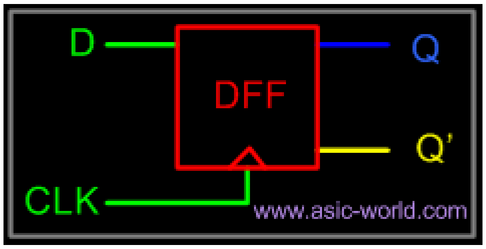
INTRODUCTION

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

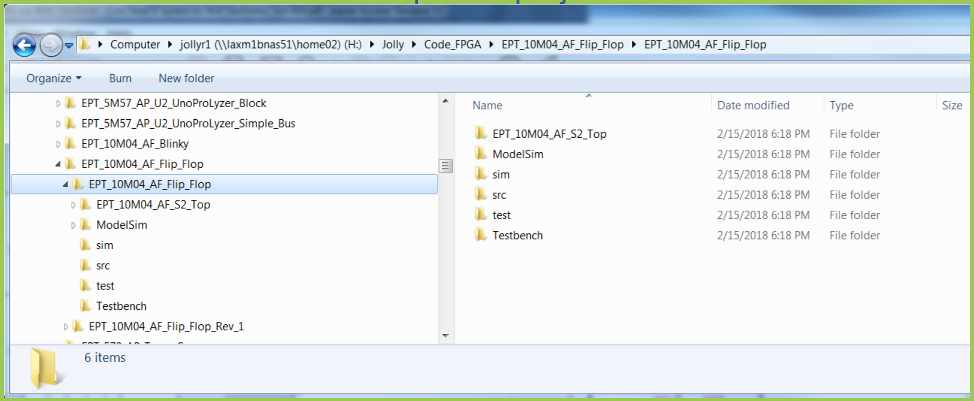
One can describe a simple Flip flop as that in the above figure, as well as a complicated design having 1 million gates. Verilog is one of the HDL languages available in the industry for hardware designing. It allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate 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.

////////////////////////////////////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.

Copy the Lesson 1 Folder over to our local drive from the DVD. In this folder you will notice a top level project folder.



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

- EPT_5M57_AP_U2_UnoProLyzer_Block
- EPT_5M57_AP_U2_UnoProLyzer_Simple_Bus
- 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

Name	Date modified	Type	Size
EPT_10M04_AF_S2_Top	2/15/2018 6:18 PM	File folder	
ModelSim	2/15/2018 6:18 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	

