT_10M04_AF_S2_Top – This folder contains all the Altera project level files such as pin, sdc, configuration, programming object files etc.**
- **ModelSim – This folder contains all the compiled object files for ModelSims use. It also includes the *.do files. These 'do' files are the make files for ModelSim. They tell the compiler which source files to compile and allow compile time options.**
- **Sim – This folder contains special source files that have non-synthesizeable constructs in them.**
- **Src – This folder contains all your source files. Only synthesizeable code should go into this folder.**
- **Test – This folder contains source code for the models which are used to model behavior of devices that are not part of the FPGA. These source files can contain non-synthesizable code.**
- **Testbench -- This folder contains the main testbench source code. The testbench controls the operation of the user source code, test models and simulation code. It provides the main stimulus such as the clock and reset.**

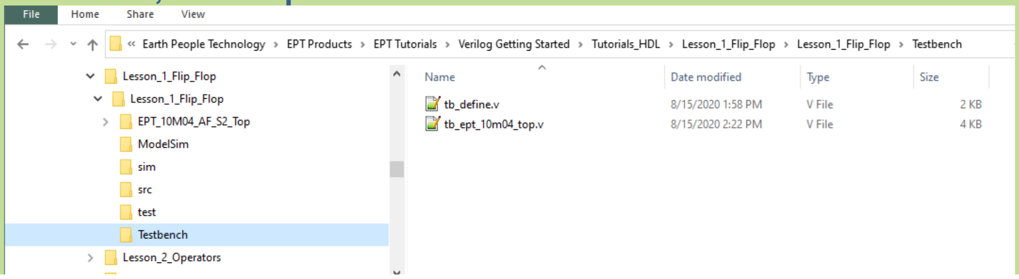
In the src folder, create a file named "EPT_10M04_AF_S2_Top.v". Then open this file in an editor, I prefer to use NotePad++. Add in the D flip-flop code and add comments to describe the file and the parts of the file.



EPT_10M04_AF_S2_Top.v

```
//# Revision History:
//#           DATE         VERSION      DETAILS
//#           1/15/17      1            Created
//#
//#
//#
//#################################################################


//*****************************************************************
//* Module Declaration
//*****************************************************************

// D flip-flop Code
  2 module EPT_10M04_AF_S2_Top ( d, clk, q, q_bar);
  3 input d ,clk;
  4 output q, q_bar;
  5 wire d ,clk;
  6 reg q, q_bar;
  7
  8 always @ (posedge clk)
  9 begin
 10   q <= d;
 11   q_bar <=  ! d;
 12 end
 13
 14 endmodule
```

Next, setup the Testbench file to provide a stimulus for our user code. The Testbench file provides all the hardware external devices that the FPGA needs to operate the user code. These include the oscillator, reset, push buttons, communications, etc… Open the Testbench folder and notice the two files in it.

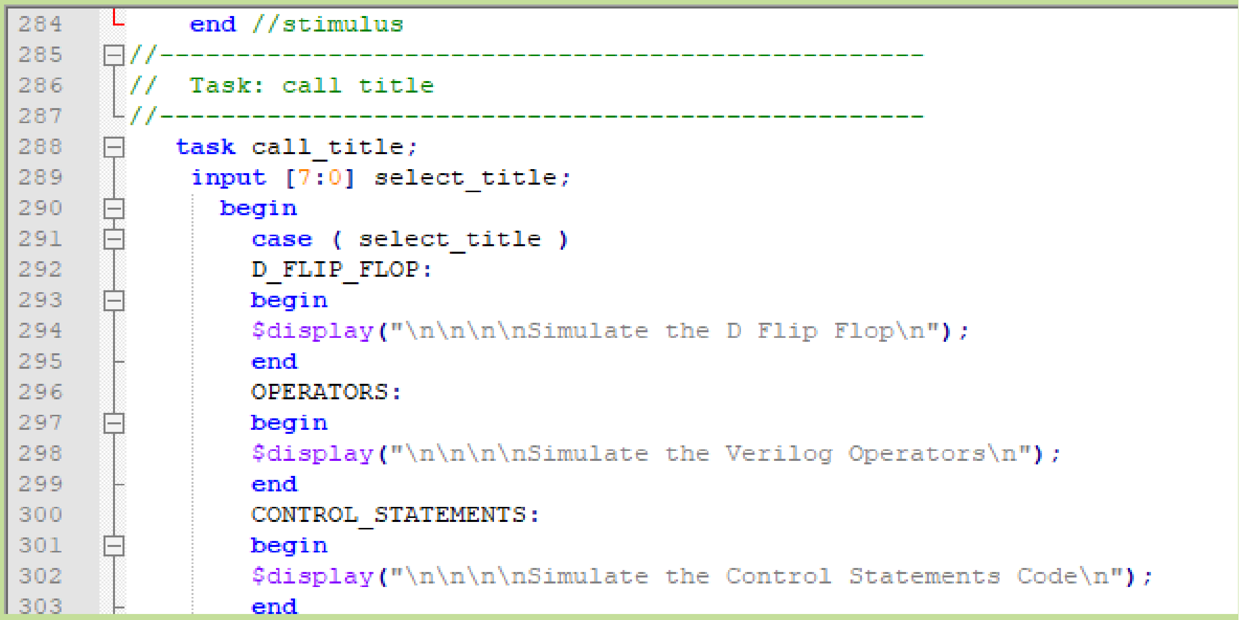


« Earth People Technology › EPT Products › EPT Tutorials › Verilog Getting Started › Tutorials_HDL › Lesson_1_Flip_Flop › Lesson_1_Flip_Flop › Testbench

- Lesson_1_Flip_Flop
  - Lesson_1_Flip_Flop
    - EPT_10M04_AF_S2_Top
    - ModelSim
    - sim
    - src
    - test
    - Testbench
- Lesson_2_Operators

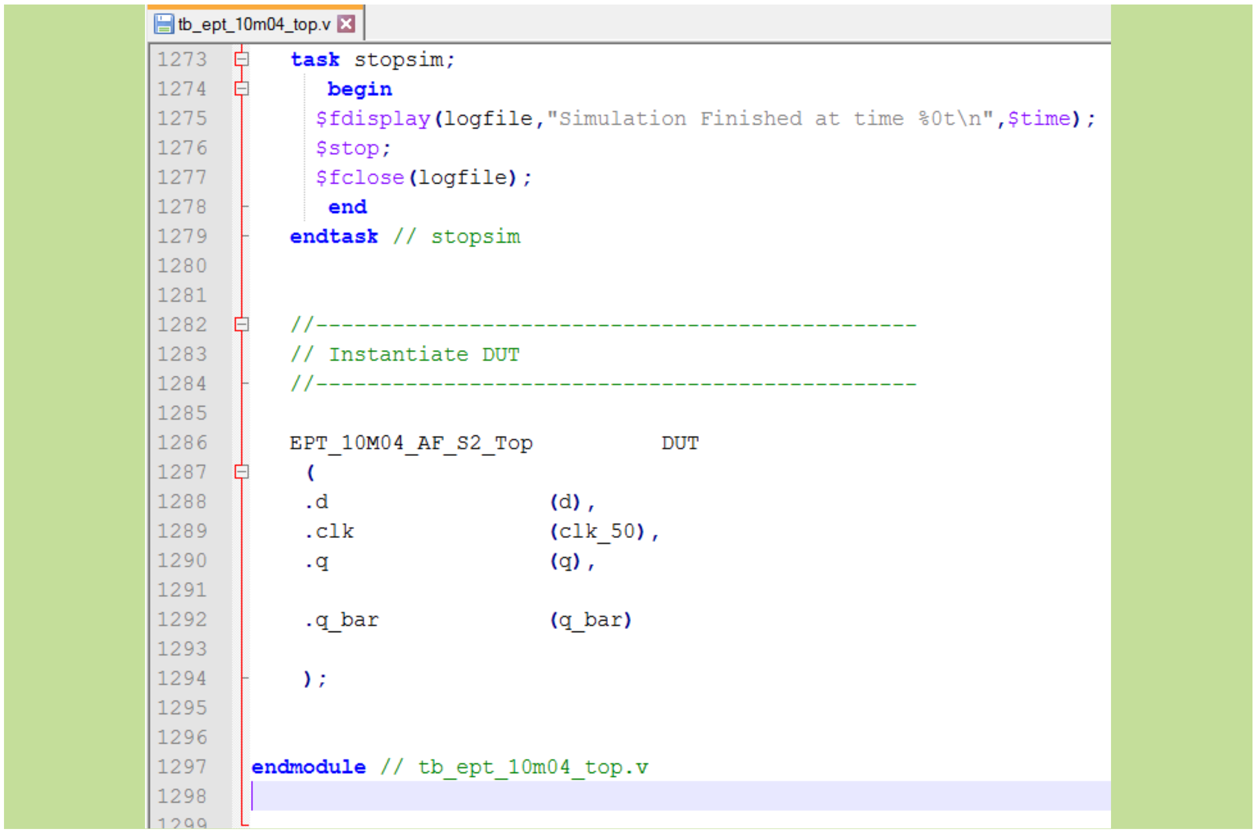
Name	Date modified	Type	Size
tb_define.v	8/15/2020 1:58 PM	V File	2 KB
tb_ept_10m04_top.v	8/15/2020 2:22 PM	V File	4 KB

- **tb_define.v contains pre-defined parameters for the testbench execution**
- **tb_ept_10m04_top.v contains declarations, stimulus, tasks and leif modules to exercise the user code.**



```
    end //stimulus
//-------------------------------------------------
//  Task: call title
//-------------------------------------------------
  task call_title;
   input [7:0] select_title;
     begin
       case ( select_title )
       D_FLIP_FLOP:
       begin
       $display("\n\n\n\nSimulate the D Flip Flop\n");
       end
       OPERATORS:
       begin
       $display("\n\n\n\nSimulate the Verilog Operators\n");
       end
       CONTROL_STATEMENTS:
       begin
       $display("\n\n\n\nSimulate the Control Statements Code\n");
       end
```

Inside the tb_ept_10m04_top.v file, we see 'module' declaration along with the name of test bench. The parameter declarations are listed to allow certain registers to have constant values. We will discuss parameters and the details of the Verilog files later in the tutorial. For now, we will focus on getting started and performing our first simulation. Scroll down the testbench file and locate the "Instantiate DUT" section.


tb_ept_10m04_top.v

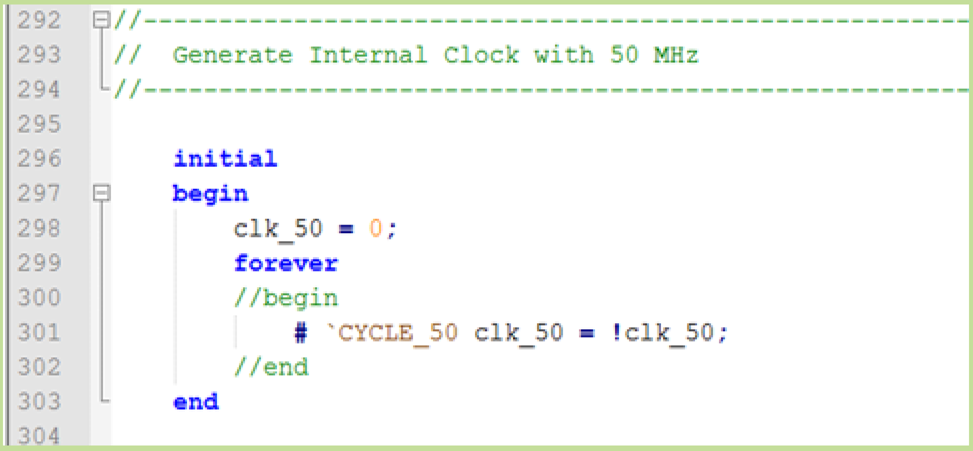
```
    task stopsim;
      begin
     $fdisplay(logfile,"Simulation Finished at time %0t\n",$time);
     $stop;
     $fclose(logfile);
      end
    endtask // stopsim


    //------------------------------------------------
    // Instantiate DUT
    //------------------------------------------------

    EPT_10M04_AF_S2_Top          DUT
     (
     .d                    (d),
     .clk                  (clk_50),
     .q                    (q),

     .q_bar                (q_bar)

     );


endmodule // tb_ept_10m04_top.v

```

You can see that the testbench uses the name of the module declared in 'EPT_10M04_AF_S2_Top.v'. This is called the leif instantiation. When ModelSim starts the compilation process, it will search the directory path for the module 'EPT_10M04_AF_S2_Top'. This instantiation must include the inputs and outputs declared in the module. In this case:

- **'d' – Input into the D flip flop**
- **'clk' – Input into the D flip flop**
- **'q' – Output from the D flip flop**
- **'q_bar' – Output from the D flip flop**

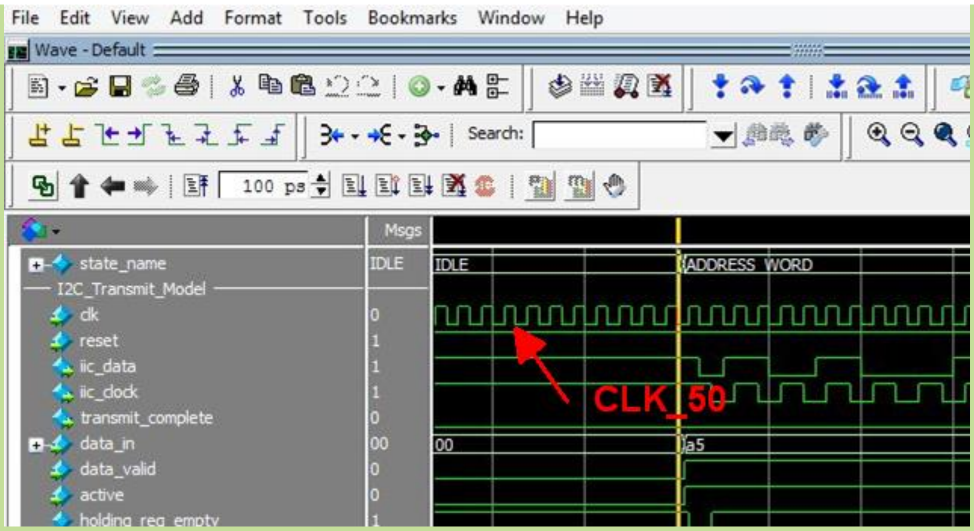
Using the leif instantiation module, we have now connected the testbench with the user code. When ModelSim starts the simulation process, testbench will control the stimulus to the inputs and accept the outputs from the user code.

To add stimulus, we use Verilog code. To add a clock, we use the 'forever' key word.

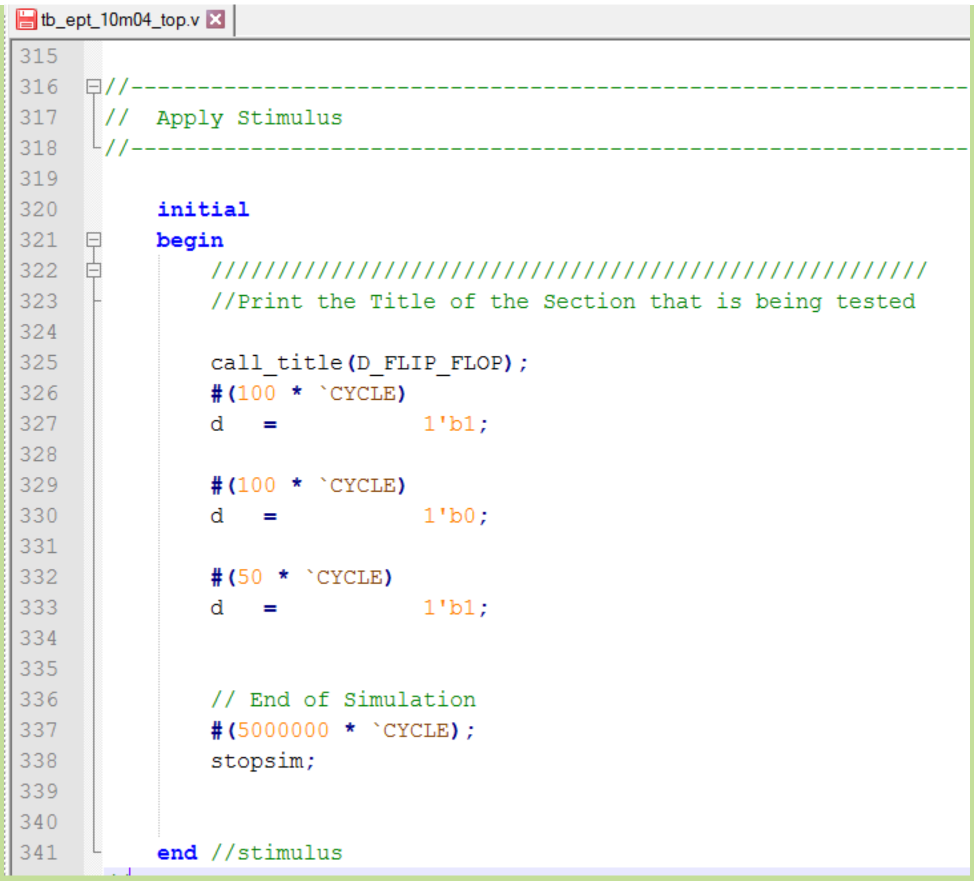


```
//-------------------------------------------------------
//   Generate Internal Clock with 50 MHz
//-------------------------------------------------------

    initial
    begin
        clk_50 = 0;
        forever
        //begin
            # `CYCLE_50 clk_50 = !clk_50;
        //end
    end

```

The parameter 'CYCLE_50' is defined in the tb_define.v file. We add the '#' character at the beginning of the line to inform the compiler to "add the following" as a delay in simulator steps. During simulation, the simulator will start a timer that halts this signal (and only this signal) and waits for the timer to expire. After the delay has expired, the signal is set equal to its complement state. Then, because of the 'forever' keyword, the process starts over, delay, then set signal to its complement. The result is a clock at 50MHz. We will go through the details of all these details later in the tutorial.



Next, add the stimulus for 'd' input. We do this using the 'initial' block. Everything between the 'begin' 'end' keywords is executed once per simulation.

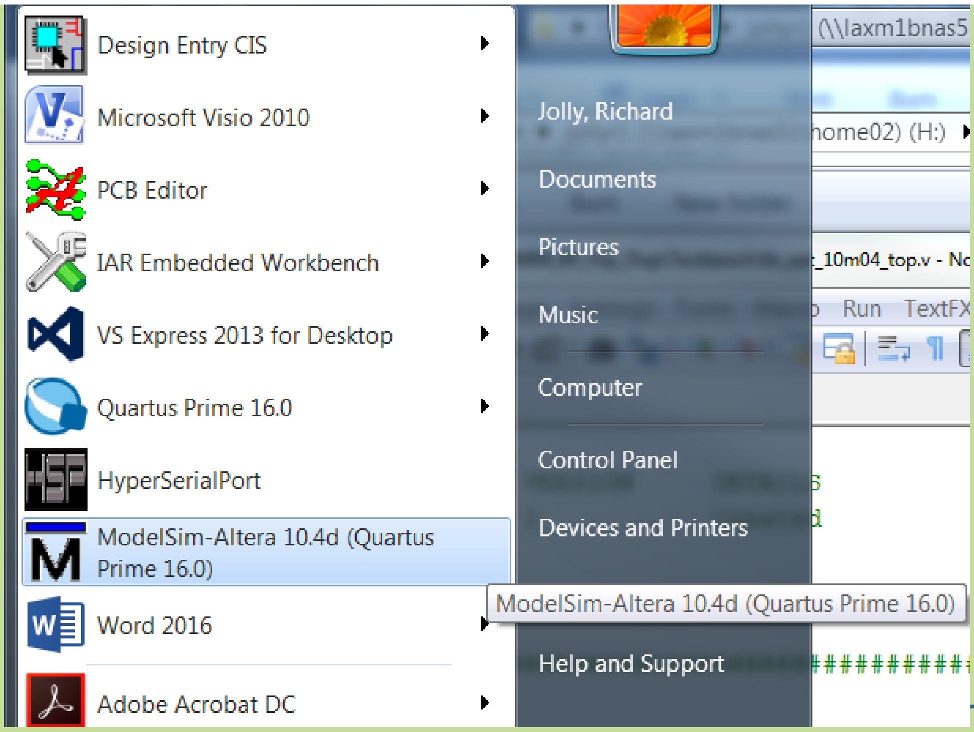


tb_ept_10m04_top.v

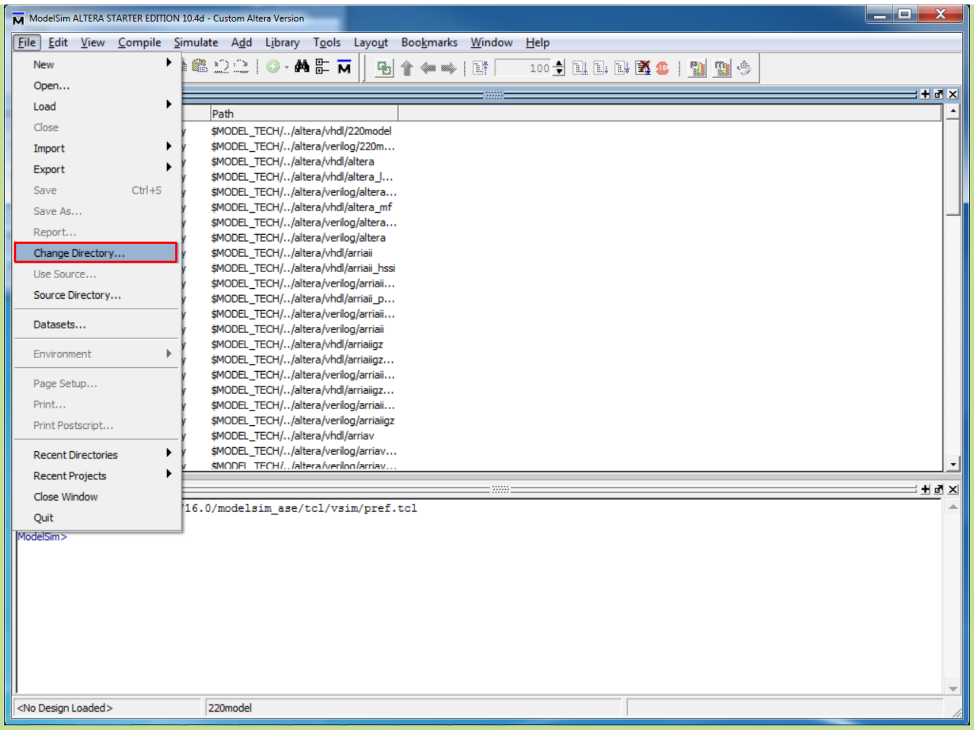
```

//------------------------------------------------------------------
//  Apply Stimulus
//------------------------------------------------------------------

    initial
    begin
        ///////////////////////////////////////////////////////
        //Print the Title of the Section that is being tested

        call_title(D_FLIP_FLOP);
        #(100 * `CYCLE)
        d   =           1'b1;

        #(100 * `CYCLE)
        d   =           1'b0;

        #(50 * `CYCLE)
        d   =           1'b1;


        // End of Simulation
        #(5000000 * `CYCLE);
        stopsim;


    end //stimulus
```

You can see here that the stimulus ‘d’ is set to 1’b1 after a delay of of 100 * CYCLE. Where ‘CYCLE’ is defined in the tb_define.v file. Then, after another delay, the ‘d’ is set to 1’b0. Finally, a delay of 50*CYCLE is added then the ‘d’ is set to 1’b1. The result is a toggle from high to low to high on the ‘d’ signal.

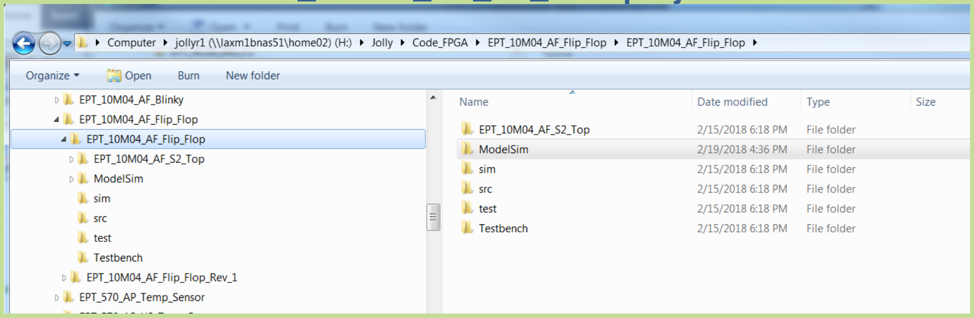
Next, get the ModelSim loaded up on your laptop/PC. Follow the install guide on how to install the ModelSim application.



Then, open the application and Go to File->Change Directory

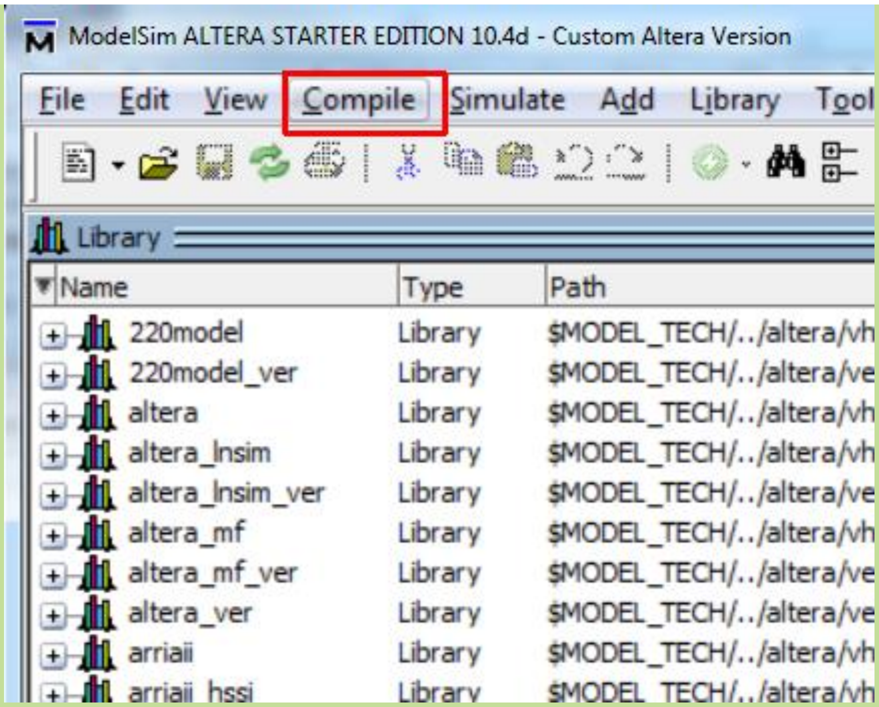


At the dialog box, locate the ModelSim folder under the EPT_10M04_AF_S2_TOP project.


Computer ▸ jollyr1 (\\laxm1bnas51\home02) (H:) ▸ Jolly ▸ Code_FPGA ▸ EPT_10M04_AF_Flip_Flop ▸ EPT_10M04_AF_Flip_Flop ▸
Organize ▾ Open Burn New folder
EPT_10M04_AF_Blinky
EPT_10M04_AF_Flip_Flop
EPT_10M04_AF_Flip_Flop
EPT_10M04_AF_S2_Top
ModelSim
sim
src
test
Testbench
EPT_10M04_AF_Flip_Flop_Rev_1
EPT_570_AP_Temp_Sensor

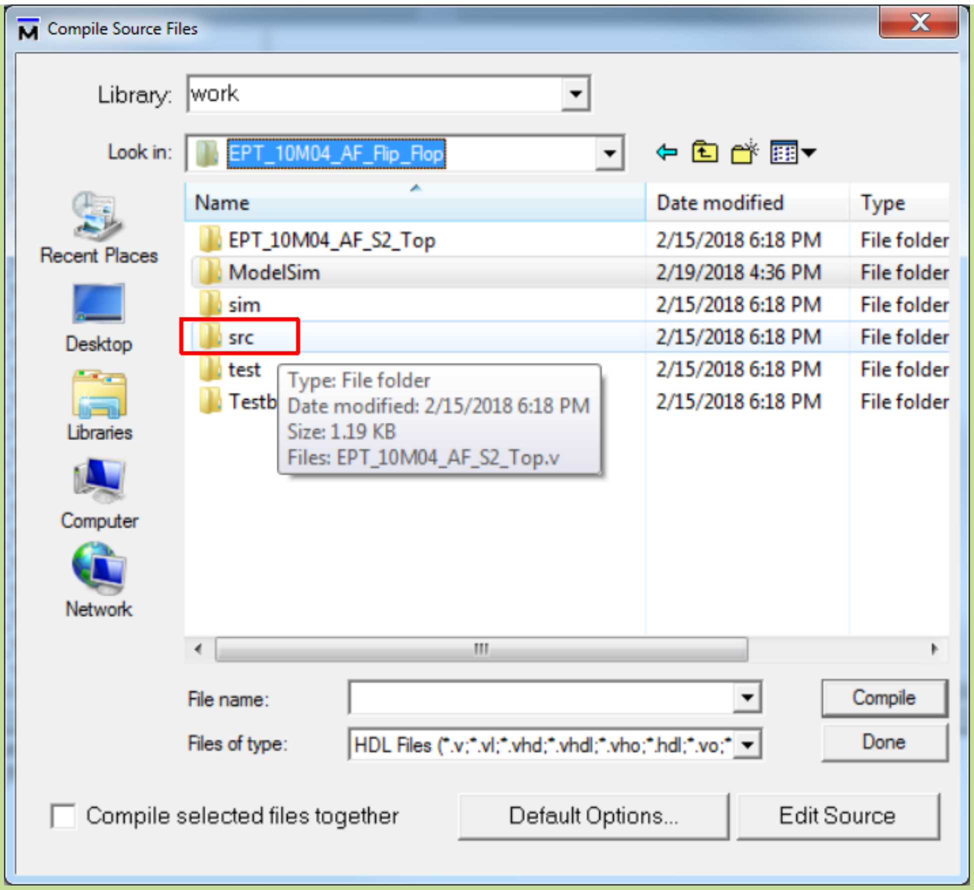
Name	Date modified	Type	Size
EPT_10M04_AF_S2_Top	2/15/2018 6:18 PM	File folder	
ModelSim	2/19/2018 4:36 PM	File folder	
sim	2/15/2018 6:18 PM	File folder	
src	2/15/2018 6:18 PM	File folder	
test	2/15/2018 6:18 PM	File folder	
Testbench	2/15/2018 6:18 PM	File folder	

Select the ModelSim folder. Next, we will compile each module individually. Click on the 'Compile' menu item. Select the 'Compile' tab.


ModelSim ALTERA STARTER EDITION 10.4d - Custom Altera Version
File Edit View Compile Simulate Add Library Tool
Library

Name	Type	Path
220model	Library	$MODEL_TECH/../altera/vh
220model_ver	Library	$MODEL_TECH/../altera/ve
altera	Library	$MODEL_TECH/../altera/vh
altera_lnsim	Library	$MODEL_TECH/../altera/vh
altera_lnsim_ver	Library	$MODEL_TECH/../altera/ve
altera_mf	Library	$MODEL_TECH/../altera/vh
altera_mf_ver	Library	$MODEL_TECH/../altera/ve
altera_ver	Library	$MODEL_TECH/../altera/ve
arriaii	Library	$MODEL_TECH/../altera/vh
arriaii_hssi	Library	$MODEL_TECH/../altera/vh

In the 'Compile Source Files' window, select the 'src' folder.



Compile Source Files

Library: work

Look in: EPT_10M04_AF_Flip_Flop

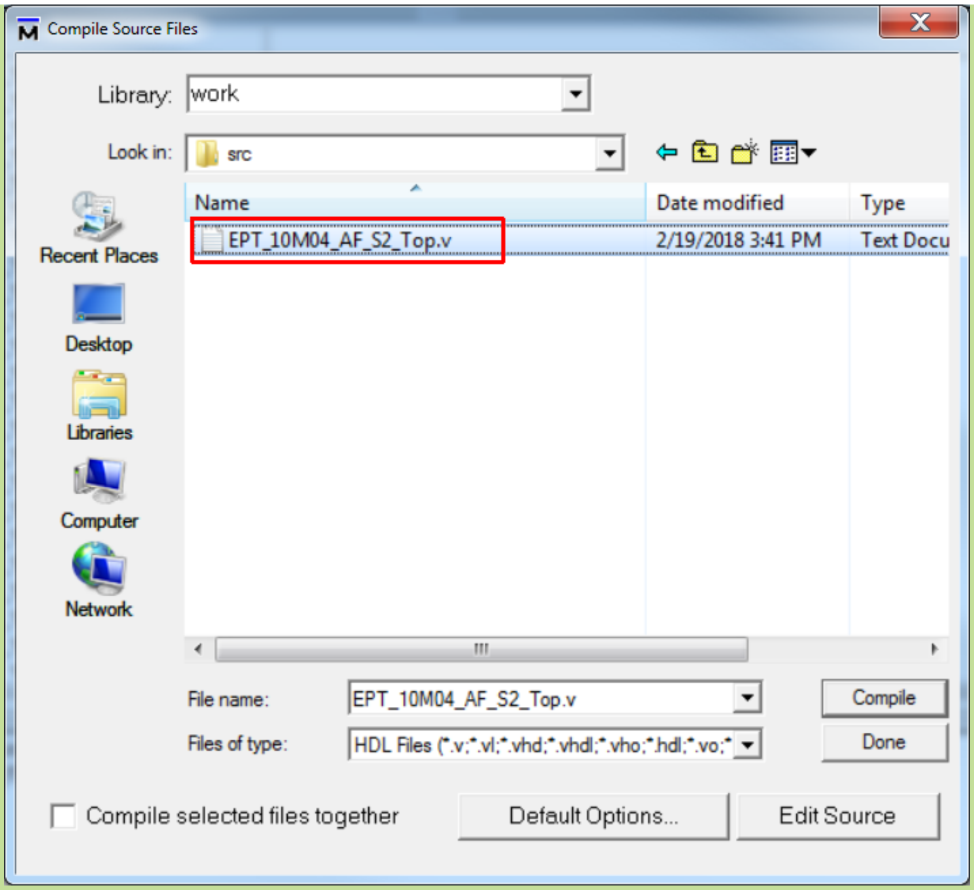
Name	Date modified	Type
EPT_10M04_AF_S2_Top	2/15/2018 6:18 PM	File folder
ModelSim	2/19/2018 4:36 PM	File folder
sim	2/15/2018 6:18 PM	File folder
src	2/15/2018 6:18 PM	File folder
test	2/15/2018 6:18 PM	File folder
Testb	2/15/2018 6:18 PM	File folder

Type: File folder
Date modified: 2/15/2018 6:18 PM
Size: 1.19 KB
Files: EPT_10M04_AF_S2_Top.v

File name:

Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

Select the 'EPT_10M04)AF_S2_Top.v' file.



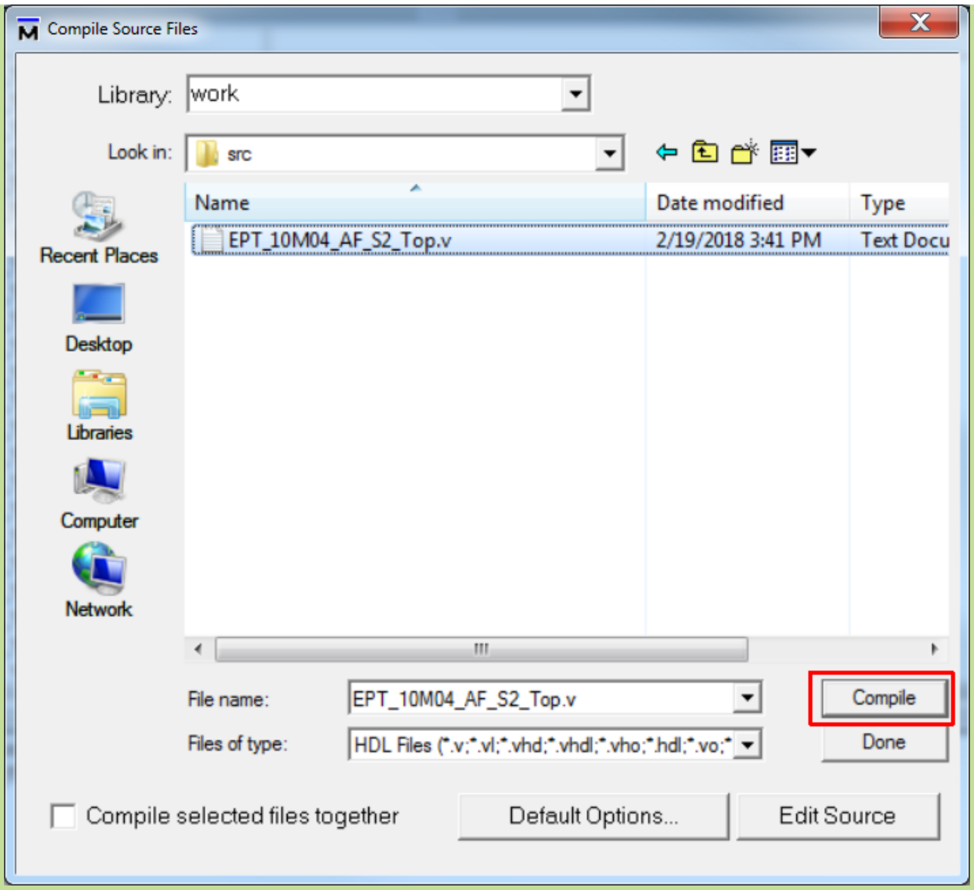
Name	Date modified	Type
EPT_10M04_AF_S2_Top.v	2/19/2018 3:41 PM	Text Docu

File name: EPT_10M04_AF_S2_Top.v

Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

☐ Compile selected files together

Click the 'Compile' button.