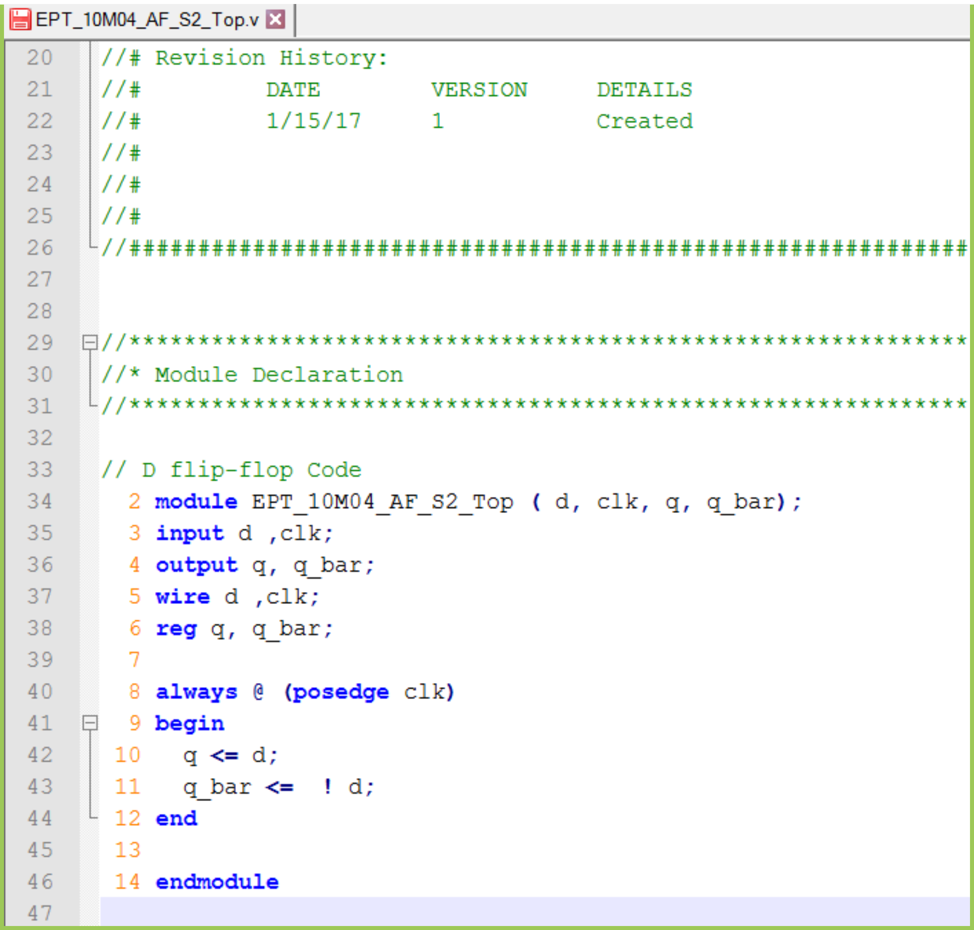
6 items

The project level folder is called "EPT_10M04_AF_Flip_Flop". Under this project level folder are various revisions of the project. 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 current revision of the project, this folder does not include a "_Rev_x" suffix. The project is organized with the following folders:

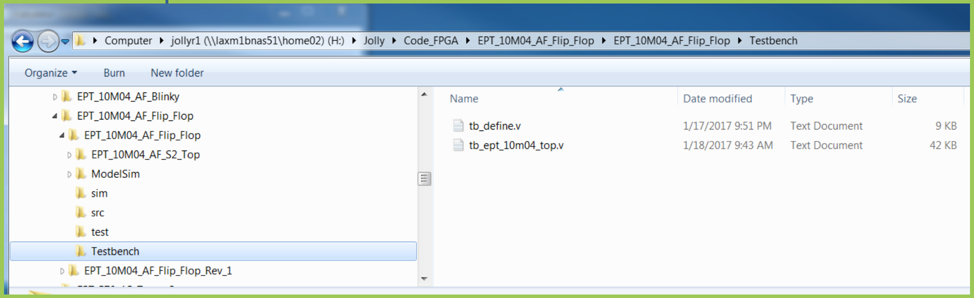
- EPT_10M04_AF_S2_Top – This folder contains all the Altera project level files such as pin, sdc, configuration, programming object files etc.
- ModelSim – This folder contains all the compiled object files for ModelSims use. It also includes the *.do files. These 'do' files are the make files for ModelSim. 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.