Compile Source Files

Library: work

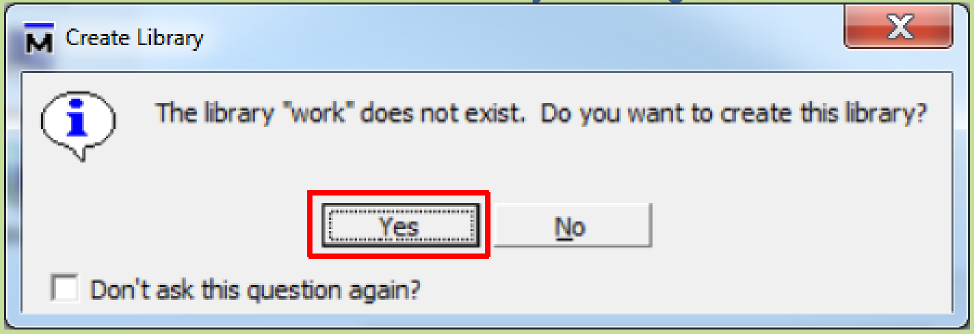
Look in: src

Name	Date modified	Type
EPT_10M04_AF_S2_Top.v	2/19/2018 3:41 PM	Text Docu

File name: EPT_10M04_AF_S2_Top.v

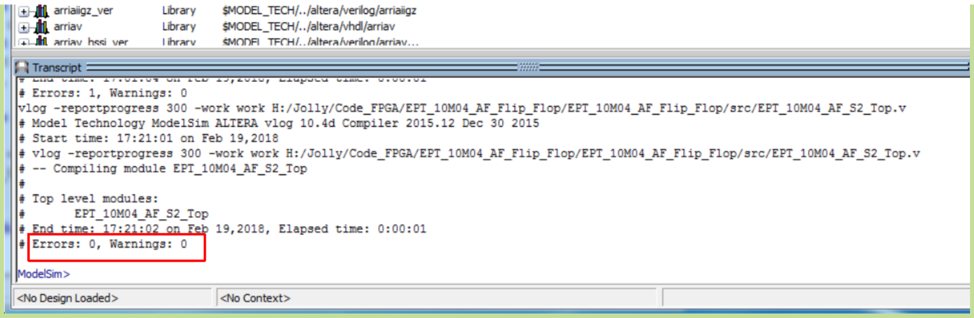
Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

You will receive the 'Create Library' message box. Select Yes.


Create Library

The library "work" does not exist. Do you want to create this library?

After the file is compiled, the log will indicate the status of compilation.

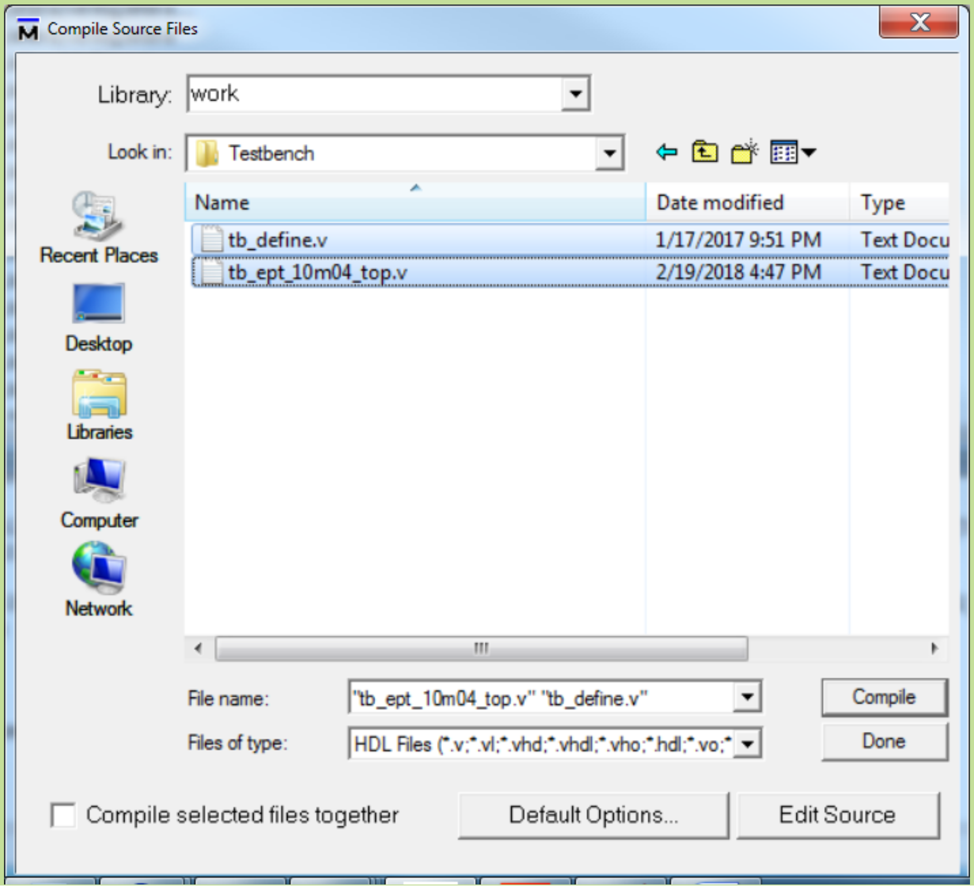


```
arriaiigz_ver      Library   $MODEL_TECH/../altera/verilog/arriaiigz
arriav             Library   $MODEL_TECH/../altera/vhdl/arriav
arriav_hssi_ver    Library   $MODEL_TECH/../altera/verilog/arriav...
```

```
# Errors: 1, Warnings: 0
vlog -reportprogress 300 -work work H:/Jolly/Code_FPGA/EPT_10M04_AF_Flip_Flop/EPT_10M04_AF_Flip_Flop/src/EPT_10M04_AF_S2_Top.v
# Model Technology ModelSim ALTERA vlog 10.4d Compiler 2015.12 Dec 30 2015
# Start time: 17:21:01 on Feb 19,2018
# vlog -reportprogress 300 -work work H:/Jolly/Code_FPGA/EPT_10M04_AF_Flip_Flop/EPT_10M04_AF_Flip_Flop/src/EPT_10M04_AF_S2_Top.v
# -- Compiling module EPT_10M04_AF_S2_Top
#
# Top level modules:
#       EPT_10M04_AF_S2_Top
# End time: 17:21:02 on Feb 19,2018, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0

ModelSim>
```

Next, repeat the compile steps for the

- **tb_define.v**
- **tb_ept_10m04_top.v**



Library: work

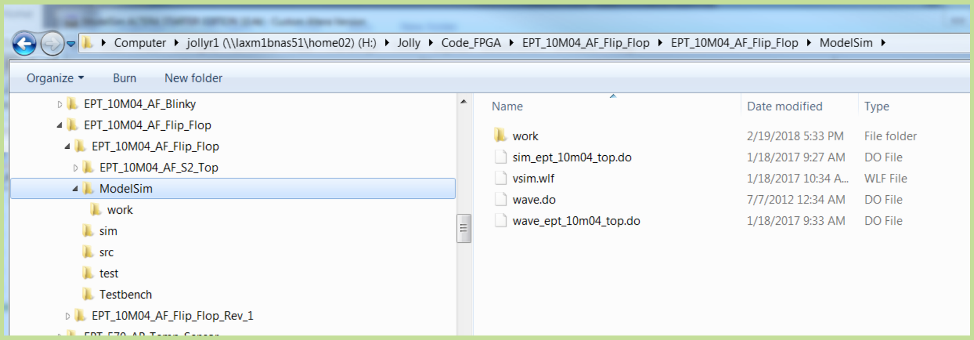
Look in: Testbench

Name	Date modified	Type
tb_define.v	1/17/2017 9:51 PM	Text Docu
tb_ept_10m04_top.v	2/19/2018 4:47 PM	Text Docu

File name: "tb_ept_10m04_top.v" "tb_define.v"

Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

Note, you will not get receive the message box asking to create the 'work' folder this time. After the compilation has completed, we are ready to run the Simulation. The simulation enviroment is controlled by the *.do files. The *.do files act as makefiles to control which files get simulated and add compile options. There are two *.do files needed for our simulation. They are found in the ModelSim folder.

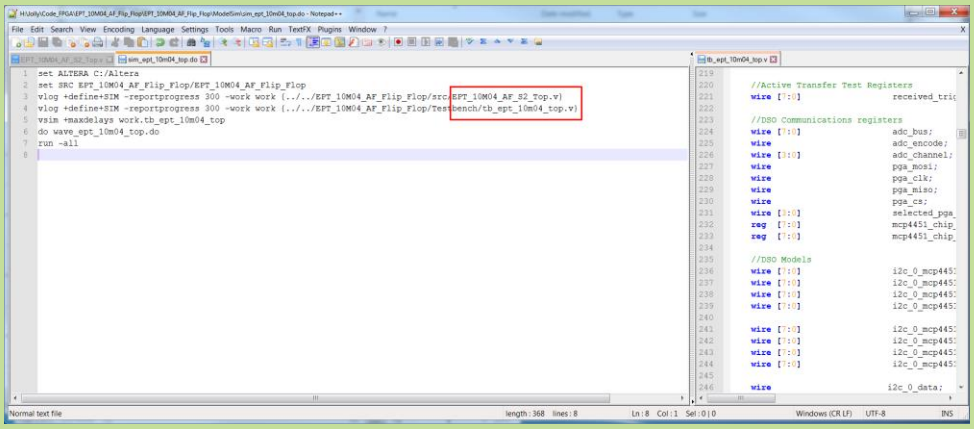


```
Computer ▸ jollyr1 (\\laxm1bnas51\home02) (H:) ▸ Jolly ▸ Code_FPGA ▸ EPT_10M04_AF_Flip_Flop ▸ EPT_10M04_AF_Flip_Flop ▸ ModelSim ▸
```

Name	Date modified	Type
work	2/19/2018 5:33 PM	File folder
sim_ept_10m04_top.do	1/18/2017 9:27 AM	DO File
vsim.wlf	1/18/2017 10:34 A...	WLF File
wave.do	7/7/2012 12:34 AM	DO File
wave_ept_10m04_top.do	1/18/2017 9:33 AM	DO File

- **Sim_ept10m04_top.do – Contains the files and compile options.**
- **Wave_ept_10m04_top.do – Contains the display flags for the 'Waveforms' window.**

When we look inside the sim_ept_10m04_top.do file we see the two files to simulate.



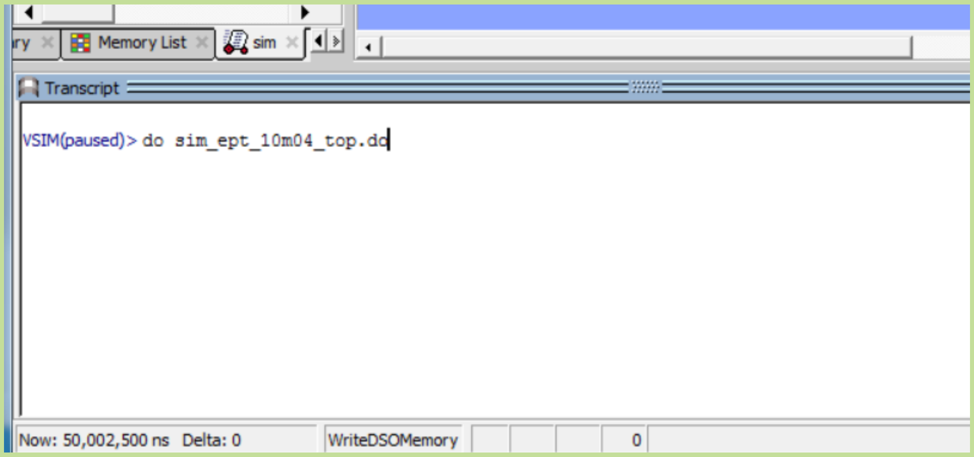
```
H:\Jolly\Code_FPGA\EPT_10M04_AF_Flip_Flop\EPT_10M04_AF_Flip_Flop\ModelSim\sim_ept_10m04_top.do - Notepad++
```

```
set ALTERA C:/Altera
set SRC EPT_10M04_AF_Flip_Flop/EPT_10M04_AF_Flip_Flop
vlog +define+SIM -reportprogress 300 -work work {../../EPT_10M04_AF_Flip_Flop/src/EPT_10M04_AF_S2_Top.v}
vlog +define+SIM -reportprogress 300 -work work {../../EPT_10M04_AF_Flip_Flop/Testbench/tb_ept_10m04_top.v}
vsim +maxdelays work.tb_ept_10m04_top
do wave_ept_10m04_top.do
run -all
```

```
//Active Transfer Test Registers
wire [7:0]                 received_tri
//DSO Communications registers
wire [7:0]                 adc_bus;
wire                       adc_encode;
wire [3:0]                 adc_channel;
wire                       pga_mosi;
wire                       pga_clk;
wire                       pga_miso;
wire                       pga_cs;
wire [3:0]                 selected_pga_
reg  [7:0]                 mcp4451_chip_
reg  [7:0]                 mcp4451_chip_

//DSO Models
wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445

wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445
wire [7:0]                 i2c_0_mcp445

wire                      i2c_0_data;
```

There are also simulate options that use the '+define' keyword. At the endof the file we see the 'do wave_ept_10m04_top.do' instruction. This will add the display flags to the 'Waveforms' window.

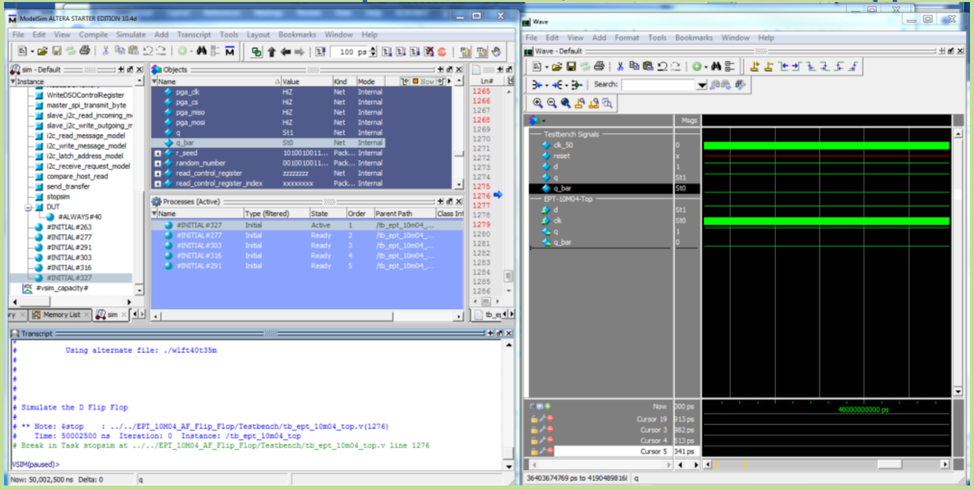
Start the simulation by typing do sim_ept_10m04_top.do into the command window.



```
VSIM(paused)> do sim_ept_10m04_top.do
```

```
Now: 50,002,500 ns   Delta: 0        WriteDSOMemory        0
```

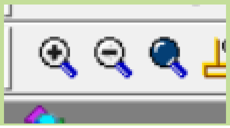
When the simulator completes its run, the Wave window appears.

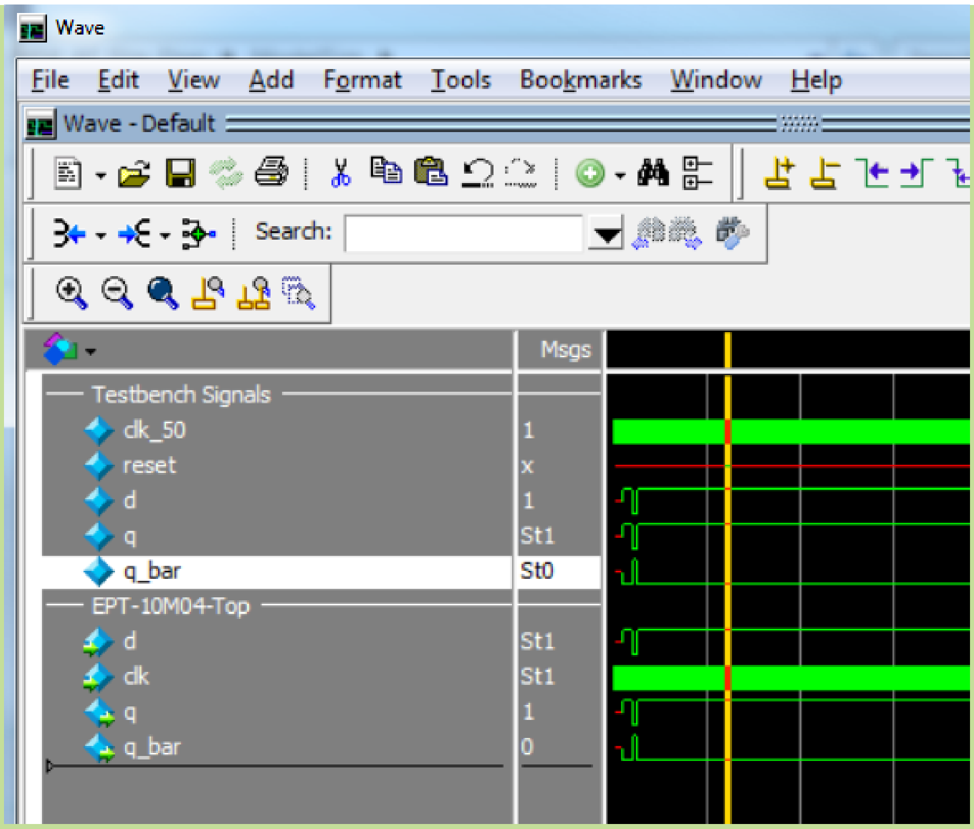


```
#          Using alternate file: ./wlft40t35m
#
#
#
#
#
# Simulate the D Flip Flop
#
# ** Note: $stop    : ../../EPT_10M04_AF_Flip_Flop/Testbench/tb_ept_10m04_top.v(1276)
#    Time: 50002500 ns  Iteration: 0  Instance: /tb_ept_10m04_top
# Break in Task stopsim at ../../EPT_10M04_AF_Flip_Flop/Testbench/tb_ept_10m04_top.v line 1276

VSIM(paused)>
```

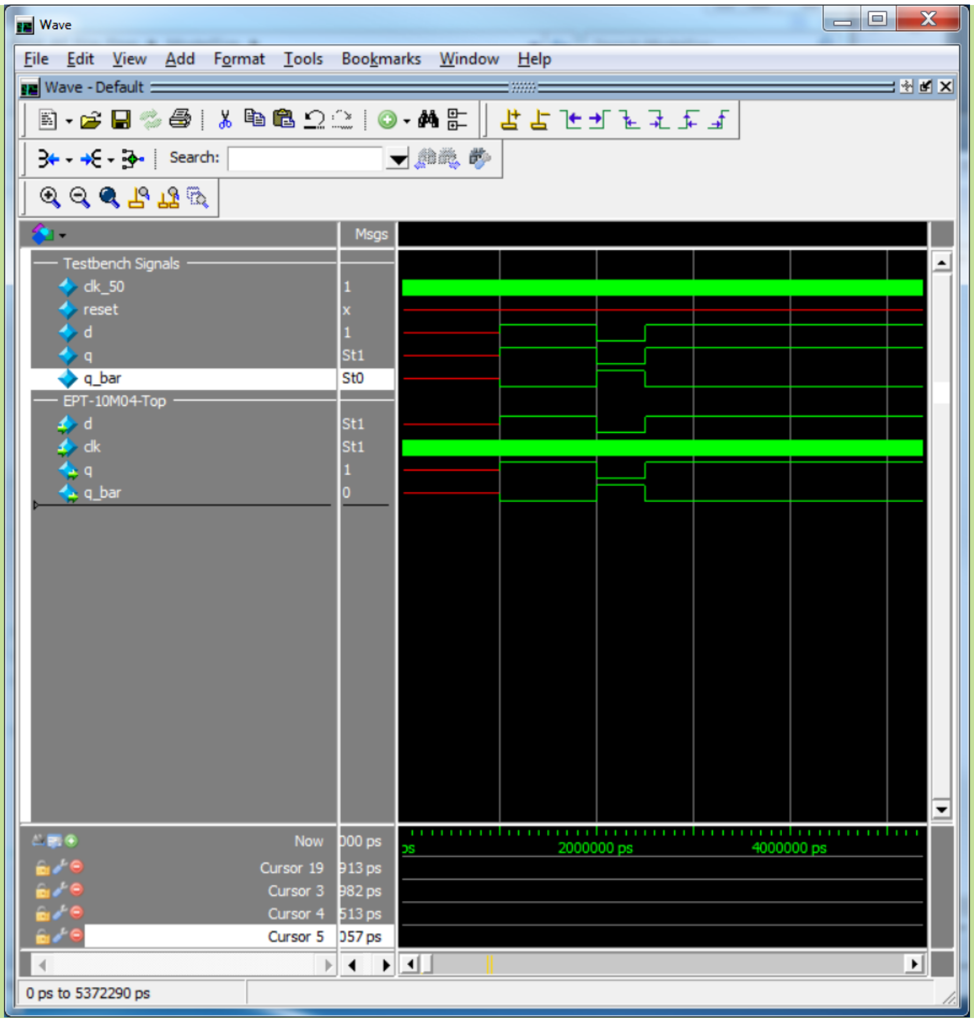
```
Now: 50,002,500 ns   Delta: 0
```

Zoom into the first 10 microseconds of the simulation usin the Zoom buttons.



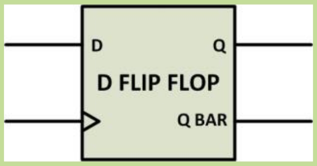


Now we see the 'd' stimulus toggling and the 'q' output following the input. Zoom in even farther.

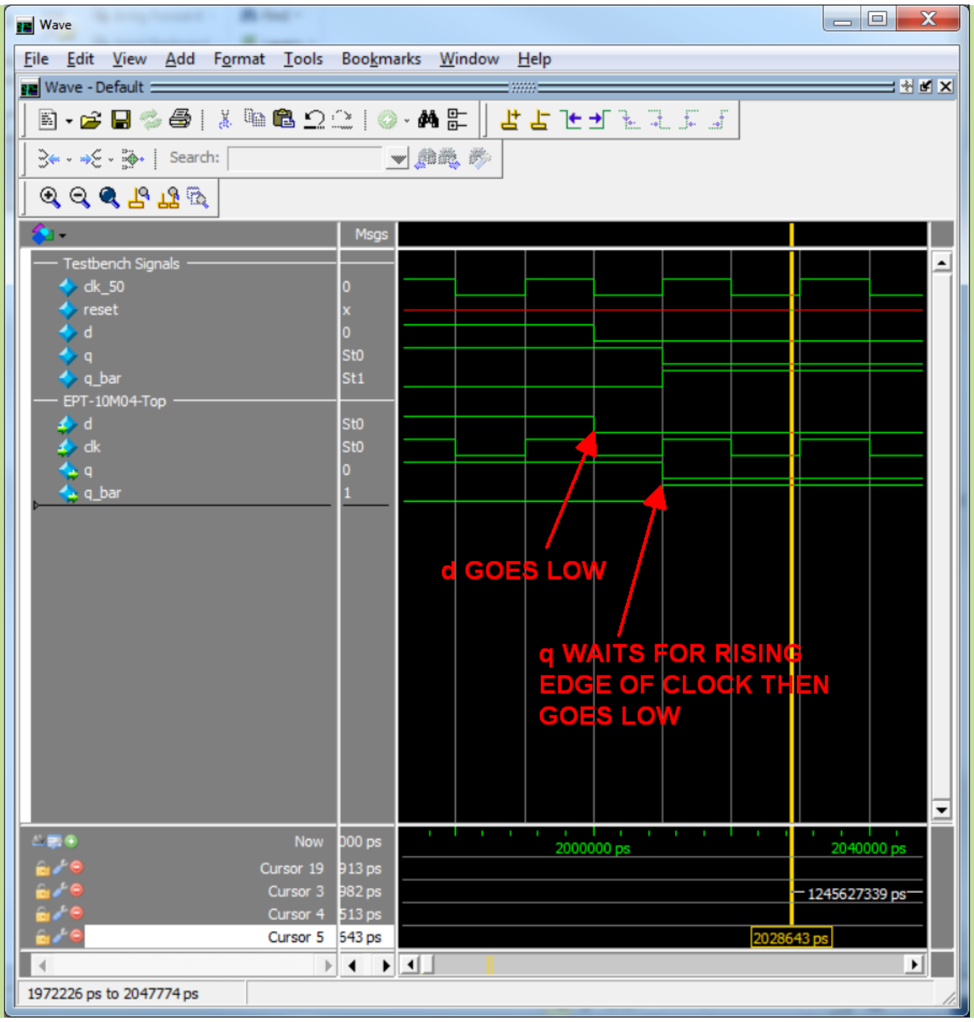

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
Testbench Signals
clk_50 1
reset x
d 1
q St1
q_bar St0
EPT-10M04-Top
d St1
clk St1
q 1
q_bar 0
Now 000 ps
Cursor 19 913 ps
Cursor 3 982 ps
Cursor 4 513 ps
Cursor 5 057 ps
2000000 ps
4000000 ps
0 ps to 5372290 ps

We can see after first delay that 'd' signal is asserted high. Before this assertion the signal is not defined in the simulation, so technically it is a Don't Care (indicated by the red 'floating' line). Once the 'd' is asserted, the delay is added by the simulator. After this delay the 'd' signal is de-asserted. Then, another dealy and the 'd' is asserted high.

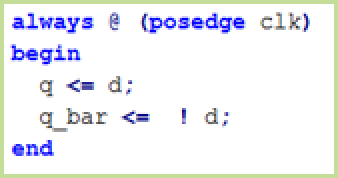
Now that we know what the simulator is doing, we can exam the user code, the D Flip Flop.



The D Flip Flop provides a registered output. This means that the input 'd' will be applied to the output 'q' but only after one rising edge of the clock. Because the output 'q' is synchronous to the clock, it is called a synchronous register. We can really see what this means when we zoom in even closer in the simulation.


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
Testbench Signals
clk_50 0
reset x
d 0
q St0
q_bar St1
EPT-10M04-Top
d St0
clk St0
q 0
q_bar 1
d GOES LOW
q WAITS FOR RISING EDGE OF CLOCK THEN GOES LOW
Now 000 ps
Cursor 19 913 ps
Cursor 3 982 ps
Cursor 4 513 ps
Cursor 5 543 ps
2000000 ps
2040000 ps
1245627339 ps
2028643 ps
1972226 ps to 2047774 ps

Examining the user code, we can see this synchronous behavior occuring because of the always statement.



```
always @ (posedge clk)
begin
  q <= d;
  q_bar <=  ! d;
end
```

The always keyword is used to cause a process to occur when an event happens. The event is what is in the paranthesis. In this case, the event is the rising edge of the 'clk' signal). Verilog uses the 'posedge' keyword to describe rising edge and 'clk' is our input clock from the testbench. So, what the Verilog code is telling us is that whenever we get a rising edge on the clock, the output 'q' is equal to 'd'. Also the 'q_bar' output is set to the complement of 'd'. This is exactly what the simluation is showing us.

We will cover the details of the Verilog keywords, description of synchronous code and combinatorial code later in this tutorial. This first lesson was designed to get you started with using ModelSim.

VERILOG ABSTRACTION LEVELS

Verilog Abstraction Levels

Verilog supports designing at many different levels of abstraction. Three of them are very important:

- Behavioral level
- Register-Transfer Level
- Gate Level

Behavioral level

Behavioral verilog deals with the logic or behavior of a system. It handles complex logic implementation and which is why in industry all implement the behavioral models of the system called as RTL.

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Register-Transfer Level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

RTL is an acronym for register transfer level. This implies that your **Verilog** code describes how data is transformed as it is passed from register to register. The transforming of the data is performed by the combinational logic that exists between the registers. Don't worry! RTL code also applies to pure combinational logic - you don't have to use registers. To show you what we mean by RTL code, let's consider a simple example.

```
module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
```

The AOI gate that we have used as an example so far has actually been written in RTL form. This means that continuous assignments are a valid way of describing designs for input to RTL synthesis tools. What other code techniques can we use? How about:

```
module MUX2 (input SEL, A, B, output F);
```

```
input SEL, A, B;
output F;

     INV G1 (SEL, SELB);
     AOI G2 (SELB, A, SEL, B, FB);
     INV G3 (.A(FB), .F(F));
endmodule
```

Gate Level

Modeling done at this level is usually called gate level modeling as it involves gates and has a one to one relation between a hardware schematic and the Verilog code. Verilog supports a few basic logic gates known as primitives as they can be instantiated like modules since they are already predefined.

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates).

```
module gates (
input a, b,
output c, d, e);

     and(c, a, b);  // c is the output, a and b are inputs
     or(d, a, b); // d is the output, a and b are inputs
     xor( e, a, b)  // e is the output, a and b are inputs
endmodule
```

MODULES, PORTS, DATA TYPES AND OPERATORS

Modules

Since Verilog is a HDL (hardware description language - one used for the conceptual design of integrated circuits), it also needs to have these things. In Verilog, we call our "black boxes" **module**. This is a reserved word within the program used to refer to things with inputs, outputs, and internal logic workings; they're the rough equivalents of functions with returns in other programming languages.

Module instances are also examples of synthesizable RTL statements. However, one of the reasons to use synthesis technology is to be able to describe the design at a higher level of abstraction than using a collection of module instances or low-level binary operators in a continuous assignment. We would like to be able to describe what the design does and leave the consideration of how the design is implemented up to the synthesis tool. This is a first step (and a pretty big conceptual one) on the road to high-level design. We are going to use a feature of the Verilog language that allows us to specify the functionality of a design (the ‘what') that can be interpreted by a synthesis tool.

In Verilog, after we have declared the module name and port names, we can define the direction of each port. The code for this is shown below.

```
module arbiter (
// Two slashes make a comment line.
clock       , // clock
reset       , // Active high, syn reset
req_0       , // Request 0
req_1       , // Request 1
gnt_0       , // Grant 0
gnt_1         // Grant 1
);
//-------------Input Ports-----------------------------
// Note : all commands are semicolon-delimited
input             clock               ;
input             reset               ;
input             req_0               ;
input             req_1               ;
//-------------Output Ports----------------------------
output            gnt_0               ;
output            gnt_1               ;
//-------------Internal Signals--------------------------
reg              count_1               ;
reg              stop_1                ;

//-------------Add User Code---------------------------
  …
endmodule
```

You could download file one_day1.v here

Types of Ports

Port	Description
Input	The design module can only receive values from outside using its `input` ports
Output	The design module can only send values to the outside using its `output` ports
Inout	The design module can either send or receive values using its `inout` ports

Ports are by default considered as nets of type `wire`.

Syntax

```
input  [net_type] [range] list_of_names; // Input port
```

```
inout  [net_type] [range] list_of_names; // Input & Output port
output [net_type] [range] list_of_names; // Output port driven by a
wire
```

```
output [var_type] [range] list_of_names;  // Output port driven by a
variable
```

<u>Example</u>

In the code shown below, there are three `input` ports, one `output` port and one `inout` port.

```
module my_design (

input wire      clk,
input           en,
input           rw,
inout [15:0]    data,
output          int_1 );

// Design behavior as Verilog code


endmodule
```

It is illegal to use the same *name* for multiple ports.

```
input  aport;         // First declaration - valid
input  aport;         // Error - already declared
output aport;         // Error - already declared
```

Bi-Directional Ports Example -

```
inout read_enable; // port named read_enable is bi-directional
```

How to define vector signals (signals composed of sequences of more than one bit)? Verilog provides a simple way to define these as well.

Vector Signals Example -

```
inout [7:0] address; //port "address" is bidirectional
```

Note the [7:0] means we're using the little-endian convention - you start with 0 at the rightmost bit to begin the vector, then move to the left. If we had done [0:7], we would be using the big-endian convention and moving from left to right. Endianness is a purely arbitrary way of deciding which way your data will "read," but does differ between systems, so using the right endianness consistently is important. As an analogy, think of some languages (English) that are written left-to-right (big-endian) versus others (Arabic) written right-to-left (little-endian). Knowing which way the language flows is crucial to being able to read it, but the direction of flow itself was arbitrarily set years back.

Summary

- We learnt how a block/module is defined in Verilog.
- We learnt how to define ports and port directions.
- We learnt how to declare vector/scalar ports.

Data Types

Verilog Language has two primary data types:

- **Nets** - represent structural connections between components.
- **Registers** - represent variables used to store data.

Every signal has a data type associated with it:

- **Explicitly declared** with a declaration in your Verilog code.
- **Implicitly declared** with no declaration when used to connect structural building blocks in your code. Implicit declaration is always a net type "wire" and is one bit with

Types of Nets

Each net type has a functionality that is used to model different types of hardware (such as PMOS, NMOS, CMOS, etc)

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate).

Summary

- Wire data type is used for connecting two points.
- Reg data type is used for storing values.

Register Data Types

- Registers store the last value assigned to them until another assignment statement changes their value.
- Registers represent data storage constructs.
- You can create regs arrays called memories.
- register data types are used as variables in procedural blocks.
- A register data type is required if a signal is assigned a value within a procedural block
- Procedural blocks begin with keyword initial and always.

Data Types	Functionality

reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Note : Of all register types, reg is the one which is most widely used

Operators

Operators are the same things here as they are in other programming languages. They take two values and compare (or otherwise operate on) them to yield a third result - common examples are addition, equals, logical-and... To make life easier for us, nearly all operators (at least the ones in the list below) are exactly the same as their counterparts in the C programming language.

Arithmetic Operators

- Binary: +, -, *, /, % (the modulus operator)
- Unary: +, - (This is used to specify the sign)
- Integer division truncates any fractional part
- The result of a modulus operation takes the sign of the first operand
- If any operand bit value is the unknown value x, then the entire result value is x
- Register data types are used as unsigned values (Negative numbers are stored in two's complement form)

Example

```
module arithmetic_operators();

initial begin
  $display (" 5  +  10 = %d", 5  + 10);
  $display (" 5  -  10 = %d", 5  - 10);
  $display (" 10 -  5  = %d", 10 - 5);
  $display (" 10 *  5  = %d", 10 * 5);
  $display (" 10 /  5  = %d", 10 / 5);
  $display (" 10 /  -5 = %d", 10 / -5);
  $display (" 10 %s  3  = %d","%", 10 % 3);
  $display (" +5       = %d", +5);
  $display (" -5       = %d", -5);
  #10 $finish;
end

endmodule
```

You could download file arithmetic_operators.v here

```
 5  +  10 =  15
 5  -  10 =  -5
 10 -  5  =   5
 10 *  5  =  50
 10 /  5  =  2
 10 /  -5 = -2
 10 %  3  =   1
 +5       =  5
 -5       =  -5
```

Relational Operators

Operator	Description
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

- The result is a scalar value (example a < b)
- 0 if the relation is false (a is bigger then b)
- 1 if the relation is true (a is smaller then b)
- x if any of the operands has unknown x bits (if a or b contains X)

Note: If any operand is x or z, then the result of that test is treated as false (0)

Example

```
module relational_operators();

initial begin
  $display (" 5   <= 10 = %b", (5     <= 10));
  $display (" 5   >= 10 = %b", (5     >= 10));
  $display (" 1'bx <= 10 = %b", (1'bx  <= 10));
  $display (" 1'bz <= 10 = %b", (1'bz  <= 10));
  #10 $finish;
end

endmodule
```

You could download file relational_operators.v here

```
 5    <=  10 = 1
 5    >=  10 = 0
1'bx  <=  10 = x
1'bz  <=  10 = x
```

Equality Operators

There are two types of Equality operators. Case Equality and Logical Equality.

Operator	Description
a === b	a equal to b, including x and z (Case equality)
a !== b	a not equal to b, including x and z (Case inequality)
a == b	a equal to b, result may be unknown (logical equality)

a != b a not equal to b, result may be unknown (logical equality)

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- Result is 0 (false) or 1 (true)
- For the == and != operators, the result is x, if either operand contains an x or a z
- For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true

Note : The result is always 0 or 1.

Example

```
module equality_operators();

initial begin
   // Case Equality
   $display (" 4'bx001 === 4'bx001 = %b", (4'bx001 === 4'bx001));
   $display (" 4'bx0x1 === 4'bx001 = %b", (4'bx0x1 === 4'bx001));
   $display (" 4'bz0x1 === 4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
   $display (" 4'bz0x1 === 4'bz001 = %b", (4'bz0x1 === 4'bz001));
   // Case Inequality
   $display (" 4'bx0x1 !== 4'bx001 = %b", (4'bx0x1 !== 4'bx001));
   $display (" 4'bz0x1 !== 4'bz001 = %b", (4'bz0x1 !== 4'bz001));
   // Logical Equality
   $display (" 5    == 10    = %b", (5         ==   10));
   $display (" 5    == 5     = %b", (5         ==   5));
   // Logical Inequality
   $display (" 5    != 5     = %b", (5         !=   5));
   $display (" 5    != 6     = %b", (5         !=   6));
   #10 $finish;
end
```

```
endmodule
```

You could download file equality_operators.v here

```
4'bx001 ===  4'bx001 = 1
 4'bx0x1 ===  4'bx001 = 0
 4'bz0x1 ===  4'bz0x1 = 1
 4'bz0x1 ===  4'bz001 = 0
 4'bx0x1 !==  4'bx001 = 1
 4'bz0x1 !==  4'bz001 = 1
 5       ==   10      = 0
 5       ==   5       = 1
 5       !=   5       = 0
 5       !=   6       = 1
```

Logical Operators

Operator	Description
!	logic negation
&&	logical and
||	logical or

- Expressions connected by && and || are evaluated from left to right
- Evaluation stops as soon as the result is known
- The result is a scalar value:
    - 0 if the relation is false
    - 1 if the relation is true
    - x if any of the operands has x (unknown) bits

Example

```
module logical_operators();

initial begin
   // Logical AND
   $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
   $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
   $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
   // Logical OR
   $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
   $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
   $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
   // Logical Negation
   $display ("! 1'b1      = %b", (!   1'b1));
   $display ("! 1'b0      = %b", (!   1'b0));
   #10  $finish;
end

endmodule
```

You could download file logical_operators.v here

```
1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1
1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
! 1'b1      = 0
! 1'b0      = 1
```

Bit-wise Operators

Bitwise operators perform a bit wise operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be extended on the left side with zeroes to match the length of the longer operand.