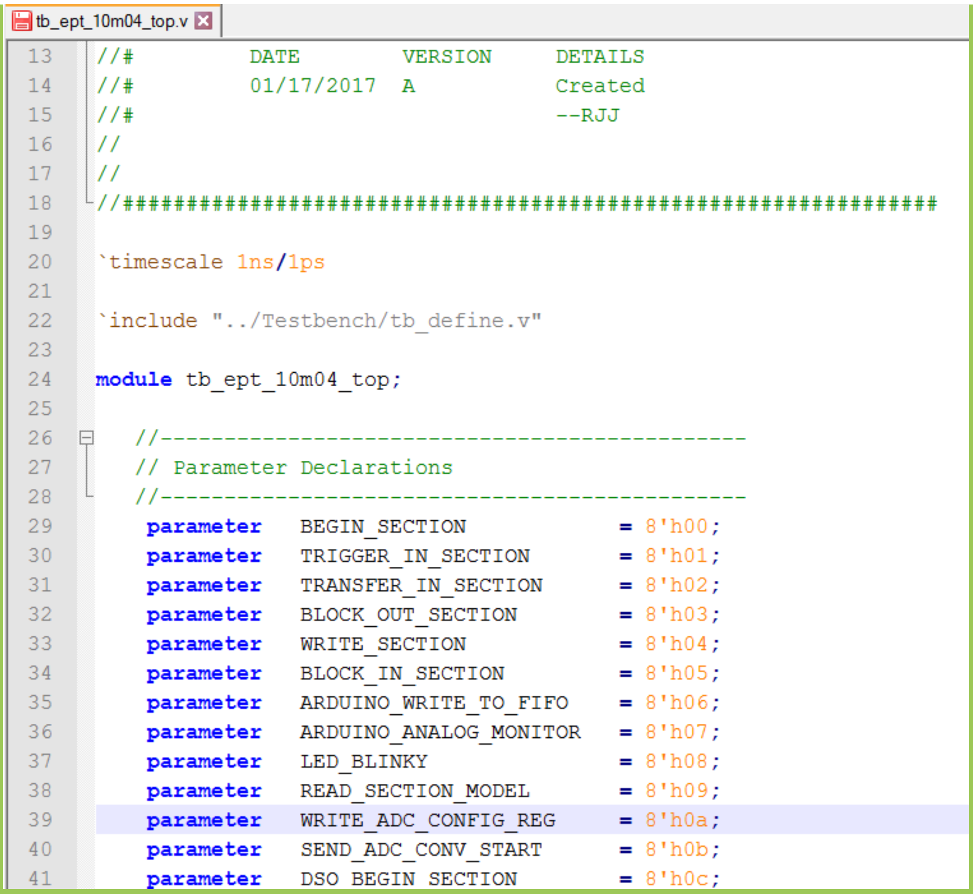
```
//# Revision History:
//#          DATE         VERSION      DETAILS
//#          1/15/17      1            Created
//#
//#
//#
//###############################################################


//***************************************************************
//* 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/clock, reset, push buttons, communications, etc… Open the Testbench folder and notice the two files in it.


Computer ▸ jollyr1 (\\laxm1bnas51\home02) (H:) ▸ Jolly ▸ Code_FPGA ▸ EPT_10M04_AF_Flip_Flop ▸ EPT_10M04_AF_Flip_Flop ▸ Testbench
Organize ▾
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

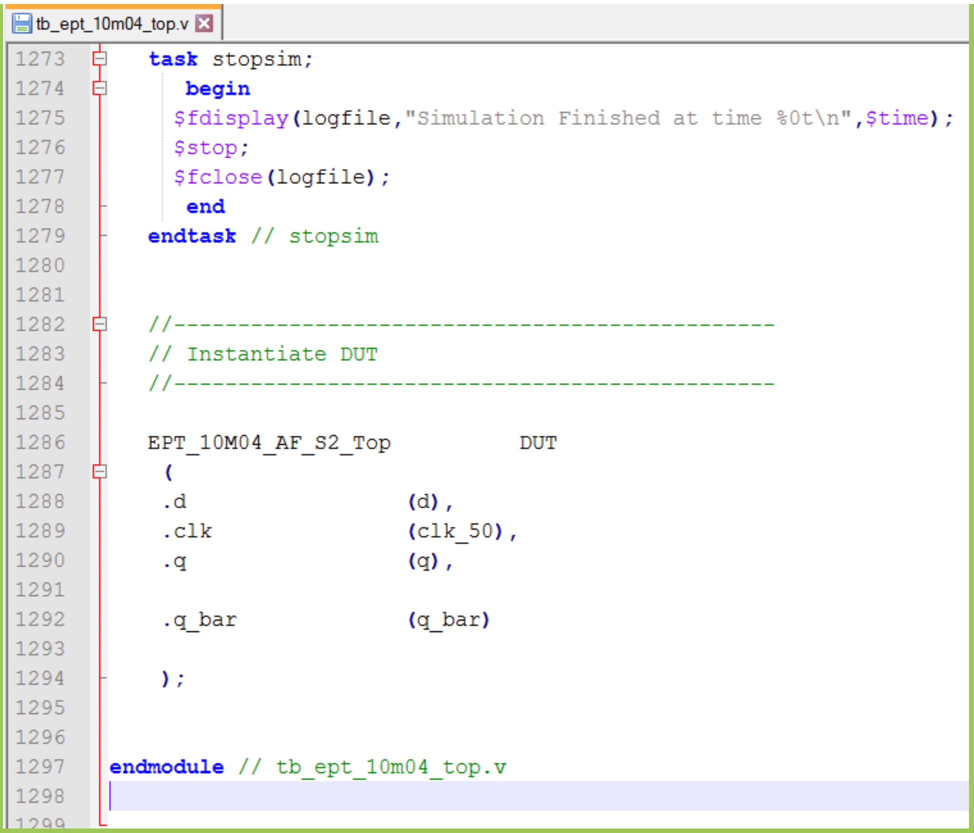
Name	Date modified	Type	Size
tb_define.v	1/17/2017 9:51 PM	Text Document	9 KB
tb_ept_10m04_top.v	1/18/2017 9:43 AM	Text Document	42 KB

- 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.