Operator	Description
~	negation
&	and
|	inclusive or
^	exclusive or
^~ or ~^	exclusive nor (equivalence)

- Computations include unknown bits, in the following way:
    - ~x = x
    - 0&x = 0
    - 1&x = x&x = x
    - 1|x = 1
    - 0|x = x|x = x
    - 0^x = 1^x = x^x = x
    - 0^~x = 1^~x = x^~x = x
- When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

Example

```
module bitwise_operators();

initial begin
  // Bit Wise Negation
   $display (" ~4'b0001        = %b", (~4'b0001));
   $display (" ~4'bx001        = %b", (~4'bx001));
   $display (" ~4'bz001        = %b", (~4'bz001));
   // Bit Wise AND
   $display (" 4'b0001 &  4'b1001 = %b", (4'b0001 &  4'b1001));
   $display (" 4'b1001 &  4'bx001 = %b", (4'b1001 &  4'bx001));
   $display (" 4'b1001 &  4'bz001 = %b", (4'b1001 &  4'bz001));
   // Bit Wise OR
   $display (" 4'b0001 |  4'b1001 = %b", (4'b0001 |  4'b1001));
   $display (" 4'b0001 |  4'bx001 = %b", (4'b0001 |  4'bx001));
   $display (" 4'b0001 |  4'bz001 = %b", (4'b0001 |  4'bz001));
   // Bit Wise XOR
   $display (" 4'b0001 ^  4'b1001 = %b", (4'b0001 ^  4'b1001));
   $display (" 4'b0001 ^  4'bx001 = %b", (4'b0001 ^  4'bx001));
   $display (" 4'b0001 ^  4'bz001 = %b", (4'b0001 ^  4'bz001));
   // Bit Wise XNOR
   $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
   $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
   $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
   #10 $finish;
end

endmodule
```

You could download file bitwise_operators.v here

```
 ~4'b0001        = 1110
 ~4'bx001        = x110
```

```
~4'bz001            = x110
4'b0001 &  4'b1001 = 0001
4'b1001 &  4'bx001 = x001
4'b1001 &  4'bz001 = x001
4'b0001 |  4'b1001 = 1001
4'b0001 |  4'bx001 = x001
4'b0001 |  4'bz001 = x001
4'b0001 ^  4'b1001 = 1000
4'b0001 ^  4'bx001 = x000
4'b0001 ^  4'bz001 = x000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111
```

Reduction Operators

Operator	**Description**
&	and
~&	nand
|	or
~|	nor
^	xor
^~ or ~^	xnor

- Reduction operators are unary.

- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
    - Unknown bits are treated as described before.

Example

```
module reduction_operators();

initial begin
  // Bit Wise AND reduction
  $display (" & 4'b1001 = %b", (&  4'b1001));
  $display (" & 4'bx111 = %b", (&  4'bx111));
  $display (" & 4'bz111 = %b", (&  4'bz111));
  // Bit Wise NAND reduction
  $display (" ~& 4'b1001 = %b", (~& 4'b1001));
  $display (" ~& 4'bx001 = %b", (~& 4'bx001));
  $display (" ~& 4'bz001 = %b", (~& 4'bz001));
  // Bit Wise OR reduction
  $display (" | 4'b1001 = %b", (|  4'b1001));
  $display (" | 4'bx000 = %b", (|  4'bx000));
  $display (" | 4'bz000 = %b", (|  4'bz000));
  // Bit Wise NOR reduction
  $display (" ~| 4'b1001 = %b", (~| 4'b1001));
  $display (" ~| 4'bx001 = %b", (~| 4'bx001));
  $display (" ~| 4'bz001 = %b", (~| 4'bz001));
  // Bit Wise XOR reduction
  $display (" ^ 4'b1001 = %b", (^  4'b1001));
  $display (" ^ 4'bx001 = %b", (^  4'bx001));
  $display (" ^ 4'bz001 = %b", (^  4'bz001));
  // Bit Wise XNOR
  $display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
  $display (" ~^ 4'bx001 = %b", (~^ 4'bx001));
  $display (" ~^ 4'bz001 = %b", (~^ 4'bz001));
  #10 $finish;
```

```
end

endmodule
```

You could download file reduction_operators.v here

```
 &  4'b1001 = 0
&  4'bx111 = x
&  4'bz111 = x
~& 4'b1001 = 1
~& 4'bx001 = 1
~& 4'bz001 = 1
|  4'b1001 = 1
|  4'bx000 = x
|  4'bz000 = x
~| 4'b1001 = 0
~| 4'bx001 = 0
~| 4'bz001 = 0
^  4'b1001 = 0
^  4'bx001 = x
^  4'bz001 = x
~^ 4'b1001 = 1
~^ 4'bx001 = x
~^ 4'bz001 = x
```

Shift Operators

Operator	Description
<<	left shift
>>	right shift

- The left operand is shifted by the number of bit positions given by the right operand.
- The vacated bit positions are filled with zeroes.

Example

```
module shift_operators();

initial begin
  // Left Shift
  $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
  $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
  $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
  // Right Shift
  $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
  $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
  $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
  #10 $finish;
end

endmodule
```

You could download file shift_operators.v here

```
4'b1001 << 1 = 0010
 4'b10x1 << 1 = 0x10
 4'b10z1 << 1 = 0z10
 4'b1001 >> 1 = 0100
 4'b10x1 >> 1 = 010x
 4'b10z1 >> 1 = 010z
```

Concatenation Operator

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
    - Example: + {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits
- Unsized constant numbers are not allowed in concatenations.

Example

```
module concatenation_operator();

initial begin
  // concatenation
  $display (" {4'b1001,4'b10x1}  = %b", {4'b1001,4'b10x1});
  #10 $finish;
end

endmodule
```

You could download file concatenation_operator.v here

```
{4'b1001,4'b10x1}  = 100110x1
```

Replication Operator

Replication operator is used to replicate a group of bits n times. Say you have a 4 bit variable and you want to replicate it 4 times to get a 16 bit variable: then we can use the replication operator.

Operator	**Description**
{n{m}}	Replicate value m, n times

- Repetition multipliers (must be constants) can be used:
  - {3{a}} // this is equivalent to {a, a, a}
- Nested concatenations and replication operator are possible:
  - {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

Example

```
module replication_operator();

initial begin
  // replication
  $display (" {4{4'b1001}}     = %b", {4{4'b1001}});
  // replication and concatenation
  $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
  #10 $finish;
end

endmodule
```

You could download file replication_operator.v here

```
{4{4'b1001}}      = 1001100110011001
```

{4{4'b1001,1'bz} = 1001z1001z1001z1001z

Conditional Operators

- The conditional operator has the following C-like format:
  - cond_expr ? true_expr : false_expr
- The true_expr or the false_expr is evaluated and used as a result depending on what cond_expr evaluates to (true or false).

Example

```
module conditional_operator();

wire out;
reg enable,data;
// Tri state buffer
assign out = (enable) ? data : 1'bz;

initial begin
  $display ("time\t enable data out");
  $monitor ("%g\t %b    %b   %b",$time,enable,data,out);
  enable = 0;
  data = 0;
  #1 data = 1;
  #1 data = 0;
  #1 enable = 1;
  #1 data = 1;
  #1 data = 0;
  #1 enable = 0;
  #10 $finish;
end
```

```
endmodule
```

You could download file conditional_operator.v here

```
time       enable data out
0          0    0   z
1          0    1   z
2          0    0   z
3          1    0   0
4          1    1   1
5          1    0   0
6          0    0   z
```

Operator Precedence

Operator	Symbols
Unary, Multiply, Divide, Modulus	!, ~, *, /, %
Add, Subtract, Shift	+, - , <<, >>
Relation, Equality	<,>,<=,>=,==,!=,===,!==
Reduction	&, !&,^,^~,|,~|
Logic	&&, ||
Conditional	? :

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Reduction	~	Bitwise negation
	~&	nand

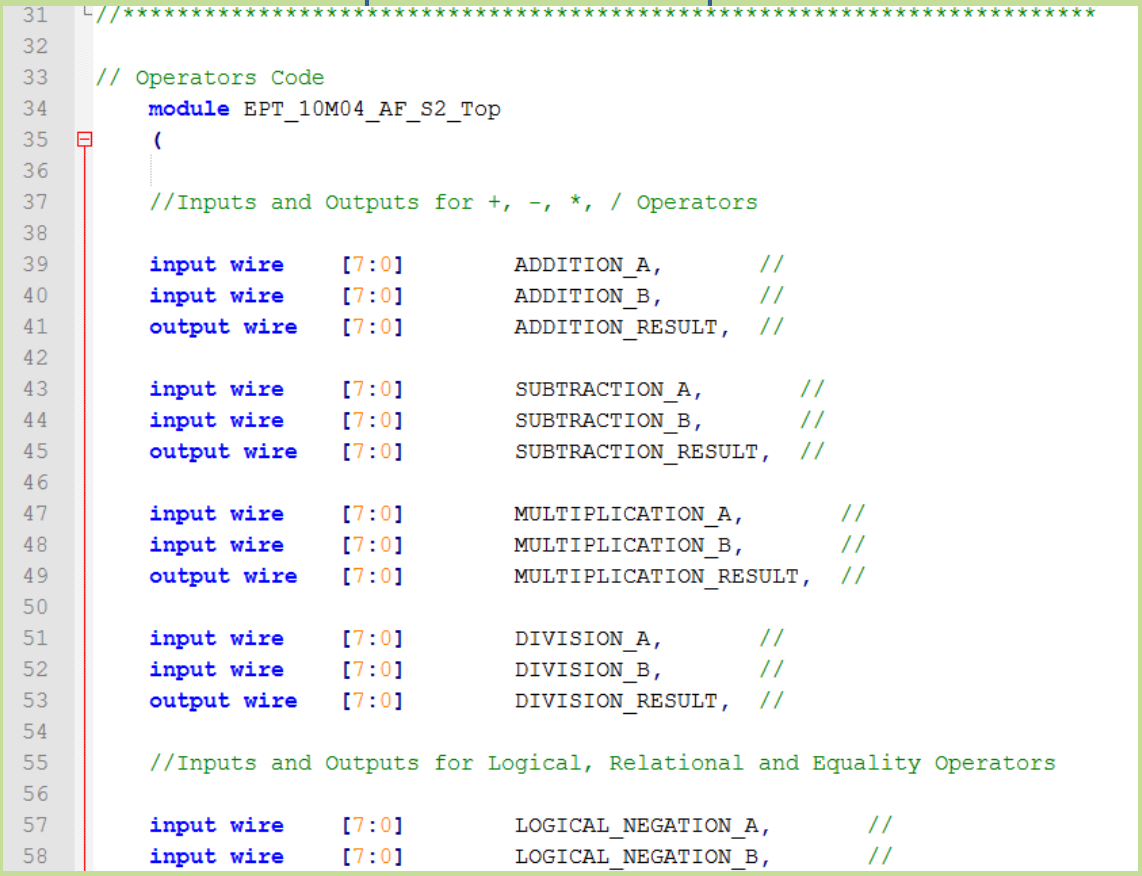
	|	or
	~|	nor
	^	xor
	^~	xnor
	~^	xnor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

///////////////////////////////////////Lesson #2/////////////////////////////////////

In this lesson, we will use the operators in Verilog code.
First, let's open the Lesson 2 and explore the operators code. Go to the xx_Project_xx_DVD-> Verilog Getting Started->Tutorials_HDL. Locate the Lesson_2_Operators folder in the DVD.

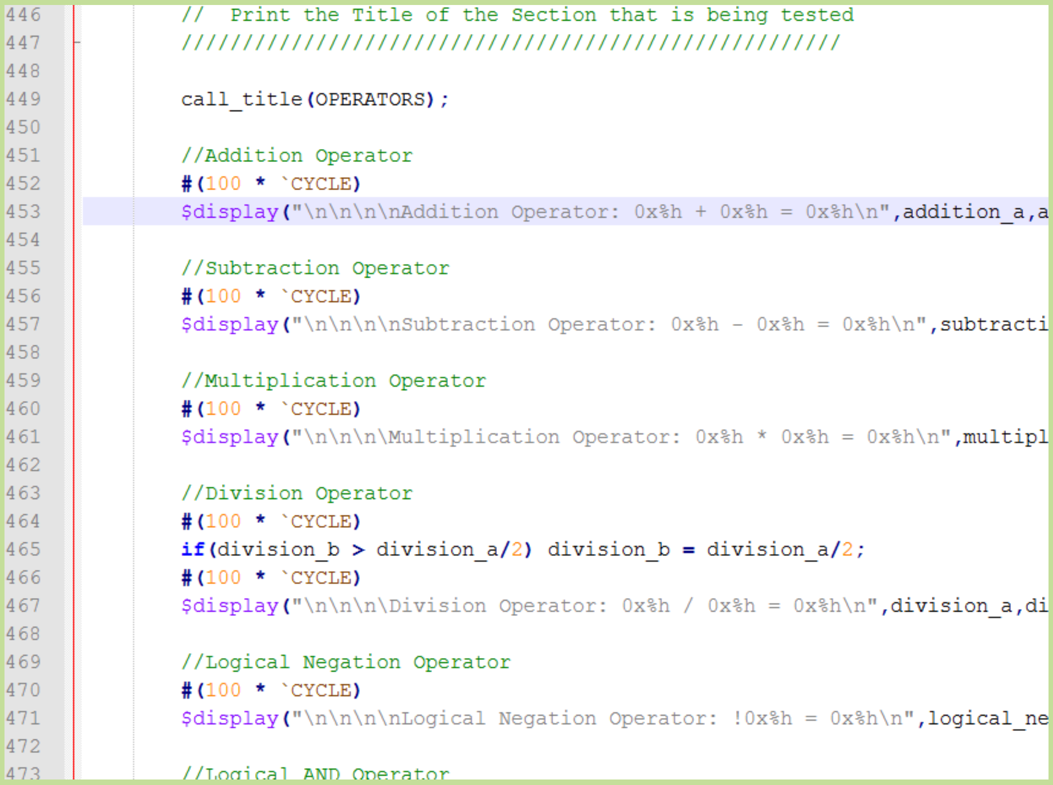
Go through the initial steps as outlined in Lesson 1. Those initial steps open ModelSim, Change Directory to the ModelSim Folder, Compile the source file, then start the simulation using the "do sim_ept_10m04_top.do"

Open the Top Level file in the "src' folder. Examine the standalone module that will perform the selected operation.



```
//**************************************************************************

// Operators Code
    module EPT_10M04_AF_S2_Top
    (

    //Inputs and Outputs for +, -, *, / Operators

    input wire     [7:0]        ADDITION_A,       //
    input wire     [7:0]        ADDITION_B,       //
    output wire    [7:0]        ADDITION_RESULT,  //

    input wire     [7:0]        SUBTRACTION_A,       //
    input wire     [7:0]        SUBTRACTION_B,       //
    output wire    [7:0]        SUBTRACTION_RESULT,  //

    input wire     [7:0]        MULTIPLICATION_A,       //
    input wire     [7:0]        MULTIPLICATION_B,       //
    output wire    [7:0]        MULTIPLICATION_RESULT,  //

    input wire     [7:0]        DIVISION_A,       //
    input wire     [7:0]        DIVISION_B,       //
    output wire    [7:0]        DIVISION_RESULT,  //

    //Inputs and Outputs for Logical, Relational and Equality Operators

    input wire     [7:0]        LOGICAL_NEGATION_A,       //
    input wire     [7:0]        LOGICAL_NEGATION_B,       //
```

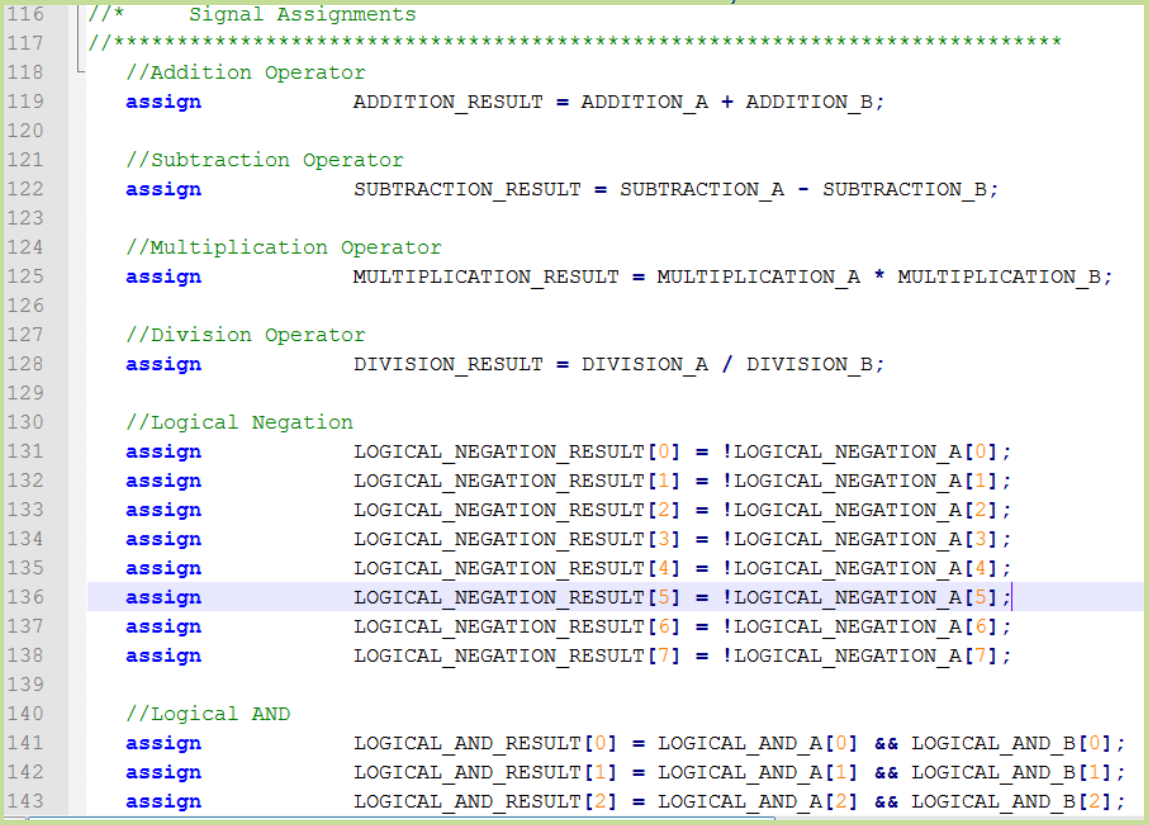
Next, open and view the test bench to exercise each operator and display the results.



```
//  Print the Title of the Section that is being tested
////////////////////////////////////////////////////////

call_title(OPERATORS);

//Addition Operator
#(100 * `CYCLE)
$display("\n\n\n\nAddition Operator: 0x%h + 0x%h = 0x%h\n",addition_a,a

//Subtraction Operator
#(100 * `CYCLE)
$display("\n\n\n\nSubtraction Operator: 0x%h - 0x%h = 0x%h\n",subtracti

//Multiplication Operator
#(100 * `CYCLE)
$display("\n\n\n\Multiplication Operator: 0x%h * 0x%h = 0x%h\n",multipl

//Division Operator
#(100 * `CYCLE)
if(division_b > division_a/2) division_b = division_a/2;
#(100 * `CYCLE)
$display("\n\n\n\Division Operator: 0x%h / 0x%h = 0x%h\n",division_a,di

//Logical Negation Operator
#(100 * `CYCLE)
$display("\n\n\n\nLogical Negation Operator: !0x%h = 0x%h\n",logical_ne

//Logical AND Operator
```

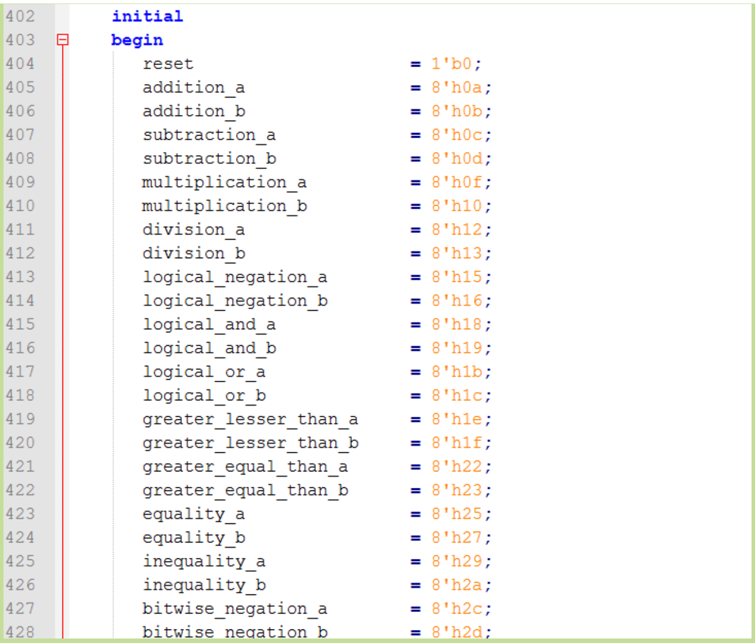
The top level module is created following the syntax rules of Verilog. Use the keyword "module" followed by the name of the module. Then, add an opening parenthesis and define the inputs and outputs. This tutorial will cover in greater detail the keywords "module", "input", and "output", signals, registers and assignment statements later on. For now, just follow the coding. This module simply takes in two operands, performs an operation then outputs the results. This lesson is designed to get you familiar with how operators work and how to use them. This lesson specifically takes in two operands of eight bits each and the output is eight bits. The reason for this is in the future you will write code that manipulates both multi-bit registers and single bit signals. The operators will act differently on registers and signals. So, you will need to understand this and how to use operators.

Lets look at the code,

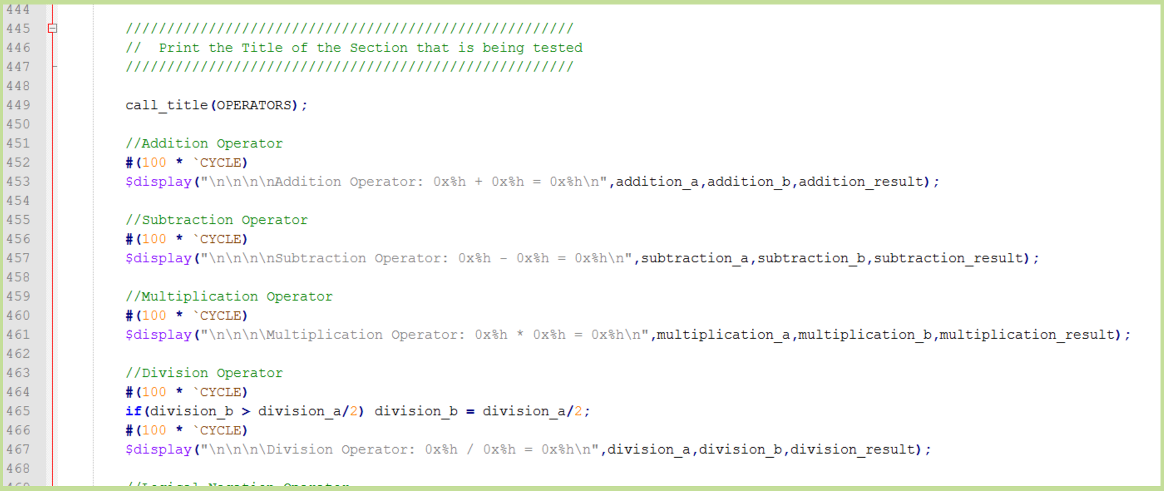


```
//*     Signal Assignments
//**************************************************************************
   //Addition Operator
   assign             ADDITION_RESULT = ADDITION_A + ADDITION_B;

   //Subtraction Operator
   assign             SUBTRACTION_RESULT = SUBTRACTION_A - SUBTRACTION_B;

   //Multiplication Operator
   assign             MULTIPLICATION_RESULT = MULTIPLICATION_A * MULTIPLICATION_B;

   //Division Operator
   assign             DIVISION_RESULT = DIVISION_A / DIVISION_B;

   //Logical Negation
   assign             LOGICAL_NEGATION_RESULT[0] = !LOGICAL_NEGATION_A[0];
   assign             LOGICAL_NEGATION_RESULT[1] = !LOGICAL_NEGATION_A[1];
   assign             LOGICAL_NEGATION_RESULT[2] = !LOGICAL_NEGATION_A[2];
   assign             LOGICAL_NEGATION_RESULT[3] = !LOGICAL_NEGATION_A[3];
   assign             LOGICAL_NEGATION_RESULT[4] = !LOGICAL_NEGATION_A[4];
   assign             LOGICAL_NEGATION_RESULT[5] = !LOGICAL_NEGATION_A[5];
   assign             LOGICAL_NEGATION_RESULT[6] = !LOGICAL_NEGATION_A[6];
   assign             LOGICAL_NEGATION_RESULT[7] = !LOGICAL_NEGATION_A[7];

   //Logical AND
   assign             LOGICAL_AND_RESULT[0] = LOGICAL_AND_A[0] && LOGICAL_AND_B[0];
   assign             LOGICAL_AND_RESULT[1] = LOGICAL_AND_A[1] && LOGICAL_AND_B[1];
   assign             LOGICAL_AND_RESULT[2] = LOGICAL_AND_A[2] && LOGICAL_AND_B[2];
```

The first four operators "+, -, *, /" work on the entire eight registers in total. If we look at the test bench, we will see that we can access the inputs by placing an eight value on the inputs at the start of the simulation. Then, use the "$display" keyword to display the outputs on the log window.



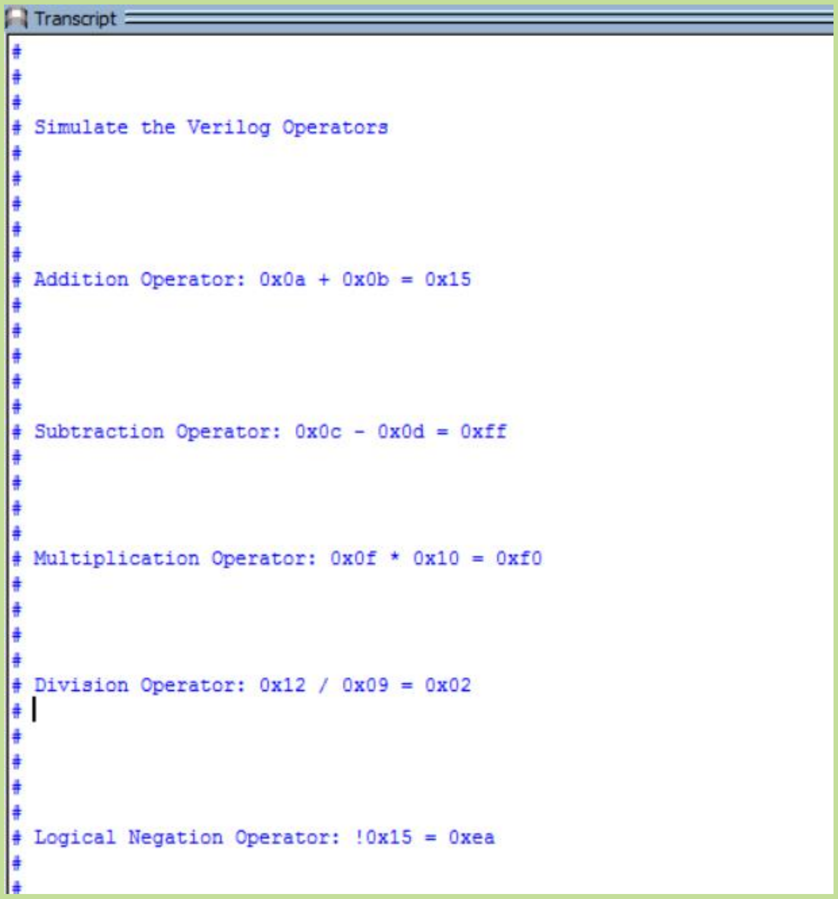
```
initial
begin
    reset                       = 1'b0;
    addition_a                  = 8'h0a;
    addition_b                  = 8'h0b;
    subtraction_a               = 8'h0c;
    subtraction_b               = 8'h0d;
    multiplication_a            = 8'h0f;
    multiplication_b            = 8'h10;
    division_a                  = 8'h12;
    division_b                  = 8'h13;
    logical_negation_a          = 8'h15;
    logical_negation_b          = 8'h16;
    logical_and_a               = 8'h18;
    logical_and_b               = 8'h19;
    logical_or_a                = 8'h1b;
    logical_or_b                = 8'h1c;
    greater_lesser_than_a       = 8'h1e;
    greater_lesser_than_b       = 8'h1f;
    greater_equal_than_a        = 8'h22;
    greater_equal_than_b        = 8'h23;
    equality_a                  = 8'h25;
    equality_b                  = 8'h27;
    inequality_a                = 8'h29;
    inequality_b                = 8'h2a;
    bitwise_negation_a          = 8'h2c;
    bitwise_negation_b          = 8'h2d;
```



```
//////////////////////////////////////////////////////
//  Print the Title of the Section that is being tested
//////////////////////////////////////////////////////

call_title(OPERATORS);

//Addition Operator
#(100 * `CYCLE)
$display("\n\n\n\nAddition Operator: 0x%h + 0x%h = 0x%h\n",addition_a,addition_b,addition_result);

//Subtraction Operator
#(100 * `CYCLE)
$display("\n\n\n\nSubtraction Operator: 0x%h - 0x%h = 0x%h\n",subtraction_a,subtraction_b,subtraction_result);

//Multiplication Operator
#(100 * `CYCLE)
$display("\n\n\n\Multiplication Operator: 0x%h * 0x%h = 0x%h\n",multiplication_a,multiplication_b,multiplication_result);

//Division Operator
#(100 * `CYCLE)
if(division_b > division_a/2) division_b = division_a/2;
#(100 * `CYCLE)
$display("\n\n\n\Division Operator: 0x%h / 0x%h = 0x%h\n",division_a,division_b,division_result);
```

You can see the testbench "$display" manipulates the operands by simply using the "%h" syntax. This similar to the C language. When you want to display a number to the log screen add the "%h" inside a string. Then close the string and place the name of the register inside the paranthesis. Run the

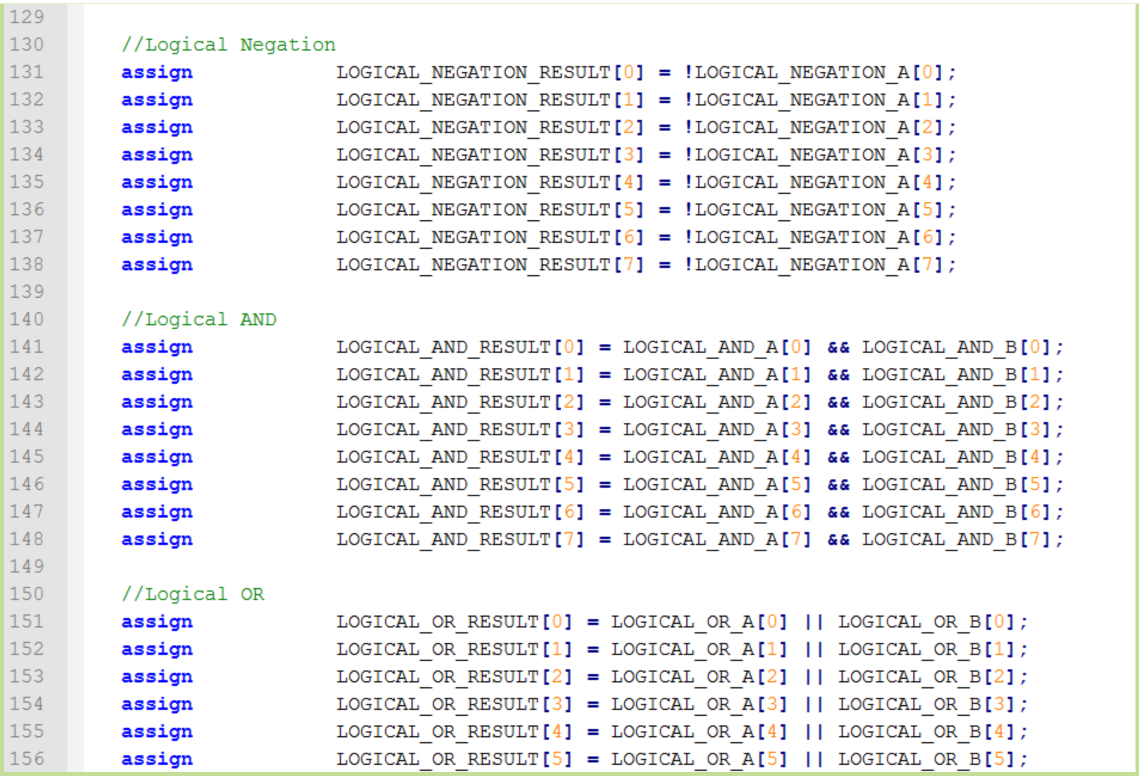
ModelSim simulation using the *.do file by following the steps in Lesson #1. The output on the log screen looks like the following:



```
#
#
#
# Simulate the Verilog Operators
#
#
#
#
#
# Addition Operator: 0x0a + 0x0b = 0x15
#
#
#
#
#
# Subtraction Operator: 0x0c - 0x0d = 0xff
#
#
#
#
# Multiplication Operator: 0x0f * 0x10 = 0xf0
#
#
#
#
# Division Operator: 0x12 / 0x09 = 0x02
#
#
#
#
#
# Logical Negation Operator: !0x15 = 0xea
#
#
```

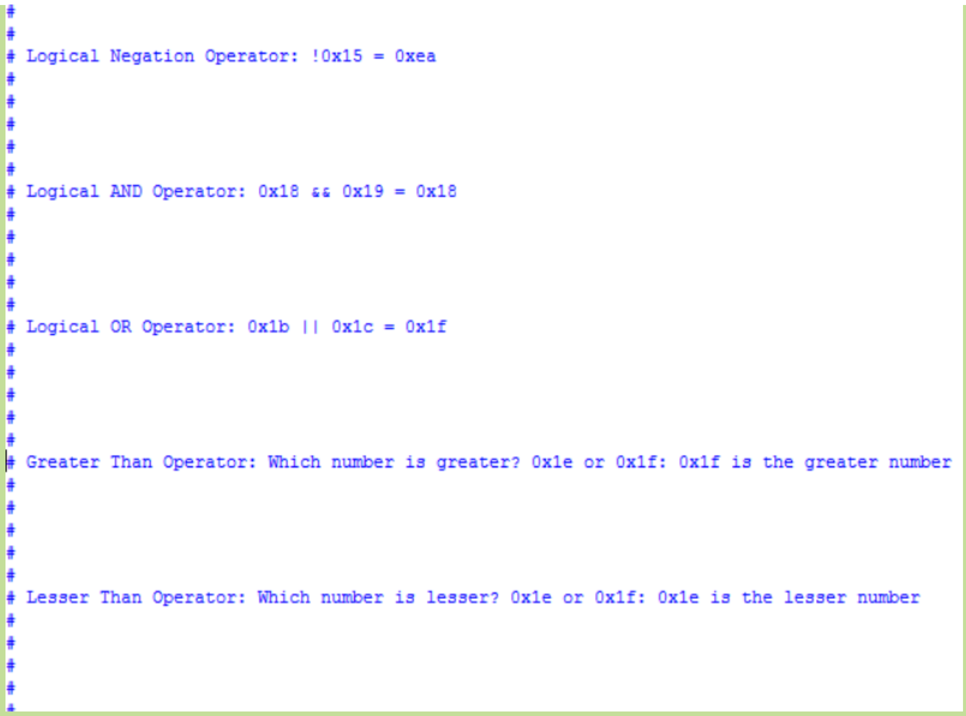
You can see the result of the operation in the log window after running the simulation. Note that the first four operators work on the entire eight bits of the input registers.

The "Logical Operators" only operate on single bit signals. When we use an eight bit register, the operator has to be assigned to each individual bit in separate assignments.



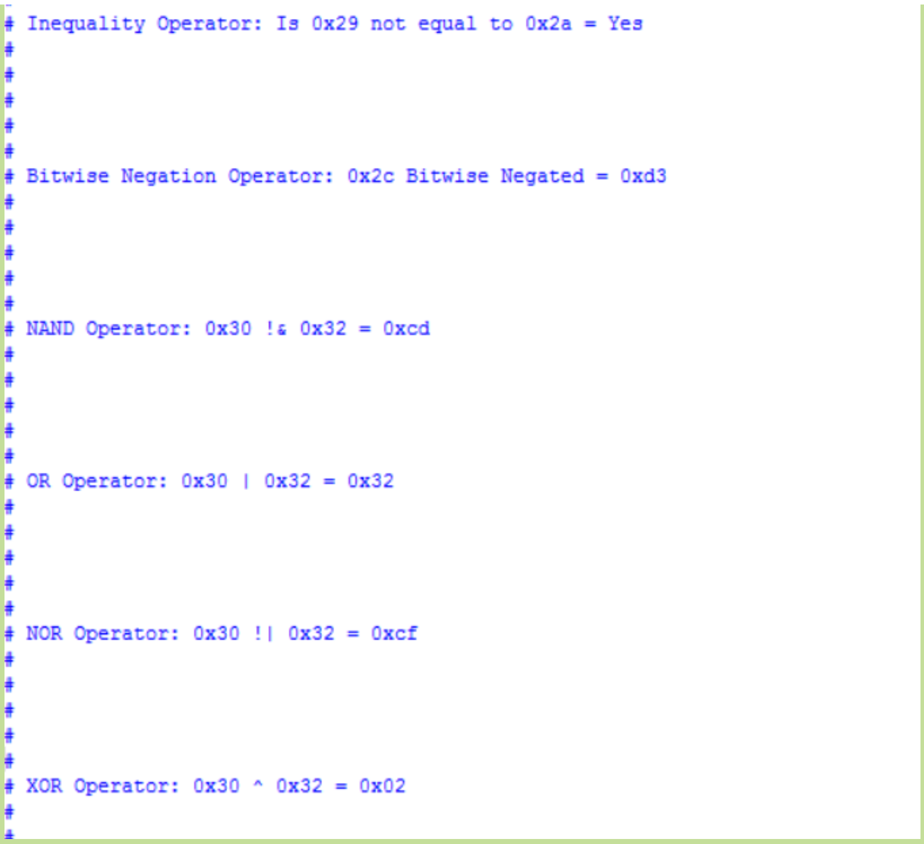
```

//Logical Negation
assign          LOGICAL_NEGATION_RESULT[0] = !LOGICAL_NEGATION_A[0];
assign          LOGICAL_NEGATION_RESULT[1] = !LOGICAL_NEGATION_A[1];
assign          LOGICAL_NEGATION_RESULT[2] = !LOGICAL_NEGATION_A[2];
assign          LOGICAL_NEGATION_RESULT[3] = !LOGICAL_NEGATION_A[3];
assign          LOGICAL_NEGATION_RESULT[4] = !LOGICAL_NEGATION_A[4];
assign          LOGICAL_NEGATION_RESULT[5] = !LOGICAL_NEGATION_A[5];
assign          LOGICAL_NEGATION_RESULT[6] = !LOGICAL_NEGATION_A[6];
assign          LOGICAL_NEGATION_RESULT[7] = !LOGICAL_NEGATION_A[7];

//Logical AND
assign          LOGICAL_AND_RESULT[0] = LOGICAL_AND_A[0] && LOGICAL_AND_B[0];
assign          LOGICAL_AND_RESULT[1] = LOGICAL_AND_A[1] && LOGICAL_AND_B[1];
assign          LOGICAL_AND_RESULT[2] = LOGICAL_AND_A[2] && LOGICAL_AND_B[2];
assign          LOGICAL_AND_RESULT[3] = LOGICAL_AND_A[3] && LOGICAL_AND_B[3];
assign          LOGICAL_AND_RESULT[4] = LOGICAL_AND_A[4] && LOGICAL_AND_B[4];
assign          LOGICAL_AND_RESULT[5] = LOGICAL_AND_A[5] && LOGICAL_AND_B[5];
assign          LOGICAL_AND_RESULT[6] = LOGICAL_AND_A[6] && LOGICAL_AND_B[6];
assign          LOGICAL_AND_RESULT[7] = LOGICAL_AND_A[7] && LOGICAL_AND_B[7];

//Logical OR
assign          LOGICAL_OR_RESULT[0] = LOGICAL_OR_A[0] || LOGICAL_OR_B[0];
assign          LOGICAL_OR_RESULT[1] = LOGICAL_OR_A[1] || LOGICAL_OR_B[1];
assign          LOGICAL_OR_RESULT[2] = LOGICAL_OR_A[2] || LOGICAL_OR_B[2];
assign          LOGICAL_OR_RESULT[3] = LOGICAL_OR_A[3] || LOGICAL_OR_B[3];
assign          LOGICAL_OR_RESULT[4] = LOGICAL_OR_A[4] || LOGICAL_OR_B[4];
assign          LOGICAL_OR_RESULT[5] = LOGICAL_OR_A[5] || LOGICAL_OR_B[5];
```

The result of the logical operator on an eight bit register show the number as a complete eight bit number:



```
#
#
# Logical Negation Operator: !0x15 = 0xea
#
#
#
#
#
# Logical AND Operator: 0x18 && 0x19 = 0x18
#
#
#
#
#
# Logical OR Operator: 0x1b || 0x1c = 0x1f
#
#
#
#
#
# Greater Than Operator: Which number is greater? 0x1e or 0x1f: 0x1f is the greater number
#
#
#
#
#
# Lesser Than Operator: Which number is lesser? 0x1e or 0x1f: 0x1e is the lesser number
#
#
#
#
#
```

The rest of the operators operate on the entire eight registers. This is an important distinction that you must handle appropriately when writing your code. The results of the other operators:



```
# Inequality Operator: Is 0x29 not equal to 0x2a = Yes
#
#
#
#
#
# Bitwise Negation Operator: 0x2c Bitwise Negated = 0xd3
#
#
#
#
#
# NAND Operator: 0x30 !& 0x32 = 0xcd
#
#
#
#
#
# OR Operator: 0x30 | 0x32 = 0x32
#
#
#
#
#
# NOR Operator: 0x30 !| 0x32 = 0xcf
#
#
#
#
#
# XOR Operator: 0x30 ^ 0x32 = 0x02
#
#
```

Feel free to explore the operators by changing the initial values of these numbers and seeing the results after running the simulation.

CONTROL STATEMENTS

Control statements in Verilog include: **if, else, repeat, while, for, case** - it's Verilog that looks exactly like C. Even though the functionality appears to be the same as in C, Verilog is an HDL, so the descriptions should translate to hardware. This means you've got to be careful when using control statements (otherwise your designs might not be implementable in hardware).

If-else

If-else statements check a condition to decide whether or not to execute a portion of code. If a condition is satisfied, the code is executed. Else, it runs this other portion of code.

```
// begin and end act like curly braces in C/C++.
if (enable == 1'b1) begin
  data = 10; // Decimal assigned
  address = 16'hDEAD; // Hexadecimal
  wr_enable = 1'b1; // Binary
```

```
end else begin
  data = 32'b0;
  wr_enable = 1'b0;
  address = address + 1;
end
```
You could download file one_day2.v here

One could use any operator in the condition checking, as in the case of C language. If needed we can have nested if else statements; statements without else are also ok, but they have their own problem, when modeling combinational logic, in case they result in a Latch (this is not always true).

Case

Case statements are used where we have one variable which needs to be checked for multiple values. like an address decoder, where the input is an address and it needs to be checked for all the values that it can take. Instead of using multiple nested if-else statements, one for each value we're looking for, we use a single case statement: this is similar to switch statements in languages like C++.

Case statements begin with the reserved word **case** and end with the reserved word **endcase** (Verilog does not use brackets to delimit blocks of code). The cases, followed with a colon and the statements you wish executed, are listed within these two delimiters. It's also a good idea to have a **default** case. Just like with a finite state machine (FSM), if the Verilog machine enters into a non-covered statement, the machine hangs. Defaulting the statement with a return to idle keeps us safe.

```
case(address)
  0 : memory_cell_0 = 16’hffe0;
  1 : memory_cell_1 = 16’hac81;
```

```
   2 : memory_cell_2 = 16'h3f76;
   default : memory_cell_1 = 16'h0000;
endcase
```
You could download file one_day3.v here

Note: One thing that is common to if-else and case statement is that, if you don't cover all the cases (don't have 'else' in If-else or 'default' in Case), and you are trying to write a combinational statement, the synthesis tool will infer Latch.

While

A while statement executes the code within it repeatedly if the condition it is assigned to check returns true. While loops are not synthesizable in hardware and not normally used for models in real life, but they are used in test benches. As with other statement blocks, they are delimited by begin and end.

```
while (free_time) begin
  $display ("Continue with webpage development");
end
```
You could download file one_day4.v here

As long as free_time variable is set, code within the begin and end will be executed. i.e print "Continue with web development". Let's looks at a stranger example, which uses most of Verilog constructs. Well, you heard it right. Verilog has fewer reserved words than VHDL, and in this few, we use even lesser for actual coding. So good of Verilog... so right.

```
module counter (clk,rst,enable,count);
input clk, rst, enable;
output [3:0] count;
reg [3:0] count;

always @ (posedge clk or posedge rst)
if (rst) begin
  count <= 0;
end else begin : COUNT
  while (enable) begin
    count <= count + 1;
    disable COUNT;
  end
end

endmodule
```

You could download file one_day5.v here

The example above uses most of the constructs of Verilog. You'll notice a new block called always - this illustrates one of the key features of Verilog. Most software languages, as we mentioned before, execute sequentially - that is, statement by statement. Verilog programs, on the other hand, often have many statements executing in parallel. All blocks marked always will run - simultaneously - when one or more of the conditions listed within it is fulfilled.

In the example above, the always block will run when either rst or clk reaches a **positive edge** - that is, when their value has risen from 0 to 1. You can have two or more **always** blocks in a program going at the same time (not shown here, but commonly used).

We can disable a block of code, by using the reserve word disable. In the above example, after each counter increment, the COUNT block of code (not shown here) is disabled.

For loop

For loops in Verilog are almost exactly like for loops in C or C++. The only difference is that the ++ and -- operators are not supported in Verilog. Instead of writing i++ as you would in C, you need to write out its full operational equivalent, i = i + 1.

```
      for (i = 0; i < 16; i = i +1) begin
             $display ("Current value of i is %d", i);
      end
```

You could download file one_day6.v here

This code will print the numbers from 0 to 15 in order. Be careful when using for loops for register transfer logic (RTL) and make sure your code is actually sanely implementable in hardware... and that your loop is not infinite. The for keyword is synthesizable in hardware.

Repeat

Repeat is similar to the for loop we just covered. Instead of explicitly specifying a variable and incrementing it when we declare the for loop, we tell the program how many times to run through the code, and no variables are incremented (unless we want them to be, like in this example).

```
repeat (16) begin
```

```
    $display ("Current value of i is %d", i);
    i = i + 1;
end
```

You could download file one_day7.v here

The output is exactly the same as in the previous for-loop program example. It is relatively rare to use a repeat (or for-loop) in actual hardware implementation. However, the repeat keyword is synthesizable in hardware.

Summary

- While, if-else, case(switch) statements are the same as in C language.
- If-else and case statements require all the cases to be covered for combinational logic.
- For-loop is the same as in C, but no ++ and -- operators.
- Repeat is the same as the for-loop but without the incrementing variable.

////////////////////////////////Lesson #3///

In this lesson, let's explore the Verilog control statements. The control statements are a big contributor to a lot of Verilog code out there in the world. The control statements encompass both synthesizable and non-synthesizable Verilog code.

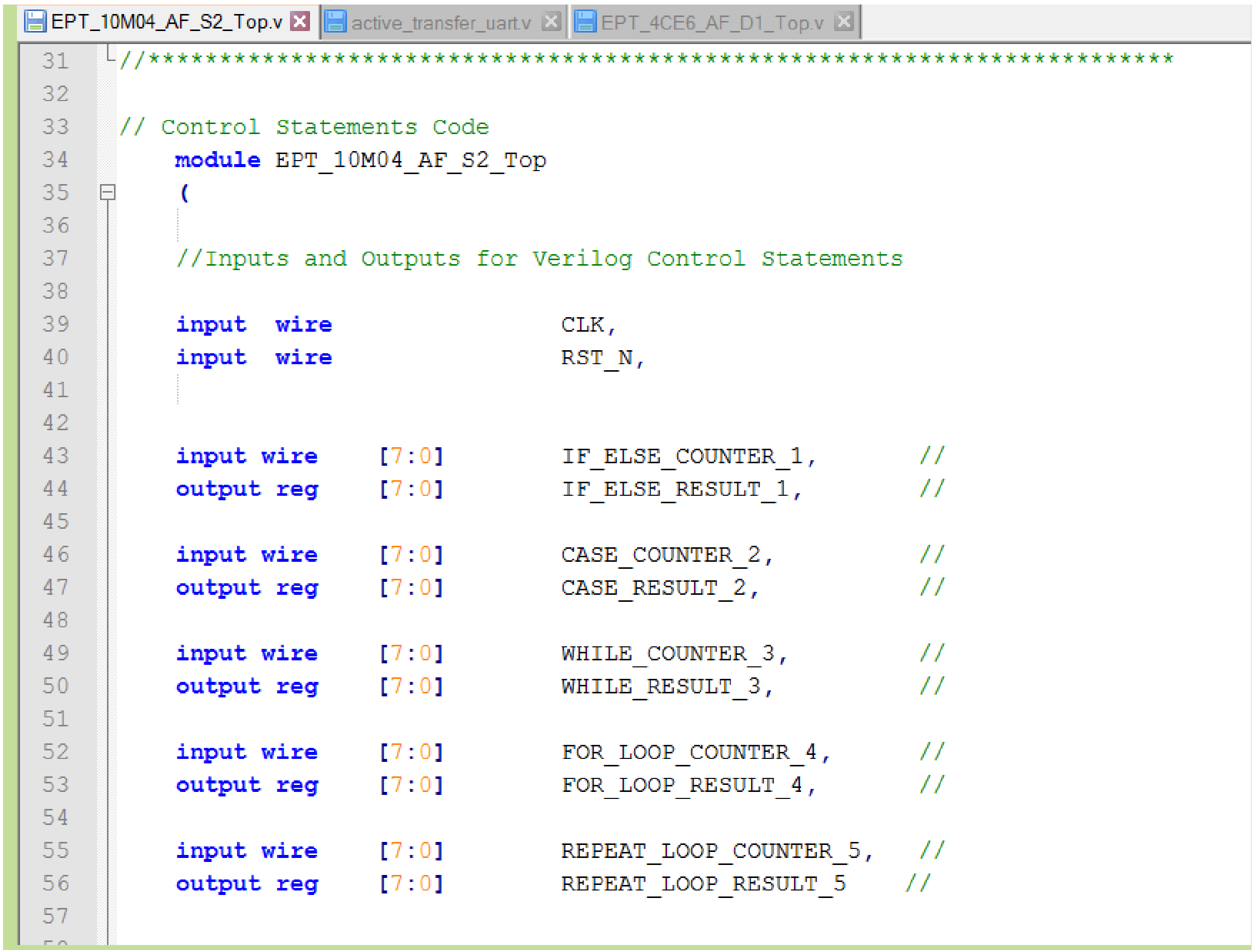
Go to the xx_Project_xx_DVD-> Verilog Getting Started->Tutorials_HDL. Copy the top level Lesson 3 folder to the users local drive. Go through the initial steps as outlined in Lesson 1. Those initial steps open ModelSim, Change

Directory to the ModelSim Folder, Compile the source file, then start the simulation using the “do sim_ept_10m04_top.do”

This lesson will introduce the user to Verilog Control Statements. It provides a Testbench and user code. The Testbench will exercise each control and display the results to the log window of ModelSim. The user code is organized as a module with only synthesizable code that mimics the project that will go into an FPGA. The user code includes non-synthesizable code that can only be used in a testbench. There are three files used in Lesson 3.

- **EPT_10M04_AF_S2_Top.v**
- **tb_ept_10m04_top.v**
- **tb_define.v**

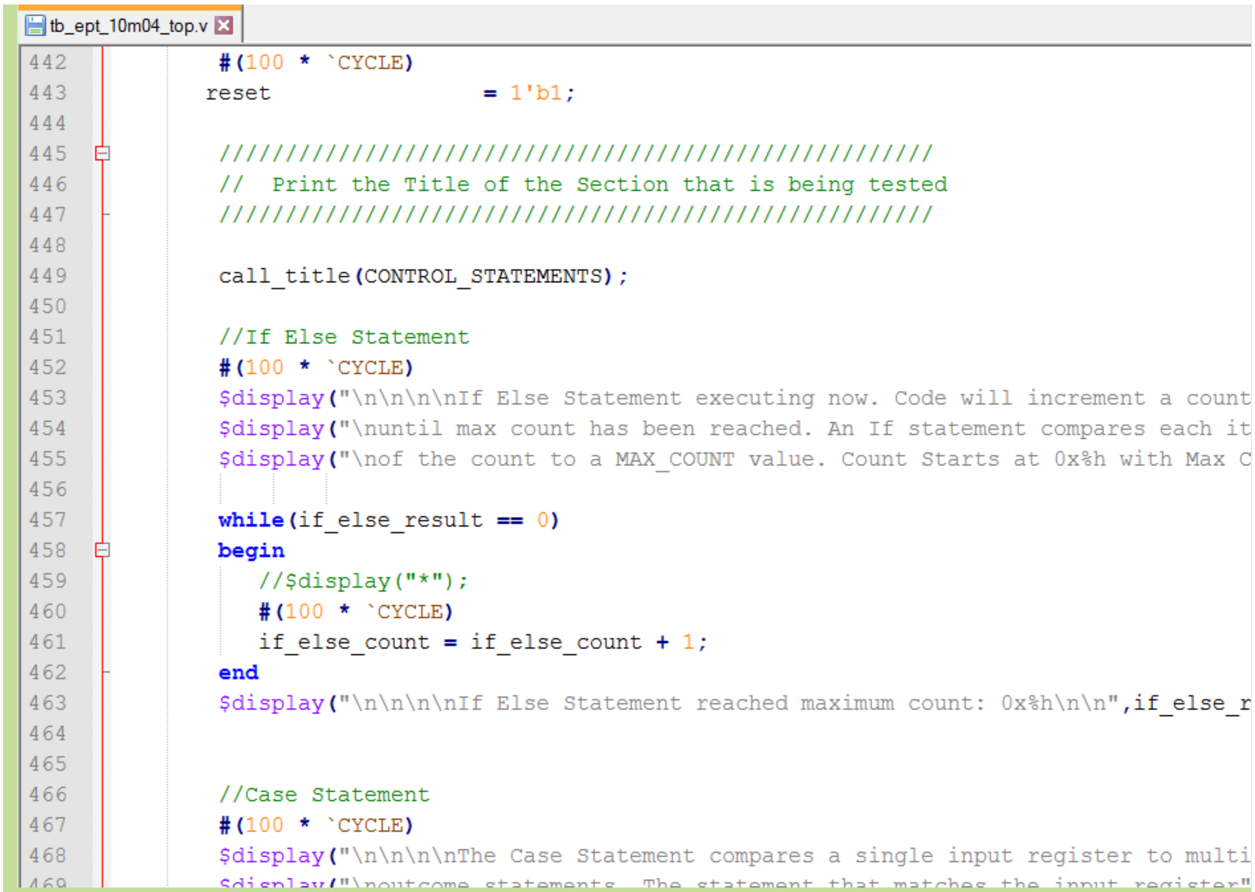
The user code:



EPT_10M04_AF_S2_Top.v | active_transfer_uart.v | EPT_4CE6_AF_D1_Top.v

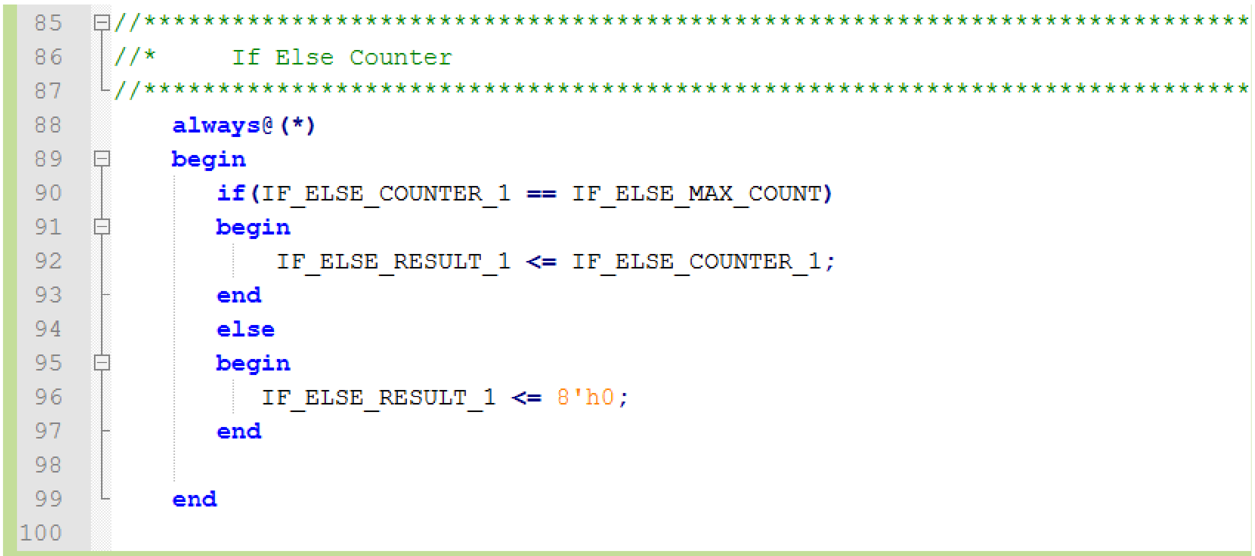
```
//*********************************************************************

// Control Statements Code
    module EPT_10M04_AF_S2_Top
    (

    //Inputs and Outputs for Verilog Control Statements

    input  wire                 CLK,
    input  wire                 RST_N,


    input wire     [7:0]        IF_ELSE_COUNTER_1,        //
    output reg     [7:0]        IF_ELSE_RESULT_1,         //

    input wire     [7:0]        CASE_COUNTER_2,           //
    output reg     [7:0]        CASE_RESULT_2,            //

    input wire     [7:0]        WHILE_COUNTER_3,          //
    output reg     [7:0]        WHILE_RESULT_3,           //

    input wire     [7:0]        FOR_LOOP_COUNTER_4,       //
    output reg     [7:0]        FOR_LOOP_RESULT_4,        //

    input wire     [7:0]        REPEAT_LOOP_COUNTER_5,    //
    output reg     [7:0]        REPEAT_LOOP_RESULT_5     //
```

The Testbench code contains the stimulus for the user code:


tb_ept_10m04_top.v

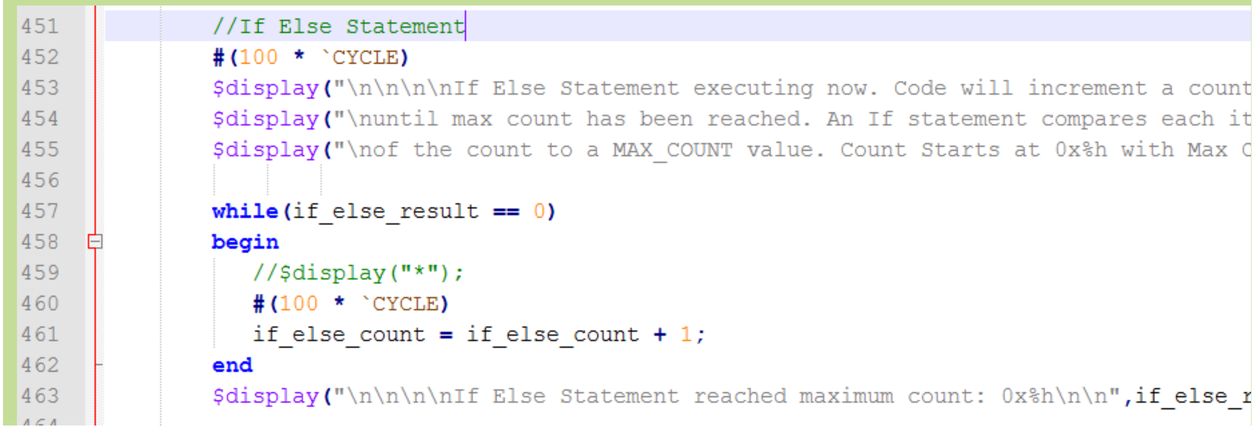
```
    #(100 * `CYCLE)
   reset                  = 1'b1;

    ///////////////////////////////////////////////////////
    //  Print the Title of the Section that is being tested
    ///////////////////////////////////////////////////////

    call_title(CONTROL_STATEMENTS);

    //If Else Statement
    #(100 * `CYCLE)
    $display("\n\n\n\nIf Else Statement executing now. Code will increment a count
    $display("\nuntil max count has been reached. An If statement compares each it
    $display("\nof the count to a MAX_COUNT value. Count Starts at 0x%h with Max C

    while(if_else_result == 0)
    begin
       //$display("*");
       #(100 * `CYCLE)
       if_else_count = if_else_count + 1;
    end
    $display("\n\n\n\nIf Else Statement reached maximum count: 0x%h\n\n",if_else_r


    //Case Statement
    #(100 * `CYCLE)
    $display("\n\n\n\nThe Case Statement compares a single input register to multi
    $display("\noutcome statements. The statement that matches the input register"
```

The If – Else control statement user code takes in an eight bit count value from the Testbench and compares it to a maximum count value. If the incoming value is lesser than the max count, it returns a value of zero. If the incoming value is greater than max count, it returns the incoming value. This shows how the "if" a value is equal to zero, "then" execute a statement, "else" execute a different statement.



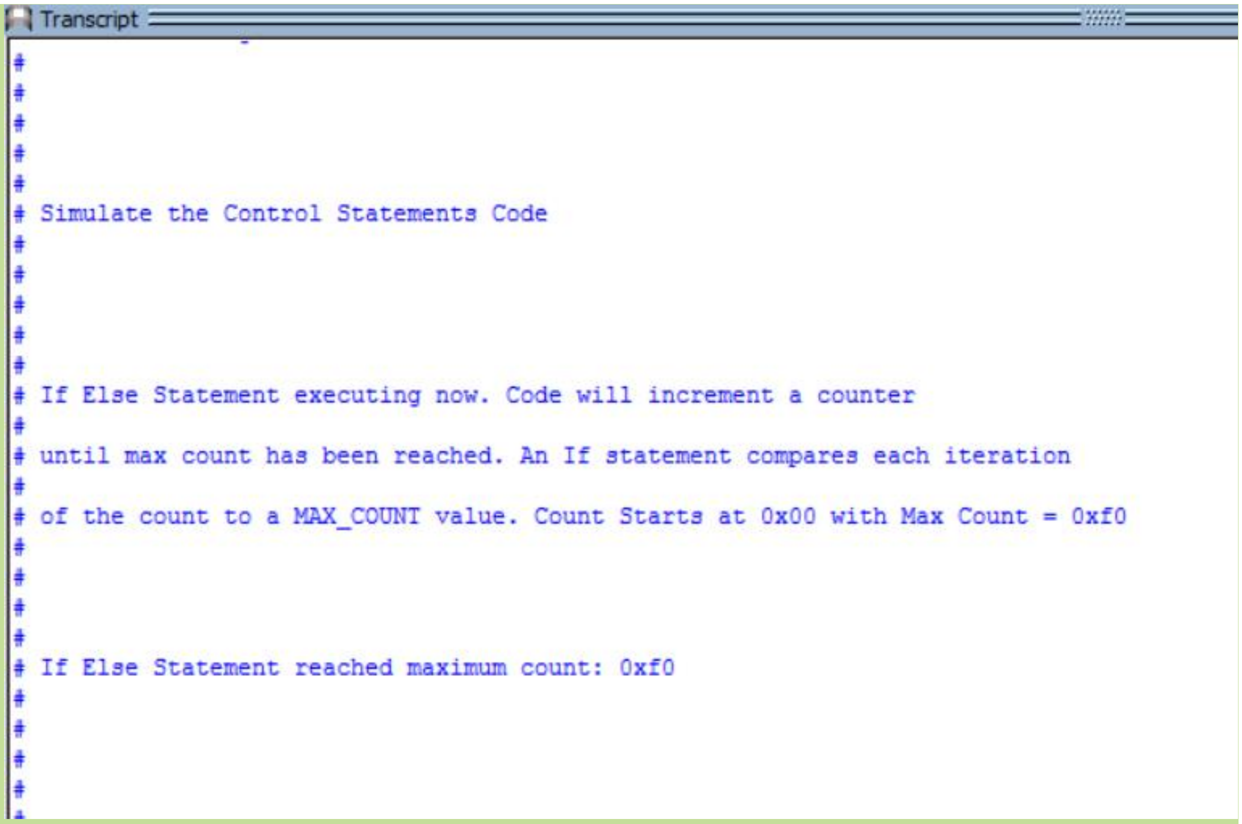
```
//***********************************************************************
//*      If Else Counter
//***********************************************************************
    always@(*)
    begin
        if(IF_ELSE_COUNTER_1 == IF_ELSE_MAX_COUNT)
        begin
            IF_ELSE_RESULT_1 <= IF_ELSE_COUNTER_1;
        end
        else
        begin
            IF_ELSE_RESULT_1 <= 8'h0;
        end

    end
```

The Testbench produces a count by using the "+" operator and adding a '1' to the current count value. The resulting incremented count is transmitted to the IF ELSE Block in the user code.



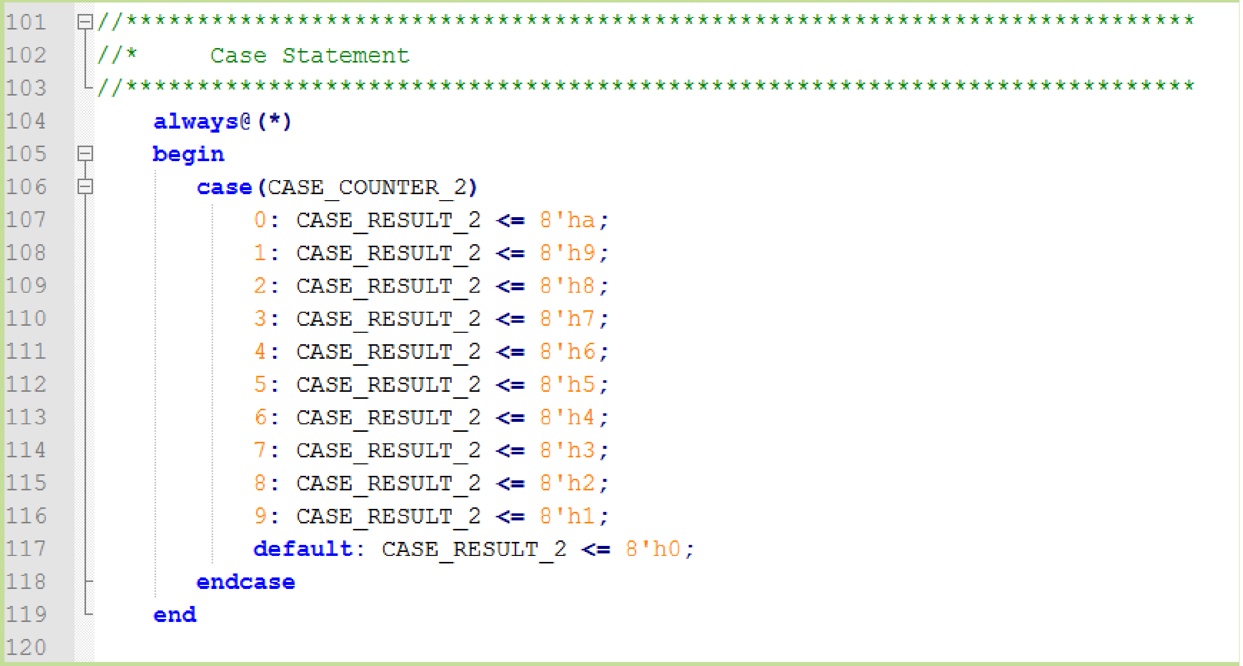
```
        //If Else Statement
        #(100 * `CYCLE)
        $display("\n\n\n\nIf Else Statement executing now. Code will increment a count
        $display("\nuntil max count has been reached. An If statement compares each it
        $display("\nof the count to a MAX_COUNT value. Count Starts at 0x%h with Max C

        while(if_else_result == 0)
        begin
            //$display("*");
            #(100 * `CYCLE)
            if_else_count = if_else_count + 1;
        end
        $display("\n\n\n\nIf Else Statement reached maximum count: 0x%h\n\n",if_else_r
```

Here, the Testbench uses the 'while' statement to cause the Testbench to compare the control and if it is true, execute the statements in the loop. When the statements have completed, the control is again compared to a value. If true, the loop continues. This cycle continues until the while control is false. The user code is comparing the each incremented count to the max value. When the count is greater than the max count, the result is transmitted to the Testbench and compared to the while control. At this point the while control will be false and the loop is exited. The next statement to execute is the "$display("If Else Statement reached……"). The result on the log window of the ModelSim is:



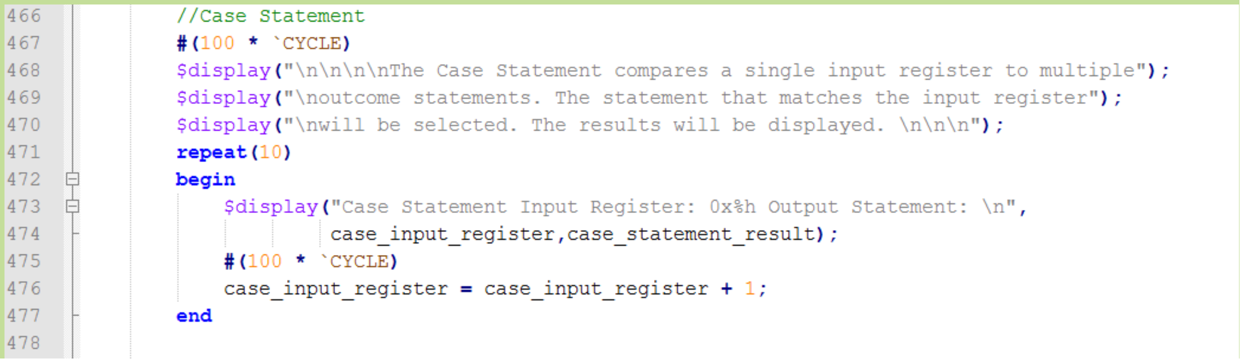
```
#
#
#
#
#
# Simulate the Control Statements Code
#
#
#
#
#
# If Else Statement executing now. Code will increment a counter
#
# until max count has been reached. An If statement compares each iteration
#
# of the count to a MAX_COUNT value. Count Starts at 0x00 with Max Count = 0xf0
#
#
#
#
# If Else Statement reached maximum count: 0xf0
#
#
#
#
```

The Case Control Statement uses the case->select statement->execute statement.
case('control')
1: 'statement';
2: 'statement':
.
.
Endcase



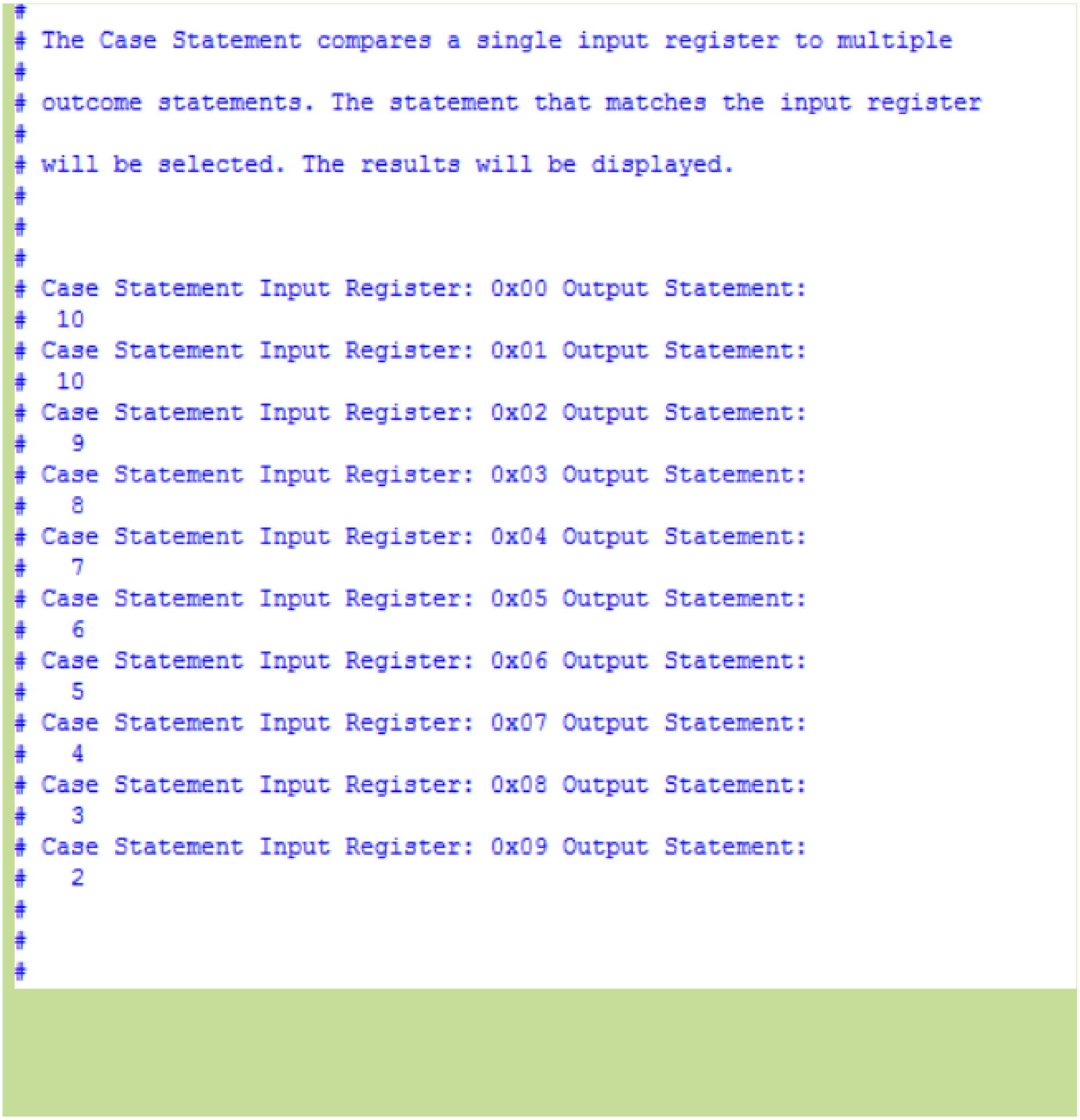
```
//**************************************************************************
//*      Case Statement
//**************************************************************************
    always@(*)
    begin
        case(CASE_COUNTER_2)
            0: CASE_RESULT_2 <= 8'ha;
            1: CASE_RESULT_2 <= 8'h9;
            2: CASE_RESULT_2 <= 8'h8;
            3: CASE_RESULT_2 <= 8'h7;
            4: CASE_RESULT_2 <= 8'h6;
            5: CASE_RESULT_2 <= 8'h5;
            6: CASE_RESULT_2 <= 8'h4;
            7: CASE_RESULT_2 <= 8'h3;
            8: CASE_RESULT_2 <= 8'h2;
            9: CASE_RESULT_2 <= 8'h1;
            default: CASE_RESULT_2 <= 8'h0;
        endcase
    end
```

The Testbench transmits an eight bit count value to the user code. This count value determines which statement is selected. Then the statement is executed.