```
//#         DATE        VERSION     DETAILS
//#         01/17/2017  A           Created
//#                                 --RJJ
//
//
//################################################################

`timescale 1ns/1ps

`include "../Testbench/tb_define.v"

module tb_ept_10m04_top;

   //----------------------------------------------
   // Parameter Declarations
   //----------------------------------------------
    parameter   BEGIN_SECTION            = 8'h00;
    parameter   TRIGGER_IN_SECTION       = 8'h01;
    parameter   TRANSFER_IN_SECTION      = 8'h02;
    parameter   BLOCK_OUT_SECTION        = 8'h03;
    parameter   WRITE_SECTION            = 8'h04;
    parameter   BLOCK_IN_SECTION         = 8'h05;
    parameter   ARDUINO_WRITE_TO_FIFO    = 8'h06;
    parameter   ARDUINO_ANALOG_MONITOR   = 8'h07;
    parameter   LED_BLINKY               = 8'h08;
    parameter   READ_SECTION_MODEL       = 8'h09;
    parameter   WRITE_ADC_CONFIG_REG     = 8'h0a;
    parameter   SEND_ADC_CONV_START      = 8'h0b;
    parameter   DSO_BEGIN_SECTION        = 8'h0c;
```

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.



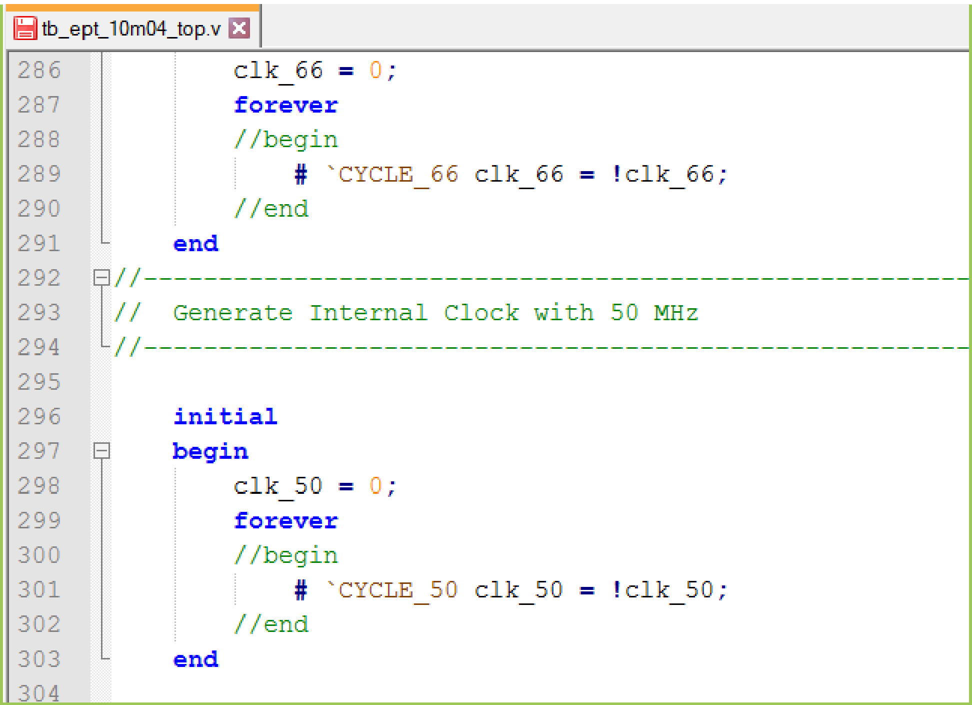
```
task stopsim;
  begin
 $fdisplay(logfile,"Simulation Finished at time %0t\n",$time);
 $stop;
 $fclose(logfile);
  end
endtask // stopsim