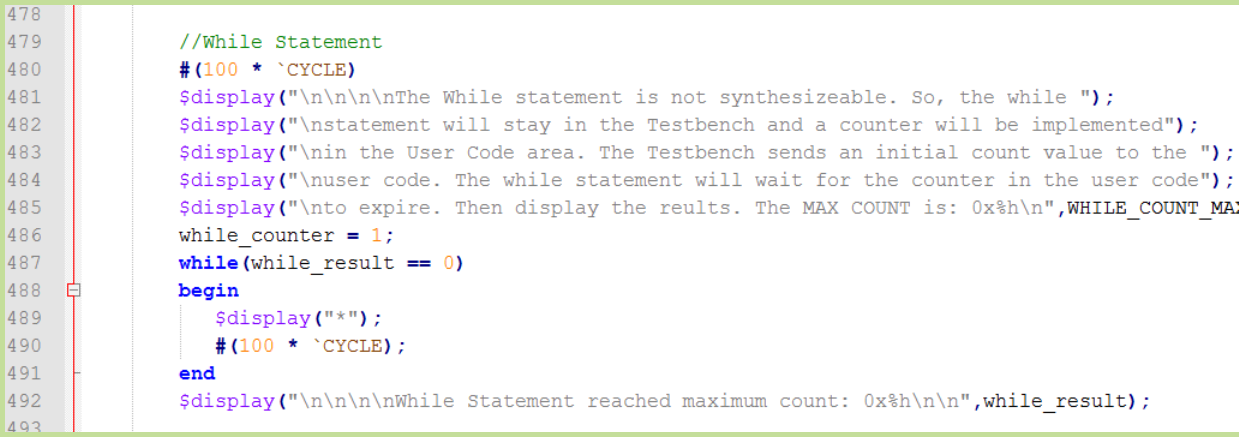
```
        //Case Statement
        #(100 * `CYCLE)
        $display("\n\n\n\nThe Case Statement compares a single input register to multiple");
        $display("\noutcome statements. The statement that matches the input register");
        $display("\nwill be selected. The results will be displayed. \n\n\n");
        repeat(10)
        begin
            $display("Case Statement Input Register: 0x%h Output Statement: \n",
                     case_input_register,case_statement_result);
            #(100 * `CYCLE)
            case_input_register = case_input_register + 1;
        end
```

The Testbench uses the 'repeat' control statement to cause the statements between begin/end to execute ten times. Each iteration through the repeat cycle causes the 'case_input_register' to increment by one. This value is transmitted to the user code and selects a statement to execute based on the case control. The result on the log window of ModelSim is:



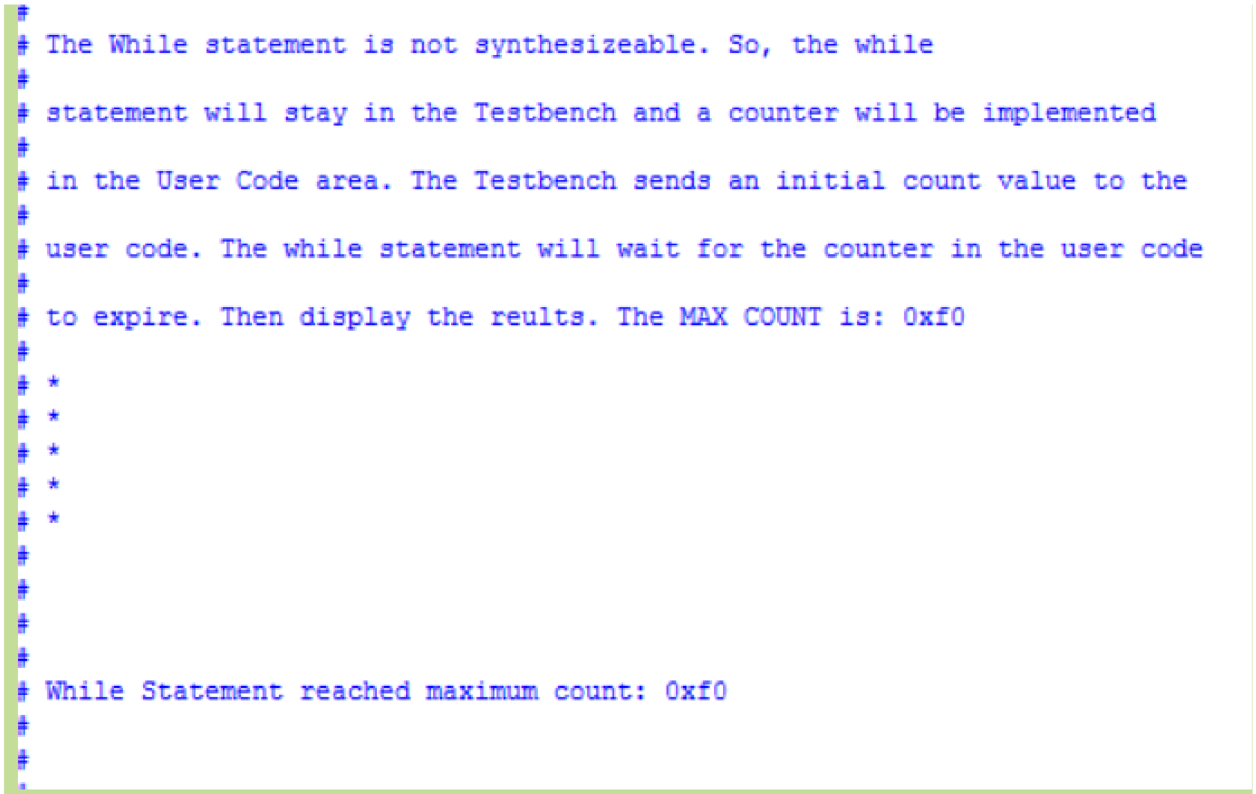
```
#
# The Case Statement compares a single input register to multiple
#
# outcome statements. The statement that matches the input register
#
# will be selected. The results will be displayed.
#
#
#
# Case Statement Input Register: 0x00 Output Statement:
#  10
# Case Statement Input Register: 0x01 Output Statement:
#  10
# Case Statement Input Register: 0x02 Output Statement:
#   9
# Case Statement Input Register: 0x03 Output Statement:
#   8
# Case Statement Input Register: 0x04 Output Statement:
#   7
# Case Statement Input Register: 0x05 Output Statement:
#   6
# Case Statement Input Register: 0x06 Output Statement:
#   5
# Case Statement Input Register: 0x07 Output Statement:
#   4
# Case Statement Input Register: 0x08 Output Statement:
#   3
# Case Statement Input Register: 0x09 Output Statement:
#   2
#
#
#
```

The while statement is not synthesizable. So, it will be implemented in the Testbench while the user code will perform an incremented count and produce a non-zero result when the count reaches maximum count.



```

        //While Statement
        #(100 * `CYCLE)
        $display("\n\n\n\nThe While statement is not synthesizeable. So, the while ");
        $display("\nstatement will stay in the Testbench and a counter will be implemented");
        $display("\nin the User Code area. The Testbench sends an initial count value to the ");
        $display("\nuser code. The while statement will wait for the counter in the user code");
        $display("\nto expire. Then display the reults. The MAX COUNT is: 0x%h\n",WHILE_COUNT_MA
        while_counter = 1;
        while(while_result == 0)
        begin
           $display("*");
           #(100 * `CYCLE);
        end
        $display("\n\n\n\nWhile Statement reached maximum count: 0x%h\n\n",while_result);
```

The while loop takes the value it receives from result of the user code while block and compares it to the control of the while statement. If it is zero, the statements in the Testbench while loop execute. This continues as the user code increments the counter based on the clock supplied by the Test bench. In this case it is 50 MHz. When the counter reaches the max value, the user code transmits this value. The Testbech while loop compares this to zero and determines the while control is false and exits the loop. The result in the log window of ModelSIm:



```
#
# The While statement is not synthesizeable. So, the while
#
# statement will stay in the Testbench and a counter will be implemented
#
# in the User Code area. The Testbench sends an initial count value to the
#
# user code. The while statement will wait for the counter in the user code
#
# to expire. Then display the reults. The MAX COUNT is: 0xf0
#
# *
# *
# *
# *
# *
#
#
#
#
# While Statement reached maximum count: 0xf0
#
#
```

The For Loop is used to execute statements within a pre-set range. Usually the for loop repeats the same statement with a slight modification based on the index counter of the for loop.

VARIABLE ASSIGNMENT

In digital there are two types of elements, combinational and sequential. Of course we know this. But the question is "How do we model this in Verilog ?". Well Verilog provides two ways to model the combinational logic and only one way to model sequential logic.

- Combinational elements can be modeled using assign and always statements.
- Sequential elements can be modeled using only always statement.
- There is a third block, which is used in test benches only: it is called Initial statement.

Initial Blocks

An initial block, as the name suggests, is executed only once when simulation starts. This is useful in writing test benches. If we have multiple initial blocks, then all of them are executed at the beginning of simulation.

Example

```
initial  begin
                clk = 0;
                reset = 0;
                req_0 = 0;
                req_1 = 0;
end
```

You could download file one_day8.v here

In the above example, at the beginning of simulation, (i.e. when time = 0), all the variables inside the begin and end block are driven zero.

Always Blocks

As the name suggests, an always block executes always, unlike initial blocks which execute only once (at the beginning of simulation). A second difference is that an always block should have a sensitive list or a delay associated with it.

The sensitive list is the one which tells the always block when to execute the block of code, as shown in the figure below. The @ symbol after reserved word ' always', indicates that the block will be triggered "at" the condition in parenthesis after symbol @.

An always block indicates a set of procedural instructions that happen in the order they are written. The reg data type can hold on to its value while the rest of the always block is completed, while the 'wire' data type (the default one) does not. Reg is what has to be used in every always block, even though that particular block is short and ends immediately. Assign is used for wire types and can be thought of as connecting physical wires between pieces of hardware, or a path for a signal to travel.

One important note about always block: it can not drive wire data type, but can drive reg and integer data types.

```
always  @ (a or b or sel)
begin
  y = 0;
  if (sel == 0) begin
    y = a;
```

```
   end else begin
     y = b;
   end
end
```

You could download file one_day9.v here

The above example is a 2:1 mux, with input a and b; sel is the select input and y is the mux output. In any combinational logic, output changes whenever input changes. This theory when applied to always blocks means that the code inside always blocks needs to be executed whenever the input variables (or output controlling variables) change. These variables are the ones included in the sensitive list, namely a, b and sel.

There are two types of sensitive list: level sensitive (for combinational circuits) and edge sensitive (for flip-flops). The code below is the same 2:1 Mux but the output y is now a flip-flop output.

```
always  @ (posedge clk )
if (reset == 0) begin
  y <= 0;
end else if (sel == 0) begin
  y <= a;
end else begin
  y <= b;
end
```

You could download file one_day10.v here

We normally have to reset flip-flops, thus every time the clock makes the transition from 0 to 1 (posedge), we check if reset is asserted (synchronous reset), then we go on with normal logic. If we look closely we see that in the case of combinational logic we had "=" for assignment, and for the sequential block we had the "<=" operator. Well, "=" is blocking assignment and "<=" is nonblocking assignment. "=" executes code sequentially inside a begin / end, whereas nonblocking "<=" executes in parallel.

We can have an always block without sensitive list, in this case we need to have a delay as shown in the code below.

```
always  begin
   #5  clk = ~clk;
end
```

You could download file one_day11.v here

#5 in front of the statement delays its execution by 5 time units.

Assign Statement

An assign statement is used for modeling only combinational logic and it is executed continuously. So the assign statement is called 'continuous assignment statement' as there is no sensitive list.

```
assign out = (enable) ? data : 1'bz;
```

You could download file one_day12.v here

The above example is a tri-state buffer. When enable is 1, data is driven to out, else out is pulled to high-impedance. We can have nested conditional operators to construct mux, decoders and encoders.

```
assign out = data;
```

You could download file one_day13.v here

The assignment statement starts with the keyword assign followed by the signal name which can be either a single signal or a concatenation of different signal nets. The driver strength and delay are optional and are mostly used for dataflow modeling than synthesizing into real hardware. The expression or signal on the right hand side is evaluated and assigned to the net or expression of nets on the left hand side.

```
 assign <net_expression> = [drive_strength] [delay] <expression of
different signals or constant value>
```

Rules:

- LHS should always be a scalar or vector net or a concatenation of scalar or vector nets and never a scalar or vector register.
- RHS can contain scalar or vector registers and function calls.
- Whenever any operand on the RHS changes in value, LHS will be updated with the new value.
- Assign statements are also called continuous assignments and are always active.

Task and Function

Often times we find certain pieces of code to be repetitive and called multiple times within the RTL. They mostly do not consume time and might involve complex calculations that need to be done with different values. In such cases, we can declare a ‘function’ and place the repetitive code inside the function and allow it to return the result. This will reduce the amount of lines in the RTL drastically since all you need to do now is to do a function call and pass data on which the computation needs to be performed. In fact, this is very similar to the functions in C.

The purpose of a function is to return the value that is to be used in an expression. A function definition always start with the keyword ‘function’ followed by the return type, name and a port list enclosed in parantheses. Verilog knows that a function definition is over when it finds the ‘endfunction’ keyword. Note that a function shall have at least one input declared and the return type will be void if the function does not return anything.

Syntax

```
function [automatic] [return_type] name ([port_list]);
      [statements]
endfunction
```

The keyword ‘automatic’ will make the function reentrant and items declared within the task are dynamically allocated rather than shared between different invocations of the task. This will be useful for recursive functions and when the same function is exectued concurrently by N processes when forked.

Code below is used for calculating even parity.

```
function parity;
input [31:0] data;
integer i;
begin
  parity = 0;
  for (i= 0; i < 32; i = i + 1) begin
```

```
      parity = parity ^ data[i];
    end
  end
  endfunction
```

You could download file one_day14.v here

Functions and tasks have the same syntax; one difference is that tasks can have delays, whereas functions can not have any delay. This means that function can be used for modeling combinational logic.

Function rules

- A function cannot contain any time-controlled statements like #, @, wait, posedge, negedge
- A function cannot start a task because it may consume simulation time, but can call other functions
- A function should have atleast one input
- A function cannot have non-blocking assignments or force-release or assign-deassign
- A function cannot have any triggers
- A function cannot have an output or inout

A second difference is that functions can return a value, whereas tasks can not.

///////////////////////////////////Lesson #4///

In this lesson, let's explore the Verilog Blocks, Tasks and Functions. These items are fundamental when writing test benches. Blocks, Tasks and Functions encompass both synthesizable and non-synthesizable Verilog code.

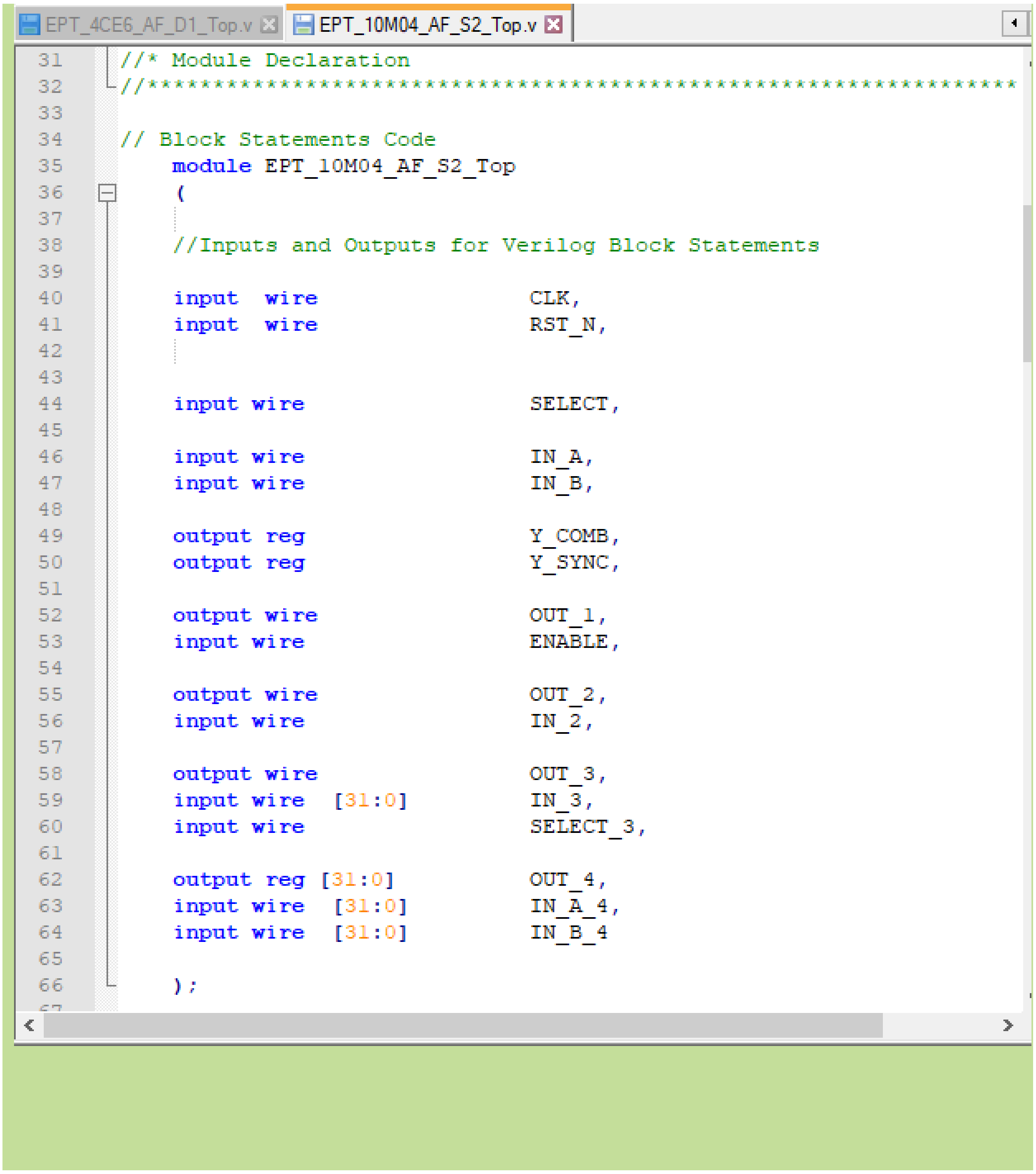
Go to the xx_Project_xx_DVD-> Verilog Getting Started->Tutorials_HDL and copy the Lesson 4 HDL folder to the users local drive. Go through the initial

steps as outlined in Lesson 1. Those initial steps open ModelSim, Change Directory to the ModelSim Folder, Compile the source file, then start the simulation using the “do sim_ept_10m04_top.do”

This lesson will introduce the user to Verilog Blocks, Tasks and Functions. It provides a Testbench and user code. The Testbench will exercise each item and display the results to the log window of ModelSim. The user code is organized as a module it includes non-synthesizable code that can only be used in a testbench. There are three files used in Lesson 4.

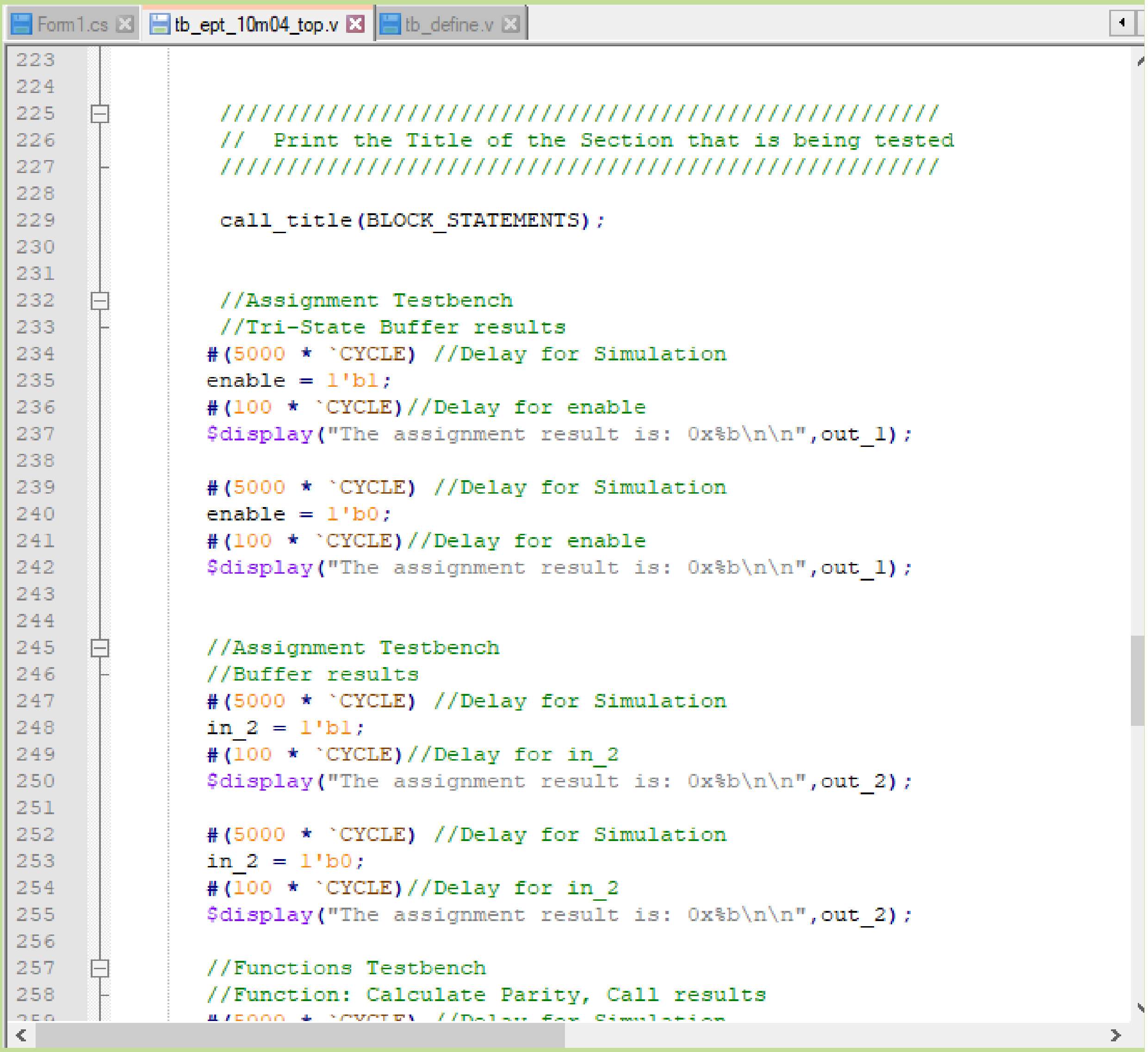
- **EPT_10M04_AF_S2_Top.v**
- **tb_ept_10m04_top.v**
- **tb_define.v**

The user code:


EPT_4CE6_AF_D1_Top.v EPT_10M04_AF_S2_Top.v

```
//* Module Declaration
//*******************************************************************

// Block Statements Code
    module EPT_10M04_AF_S2_Top
    (

    //Inputs and Outputs for Verilog Block Statements

    input   wire                CLK,
    input   wire                RST_N,


    input wire                  SELECT,

    input wire                  IN_A,
    input wire                  IN_B,

    output reg                  Y_COMB,
    output reg                  Y_SYNC,

    output wire                 OUT_1,
    input wire                  ENABLE,

    output wire                 OUT_2,
    input wire                  IN_2,

    output wire                 OUT_3,
    input wire   [31:0]         IN_3,
    input wire                  SELECT_3,

    output reg [31:0]           OUT_4,
    input wire   [31:0]         IN_A_4,
    input wire   [31:0]         IN_B_4

    );
```

The Testbench code contains the stimulus for the user code:

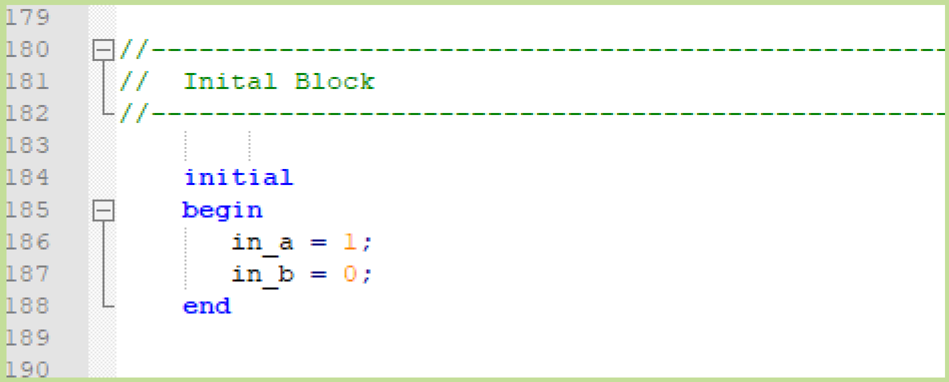


Form1.cs | tb_ept_10m04_top.v | tb_define.v

```


     ////////////////////////////////////////////////////////
     //  Print the Title of the Section that is being tested
     ////////////////////////////////////////////////////////

     call_title(BLOCK_STATEMENTS);


     //Assignment Testbench
     //Tri-State Buffer results
    #(5000 * `CYCLE) //Delay for Simulation
    enable = 1'b1;
    #(100 * `CYCLE)//Delay for enable
    $display("The assignment result is: 0x%b\n\n",out_1);

    #(5000 * `CYCLE) //Delay for Simulation
    enable = 1'b0;
    #(100 * `CYCLE)//Delay for enable
    $display("The assignment result is: 0x%b\n\n",out_1);


    //Assignment Testbench
    //Buffer results
    #(5000 * `CYCLE) //Delay for Simulation
    in_2 = 1'b1;
    #(100 * `CYCLE)//Delay for in_2
    $display("The assignment result is: 0x%b\n\n",out_2);

    #(5000 * `CYCLE) //Delay for Simulation
    in_2 = 1'b0;
    #(100 * `CYCLE)//Delay for in_2
    $display("The assignment result is: 0x%b\n\n",out_2);

    //Functions Testbench
    //Function: Calculate Parity, Call results
```

Initial Block

The Initial statement starts at line 180 in the tb_ept_10m04_top.



```
179
180  //-----------------------------------------------------
181  //   Inital Block
182  //-----------------------------------------------------
183
184      initial
185      begin
186         in_a = 1;
187         in_b = 0;
188      end
189
190
```

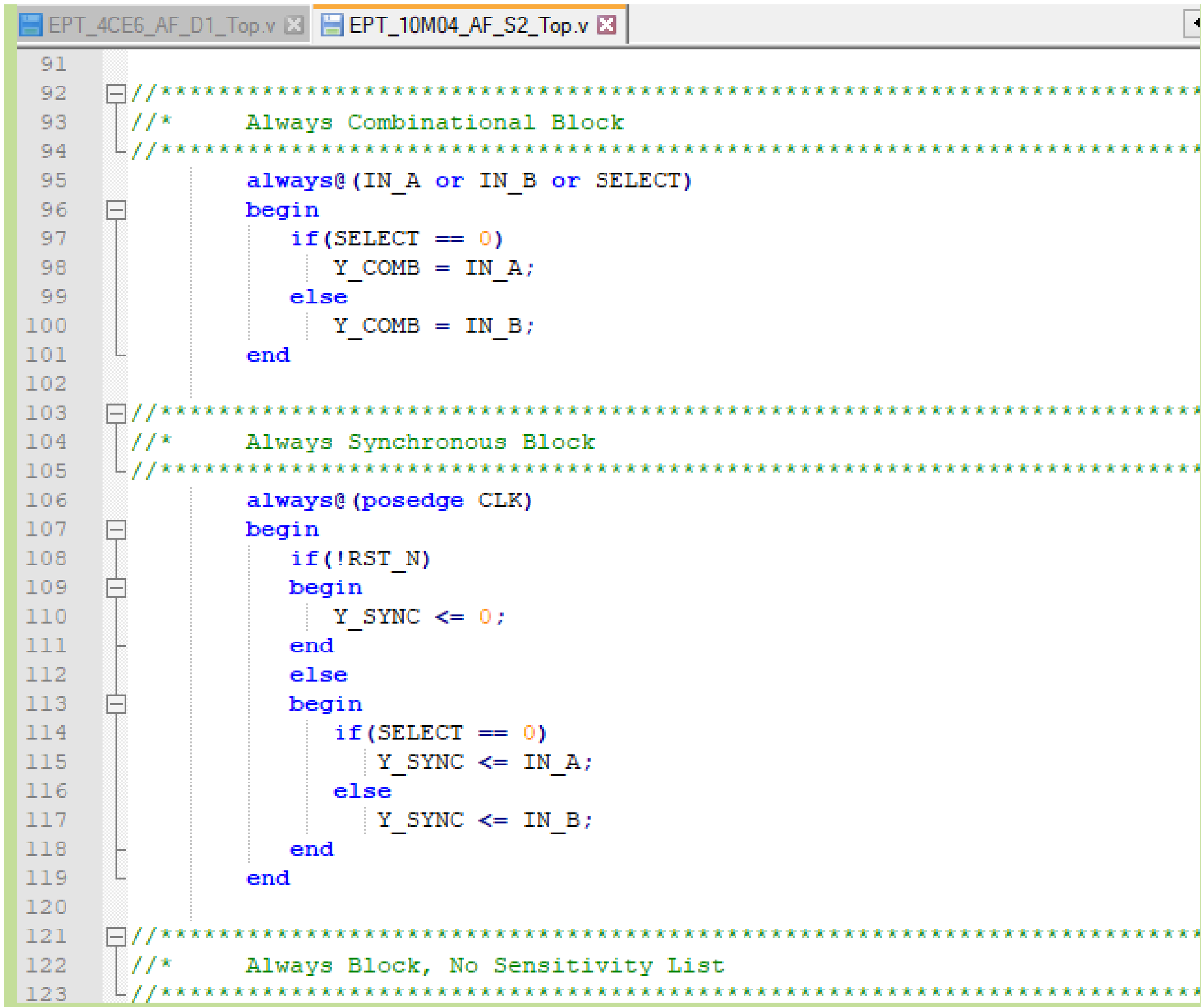
The Initial block is started at the beginning of a simulation at time 0 unit. This block will be executed only once during the entire simulation. Execution of an initial block finishes once all the statements within the block are executed. In the case of Lesson 4, the initial block is used to give a value to registers, in_a and in_b at the beginning of simulation. The registers are set to:

- **in_a = 1**
- **in_b = 0**

There are other Initial blocks in the Lesson 4 testbench. However, these are needed to provide registers and signals with valid settings. We will cover these other testbench Initial Blocks in future Lessons.

Always Combinational Block

The Always Block are used to describe events that should happen under certain conditions. The Lesson 4 Always Block is explored using the various instantiations on line 92 of the EPT_10M04_AF_S2_Top.v file.

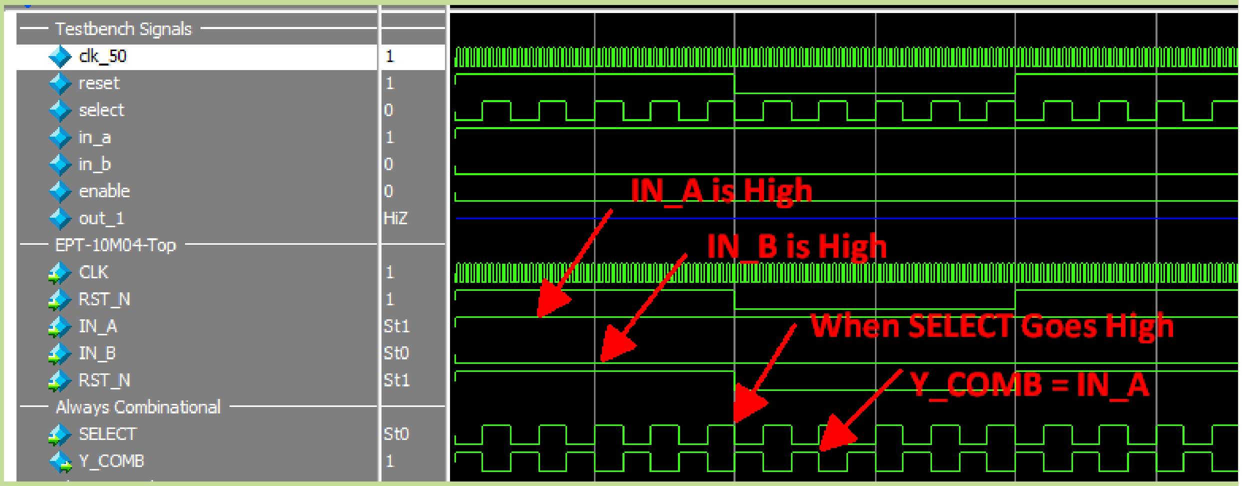


```
//*************************************************************************
//*      Always Combinational Block
//*************************************************************************
        always@(IN_A or IN_B or SELECT)
        begin
            if(SELECT == 0)
                Y_COMB = IN_A;
            else
                Y_COMB = IN_B;
        end

//*************************************************************************
//*      Always Synchronous Block
//*************************************************************************
        always@(posedge CLK)
        begin
            if(!RST_N)
            begin
                Y_SYNC <= 0;
            end
            else
            begin
                if(SELECT == 0)
                    Y_SYNC <= IN_A;
                else
                    Y_SYNC <= IN_B;
            end
        end

//*************************************************************************
//*      Always Block, No Sensitivity List
//*************************************************************************
```

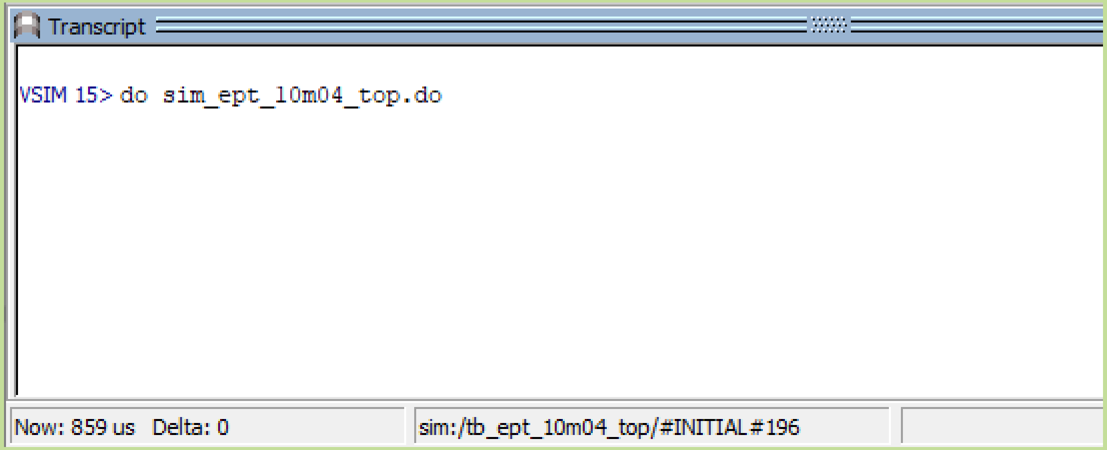
The first Always Block is a Combinational method. This block uses a Sensitivity List immediately following the statement "always". The sensitive list is the one which tells the always block when to execute the block of code.

Here, the Project Top uses the 'while' statement to cause the Always Combinational Block to compare the signals in the Sensitivity List and if true, execute the statements in the loop. When the statements have completed, the control is again compared to a value. If true, the statements are executed again.

This cycle continues until the while control is false. This comparison continually happens until the end of simulation. The output looks like the following:



Run the simulation as per the instructions outlined in Lesson 1. In the command window of the ModelSim, type 'do sim_ept_10m04_top.do'



```
VSIM 15> do sim_ept_10m04_top.do
```

```
Now: 859 us  Delta: 0    sim:/tb_ept_10m04_top/#INITIAL#196
```

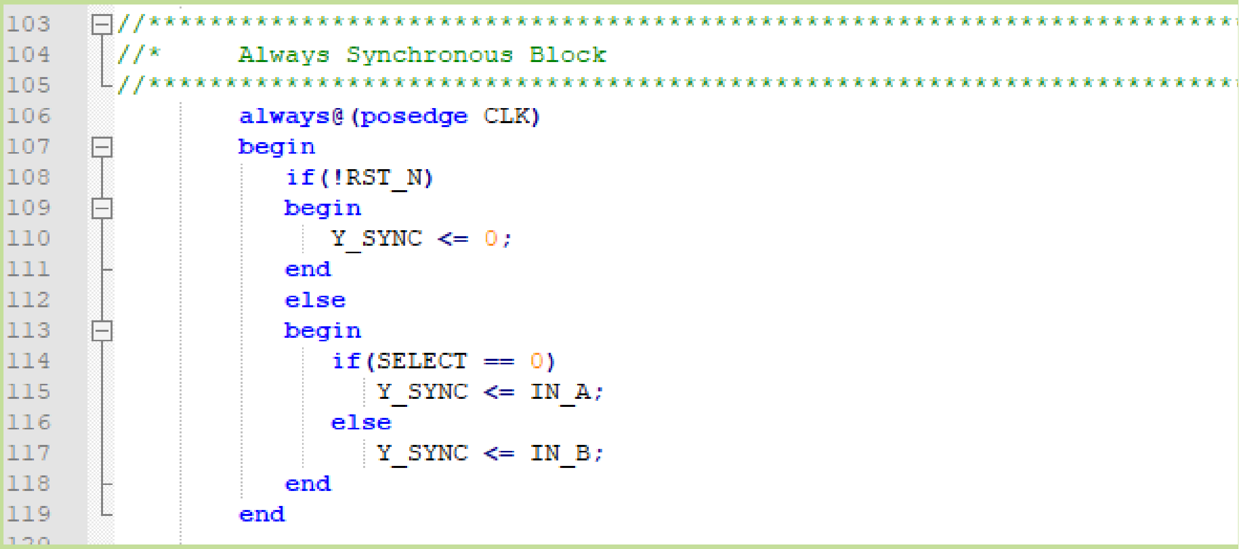
The Combinational Block uses Always->Sensitivity List True->execute statement.

```
if(SELECT == 1)
  Y_COMB = IN_A
else
  Y_COMB = IN_B
```

From the Simulation, we can see that Y_COMB oscillates from low to high with the input SELECT. The user can experiment with this simulation and change the polarity of IN_A and IN_B. Run the simulation again and check the results for the change.

Always Synchronous Block

The Synchronous Always Block starts on line 103 of the EPT_10M04_AF_S2_Top.v file.

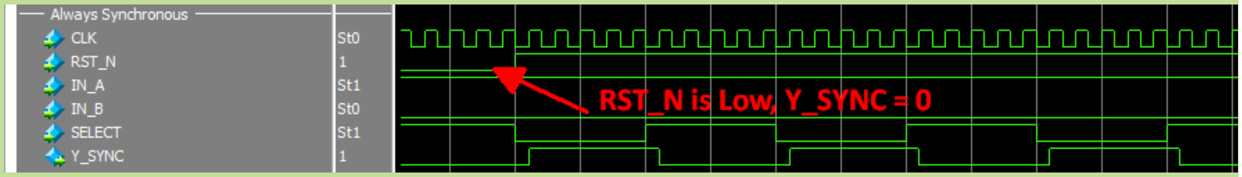


```
//*********************************************************************
//*     Always Synchronous Block
//*********************************************************************
        always@(posedge CLK)
        begin
            if(!RST_N)
            begin
                Y_SYNC <= 0;
            end
            else
            begin
                if(SELECT == 0)
                    Y_SYNC <= IN_A;
                else
                    Y_SYNC <= IN_B;
            end
        end
```

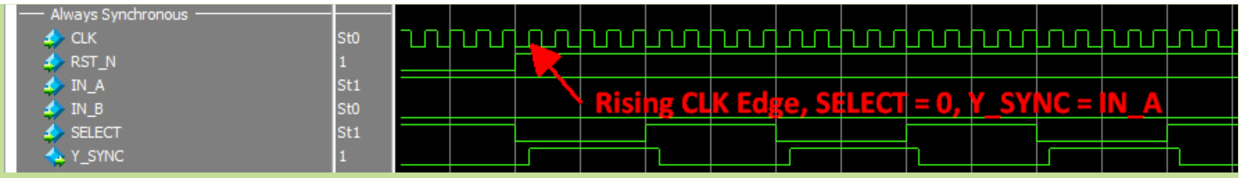
The Synchronous Always Block is different from the Combinational Always Block in that the Synchronous executes the statements only when the clock signal asserts. In hardware, the FPGA will have a clock signal from a separate oscillator. This clock signal is added to an FPGA pin. This clock pin will be used internal in the FPGA to provide the clock for synchronous code. The Always Block will add the clock signal to the Sensitivity List.

```
always@(posedge CLK)
begin
   …
   …
end
```

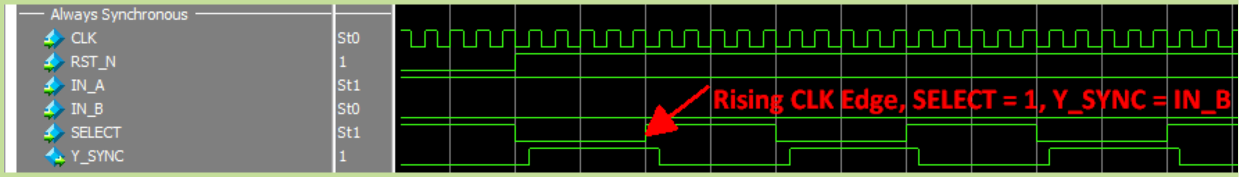
The Always Synchronous block above execute the statements in the begin/end frame only when the signal, CLK, is rising. Run the simulation to see what the synchronous always block does.