//-----------------------------------------------
// 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

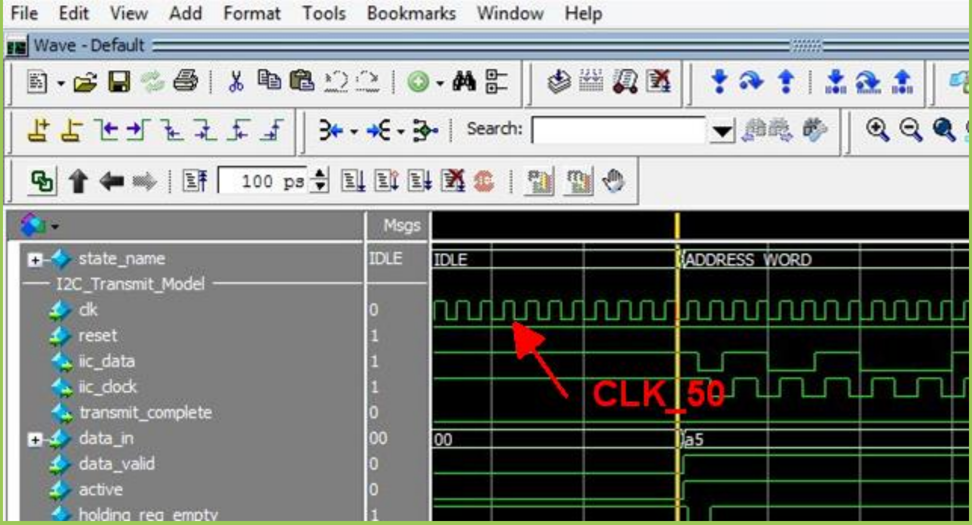
Using the leif instantiation module, we have now connected the testbench with the user code. When ModelSim starts the simulation process, testbench will control the stimulus to the inputs and accept the outputs from the user code.

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

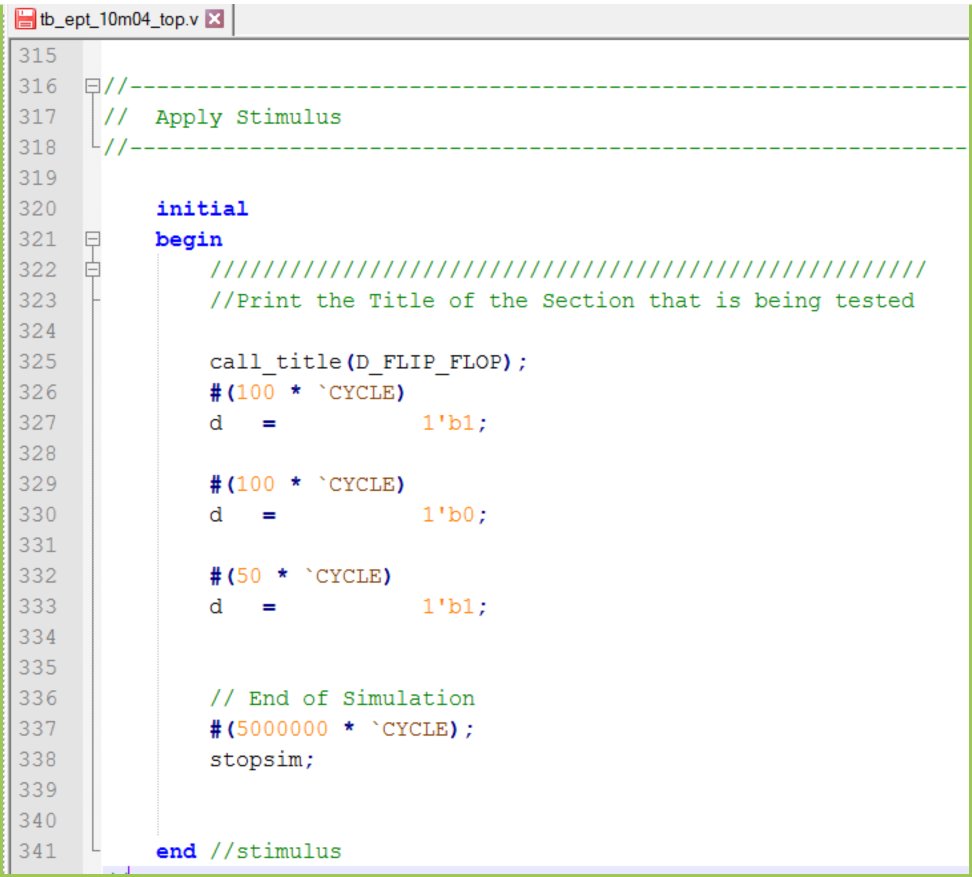


```
            clk_66 = 0;
            forever
            //begin
                # `CYCLE_66 clk_66 = !clk_66;
            //end
        end
//-----------------------------------------------------------
//  Generate Internal Clock with 50 MHz
//-----------------------------------------------------------