Starting at the leftmost part of the simulation, the RST_N is asserted low. So, Y_SYNC is set low. Once the RST_N asserts high, the always statements execute when the CLK signal goes high.



In this case, SELECT = 0 so, Y_SYNC = IN_A= 1. Notice that Y_SYNC changes to to equal IN_A only after the clock has completed the rising edge. You can see from the simulation that SELECT goes from 1 to 0, but the Y_SYNC does not change value until after CLK = 1. This is fundamental of synchronous operation. After five CLK cycles, the SELECT = 1.

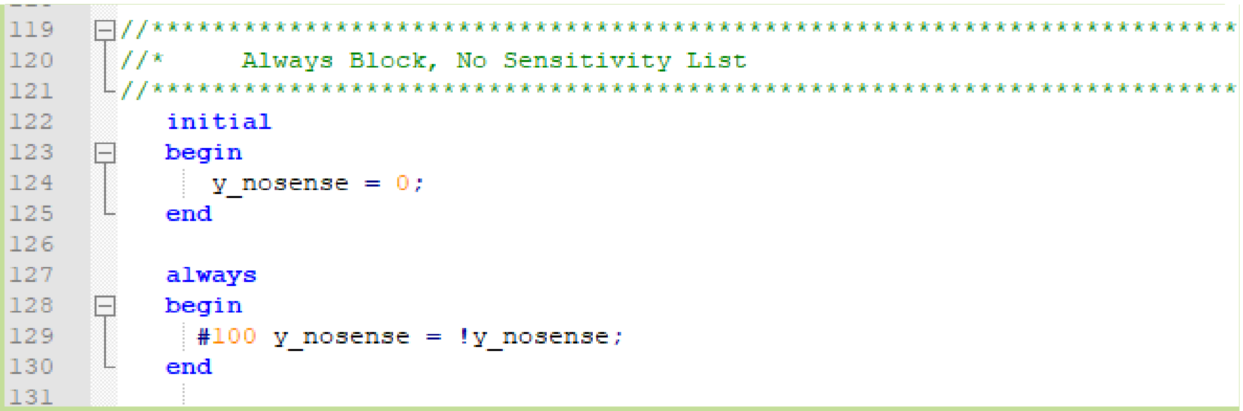


When SELECT = 1, Y_SYNC = IN_B=0. Again, notice that Y_SYNC does not change immediately. It only changes after the rising edge of CLK.

Next the user is encouraged to experiment with the Always Blocks. Change the polarity of the SELECT and re-run the simulation. Notice how the results change.

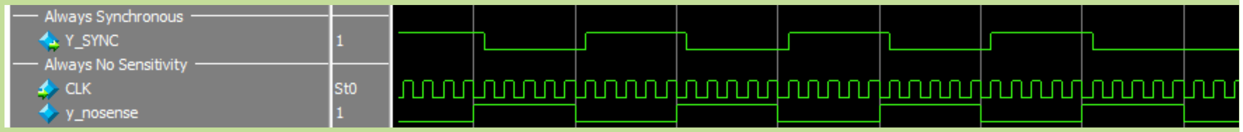
Always Block No Sensitivity List

The Always Block with no Sensitivity List starts on line 119 of the EPT_10M04_AF_S2_Top.v file.



```
//***********************************************************************
//*      Always Block, No Sensitivity List
//***********************************************************************
   initial
   begin
      y_nosense = 0;
   end

   always
   begin
     #100 y_nosense = !y_nosense;
   end
```

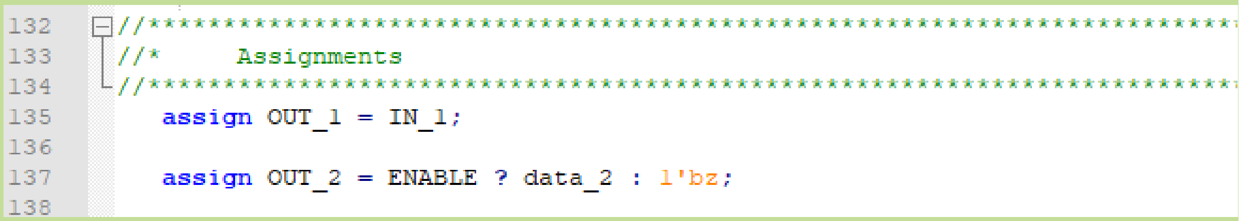
The Sensitivity List of the Always Block has several variations. One such variation is the Always block without Sensitivity List. This block will execute the statements on each simulation time cycle. Remember, this is in deference to the Always Block with Sensitivity List in that the statements are only executed once the events in the Sensitivity List occur. In this exercise, we add a delay of '#100' which means delay the next statement for 100 simulations cycles. The results of y_nosense = !y_nonsense is alternating high and low. The delay causes the oscillation to run slower. Run the simulation to see what the no sensitivity always block does.



Earlier, we set up CLK to equal 50MHz. The 50MHz clock uses a delay of #10 simulation cycles to produce the 50MHz period. The period of our delay #100 is equal to 10 cycles of the 50MHz clock. You can see the Always Block with no Sensitivity List is executing at each simulation cycle. The user is encouraged to experiment with this simulation by changing the value of the delay and re-running the simulation. Notice what the changes are.

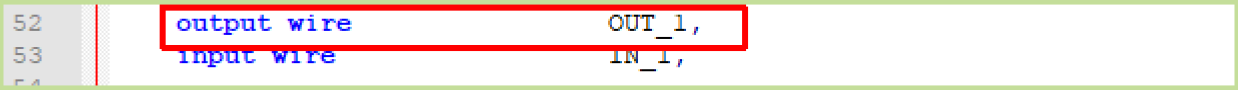
Assignment Statements

The Always Block with no Sensitivity List starts on line 119 of the EPT_10M04_AF_S2_Top.v file.



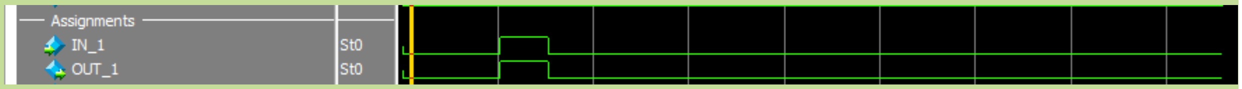
```
//*************************************************************************
//*     Assignments
//*************************************************************************
    assign OUT_1 = IN_1;

    assign OUT_2 = ENABLE ? data_2 : 1'bz;

```

The Assignment Statements are used for combinatorial and register logic. So the assign statement is called 'continuous assignment statement' as there is no sensitive list. The value can either be a constant or an expression comprising a group of signals. Please note the signal on the LHS must be declared as a wire. It can never be declared as a register.



```
    output wire                 OUT_1,
    input wire                  IN_1,
```

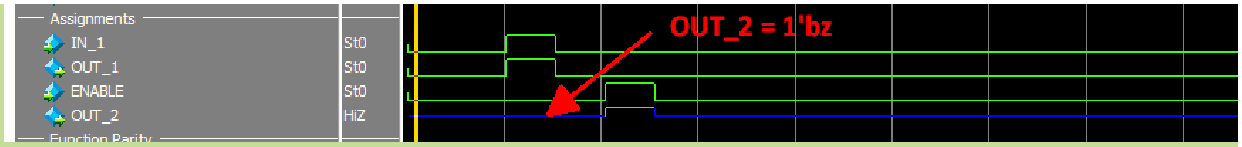
Because the assignment is continuous, OUT_1 above will simply follow any signal applied to the IN_1. Run the simulation to verify this.



For OUT_2, this syntax is legal and produces a reliable output. This is called an inline if-then-else statement.

- **If ENABLE = 1, then OUT_2 = data_2 (assign OUT_2 = ENABLE ? data_2 : 1’bz)**
- **Else if ENABLE =0, then OUT_2 = 1’bz (assign OUT_2 = ENABLE ? data_2 : 1’bz)**

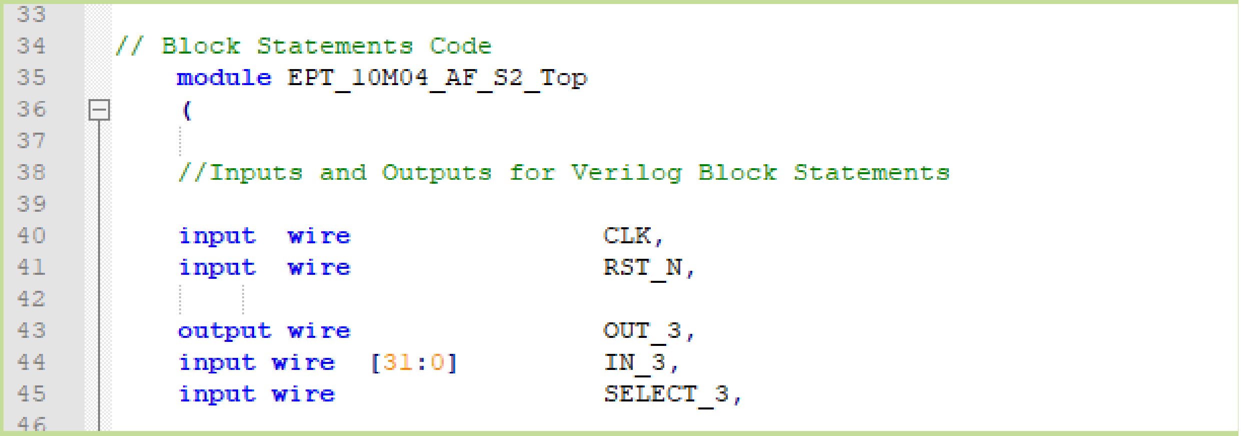
The 1’bz output indicates the tri-state condition. Remember that signals can have one of four states, high, low, don’t care and tri-state. Run the simulation to see the results.



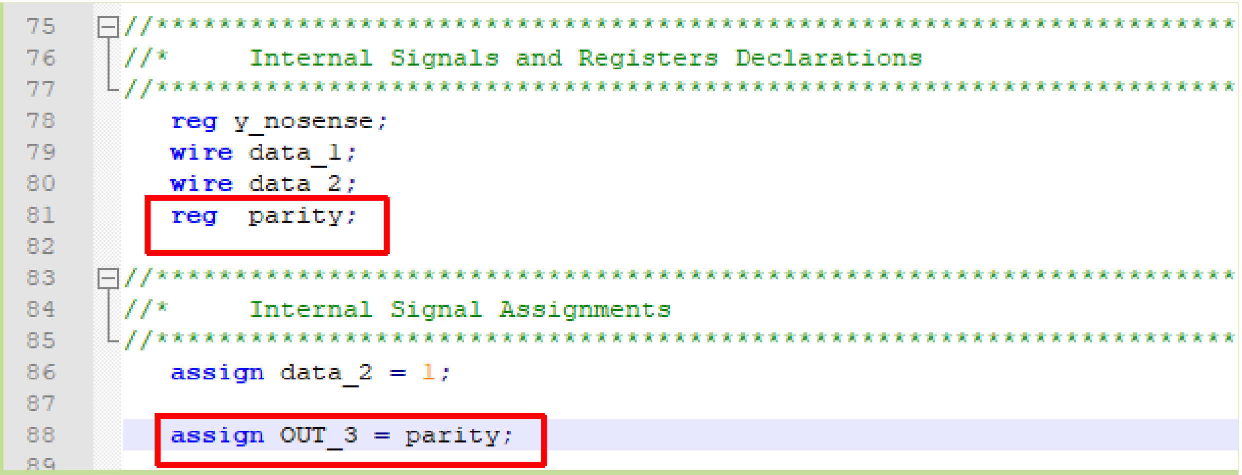
Notice that the blue line in ModelSim indicates the tri-state condition of 1'bz. From the inline if-then-else, when ENABLE = 0, OUT_2 = 1'bz. When ENABLE = 1, the first part of the inline if-then-else executes and the OUT_2 = data_2 = 1. The user is encouraged to experiment with the simulation and change the polarity of the signals and re-run the simulation to examine the results.

Functions

The Functions starts on line 139 of the EPT_10M04_AF_S2_Top.v file. Functions are declared with the "function" keyword. This keyword starts the function, and it ends with the keyword "endfunction". The function can return only one signal or vector, but can have multiple inputs. The function in Lesson 4 calculates the parity of a 32 bit word. This is useful for ensuring digital communications between two devices was successful. It is also a perfect example of the use of functions. This calculation would be repeated for each 32 bit word transferred. This might be performed thousands of times per transaction. The function is listed and called inside the top level project. The testbench provides a signal to start the calculation. For the parity function notice the signal inputs.

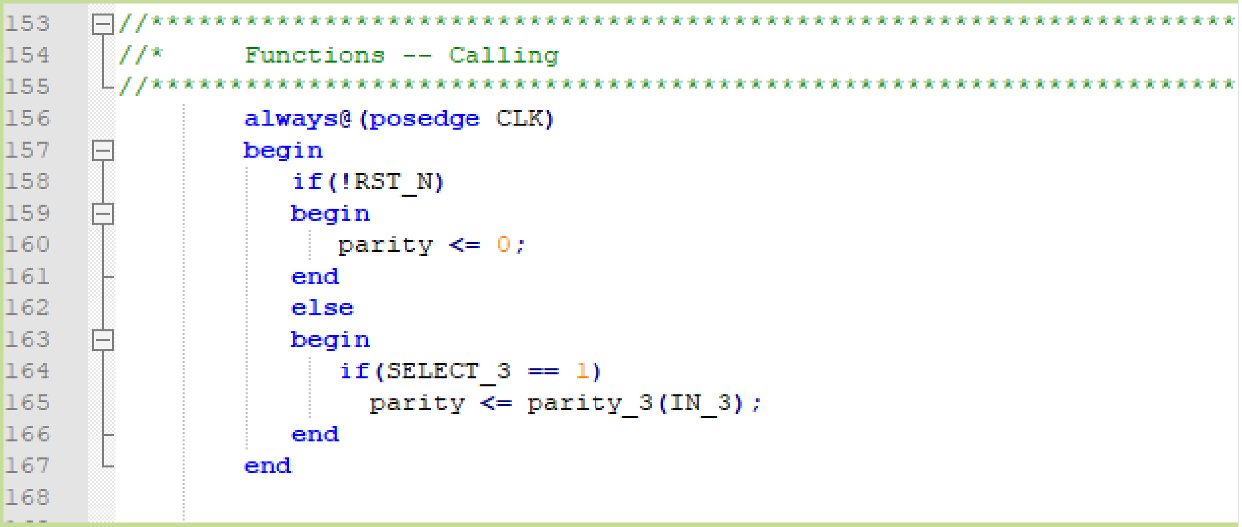


```
// Block Statements Code
    module EPT_10M04_AF_S2_Top
    (

    //Inputs and Outputs for Verilog Block Statements

    input  wire                 CLK,
    input  wire                 RST_N,

    output wire                 OUT_3,
    input wire   [31:0]         IN_3,
    input wire                  SELECT_3,
```



```
75  //**********************************************************************
76  //*      Internal Signals and Registers Declarations
77  //**********************************************************************
78     reg y_nosense;
79     wire data_1;
80     wire data_2;
81     reg  parity;
82
83  //**********************************************************************
84  //*      Internal Signal Assignments
85  //**********************************************************************
86     assign data_2 = 1;
87
88     assign OUT_3 = parity;
```

The OUT_3 signal is assigned the value of parity. Parity is declared as a 'reg'.

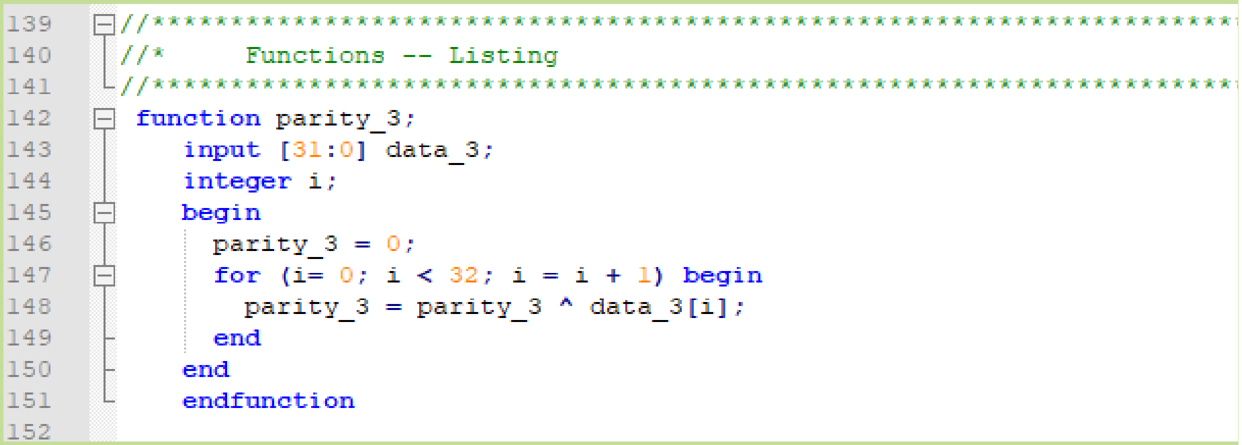


```
153 //**********************************************************************
154 //*      Functions -- Calling
155 //**********************************************************************
156         always@(posedge CLK)
157         begin
158            if(!RST_N)
159            begin
160               parity <= 0;
161            end
162            else
163            begin
164               if(SELECT_3 == 1)
165                 parity <= parity_3(IN_3);
166            end
167         end
168
```

In the function call, the return value of the function is assigned to parity. This return must be of type 'reg'. During the assignment above of OUT_3 to partiy, OUT_3 must be declared as 'wire'. The function call on line 165 takes in one input signal. In this function, the input is a 32 bit vector. The function performs the action, then returns a signal and assigns it to parity. For this function, the return is a single bit. In the function call above for parity_3(), the output assigned signal, parity, is set to zero when RST_N asserts. This is not

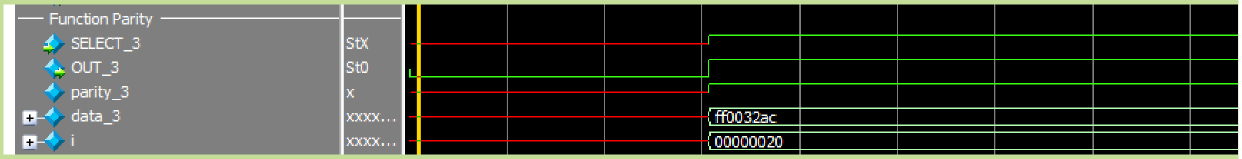
necessary but the signal, parity, will show up as 'Don't Care' in the simulation. This reset assertion will ensure that parity and anything using this signal will have a value during all simulation.

The listing is shown on line 139. This is the real meat of the function, where is does its action.



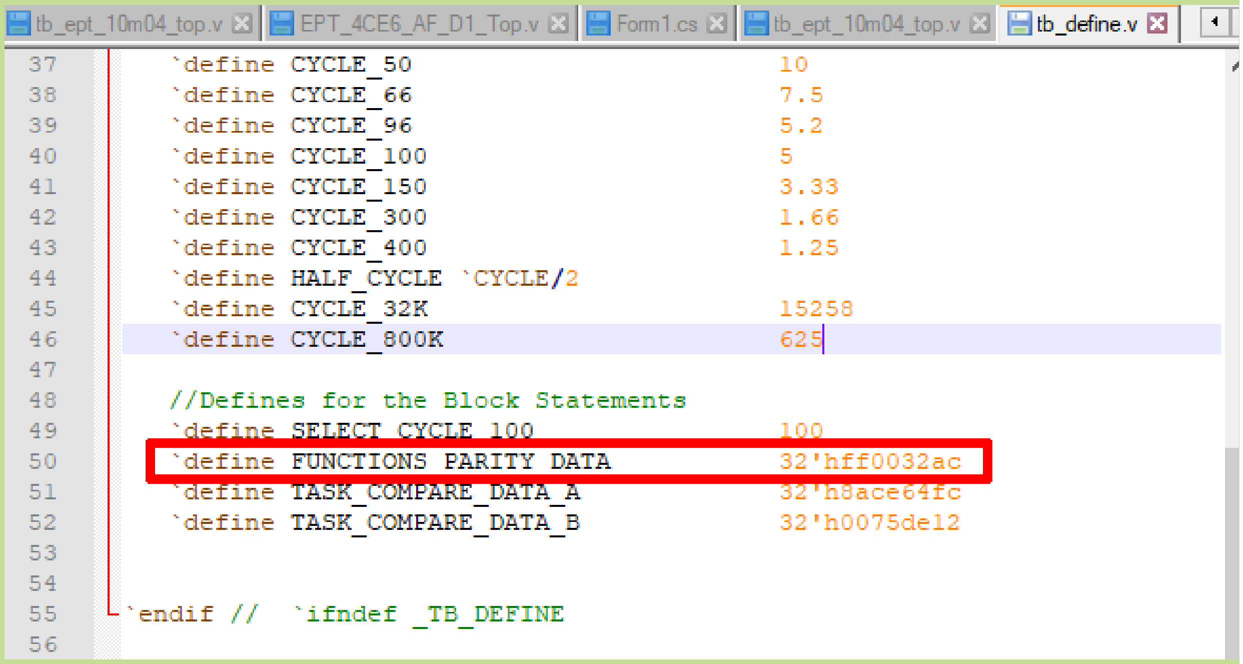
```
139 //**********************************************************************
140 //*      Functions -- Listing
141 //**********************************************************************
142  function parity_3;
143     input [31:0] data_3;
144     integer i;
145     begin
146       parity_3 = 0;
147       for (i= 0; i < 32; i = i + 1) begin
148         parity_3 = parity_3 ^ data_3[i];
149       end
150     end
151     endfunction
152
```

The function is declared with the 'function' keyword. Then the input is declared, which is a 32 bit vector. An internal 'integer' is declared to allow a counter to be used. The for loop always the ability to step through each bit of the 32 bit vector and perform the calculation. And of course 'parity_3' provides the output of the function. So, run the simulation using the do sim_ept_10m04_top.do and examine the results.



The parity_3() function is called when SELECT_3 goes high. SELECT_3 is manipulated by the testbench. The parity_3() function uses a for loop to calculate the result of parity for the IN_3 32 bit word. It happens instantly in the simulation. So, the result, OUT_3 is available immediately. The user is encouraged to experiment with the parity_3() function. Go into the file,

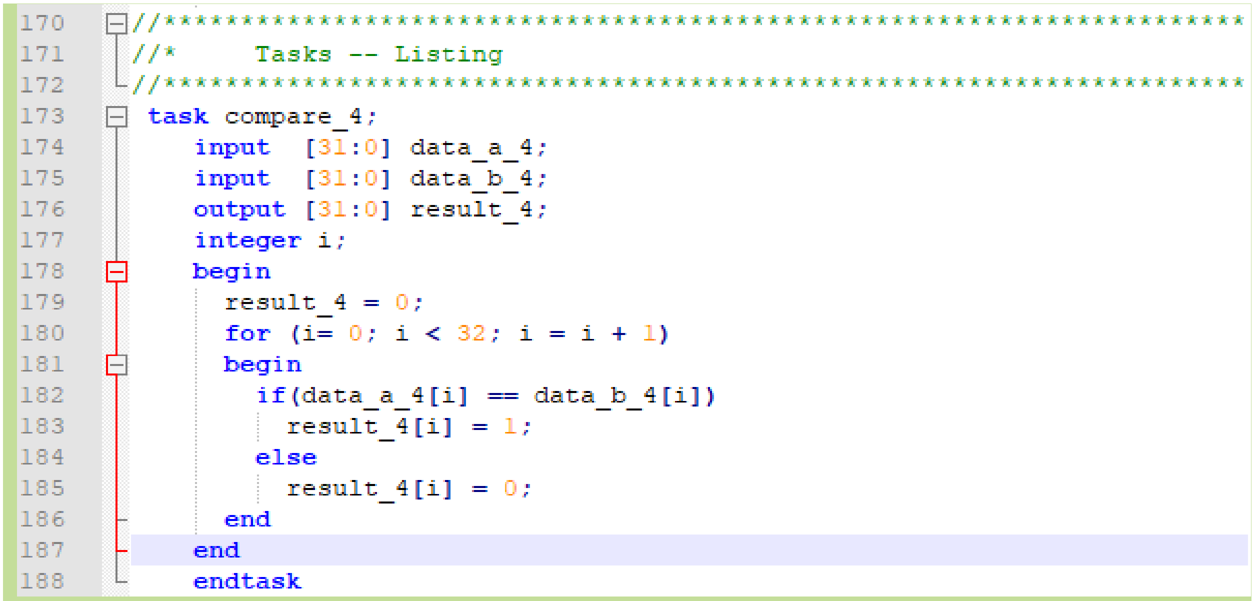
tb_define.v and change the base number, FUNCTIONS_PARITY_DATA. Then, re-run the simulation and examine the results.



```
    `define CYCLE_50                        10
    `define CYCLE_66                        7.5
    `define CYCLE_96                        5.2
    `define CYCLE_100                       5
    `define CYCLE_150                       3.33
    `define CYCLE_300                       1.66
    `define CYCLE_400                       1.25
    `define HALF_CYCLE `CYCLE/2
    `define CYCLE_32K                       15258
    `define CYCLE_800K                      625

    //Defines for the Block Statements
    `define SELECT_CYCLE_100                100
    `define FUNCTIONS_PARITY_DATA           32'hff0032ac
    `define TASK_COMPARE_DATA_A             32'h8ace64fc
    `define TASK_COMPARE_DATA_B             32'h0075de12


`endif //  `ifndef _TB_DEFINE
```

Tasks

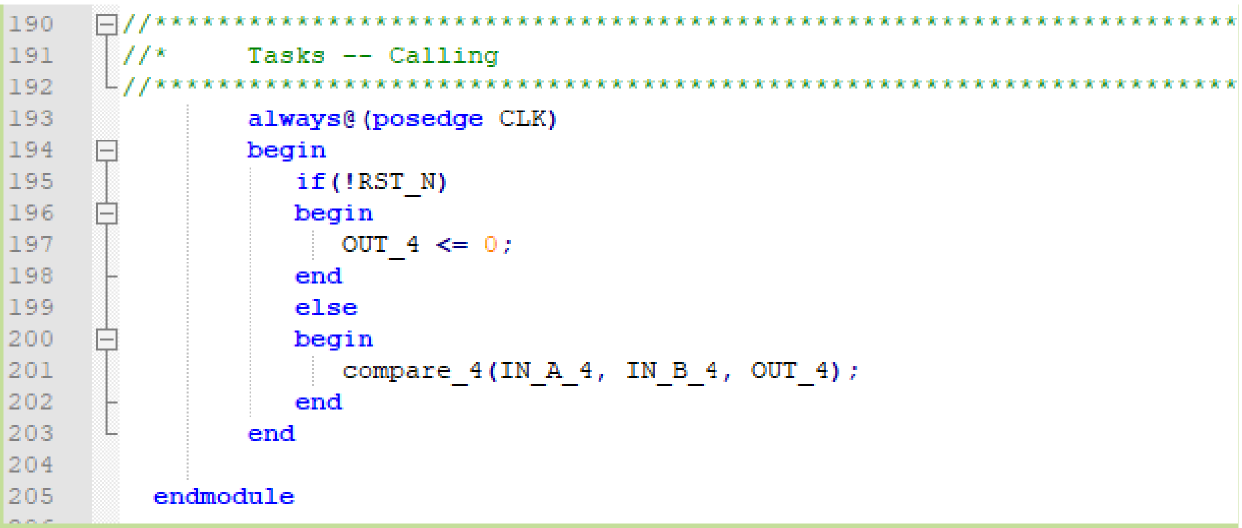
The Tasks guide starts on line 170 of the EPT_10M04_AF_S2_Top.v file. Tasks are declared with the "task" keyword. This keyword starts the function, and it ends with the keyword "endtask".



```
//***********************************************************************
//*      Tasks -- Listing
//***********************************************************************
 task compare_4;
    input  [31:0] data_a_4;
    input  [31:0] data_b_4;
    output [31:0] result_4;
    integer i;
    begin
      result_4 = 0;
      for (i= 0; i < 32; i = i + 1)
      begin
        if(data_a_4[i] == data_b_4[i])
          result_4[i] = 1;
        else
          result_4[i] = 0;
      end
    end
    endtask
```

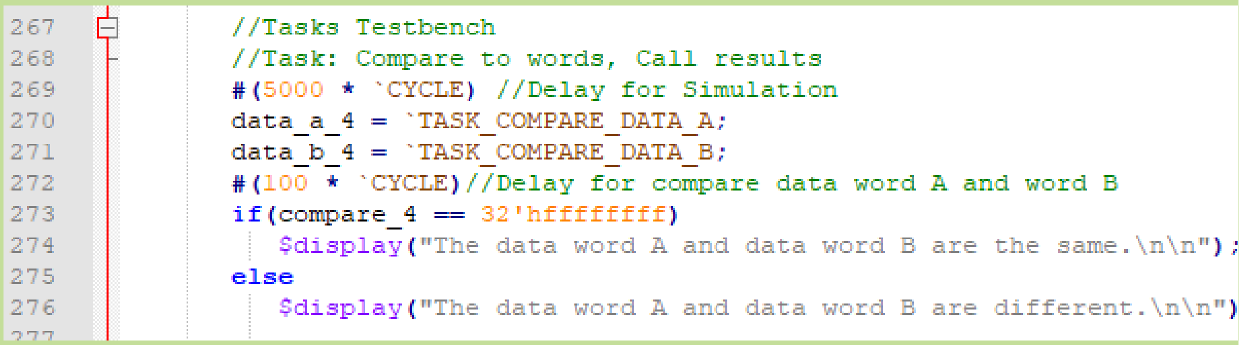
The task can have as many inputs and outputs as needed. The inputs and outputs can be signals or vectors or a mixture of both. The task will only execute when it is called. When all its statements have completed the task returns control back to the testbench.

The task in Lesson 4 will compare two vectors and return the results. The Testbench provides the stimulus for this compare. And it uses the results returned from the task to display to the command window.



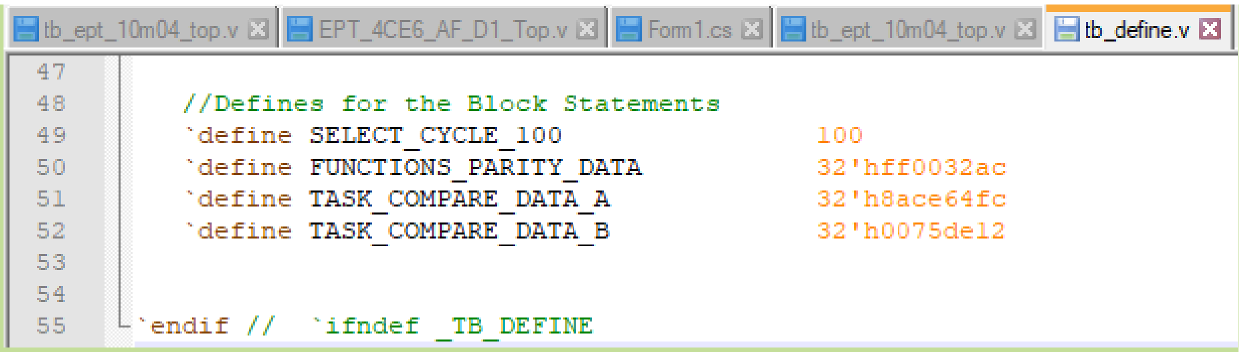
```
//*********************************************************************
//*      Tasks -- Calling
//*********************************************************************
        always@(posedge CLK)
        begin
           if(!RST_N)
           begin
              OUT_4 <= 0;
           end
           else
           begin
              compare_4(IN_A_4, IN_B_4, OUT_4);
           end
        end

  endmodule
```

The testbench provides IN_A_4 and IN_B_4 which are 32 bit vectors to the compare_4() task.



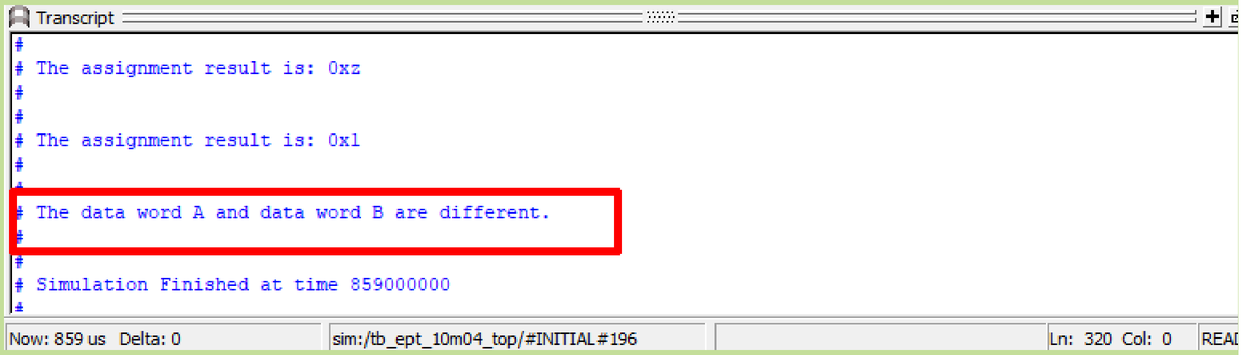
```
        //Tasks Testbench
        //Task: Compare to words, Call results
        #(5000 * `CYCLE) //Delay for Simulation
        data_a_4 = `TASK_COMPARE_DATA_A;
        data_b_4 = `TASK_COMPARE_DATA_B;
        #(100 * `CYCLE)//Delay for compare data word A and word B
        if(compare_4 == 32'hffffffff)
           $display("The data word A and data word B are the same.\n\n");
        else
           $display("The data word A and data word B are different.\n\n")
```

The numbers are provided in tb_define.v



```
    //Defines for the Block Statements
    `define SELECT_CYCLE_100                100
    `define FUNCTIONS_PARITY_DATA           32'hff0032ac
    `define TASK_COMPARE_DATA_A             32'h8ace64fc
    `define TASK_COMPARE_DATA_B             32'h0075de12


`endif //  `ifndef _TB_DEFINE
```

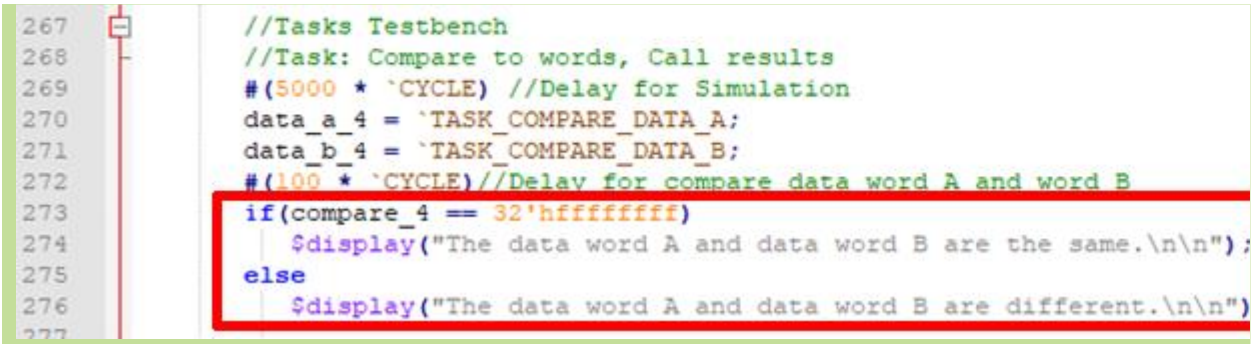
Run the simulation using the do sim_ept_10m04_top.do in the command window. The top level code performs the compare every cycle of CLK. The compare_4() task uses a for loop to count and index, i, from 0 to 31 and compares each bit of the two 32 bit vectors. If the two bits are equal, the resulting 32 bit vector places a 1 in the indexed bit. If they are not equal a 0 is placed in the indexed bit. Using the two numbers provided in tb_define.v the results:



```
#
# The assignment result is: 0xz
#
#
# The assignment result is: 0x1
#
#
# The data word A and data word B are different.
#
#
# Simulation Finished at time 859000000
#
```

```
Now: 859 us  Delta: 0    sim:/tb_ept_10m04_top/#INITIAL#196    Ln: 320  Col: 0  READ
```

The testbench reads the result vector and performs another compare to determine what should be displayed in the command window.



```
//Tasks Testbench
//Task: Compare to words, Call results
#(5000 * `CYCLE) //Delay for Simulation
data_a_4 = `TASK_COMPARE_DATA_A;
data_b_4 = `TASK_COMPARE_DATA_B;
#(100 * `CYCLE)//Delay for compare data word A and word B
if(compare_4 == 32'hffffffff)
    $display("The data word A and data word B are the same.\n\n");
else
    $display("The data word A and data word B are different.\n\n")
```

The user is encouraged to experiment with the compare_4() task. Change the numbers in the tb_define so that the numbers are equal. Re-run the simulation and notice how the results change.

This is the end of Lesson 4

TEST BENCHES

The goal of this Verilog guide is to introduce the user to producing FPGA designs by simulating the design before committing to hardware. So, we can create a simple design such as the d flip flop and add it to the simulator and add the stimulus such as clocks and inputs. Run the simulation and examine the outputs. This is all fairly manageable with hand coding the stimulus. Let's look at a more complex design and the need to provide more complex stimulus and a method to examine the output. This is were Test

Bench comes in. Verilog was designed as a modeling language first. So, it has some very useful tools to make it easy to provide a complex stimulus and methods to easily examine outputs.

Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

White Space

White space can contain the characters for blanks, tabs, newlines, and form feeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

White space characters are :

- Blank spaces
- Tabs
- Carriage returns
- New-line
- Form-feeds

Bad Code : Never write code like this.

```
module addbit(a,b,ci,sum,co);
input a,b,ci;output sum co;
wire a,b,ci,sum,co;endmodule
```

You could download file bad_code.v here

Good Code : Nice way to write code.

```
module addbit (
a,
b,
ci,
sum,
co);
input           a;
input           b;
input           ci;
output         sum;
output         co;
wire            a;
wire            b;
wire            ci;
wire            sum;
wire            co;

endmodule
```

You could download file good_code.v here

Comments

There are two forms to introduce comments.

- Single line comments begin with the token // and end with a carriage return
- Multi line comments begin with the token /* and end with the token */

Examples of Comments

```
/* This is a
  Multi line comment
  example */
module addbit (
a,
b,
ci,
sum,
co);

// Input Ports  Single line comment
input               a;
input               b;
input               ci;
// Output ports
output              sum;
output              co;
// Data Types
wire                a;
wire                b;
wire                ci;
wire                sum;
wire                co;

endmodule
```

You could download file comment.v here

Case Sensitivity

Verilog HDL is case sensitive

- Lower case letters are unique from upper case letters
- All Verilog keywords are lower case

Examples of Unique names

```
input                          // a Verilog Keyword
wire                           // a Verilog Keyword
WIRE                           // a unique name ( not a keyword)
Wire                           // a unique name (not a keyword)
```

You could download file unique_names.v here

NOTE : Never use Verilog keywords as unique names, even if the case is different.

Identifiers

Identifiers are names used to give an object, such as a register or a function or a module, a name so that it can be referenced from other places in a description.

- Identifiers must begin with an alphabetic character or the underscore character (**a-z A-Z** _)
- Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (**a-z A-Z 0-9** _ **$**)
- Identifiers can be up to 1024 characters long.

Examples of legal identifiers

data_input mu

clk_input my$clk

i386 A

Escaped Identifiers

Verilog HDL allows any character to be used in an identifier by escaping the identifier. Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

- Escaped identifiers begin with the back slash (\ )
- Entire identifier is escaped by the back slash.
- Escaped identifier is terminated by white space (Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space)
- Terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

Examples of escape identifiers

Verilog does not allow to identifier to start with a numeric character. So if you really want to use a identifier to start with a numeric value then use a escape character as shown below.

```
// There must be white space after the
// string which uses escape character
module \1dff (
q,          // Q output
\q~ ,       // Q_out output
d,          // D input
cl$k,       // CLOCK input
\reset*  // Reset input
);

```

```
input d, cl$k, \reset* ;
output q, \q~ ;

endmodule
```

You could download file escape_id.v here

Numbers in Verilog

You can specify constant numbers in decimal, hexadecimal, octal, or binary format. Negative numbers are represented in 2's complement form. When used in a number, the question mark (?) character is the Verilog alternative for the z character. The underscore character (_) is legal anywhere in a number except as the first character, where it is ignored.

Integer Numbers

Verilog HDL allows integer numbers to be specified as

- Sized or unsized numbers (Unsized size is 32 bits)
- In a radix of binary, octal, decimal, or hexadecimal
- Radix and hex digits (a,b,c,d,e,f) are case insensitive
- Spaces are allowed between the size, radix and value

Syntax: <size>'<radix><value>;

Example of Integer Numbers

Integer **Stored as**

1	00000000000000000000000000000001
8'hAA	10101010
6'b10_0011	100011
'hF	00000000000000000000000000001111

Verilog expands <value> filling the specified <size> by working from right-to-left

- When <size> is smaller than <value>, then leftmost bits of <value> are truncated
- When <size> is larger than <value>, then leftmost bits are filled, based on the value of the leftmost bit in <value>.
    - Leftmost '0' or '1' are filled with '0'
    - Leftmost 'Z' are filled with 'Z'
    - Leftmost 'X' are filled with 'X'

Note : X Stands for unknown and Z stands for high impedance, 1 for logic high or 1 and 0 for logic low or 0.

Example of Integer Numbers

Integer	**Stored as**
6'hCA	001010
6'hA	001010

16'bZ	ZZZZZZZZZZZZZZZZ
8'bx	xxxxxxxx

Real Numbers

- Verilog supports real constants and variables
- Verilog converts real numbers to integers by rounding
- Real Numbers can not contain 'Z' and 'X'
- Real numbers may be specified in either decimal or scientific notation
- < value >.< value >
- < mantissa >E< exponent >
- Real numbers are rounded off to the nearest integer when assigning to an integer.

Example of Real Numbers

Real Number	Decimal notation
1.2	1.2
0.6	0.6
3.5E6	3,500000.0

Signed and Unsigned Numbers

Verilog Supports both types of numbers, but with certain restrictions. Like in C language we don't have int and unint types to say if a number is signed integer or unsigned integer.

Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus they become signed numbers. Verilog internally represents negative numbers in 2's complement format. An optional signed specifier can be added for signed arithmetic.

Examples

Number	**Description**
32'hDEAD_BEEF	Unsigned or signed positive number
-14'h1234	Signed negative number

The example file below shows how Verilog treats signed and unsigned numbers.

```
module signed_number;
```

```

reg [31:0]  a;

initial begin
  a = 14'h1234;
  $display ("Current Value of a = %h", a);
  a = -14'h1234;
  $display ("Current Value of a = %h", a);
  a = 32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  a = -32'hDEAD_BEEF;
  $display ("Current Value of a = %h", a);
  #10  $finish;
end

endmodule
```

You could download file signed_number.v here

```
Current Value of a = 00001234
Current Value of a = ffffedcc
Current Value of a = deadbeef
Current Value of a = 21524111
```

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators.

When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column together with the escape sequence that represents the character in the left-hand column.

Special Characters in Strings

Character	**Description**
\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Example

```
//-----------------------------------------------------
// Design Name : strings
// File Name   : strings.v
// Function    : This program shows how string
//               can be stored in reg
// Coder�      : Deepak Kumar Tala
//-----------------------------------------------------
module strings();
// Declare a register variable that is 21 bytes
reg [8*21:0] string ;

initial begin
  string = "This is sample string";
  $display ("%s \n", string);
end

endmodule
```

You could download file strings.v here

Logic Values and signal Strengths

The Verilog HDL has got four logic values

Logic Value	Description
0	zero, low, false
1	one, high, true
z or **Z**	high impedance, floating
x or **X**	unknown, uninitialized, contention

Gate and Switch delays

In real circuits, logic gates have delays associated with them. Verilog provides the mechanism to associate delays with gates.

- Rise, Fall and Turn-off delays.
- Minimal, Typical, and Maximum delays.

In Verilog delays can be introduced with #'num' as in the examples below, where # is a special character to introduce delay, and 'num' is the number of ticks simulator should delay current statement execution.

- #1 a = b : Delay by 1, i.e. execute after 1 tick
- #2 not (a,b) : Delay by 2 all assignments made to a.

Real transistors have resolution delays between the input and output. This is modeled in Verilog by specifying one or more delays for the rise, fall, turn-on and turn off time seperated by commas.

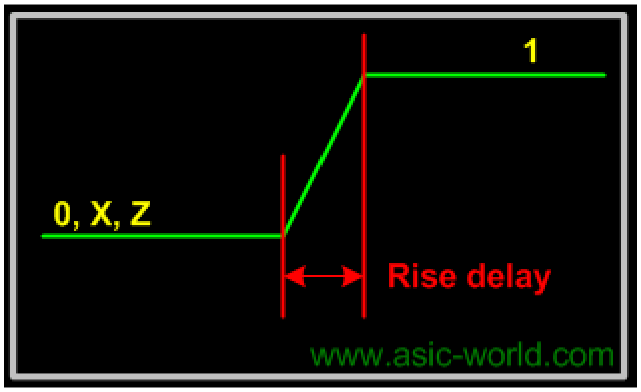
Syntax: keyword #(delay{s}) unique_name (node specifications);

Switch element	Number Of Delays	Specified delays

Switch	1	Rise, fall and turn-off times of equal length
	2	Rise and fall times
	3	Rise, fall and turn off
(r)tranif0, (r)tranif1	1	both turn on and turn off
	2	turn on, turn off
(r)tran	0	None allowed

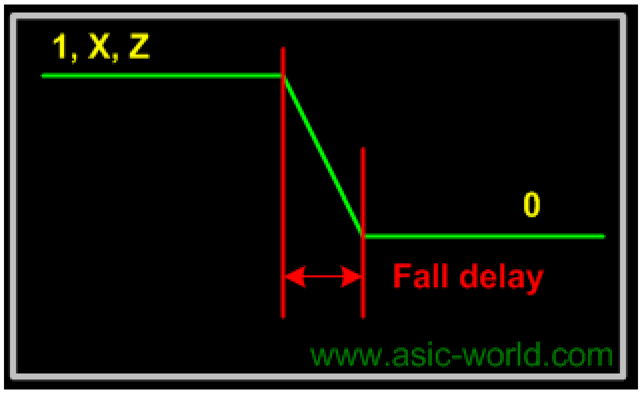
Rise Delay

The rise delay is associated with a gate output transition to 1 from another value (0, x, z).



Fall Delay

The fall delay is associated with a gate output transition to 0 from another value (1, x, z).



Turn-off Delay

The Turn-off delay is associated with a gate output transition to z from another value (0, 1, x).

Min Value

The min value is the minimum delay value that the gate is expected to have.

Typ Value

The typ value is the typical delay value that the gate is expected to have.

Max Value

The max value is the maximum delay value that the gate is expected to have.

Example

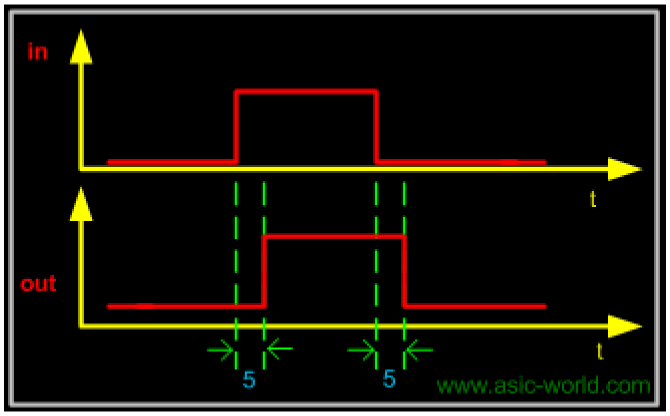
Below are some examples to show the usage of delays.

Example - Single Delay

```
module  buf_gate ();
reg in;
wire out;

buf #(5) (out,in);

initial begin
  $monitor ("Time = %g in = %b out=%b", $time, in, out);
  in = 0;
  #10 in = 1;
  #10 in = 0;
  #10 $finish;
end

endmodule
```

You could download file buf_gate.v here

```
Time = 0 in = 0 out=x
Time = 5 in = 0 out=0
Time = 10 in = 1 out=0
Time = 15 in = 1 out=1
Time = 20 in = 0 out=1
Time = 25 in = 0 out=0
```



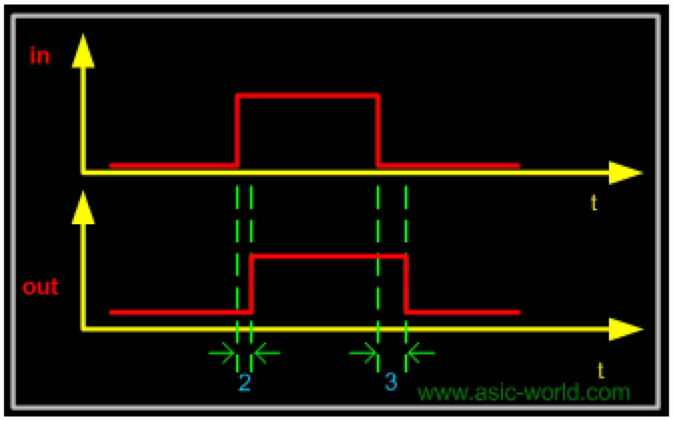
Example - Two Delays

```
module  buf_gate1 ();
reg in;
wire out;

buf #(2,3) (out,in);

initial begin
  $monitor ("Time = %g in = %b out=%b", $time, in, out);
  in = 0;
  #10 in = 1;
  #10 in = 0;
  #10 $finish;
end

```

```
endmodule
```

You could download file buf_gate1.v here

```
Time = 0 in = 0 out=x
 Time = 3 in = 0 out=0
 Time = 10 in = 1 out=0
 Time = 12 in = 1 out=1
 Time = 20 in = 0 out=1
 Time = 23 in = 0 out=0
```

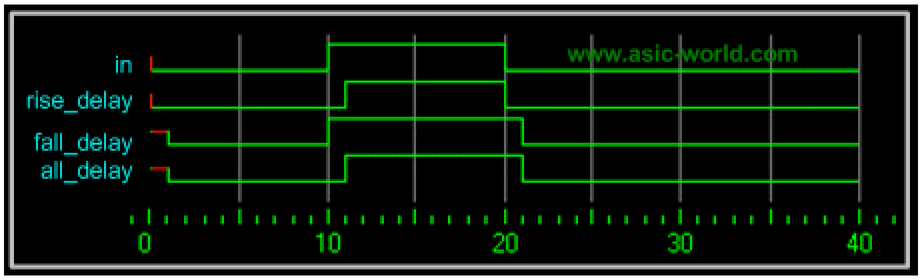


Example - All Delays

```
module delay();
 reg in;
 wire rise_delay, fall_delay, all_delay;

 initial begin
  $monitor (
     "Time=%g in=%b rise_delay=%b fall_delay=%b all_delay=%b",
     $time, in, rise_delay, fall_delay, all_delay);
  in = 0;
  #10  in = 1;
  #10  in = 0;
  #20  $finish;
 end

 buf #(1,0)U_rise (rise_delay,in);
 buf #(0,1)U_fall (fall_delay,in);
 buf #1  U_all (all_delay,in);

endmodule
```

You could download file delay.v here

```
Time = 0 in = 0 rise_delay = 0 fall_delay = x all_delay = x
Time = 1 in = 0 rise_delay = 0 fall_delay = 0 all_delay = 0
Time = 10 in = 1 rise_delay = 0 fall_delay = 1 all_delay = 0
Time = 11 in = 1 rise_delay = 1 fall_delay = 1 all_delay = 1
Time = 20 in = 0 rise_delay = 0 fall_delay = 1 all_delay = 1
Time = 21 in = 0 rise_delay = 0 fall_delay = 0 all_delay = 0
```



Example - Complex Example

```
module delay_example();

wire out1,out2,out3,out4,out5,out6;
reg b,c;

// Delay for all transitions
or      #5                 u_or     (out1,b,c);
// Rise and fall delay
and     #(1,2)             u_and    (out2,b,c);
// Rise, fall and turn off delay
nor     #(1,2,3)           u_nor    (out3,b,c);
//One Delay, min, typ and max
nand    #(1:2:3)           u_nand   (out4,b,c);
//Two delays, min,typ and max
buf     #(1:4:8,4:5:6)     u_buf    (out5,b);
//Three delays, min, typ, and max
notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (out6,b,c);

//Testbench code
initial  begin
   $monitor (
   "Time=%g b=%b c=%b  out1=%b out2=%b out3=%b out4=%b out5=%b out6=%b",
     $time, b, c , out1, out2, out3, out4, out5, out6);
   b = 0;
   c = 0;
   #10  b = 1;
   #10  c = 1;
   #10  b = 0;
   #10  $finish;
end

endmodule
```

You could download file delay_example.v here

```
Time = 0 b = 0 c=0  out1=x out2=x out3=x out4=x out5=x out6=x
 Time = 1 b = 0 c=0  out1=x out2=x out3=1 out4=x out5=x out6=x
 Time = 2 b = 0 c=0  out1=x out2=0 out3=1 out4=1 out5=x out6=z
 Time = 5 b = 0 c=0  out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
 Time = 8 b = 0 c=0  out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
 Time = 10 b = 1 c=0  out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
 Time = 12 b = 1 c=0  out1=0 out2=0 out3=0 out4=1 out5=0 out6=z
 Time = 14 b = 1 c=0  out1=0 out2=0 out3=0 out4=1 out5=1 out6=z
 Time = 15 b = 1 c=0  out1=1 out2=0 out3=0 out4=1 out5=1 out6=z
 Time = 20 b = 1 c=1  out1=1 out2=0 out3=0 out4=1 out5=1 out6=z
 Time = 21 b = 1 c=1  out1=1 out2=1 out3=0 out4=1 out5=1 out6=z
 Time = 22 b = 1 c=1  out1=1 out2=1 out3=0 out4=0 out5=1 out6=z
 Time = 25 b = 1 c=1  out1=1 out2=1 out3=0 out4=0 out5=1 out6=0
 Time = 30 b = 0 c=1  out1=1 out2=1 out3=0 out4=0 out5=1 out6=0
 Time = 32 b = 0 c=1  out1=1 out2=0 out3=0 out4=1 out5=1 out6=1
 Time = 35 b = 0 c=1  out1=1 out2=0 out3=0 out4=1 out5=0 out6=1
```

Blocking and Nonblocking assignment

Blocking assignments are executed in the order they are coded, hence they are sequential. Since they block the execution of next statement, till the current statement is executed, they are called blocking assignments. Assignment are made with "=" symbol. Example a = b;

Nonblocking assignments are executed in parallel. Since the execution of next statement is not blocked due to execution of current statement, they are called nonblocking statement. Assignments are made with "<=" symbol. Example a <= b;

Note : Correct way to spell 'nonblocking' is 'nonblocking' and not 'non-blocking'.

Example - blocking and nonblocking

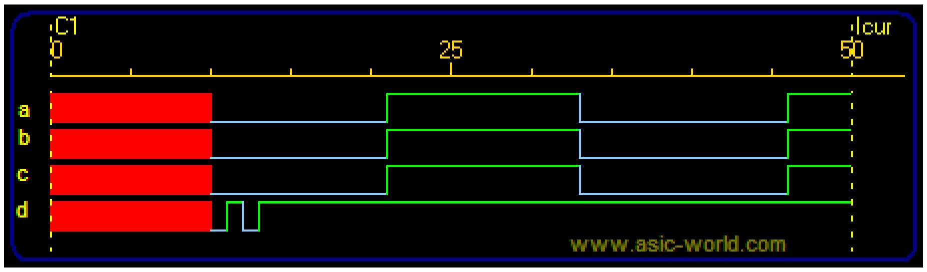
```
module blocking_nonblocking();

reg a,b,c,d;
// Blocking Assignment
initial begin
   #10  a = 0;
   #11  a = 1;
   #12  a = 0;
   #13  a = 1;
end

initial begin
   #10  b <= 0;
   #11  b <= 1;
   #12  b <= 0;
   #13  b <= 1;
end

initial begin
    c = #10 0;
    c = #11 1;
    c = #12 0;
    c = #13 1;
end

initial begin
    d  <= #10  0;
    d  <= #11  1;
    d  <= #12  0;
    d  <= #13  1;
end

initial begin
   $monitor("TIME = %g A = %b B = %b C = %b D = %b",$time, a, b, c, d);
   #50  $finish;
end

endmodule
```

You could download file blocking_nonblocking.v here

Simulator Output

```
TIME = 0 A = x B = x C = x D = x
 TIME = 10 A = 0 B = 0 C = 0 D = 0
 TIME = 11 A = 0 B = 0 C = 0 D = 1
 TIME = 12 A = 0 B = 0 C = 0 D = 0
 TIME = 13 A = 0 B = 0 C = 0 D = 1
 TIME = 21 A = 1 B = 1 C = 1 D = 1
 TIME = 33 A = 0 B = 0 C = 0 D = 1
 TIME = 46 A = 1 B = 1 C = 1 D = 1
```

Waveform



The Conditional Statement if-else

It's known fact that priority implementation takes more logic to implement than parallel implementation. So if you know the inputs are mutually exclusive, then you can code the logic in parallel if.

```
module parallel_if();

reg [3:0] counter;
wire clk,reset,enable, up_en, down_en;

always @ (posedge clk)
// If reset is asserted
if (reset == 1'b0) begin
   counter <= 4'b0000;
end else begin
  // If counter is enable and up count is mode
  if (enable == 1'b1 && up_en == 1'b1) begin
    counter <= counter + 1'b1;
  end
  // If counter is enable and down count is mode
  if (enable == 1'b1 && down_en == 1'b1) begin
    counter <= counter - 1'b1;
  end
end

endmodule
```

You could download file parallel_if.v here

The Case Statement

The case statement compares an expression to a series of cases and executes the statement or statement group associated with the first matching case:

- case statement supports single or multiple statements.
- Group multiple statements using begin and end keywords.

Syntax of a case statement look as shown below.

case ()

< case1 > : < statement >

< case2 > : < statement >

.....

default : < statement >

endcase

Normal Case

Example- case

```
module mux (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;

```

```
reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  default : $display("Error in SEL");
endcase

endmodule
```

You could download file mux.v here

Example- case without default

```
module mux_without_default (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;

reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
  2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display("Error in SEL");
endcase

endmodule
```

You could download file mux_without_default.v here

The example above shows how to specify multiple case items as a single case item.

The Verilog case statement does an identity comparison (like the === operator); one can use the case statement to check for logic x and z values as shown in the example below.

Example- case with x and z

```
module case_xz(enable);
input enable;

always @ (enable)
case(enable)
  1'bz : $display ("enable is floating");
  1'bx : $display ("enable is unknown");
  default : $display ("enable is %b",enable);
endcase

endmodule
```

You could download file case_xz.v here

The casez and casex statement

Special versions of the case statement allow the x ad z logic values to be used as "don't care":

- casez : Treats z as don't care.
- casex : Treats x and z as don't care.

Example- casez

```
module casez_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;

always @ (opcode or a or b or c)
casez(opcode)
  4'b1zzx : begin  // Don't care about lower 2:1 bit, bit 0 match with x
              out = a;
              $display("@%0dns 4'b1zzx is selected, opcode %b",$time,opcode);
            end
  4'b01?? : begin
              out = b;  // bit 1:0 is don't care
              $display("@%0dns 4'b01?? is selected, opcode %b",$time,opcode);
            end
  4'b001? : begin   // bit 0 is don't care
              out = c;
              $display("@%0dns 4'b001? is selected, opcode %b",$time,opcode);
            end
  default : begin
              $display("@%0dns default is selected, opcode %b",$time,opcode);
            end
endcase

// Testbench code goes here
always #2  a = $random;
always #2  b = $random;
always #2  c = $random;

initial  begin
   opcode = 0;
   #2  opcode = 4'b101x;
   #2  opcode = 4'b0101;
   #2  opcode = 4'b0010;
   #2  opcode = 4'b0000;
   #2  $finish;
end

endmodule
```

You could download file casez_example.v here

Simulation Output - casez

```
@0ns default is selected, opcode 0000
 @2ns 4'b1zzx is selected, opcode 101x
 @4ns 4'b01?? is selected, opcode 0101
 @6ns 4'b001? is selected, opcode 0010
 @8ns default is selected, opcode 0000
```

Example- casex

```
module casex_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;

always @ (opcode or a or b or c)
```

```
casex (opcode)
   4'b1zzx : begin  // Don't care  2:0 bits
                  out = a;
                  $display("@%0dns 4'b1zzx is selected, opcode %b",$time,opcode);
               end
   4'b01?? : begin  // bit 1:0 is don't care
                  out = b;
                  $display("@%0dns 4'b01?? is selected, opcode %b",$time,opcode);
               end
   4'b001? : begin  // bit 0 is don't care
                  out = c;
                  $display("@%0dns 4'b001? is selected, opcode %b",$time,opcode);
               end
   default : begin
                  $display("@%0dns default is selected, opcode %b",$time,opcode);
               end
endcase

// Testbench code goes here
always  #2  a = $random;
always  #2  b = $random;
always  #2  c = $random;

initial  begin
   opcode = 0;
   #2  opcode = 4'b101x;
   #2  opcode = 4'b0101;
   #2  opcode = 4'b0010;
   #2  opcode = 4'b0000;
   #2  $finish;
end

endmodule
```

You could download file casex_example.v here

Simulation Output - casex

```
@0ns default is selected, opcode 0000
 @2ns 4'b1zzx is selected, opcode 101x
 @4ns 4'b01?? is selected, opcode 0101
 @6ns 4'b001? is selected, opcode 0010
 @8ns default is selected, opcode 0000
```

Example- Comparing case, casex, casez

```
module case_compare;

reg sel;

initial begin
  #1 $display ("\n   Driving 0");
  sel = 0;
  #1 $display ("\n   Driving 1");
  sel = 1;
  #1 $display ("\n   Driving x");
  sel = 1'bx;
  #1 $display ("\n   Driving z");
  sel = 1'bz;
  #1 $finish;
end

always @ (sel)
case (sel)
  1'b0 : $display("Normal : Logic 0 on sel");
  1'b1 : $display("Normal : Logic 1 on sel");
  1'bx : $display("Normal : Logic x on sel");
  1'bz : $display("Normal : Logic z on sel");
endcase

always @ (sel)
casex (sel)
```

```
  1'b0 : $display("CASEX  : Logic 0 on sel");
  1'b1 : $display("CASEX  : Logic 1 on sel");
  1'bx : $display("CASEX  : Logic x on sel");
  1'bz : $display("CASEX  : Logic z on sel");
endcase

always @ (sel)
casez (sel)
  1'b0 : $display("CASEZ  : Logic 0 on sel");
  1'b1 : $display("CASEZ  : Logic 1 on sel");
  1'bx : $display("CASEZ  : Logic x on sel");
  1'bz : $display("CASEZ  : Logic z on sel");
endcase

endmodule
```

You could download file case_compare.v here

Simulation Output

```
Driving 0
 Normal : Logic 0 on sel
 CASEX  : Logic 0 on sel
 CASEZ  : Logic 0 on sel

     Driving 1
 Normal : Logic 1 on sel
 CASEX  : Logic 1 on sel
 CASEZ  : Logic 1 on sel

     Driving x
 Normal : Logic x on sel
 CASEX  : Logic 0 on sel
 CASEZ  : Logic x on sel

     Driving z
 Normal : Logic z on sel
 CASEX  : Logic 0 on sel
```

CASEZ : Logic 0 on sel

Looping Statements

Looping statements appear inside procedural blocks only; Verilog has four looping statements like any other programming language.

- forever
- repeat
- while
- for

The forever statement

The forever loop executes continually, the loop never ends. Normally we use forever statements in initial blocks.

syntax : forever < statement >

One should be very careful in using a forever statement: if no timing construct is present in the forever statement, simulation could hang. The code below is one such application, where a timing construct is included inside a forever statement.

Example - Free running clock generator

```
module forever_example ();

reg clk;

initial begin
  #1 clk = 0;
  forever begin
    #5 clk = !clk;
  end
end

initial begin
  $monitor ("Time = %d  clk = %b",$time, clk);
  #100 $finish;
end

endmodule
```

You could download file forever_example.v here

The repeat statement

The repeat loop executes < statement > a fixed < number > of times.

syntax : repeat (< number >) < statement >

Example- repeat

```
module repeat_example();
reg  [3:0] opcode;
reg  [15:0] data;
reg        temp;

always @ (opcode or data)
begin
  if (opcode == 10) begin
    // Perform rotate
    repeat (8) begin
      #1 temp = data[15];
      data = data << 1;
      data[0] = temp;
    end
  end
end
// Simple test code
initial begin
   $display (" TEMP  DATA");
   $monitor (" %b   %b ",temp, data);
   #1 data = 18'hF0;
   #1 opcode = 10;
   #10 opcode = 0;
   #1 $finish;
end

endmodule
```

You could download file repeat_example.v here

The while loop statement

The while loop executes as long as an < expression > evaluates as true. This is the same as in any other programming language.

syntax : while (< expression >) < statement >

Example- while

```
module while_example();

reg [5:0] loc;
reg [7:0] data;

always @ (data or loc)
begin
  loc = 0;
  // If Data is 0, then loc is 32 (invalid value)
  if (data == 0) begin
    loc = 32;
  end else begin
    while (data[0] == 0) begin
      loc = loc + 1;
      data = data >> 1;
      end
    end
    $display ("DATA = %b  LOCATION = %d",data,loc);
end

initial begin
   #1 data = 8'b11;
   #1 data = 8'b100;
   #1 data = 8'b1000;
   #1 data = 8'b1000_0000;
   #1 data = 8'b0;
   #1 $finish;
end

endmodule
```

You could download file while_example.v here

The for loop statement

The for loop is the same as the for loop used in any other programming language.

- Executes an < initial assignment > once at the start of the loop.
- Executes the loop as long as an < expression > evaluates as true.
- Executes a < step assignment > at the end of each pass through the loop.

syntax : for (< initial assignment >; < expression >, < step assignment >) < statement >

Note : verilog does not have ++ operator as in the case of C language.

Example - For

```
module for_example();

integer i;
reg [7:0] ram [0:255];

initial begin
  for (i = 0; i < 256; i = i + 1) begin
    #1 $display(" Address = %g  Data = %h",i,ram[i]);
    ram[i] <= 0; // Initialize the RAM with 0
    #1 $display(" Address = %g  Data = %h",i,ram[i]);
  end
  #1 $finish;
end

endmodule
```

You could download file for_example.v here

////////////////////////////////////Lesson #5///

In this lesson, let's explore the Test Benches.

Go to the xx_Project_xx_DVD-> Verilog Getting Started->Tutorials_HDL and copy the Lesson 5 HDL folder to the users local drive. Go through the initial steps as outlined in Lesson 1. Those initial steps open ModelSim, Change Directory to the ModelSim Folder, Compile the source file, then start the simulation using the "do sim_ept_10m04_top.do"

This lesson will to into depth on Test Benches. It provides a Testbench and user code. The Testbench will exercise each item and display the results to the log window of ModelSim. The user code is organized as a module it includes non-synthesizable code that can only be used in a testbench. There are three files used in Lesson 5.

- **EPT_10M04_AF_S2_Top.v**
- **tb_ept_10m04_top.v**
- **tb_define.v**

The user code:

The Testbench code contains the stimulus for the user code:

The Always Block are used to describe events that should happen under certain conditions. The Lesson 4 Always Block is explored using the various instantiations on line 92 of the EPT_10M04_AF_S2_Top.v file.

Sequential Logic using Procedural Coding

To model sequential logic, a procedure block must be sensitive to positive edge or negative edge of clock. To model asynchronous reset, procedure block must be sensitive to both clock and reset. All the assignments to sequential logic should be made through nonblocking assignments.

Sometimes it's tempting to have multiple edge triggering variables in the sensitive list: this is fine for simulation. But for synthesis this does not make sense, as in real life, flip-flop can have only one clock, one reset and one preset (i.e. posedge clk or posedge reset or posedge preset).

One common mistake the new beginner makes is using clock as the enable input to flip-flop. This is fine for simulation, but for synthesis, this is not right.

Example - Bad coding - Using two clocks

```
module wrong_seq();
```

```

reg q;
reg clk1, clk2, d1, d2;

always @ (posedge clk1 or posedge clk2)
if (clk1) begin
  q <= d1;
end else if (clk2) begin
  q <= d2;
end

initial begin
  $monitor ("CLK1 = %b CLK2 = %b D1 = %b D2 %b Q = %b",
    clk1, clk2, d1, d2, q);
  clk1 = 0;
  clk2 = 0;
  d1 = 0;
  d2 = 1;
  #10 $finish;
end

always
 #1 clk1 = ~clk1;

always
 #1.9 clk2 = ~clk2;

endmodule
```

You could download file wrong_seq.v here

Example - D Flip-flop with async reset and async preset

```
module dff_async_reset_async_preset();

reg clk,reset,preset,d;
reg  q;

always @ (posedge clk or posedge reset or posedge preset)
```

```
if (reset) begin
  q <= 0;
end else if (preset) begin
  q <= 1;
end else begin
  q <= d;
end

// Testbench code here
initial begin
  $monitor("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",
    clk,reset,preset,d,q);
  clk    =0;
  #1 reset  =0;
  preset=0;
  d      =0;
  #1 reset=1;
  #2 reset=0;
  #2 preset=1;
  #2 preset=0;
  repeat (4) begin
    #2 d      =~d;
  end
  #2 $finish;
end

always
 #1 clk=~clk;

endmodule
```

You could download file dff_async_reset_async_preset.v here

Example - D Flip-flop with sync reset and sync preset

```
module dff_sync_reset_sync_preset();

reg clk,reset,preset,d;
reg   q;

always @ (posedge clk)
if (reset) begin
  q <= 0;
end else if (preset) begin
  q <= 1;
end else begin
  q <= d;
end

// Testbench code here
initial begin
  $monitor("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",
    clk,reset,preset,d,q);
  clk    = 0;
  #1 reset  = 0;
  preset = 0;
  d      = 0;
  #1 reset = 1;
  #2 reset = 0;
  #2 preset = 1;
  #2 preset = 0;
  repeat (4) begin
    #2 d     = ~d;
  end
  #2 $finish;
end

always
 #1 clk = ~clk;

endmodule
```

You could download file dff_sync_reset_sync_preset.v here

A procedure can't trigger itself

One cannot trigger the block with a variable that block assigns value or drives.

```
module trigger_itself();

reg clk;

always @ (clk)
  #5 clk = !clk;

// Testbench code here
initial begin
  $monitor("TIME = %d  CLK = %b",$time,clk);
  clk = 0;
  #500 $display("TIME = %d  CLK = %b",$time,clk);
  $finish;
end

endmodule
```

You could download file trigger_itself.v here

Procedural Block Concurrency

If we have multiple always blocks inside one module, then all the blocks (i.e. all the always blocks and initial blocks) will start executing at time 0 and will continue to execute concurrently. Sometimes this leads to race conditions, if coding is not done properly.

```
module multiple_blocks ();
reg a,b;
reg c,d;
reg clk,reset;
// Combo Logic
always @ ( c)
begin
  a = c;
```

```
end
// Seq Logic
always @ (posedge clk)
if (reset) begin
  b <= 0;
end else begin
  b <= a & d;
end

// Testbench code here
initial begin
  $monitor ("TIME = %d CLK = %b C = %b D = %b A = %b B = %b",
    $time, clk,c,d,a,b);
  clk = 0;
  reset = 0;
  c = 0;
  d = 0;
  #2 reset = 1;
  #2 reset = 0;
  #2 c = 1;
  #2 d = 1;
  #2 c = 0;
  #5 $finish;
end
// Clock generator
always
 #1 clk = ~clk;

endmodule
```

You could download file multiple_blocks.v here

Race condition

```
module race_condition();
reg b;

initial begin
```

```
  b = 0;
end

initial  begin
  b = 1;
end

endmodule
```

You could download file race_condition.v here

In the code above it is difficult to say the value of b, as both blocks are supposed to execute at same time. In Verilog, if care is not taken, a race condition is something that occurs very often.

Named Blocks

Blocks can be named by adding : block_name after the keyword begin. Named blocks can be disabled using the 'disable' statement.

Example - Named Blocks

```
// This code find the lowest bit set
module named_block_disable();

reg [31:0] bit_detect;
reg [5:0]  bit_position;
integer i;

always @ (bit_detect)
```

```
begin : BIT_DETECT
  for (i = 0; i < 32 ; i = i + 1) begin
     // If bit is set, latch the bit position
     // Disable the execution of the block
     if (bit_detect[i] == 1) begin
        bit_position = i;
        disable BIT_DETECT;
     end  else begin
        bit_position = 32;
     end
  end
end

// Testbench code here
initial begin
  $monitor(" INPUT = %b  MIN_POSITION = %d", bit_detect, bit_position);
  #1 bit_detect = 32'h1000_1000;
  #1 bit_detect = 32'h1100_0000;
  #1 bit_detect = 32'h1000_1010;
  #10 $finish;
end

endmodule
```

You could download file named_block_disable.v here

In the example above, BIT_DETECT is the named block and it is disabled whenever the bit position is detected.

Procedural blocks and timing controls.

- Delay controls.
- Edge-Sensitive Event controls.
- Level-Sensitive Event controls-Wait statements.
- Named Events.

Delay Controls

Delays the execution of a procedural statement by specific simulation time.

#< time > < statement >;

Example - clk_gen

```
module clk_gen ();

reg clk, reset;

initial begin
  $monitor ("TIME = %g RESET = %b CLOCK = %b", $time, reset, clk);
  clk = 0;
  reset = 0;
  #2  reset = 1;
  #5  reset = 0;
  #10  $finish;
end

always
  #1  clk = ! clk;

endmodule
```

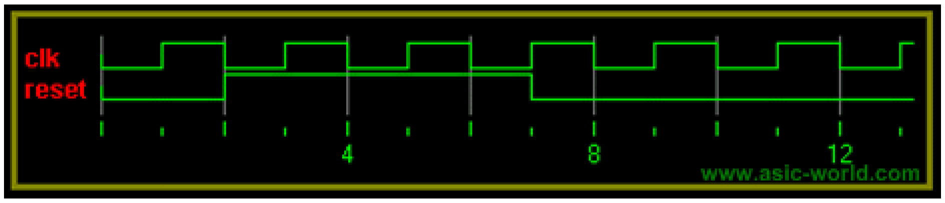
You could download file clk_gen.v here

Simulation Output

```
TIME = 0  RESET = 0 CLOCK = 0
```

```
 TIME = 1  RESET = 0 CLOCK = 1
 TIME = 2  RESET = 1 CLOCK = 0
 TIME = 3  RESET = 1 CLOCK = 1
 TIME = 4  RESET = 1 CLOCK = 0
 TIME = 5  RESET = 1 CLOCK = 1
 TIME = 6  RESET = 1 CLOCK = 0
 TIME = 7  RESET = 0 CLOCK = 1
 TIME = 8  RESET = 0 CLOCK = 0
 TIME = 9  RESET = 0 CLOCK = 1
 TIME = 10 RESET = 0 CLOCK = 0
 TIME = 11 RESET = 0 CLOCK = 1
 TIME = 12 RESET = 0 CLOCK = 0
 TIME = 13 RESET = 0 CLOCK = 1
 TIME = 14 RESET = 0 CLOCK = 0
 TIME = 15 RESET = 0 CLOCK = 1
 TIME = 16 RESET = 0 CLOCK = 0
```

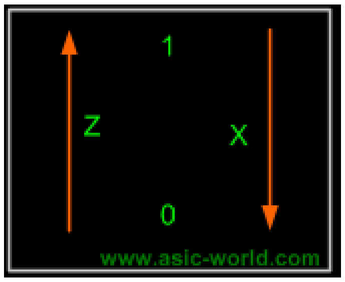
Waveform



Edge sensitive Event Controls

Delays execution of the next statement until the specified transition on a signal.

syntax : @ (< posedge >|< negedge > signal) < statement >;



Example - Edge Wait

```
module edge_wait_example();

reg enable, clk, trigger;

always @ (posedge enable)
begin
  trigger = 0;
  // Wait for 5 clock cycles
  repeat (5) begin
    @ (posedge clk) ;
  end
  trigger = 1;
end

//Testbench code here
initial begin
  $monitor ("TIME : %g CLK : %b ENABLE : %b TRIGGER : %b",
    $time, clk,enable,trigger);
  clk = 0;
  enable = 0;
  #5  enable = 1;
  #1  enable = 0;
  #10 enable = 1;
  #1  enable = 0;
```

```
   #10  $finish;
end

always
 #1  clk = ~clk;

endmodule
```

You could download file edge_wait_example.v here

Simulator Output

```
TIME : 0 CLK : 0 ENABLE : 0 TRIGGER : x
 TIME : 1 CLK : 1 ENABLE : 0 TRIGGER : x
 TIME : 2 CLK : 0 ENABLE : 0 TRIGGER : x
 TIME : 3 CLK : 1 ENABLE : 0 TRIGGER : x
 TIME : 4 CLK : 0 ENABLE : 0 TRIGGER : x
 TIME : 5 CLK : 1 ENABLE : 1 TRIGGER : 0
 TIME : 6 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 7 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 8 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 9 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 10 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 11 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 12 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 13 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 14 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 15 CLK : 1 ENABLE : 0 TRIGGER : 1
 TIME : 16 CLK : 0 ENABLE : 1 TRIGGER : 0
 TIME : 17 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 18 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 19 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 20 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 21 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 22 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 23 CLK : 1 ENABLE : 0 TRIGGER : 0
 TIME : 24 CLK : 0 ENABLE : 0 TRIGGER : 0
 TIME : 25 CLK : 1 ENABLE : 0 TRIGGER : 1
 TIME : 26 CLK : 0 ENABLE : 0 TRIGGER : 1
```

Level-Sensitive Even Controls (Wait statements)

Delays execution of the next statement until < expression > evaluates to true

syntax : wait (< expression >) < statement >;

Example - Level Wait

```
module wait_example();

reg mem_read, data_ready;
reg [7:0] data_bus, data;

always @ (mem_read or data_bus or data_ready)
begin
  data = 0;
  while (mem_read == 1'b1) begin
    // #1 is very important to avoid infinite loop
    wait (data_ready == 1) #1 data = data_bus;
  end
end

// Testbench Code here
initial begin
 $monitor ("TIME = %g READ = %b READY = %b DATA = %b",
   $time, mem_read, data_ready, data);
 data_bus = 0;
 mem_read = 0;
 data_ready = 0;
```

```
  #10 data_bus = 8'hDE;
  #10 mem_read = 1;
  #20 data_ready = 1;
  #1  mem_read = 1;
  #1  data_ready = 0;
  #10 data_bus = 8'hAD;
  #10 mem_read = 1;
  #20 data_ready = 1;
  #1  mem_read = 1;
  #1  data_ready = 0;
  #10 $finish;
end

endmodule
```

You could download file wait_example.v here

Simulator Output

```
TIME = 0  READ = 0 READY = 0 DATA = 00000000
 TIME = 20 READ = 1 READY = 0 DATA = 00000000
 TIME = 40 READ = 1 READY = 1 DATA = 00000000
 TIME = 41 READ = 1 READY = 1 DATA = 11011110
 TIME = 42 READ = 1 READY = 0 DATA = 11011110
 TIME = 82 READ = 1 READY = 1 DATA = 11011110
 TIME = 83 READ = 1 READY = 1 DATA = 10101101
 TIME = 84 READ = 1 READY = 0 DATA = 10101101
```

Intra-Assignment Timing Controls

Intra-assignment controls always evaluate the right side expression immediately and assign the result after the delay or event control.

In non-intra-assignment controls (delay or event control on the left side), the right side expression is evaluated after the delay or event control.

Example - Intra-Assignment

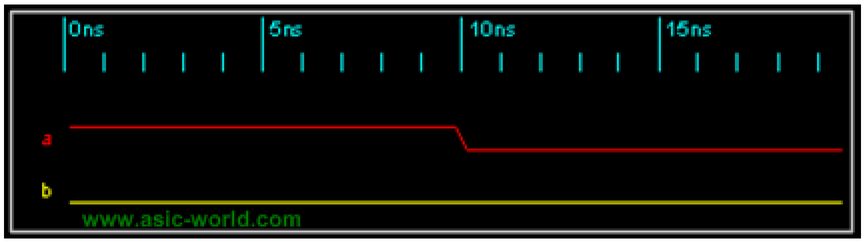
```verilog
module intra_assign();

reg a, b;

initial begin
   $monitor("TIME = %g  A = %b  B = %b",$time, a , b);
   a = 1;
   b = 0;
   a = #10 0;
   b = a;
   #20  $display("TIME = %g  A = %b  B = %b",$time, a , b);
   $finish;
end

endmodule
```

You could download file intra_assign.v here

Simulation Output

```
 TIME = 0   A = 1  B = 0
TIME = 10  A = 0  B = 0
TIME = 30  A = 0  B = 0
```

Waveform



Modeling Combo Logic with Continuous Assignments

Whenever any signal changes on the right hand side, the entire right-hand side is re-evaluated and the result is assigned to the left hand side.

Example - Tri-state Buffer

```
module tri_buf_using_assign();
reg data_in, enable;
wire pad;

assign pad = (enable) ? data_in : 1'bz;

initial begin
  $monitor ("TIME = %g ENABLE = %b DATA : %b PAD %b",
    $time, enable, data_in, pad);
  #1 enable = 0;
  #1 data_in = 1;
  #1 enable = 1;
  #1 data_in = 0;
  #1 enable = 0;
  #1 $finish;
end
```

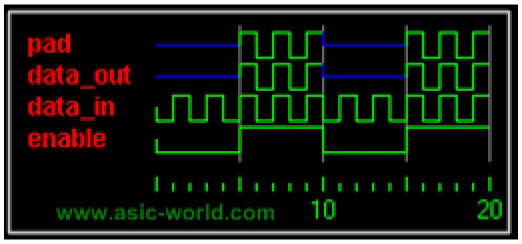
```

endmodule
```

You could download file tri_buf_using_assign.v here

Simulation Output

```
TIME = 0 ENABLE = x DATA : x PAD x
TIME = 1 ENABLE = 0 DATA : x PAD z
TIME = 2 ENABLE = 0 DATA : 1 PAD z
TIME = 3 ENABLE = 1 DATA : 1 PAD 1
TIME = 4 ENABLE = 1 DATA : 0 PAD 0
TIME = 5 ENABLE = 0 DATA : 0 PAD z
```

Waveform



Example - Mux

```
module mux_using_assign();
reg data_in_0, data_in_1;
```

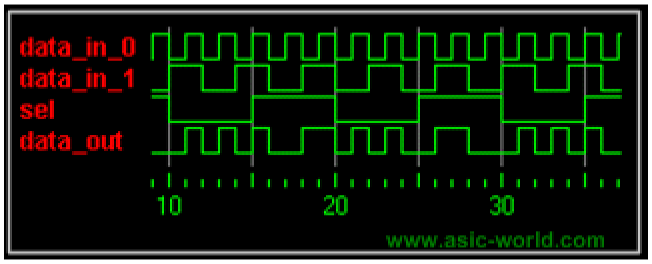
```
wire data_out;
reg  sel;

assign data_out = (sel) ? data_in_1 : data_in_0;

// Testbench code here
initial begin
   $monitor("TIME = %g SEL = %b DATA0 = %b DATA1 = %b OUT = %b",
     $time,sel,data_in_0,data_in_1,data_out);
   data_in_0 = 0;
   data_in_1 = 0;
   sel = 0;
   #10 sel = 1;
   #10 $finish;
end

// Toggel data_in_0 at #1
always
 #1 data_in_0 = ~data_in_0;

// Toggel data_in_1 at #2
always
 #2 data_in_1 = ~data_in_1;

endmodule
```

You could download file mux_using_assign.v here

Simulation Output

```
TIME = 0 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
 TIME = 1 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
 TIME = 2 SEL = 0 DATA0 = 0 DATA1 = 1 OUT = 0
 TIME = 3 SEL = 0 DATA0 = 1 DATA1 = 1 OUT = 1
 TIME = 4 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
 TIME = 5 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
 TIME = 6 SEL = 0 DATA0 = 0 DATA1 = 1 OUT = 0
 TIME = 7 SEL = 0 DATA0 = 1 DATA1 = 1 OUT = 1
 TIME = 8 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
 TIME = 9 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
 TIME = 10 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
 TIME = 11 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
 TIME = 12 SEL = 1 DATA0 = 0 DATA1 = 0 OUT = 0
 TIME = 13 SEL = 1 DATA0 = 1 DATA1 = 0 OUT = 0
 TIME = 14 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
 TIME = 15 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
 TIME = 16 SEL = 1 DATA0 = 0 DATA1 = 0 OUT = 0
 TIME = 17 SEL = 1 DATA0 = 1 DATA1 = 0 OUT = 0
 TIME = 18 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
 TIME = 19 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
```

Waveform



$display, $strobe, $monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like GTKWave. or Undertow or Debussy. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes. The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly when it is executed. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (hierarchy level). %5d, %5b etc. would give exactly 5 spaces for the number instead of the space needed. Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax

- $display ("format_string", par_1, par_2, ...);
- $strobe ("format_string", par_1, par_2, ...);
- $monitor ("format_string", par_1, par_2, ...);
- $displayb (as above but defaults to binary..);
- $strobeh (as above but defaults to hex..);
- $monitoro (as above but defaults to octal..);

$time, $stime, $realtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.

$reset, $stop, $finish

$reset resets the simulation back to time 0; $stop halts the simulator and puts it in interactive mode where the user can enter commands; $finish exits the simulator back to the operating system.

$scope, $showscope

$scope(hierarchy_name) sets the current hierarchical scope to hierarchy_name. $showscopes(n) lists all modules, tasks and block names in (and below, if n is set to 1) the current scope.

$random

$random generates a random integer every time it is called. If the sequence is to be repeatable, the first time one invokes random giving it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

$dumpfile, $dumpvar, $dumpon, $dumpoff, $dumpall

These can dump variable changes to a simulation viewer like Debussy. The dump files are capable of dumping all the variables in a simulation. This is convenient for debugging, but can be very slow.

Syntax

- $dumpfile("filename.vcd")
- $dumpvar dumps all variables in the design.
- $dumpvar(1, top) dumps all the variables in module top and below, but not modules instantiated in top.
- $dumpvar(2, top) dumps all the variables in module top and 1 level below.
- $dumpvar(n, top) dumps all the variables in module top and n-1 levels below.
- $dumpvar(0, top) dumps all the variables in module top and all level below.
- $dumpon initiates the dump.
- $dumpoff stop dumping.

$fopen, $fdisplay, $fstrobe $fmonitor and $fwrite

These commands write more selectively to files.

- $fopen opens an output file and gives the open file a handle for use by the other commands.
- $fclose closes the file and lets other programs access it.
- $fdisplay and $fwrite write formatted data to a file whenever they are executed. They are the same except $fdisplay inserts a new line after every execution and $write does not.
- $strobe also writes to a file when executed, but it waits until all other operations in the time step are complete before writing. Thus initial #1 a=1; b=0; $fstrobe(hand1, a,b); b=1; will write write 1 1 for a and b.
- $monitor writes to a file whenever any of its arguments changes.

Syntax

- handle1=$fopen("filenam1.suffix")
- handle2=$fopen("filenam2.suffix")
- $fstrobe(handle1, format, variable list) //strobe data into filenam1.suffix
- $fdisplay(handle2, format, variable list) //write data into filenam2.suffix
- $fwrite(handle2, format, variable list) //write data into filenam2.suffix all on one line. Put in the format string where a new line is desired.

Writing a testbench is as complex as writing the RTL code itself. These days ASICs are getting more and more complex and thus verifying these complex ASIC has become a challenge. Typically 60-70% of time needed for any ASIC is spent on verification/validation/testing. Even though the above facts are well known to most ASIC engineers, still engineers think that there is no glory in verification.

Memory Modeling

To help modeling of memory, Verilog provides support for two dimensions arrays. Behavioral models of memories are modeled by declaring an array of register variables; any word in the array may be accessed using an index into the array. A temporary variable is required to access a discrete bit within the array.

Syntax

```
reg [wordsize:0] array_name [0:arraysize]
```

Examples

Declaration

```
reg [7:0] my_memory [0:255];
```

Here [7:0] is the memory width and [0:255] is the memory depth with the following parameters:

- Width : 8 bits, little endian

- Depth : 256, address 0 corresponds to location 0 in the array.

Storing Values

```
my_memory[address] = data_in;
```

Reading Values

```
data_out = my_memory[address];
```

Bit Read

Sometimes there may be need to read just one bit. Unfortunately Verilog does not allow to read or write only one bit: the workaround for such a problem is as shown below.

```
data_out = my_memory[address];

data_out_it_0 = data_out[0];
```

Initializing Memories

A memory array may be initialized by reading memory pattern file from disk and storing it on the memory array. To do this, we use system tasks $readmemb and $readmemh. $readmemb is used for binary representation of memory content and $readmemh for hex representation.

Syntax

$readmemh("file_name",mem_array,start_addr,stop_addr);

Note : start_addr and stop_addr are optional.

Example - Simple memory

```
module  memory();
reg [7:0] my_memory [0:255];

initial  begin
  $readmemh("memory.list", my_memory);
end
endmodule
```

You could download file memory.v here

Example - Memory.list file

```
//Comments are allowed
1100_1100     // This is first address i.e 8'h00
1010_1010     // This is second address i.e 8'h01
@ 55           // Jump to new address 8'h55
```

```
0101_1010    // This is address 8'h55
0110_1001    // This is address 8'h56
```

You could download file memory.list here

$readmemh system task can also be used for reading testbench vectors. I will cover this in detail in the test bench section ... when I find time.

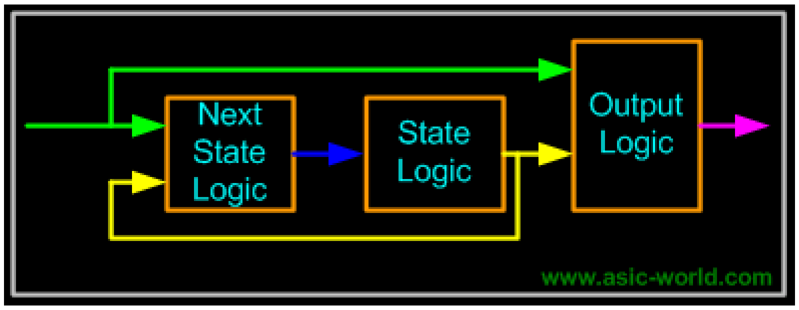
Introduction to FSM

State machines or FSM are the heart of any digital design; of course a counter is a simple form of FSM. When I was learning Verilog, I used to wonder "How do I code FSM in Verilog" and "What is the best way to code it". I will try to answer the first part of the question below and second part of the question can be found in the tidbits section.

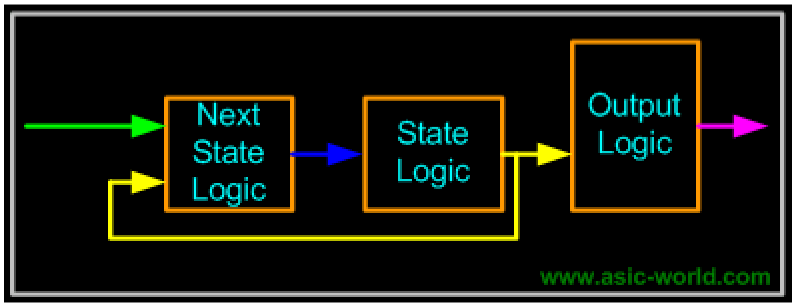
State machine Types

There are two types of state machines as classified by the types of outputs generated from each. The first is the Moore State Machine where the outputs are only a function of the present state, the second is the Mealy State Machine where one or more of the outputs are a function of the present state and one or more of the inputs.

Mealy Model



Moore Model



State machines can also be classified according to the state encoding used. Encoding style is also a critical factor which decides speed and gate complexity of the FSM. Binary, gray, one hot, one cold, and almost one hot are the different types of encoding styles used in coding FSM states.

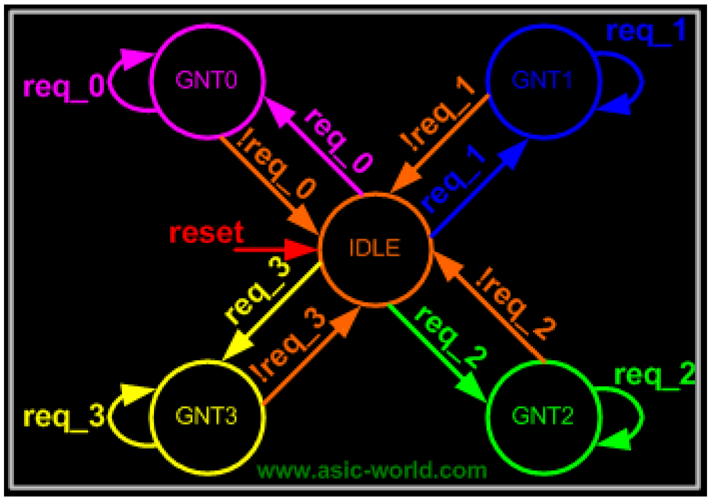
Modeling State machines.

One thing that need to be kept in mind when coding FSM is that combinational logic and sequence logic should be in two different always blocks. In the above two figures, next state logic is always the combinational logic. State Registers and Output logic are sequential logic. It is very important that any asynchronous signal to the next state logic be synchronized before being fed to the FSM. Always try to keep FSM in a separate Verilog file.

Using constants declaration like parameter or `define to define states of the FSM makes code more readable and easy to manage.

Example - Arbiter

We will be using the arbiter FSM to study FSM coding styles in Verilog.



Verilog Code

FSM code should have three sections:

- Encoding style.
- Combinational part.
- Sequential part.

Encoding Style

There are many encoding styles around, some of which are:

- Binary Encoding
- One Hot Encoding
- One Cold Encoding
- Almost One Hot Encoding
- Almost One Cold Encoding
- Gray Encoding

Of all the above types we normally use one hot and binary encoding.

One Hot Encoding

```
parameter  [4:0]  IDLE  = 5'b0_0001;
parameter  [4:0]  GNT0  = 5'b0_0010;
parameter   [4:0]   GNT1  = 5'b0_0100;
parameter   [4:0]   GNT2  = 5'b0_1000;
parameter   [4:0]   GNT3  = 5'b1_0000;
```

You could download file fsm_one_hot_params.v here

Binary Encoding

```
parameter   [2:0]   IDLE  = 3'b000;
parameter   [2:0]   GNT0  = 3'b001;
parameter   [2:0]   GNT1  = 3'b010;
parameter   [2:0]   GNT2  = 3'b011;
parameter   [2:0]   GNT3  = 3'b100;
```

You could download file fsm_binary_params.v here

Gray Encoding

```
parameter   [2:0]   IDLE  = 3'b000;
parameter   [2:0]   GNT0  = 3'b001;
parameter   [2:0]   GNT1  = 3'b011;
parameter   [2:0]   GNT2  = 3'b010;
parameter   [2:0]   GNT3  = 3'b110;
```

You could download file fsm_gray_params.v here

Combinational Section

This section can be modeled using functions, assign statements or using always blocks with a case statement. For the time being let's see the always block version

```
always @ (state or req_0 or req_1 or req_2 or req_3)
begin
  next_state = 0;
  case(state)
    IDLE : if (req_0 == 1'b1) begin
         next_state = GNT0;
           end else if (req_1 == 1'b1) begin
         next_state= GNT1;
           end else if (req_2 == 1'b1) begin
         next_state= GNT2;
           end else if (req_3 == 1'b1) begin
         next_state= GNT3;
      end else begin
         next_state = IDLE;
           end
    GNT0 : if (req_0 == 1'b0) begin
         next_state = IDLE;
           end else begin
         next_state = GNT0;
      end
    GNT1 : if (req_1 == 1'b0) begin
         next_state = IDLE;
           end else begin
         next_state = GNT1;
      end
    GNT2 : if (req_2 == 1'b0) begin
         next_state = IDLE;
           end else begin
         next_state = GNT2;
      end
    GNT3 : if (req_3 == 1'b0) begin
         next_state = IDLE;
           end else begin
         next_state = GNT3;
      end
   default : next_state = IDLE;
  endcase
end
```

You could download file fsm_combo.v here

Sequential Section

This section has to be modeled using only edge sensitive logic such as always block with posedge or negedge of clock.

```
always @ (posedge clock)
begin : OUTPUT_LOGIC
  if (reset == 1'b1) begin
    gnt_0 <= #1 1'b0;
    gnt_1 <= #1 1'b0;
    gnt_2 <= #1 1'b0;
    gnt_3 <= #1 1'b0;
    state <= #1 IDLE;
  end else begin
    state <= #1 next_state;
    case(state)
       IDLE : begin
                  gnt_0 <= #1 1'b0;
                  gnt_1 <= #1 1'b0;
                  gnt_2 <= #1 1'b0;
                  gnt_3 <= #1 1'b0;
             end
    GNT0 : begin
                gnt_0 <= #1 1'b1;
             end
          GNT1 : begin
                   gnt_1 <= #1 1'b1;
                 end
          GNT2 : begin
                   gnt_2 <= #1 1'b1;
                 end
          GNT3 : begin
```

```
                    gnt_3 <= #1  1'b1;
                  end
         default : begin
                    state <= #1  IDLE;
                  end
        endcase
      end
    end
```

You could download file fsm_seq.v here

Full Code using binary encoding

```
module fsm_full(
clock , // Clock
reset , // Active high reset
req_0 , // Active high request from agent 0
req_1 , // Active high request from agent 1
req_2 , // Active high request from agent 2
req_3 , // Active high request from agent 3
gnt_0 , // Active high grant to agent 0
gnt_1 , // Active high grant to agent 1
gnt_2 , // Active high grant to agent 2
gnt_3   // Active high grant to agent 3
);
// Port declaration here
input clock ; // Clock
input reset ; // Active high reset
input req_0 ; // Active high request from agent 0
input req_1 ; // Active high request from agent 1
input req_2 ; // Active high request from agent 2
input req_3 ; // Active high request from agent 3
output gnt_0 ; // Active high grant to agent 0
output gnt_1 ; // Active high grant to agent 1
output gnt_2 ; // Active high grant to agent 2
output gnt_3 ; // Active high grant to agent

// Internal Variables
```

```
reg     gnt_0  ;  // Active high grant to agent 0
reg     gnt_1  ;  // Active high grant to agent 1
reg     gnt_2  ;  // Active high grant to agent 2
reg     gnt_3  ;  // Active high grant to agent

parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b010;
parameter  [2:0]  GNT2  = 3'b011;
parameter  [2:0]  GNT3  = 3'b100;

reg [2:0] state, next_state;

always @ (state or req_0 or req_1 or req_2 or req_3)
begin
  next_state = 0;
  case(state)
    IDLE : if (req_0 == 1'b1) begin
        next_state = GNT0;
           end else if (req_1 == 1'b1) begin
        next_state= GNT1;
           end else if (req_2 == 1'b1) begin
        next_state= GNT2;
           end else if (req_3 == 1'b1) begin
        next_state= GNT3;
      end else begin
        next_state = IDLE;
           end
    GNT0 : if (req_0 == 1'b0) begin
        next_state = IDLE;
           end else begin
        next_state = GNT0;
     end
    GNT1 : if (req_1 == 1'b0) begin
        next_state = IDLE;
           end else begin
        next_state = GNT1;
     end
    GNT2 : if (req_2 == 1'b0) begin
        next_state = IDLE;
           end else begin
        next_state = GNT2;
     end
    GNT3 : if  (req_3 == 1'b0)  begin
        next_state = IDLE;
            end  else  begin
        next_state = GNT3;
     end
   default : next_state = IDLE;
  endcase
end

always  @  (posedge  clock)
begin : OUTPUT_LOGIC
  if  (reset)  begin
    gnt_0  <=  #1  1'b0;
    gnt_1  <=  #1  1'b0;
    gnt_2  <=  #1  1'b0;
    gnt_3  <=  #1  1'b0;
    state  <=  #1  IDLE;
  end  else  begin
    state  <=  #1  next_state;
    case (state)
   IDLE : begin
                gnt_0  <=  #1  1'b0;
                gnt_1  <=  #1  1'b0;
                gnt_2  <=  #1  1'b0;
                gnt_3  <=  #1  1'b0;
           end
   GNT0 : begin
             gnt_0  <=  #1  1'b1;
           end
         GNT1 : begin
                 gnt_1  <=  #1  1'b1;
               end
         GNT2 : begin
                 gnt_2  <=  #1  1'b1;
               end
         GNT3 : begin
                 gnt_3  <=  #1  1'b1;
               end
      default : begin
                 state  <=  #1  IDLE;
               end
    endcase
  end
```

```
end

endmodule
```

You could download file fsm_full.v here

Testbench

```
`include "fsm_full.v"

module fsm_full_tb();
reg clock , reset ;
reg req_0 , req_1 ,  req_2 , req_3;
wire gnt_0 , gnt_1 , gnt_2 , gnt_3 ;

initial begin
   $display("Time\t   R0 R1 R2 R3 G0 G1 G2 G3");
   $monitor("%g\t   %b  %b  %b  %b  %b  %b  %b  %b",
     $time, req_0, req_1, req_2, req_3, gnt_0, gnt_1, gnt_2, gnt_3);
   clock = 0;
   reset = 0;
   req_0 = 0;
   req_1 = 0;
   req_2 = 0;
   req_3 = 0;
   #10 reset = 1;
   #10 reset = 0;
   #10 req_0 = 1;
   #20 req_0 = 0;
   #10 req_1 = 1;
   #20 req_1 = 0;
   #10 req_2 = 1;
   #20 req_2 = 0;
   #10 req_3 = 1;
   #20 req_3 = 0;
   #10 $finish;
end

always
 #2 clock = ~clock;

```

```
fsm_full U_fsm_full(
clock , // Clock
reset , // Active high reset
req_0 , // Active high request from agent 0
req_1 , // Active high request from agent 1
req_2 , // Active high request from agent 2
req_3 , // Active high request from agent 3
gnt_0 , // Active high grant to agent 0
gnt_1 , // Active high grant to agent 1
gnt_2 , // Active high grant to agent 2
gnt_3   // Active high grant to agent 3
);



endmodule
```

You could download file fsm_full_tb.v here

Simulator Output

```
Time        R0 R1 R2 R3 G0 G1 G2 G3
 0          0  0  0  0  x  x  x  x
 7          0  0  0  0  0  0  0  0
 30         1  0  0  0  0  0  0  0
 35         1  0  0  0  1  0  0  0
 50         0  0  0  0  1  0  0  0
 55         0  0  0  0  0  0  0  0
 60         0  1  0  0  0  0  0  0
 67         0  1  0  0  0  1  0  0
 80         0  0  0  0  0  1  0  0
 87         0  0  0  0  0  0  0  0
 90         0  0  1  0  0  0  0  0
 95         0  0  1  0  0  0  1  0
 110        0  0  0  0  0  0  1  0
 115        0  0  0  0  0  0  0  0
 120        0  0  0  1  0  0  0  0
 127        0  0  0  1  0  0  0  1
 140        0  0  0  0  0  0  0  1
 147        0  0  0  0  0  0  0  0
```

assign and deassign

The assign and deassign procedural assignment statements allow continuous assignments to be placed onto registers for controlled periods of time. The assign procedural statement overrides procedural assignments to a **register**. The deassign procedural statement ends a continuous assignment to a register.

Example - assign and deassign

```
module assign_deassign ();

reg clk,rst,d,preset;
wire q;

initial begin
  $monitor("@%g clk %b rst %b preset %b d %b q %b",
    $time, clk, rst, preset, d, q);
  clk = 0;
  rst = 0;
  d  = 0;
  preset = 0;
  #10 rst = 1;
  #10 rst = 0;
  repeat (10) begin
    @ (posedge clk);
    d <= $random;
    @ (negedge clk) ;
    preset <= ~preset;
  end
  #1 $finish;
end
// Clock generator
always #1 clk = ~clk;

// assign and deassign q of flip flop module
always @(preset)
if (preset) begin
  assign U.q = 1; // assign procedural statement
end else begin
```

```
 deassign U.q;     // deassign procedural statement
end

d_ff U (clk,rst,d,q);

endmodule

// D Flip-Flop model
module d_ff (clk,rst,d,q);
input clk,rst,d;
output q;
reg q;

always @ (posedge clk)
if (rst) begin
  q <= 0;
end else begin
  q <= d;
end

endmodule
```

You could download file assign_deassign.v here

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
 @1  clk 1 rst 0 preset 0 d 0 q 0
 @2  clk 0 rst 0 preset 0 d 0 q 0
 @3  clk 1 rst 0 preset 0 d 0 q 0
 @4  clk 0 rst 0 preset 0 d 0 q 0
 @5  clk 1 rst 0 preset 0 d 0 q 0
 @6  clk 0 rst 0 preset 0 d 0 q 0
 @7  clk 1 rst 0 preset 0 d 0 q 0
 @8  clk 0 rst 0 preset 0 d 0 q 0
 @9  clk 1 rst 0 preset 0 d 0 q 0
 @10 clk 0 rst 1 preset 0 d 0 q 0
 @11 clk 1 rst 1 preset 0 d 0 q 0
 @12 clk 0 rst 1 preset 0 d 0 q 0
 @13 clk 1 rst 1 preset 0 d 0 q 0
```

```
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 1
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```

force and release

Another form of procedural continuous assignment is provided by the force and release procedural statements. These statements have a similar effect on the assign-deassign pair, but a force can be applied to nets as well as to registers.

One can use force and release while doing gate level simulation to work around reset connectivity problems. Also can be used insert single and double bit errors on data read from memory.

Example - force and release

```
module force_release ();

reg clk,rst,d,preset;
wire q;

initial begin
  $monitor("@%g clk %b rst %b preset %b d %b q %b",
    $time, clk, rst, preset, d, q);
  clk = 0;
  rst = 0;
  d  = 0;
  preset = 0;
  #10 rst = 1;
  #10 rst = 0;
  repeat (10) begin
    @ (posedge clk);
    d <= $random;
    @ (negedge clk) ;
    preset <= ~preset;
  end
  #1 $finish;
end
// Clock generator
always #1 clk = ~clk;

// force and release of flip flop module
always @(preset)
if (preset) begin
  force U.q = preset; // force procedural statement
end else begin
  release U.q;      // release procedural statement
end

d_ff U (clk,rst,d,q);

```

```
endmodule

// D Flip-Flop model
module d_ff (clk,rst,d,q);
input clk,rst,d;
output q;
wire q;
reg q_reg;

assign q = q_reg;

always @ (posedge clk)
if (rst) begin
  q_reg <= 0;
end else begin
  q_reg <= d;
end

endmodule
```

You could download file force_release.v here

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
@1  clk 1 rst 0 preset 0 d 0 q 0
@2  clk 0 rst 0 preset 0 d 0 q 0
@3  clk 1 rst 0 preset 0 d 0 q 0
@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
```

```
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 0
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```

Introduction

Let's assume that we have a design which requires us to have counters of various width, but with the same functionality. Maybe we can assume that we have a design which requires lots of instants of different depth and width of RAMs of similar functionality. Normally what we do is creating counters of different widths and then use them. The same rule applies to the RAM we talked about.

But Verilog provides a powerful way to overcome this problem: it provides us with something called parameter; these parameters are like constants local to that particular module.

We can override the default values, either using defparam or by passing a new set of parameters during instantiation. We call this parameter overriding.

Parameters

A parameter is defined by Verilog as a constant value declared within the module structure. The value can be used to define a set of attributes for the module which can characterize its behavior as well as its physical representation.

- Defined inside a module.
- Local scope.
- Maybe overridden at instantiation time.
    - If multiple parameters are defined, they must be overridden in the order they were defined. If an overriding value is not specified, the default parameter declaration values are used.
- Maybe changed using the defparam statement.

Parameter Override using defparam

```
module secret_number;
parameter my_secret = 0;
```

```

initial  begin
   $display ("My secret number is %d", my_secret);
end

endmodule

module defparam_example();

defparam U0.my_secret = 11;
defparam U1.my_secret = 22;

secret_number U0();
secret_number U1();

endmodule
```

You could download file defparam_example.v here

Parameter Override during instantiating.

```
module secret_number;
parameter my_secret = 0;

initial  begin
   $display ("My secret number in module is %d", my_secret);
end

endmodule

module param_overide_instance_example();

secret_number #(11) U0();
secret_number #(22) U1();

endmodule
```

You could download file param_overide_instance_example.v here

Passing more than one parameter

```
module  ram_sp_sr_sw (
clk            , // Clock Input
address        , // Address Input
data           , // Data bi-directional
cs             , // Chip Select
we             , // Write Enable/Read Enable
oe               // Output Enable
);

parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
// Actual code of RAM here

endmodule
```

You could download file param_more_then_one.v here

When instantiating more than the one parameter, parameter values should be passed in the order they are declared in the sub module.

```
module  ram_controller ();//Some ports

// Controller Code

ram_sp_sr_sw #(16,8,256)  ram(clk,address,data,cs,we,oe);

endmodule
```

You could download file param_more_then_one1.v here

N-Input Primitives

The and, nand, or, nor, xor, and xnor primitives have one output and any number of inputs

- The single output is the first terminal.
- All other terminals are inputs.

Examples

```
module n_in_primitive();

wire out1,out2,out3;
reg in1,in2,in3,in4;

// Two input AND gate
and u_and1 (out1, in1, in2);
// four input AND gate
and u_and2 (out2, in1, in2, in3, in4);
// three input XNOR gate
xnor u_xnor1 (out3, in1, in2, in3);

//Testbench Code
initial begin
   $monitor (
   "in1 = %b in2 = %b in3 = %b in4 = %b out1 = %b out2 = %b out3 = %b",
   in1, in2, in3, in4, out1, out2, out3);
   in1 = 0;
   in2 = 0;
   in3 = 0;
   in4 = 0;
   #1  in1 = 1;
   #1  in2 = 1;
   #1  in3 = 1;
   #1  in4 = 1;
   #1  $finish;
end

endmodule
```

You could download file n_in_primitive.v here

```
                        in1 = 0 in2 = 0 in3 = 0 in4 = 0 out1 = 0 out2 = 0 out3 = 1
  in1 = 1 in2 = 0 in3 = 0 in4 = 0 out1 = 0 out2 = 0 out3 = 0
  in1 = 1 in2 = 1 in3 = 0 in4 = 0 out1 = 1 out2 = 0 out3 = 1
  in1 = 1 in2 = 1 in3 = 1 in4 = 0 out1 = 1 out2 = 0 out3 = 0
  in1 = 1 in2 = 1 in3 = 1 in4 = 1 out1 = 1 out2 = 1 out3 = 0
```

N-Output Primitives

The buf and not primitives have any number of outputs and one input

- The outputs are the first terminals listed.
- The last terminal is the single input.

Examples

```
module n_out_primitive();

wire out,out_0,out_1,out_2,out_3,out_a,out_b,out_c;
wire in;

```

```
// one output Buffer gate
buf u_buf0 (out,in);
// four output Buffer gate
buf u_buf1 (out_0, out_1, out_2, out_3, in);
// three output Invertor gate
not u_not0 (out_a, out_b, out_c, in);

endmodule
```

You could download file n_out_primitive.v here

Example - "fork-join"

```
module initial_fork_join();
reg clk,reset,enable,data;

initial begin
  $monitor("%g clk=%b reset=%b enable=%b data=%b",
    $time, clk, reset, enable, data);
  fork
    #1   clk = 0;
    #10  reset = 0;
    #5   enable = 0;
    #3   data = 0;
  join
  #1  $display ("%g Terminating simulation", $time);
  $finish;
end

endmodule
```

You could download file initial_fork_join.v here

Fork : clk gets its value after 1 time unit, reset after 10 time units, enable after 5 time units, data after 3 time units. All the statements are executed in parallel.

Simulator Output

```
0 clk=x reset=x enable=x data=x
 1 clk=0 reset=x enable=x data=x
 3 clk=0 reset=x enable=x data=0
 5 clk=0 reset=x enable=0 data=0
 10 clk=0 reset=0 enable=0 data=0
 11 Terminating simulation
```

Sequential Statement Groups

The **begin - end** keywords:

- Group several statements together.
- Cause the statements to be evaluated sequentially (one at a time)
    - Any timing within the sequential groups is relative to the previous statement.
    - Delays in the sequence accumulate (each delay is added to the previous delay)
- Block finishes after the last statement in the block.

Example - sequential

```
module sequential();

reg a;

initial begin
  $monitor ("%g a = %b", $time, a);
  #10 a = 0;
  #11 a = 1;
  #12 a = 0;
  #13 a = 1;
  #14 $finish;
end

endmodule
```

You could download file sequential.v here

Simulator Output

```
 0 a = x
 10 a = 0
 21 a = 1
 33 a = 0
 46 a = 1
```

Parallel Statement Groups

The fork - join keywords:

- Group several statements together.
- Cause the statements to be evaluated in parallel (all at the same time).
    - Timing within parallel group is absolute to the beginning of the group.
    - Block finishes after the last statement completes (Statement with highest delay, it can be the first statement in the block).

Example - Parallel

```
module parallel();

reg a;

initial
fork
  $monitor ("%g a = %b", $time, a);
  #10  a = 0;
  #11  a = 1;
  #12  a = 0;
  #13  a = 1;
  #14  $finish;
join

endmodule
```

You could download file parallel.v here

Simulator Output

```
0 a = x
 10 a = 0
 11 a = 1
 12 a = 0
 13 a = 1
```

Example - Mixing "begin-end" and "fork - join"

```
module fork_join();

reg clk,reset,enable,data;

initial  begin
   $display ("Starting simulation");
   $monitor("%g clk=%b reset=%b enable=%b data=%b",
     $time, clk, reset, enable, data);
   fork : FORK_VAL
     #1  clk = 0;
     #5  reset = 0;
     #5  enable = 0;
     #2  data = 0;
   join
   #10  $display ("%g Terminating simulation", $time);
   $finish;
end

endmodule
```

You could download file fork_join.v here

Simulator Output

```
0 clk=x reset=x enable=x data=x
 1 clk=0 reset=x enable=x data=x
 2 clk=0 reset=x enable=x data=0
 5 clk=0 reset=0 enable=0 data=0
 15 Terminating simulation
```

EARTHPEOPLE Technology

Verilog Programming Guide