        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.



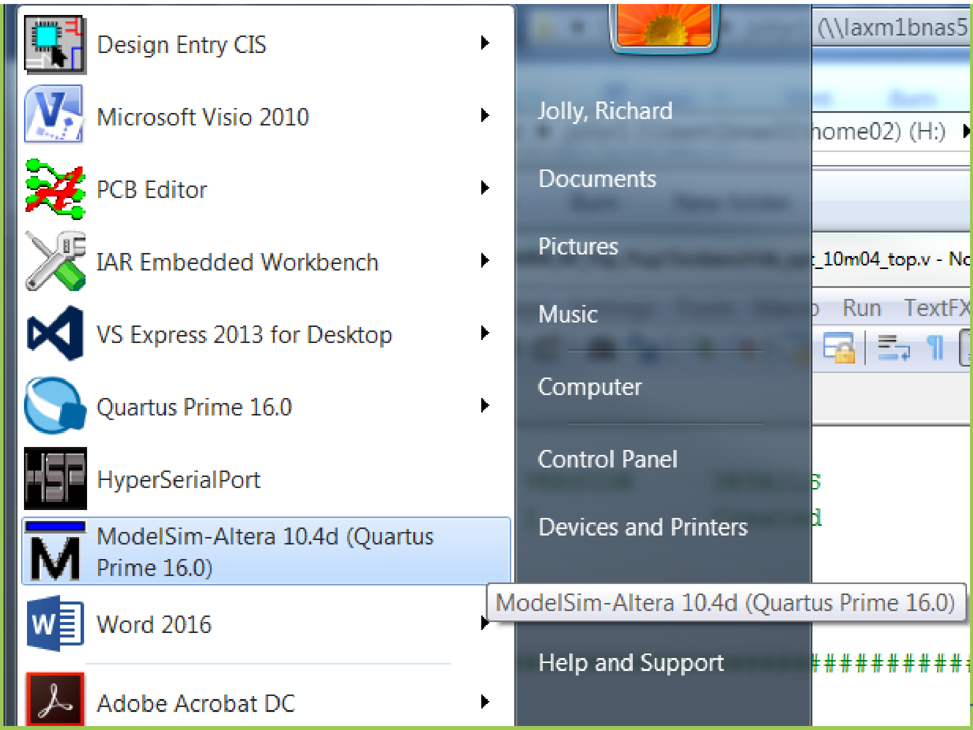
Next, add the stimulus for ‘d’ input. We do this using the ‘initial’ block. Everything between the ‘begin’ ‘end’ keywords is executed once per simulation.



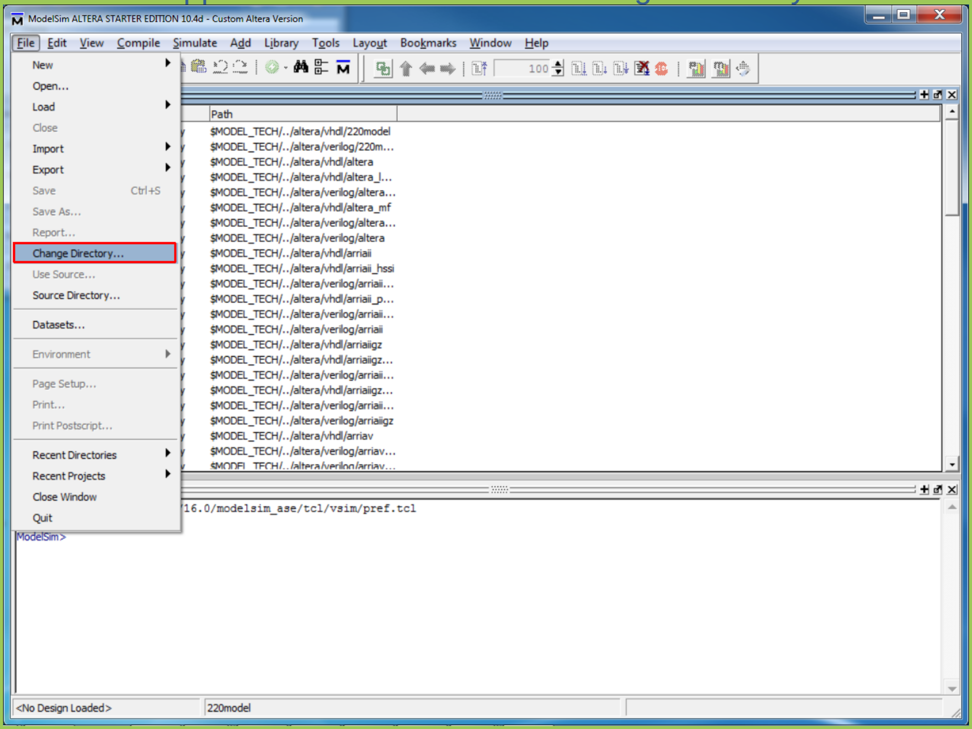
```
//-----------------------------------------------------------------
//  Apply Stimulus
//-----------------------------------------------------------------

    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.

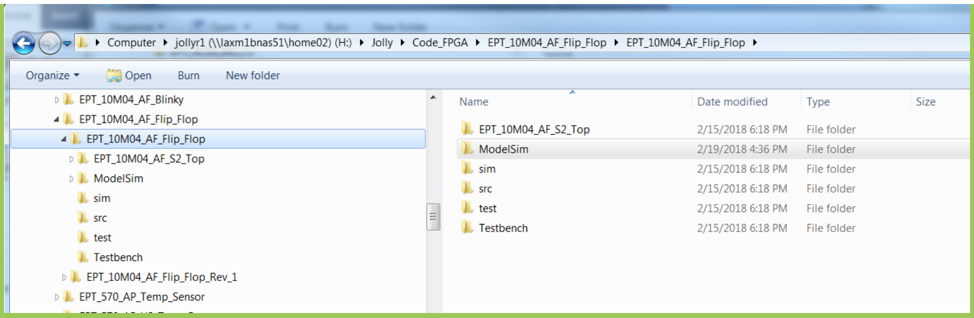
Next, get the ModelSim loaded up on your laptop/PC. Follow the install guide on how to install the ModelSim application.



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

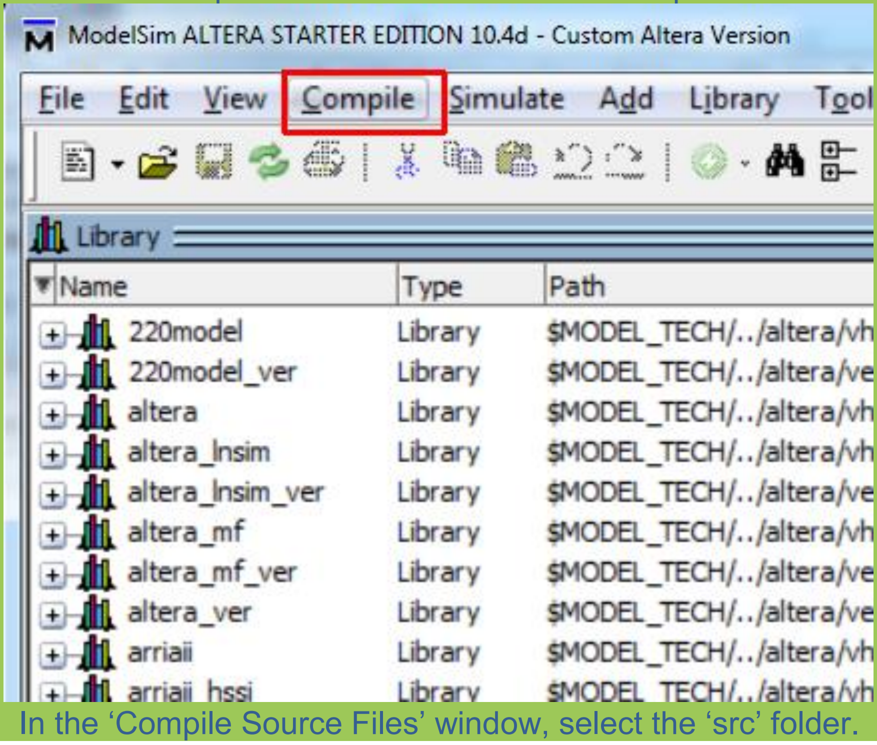


At the dialog box, locate the ModelSim folder under the EPT_10M04_AF_S2_TOP project.



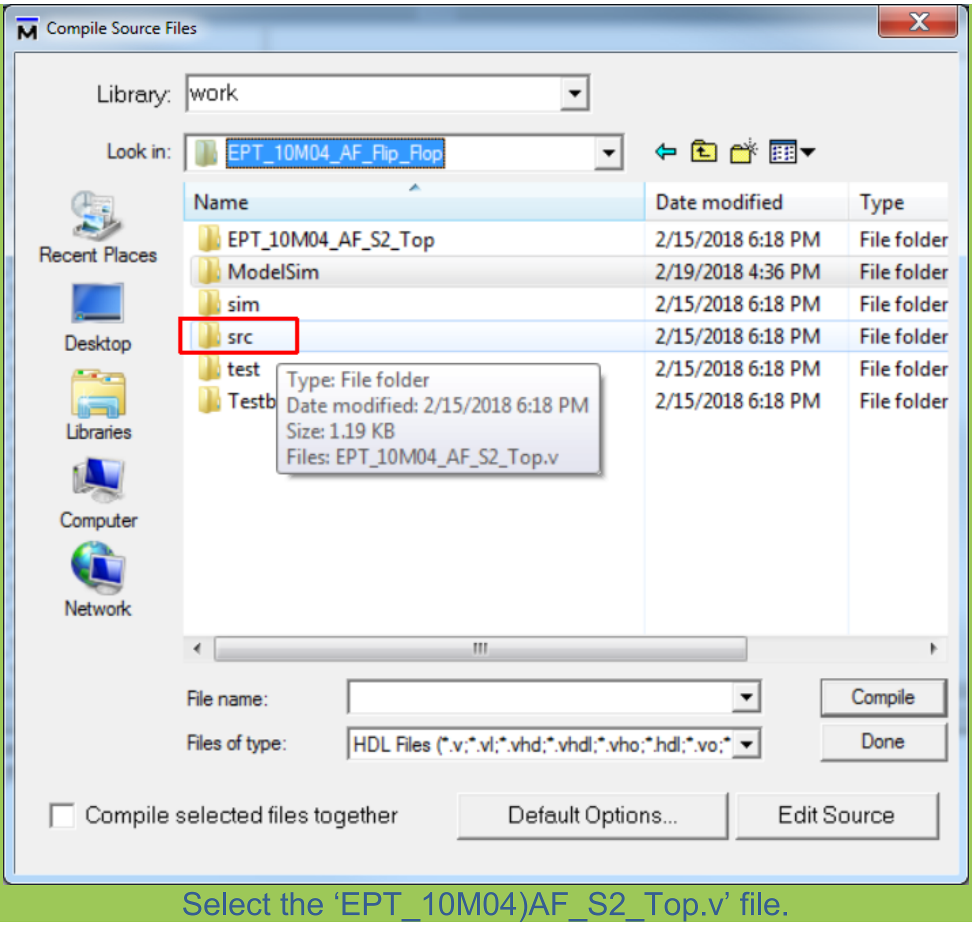
Name	Date modified	Type	Size
EPT_10M04_AF_S2_Top	2/15/2018 6:18 PM	File folder	
ModelSim	2/19/2018 4:36 PM	File folder	
sim	2/15/2018 6:18 PM	File folder	
src	2/15/2018 6:18 PM	File folder	
test	2/15/2018 6:18 PM	File folder	
Testbench	2/15/2018 6:18 PM	File folder	

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.



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

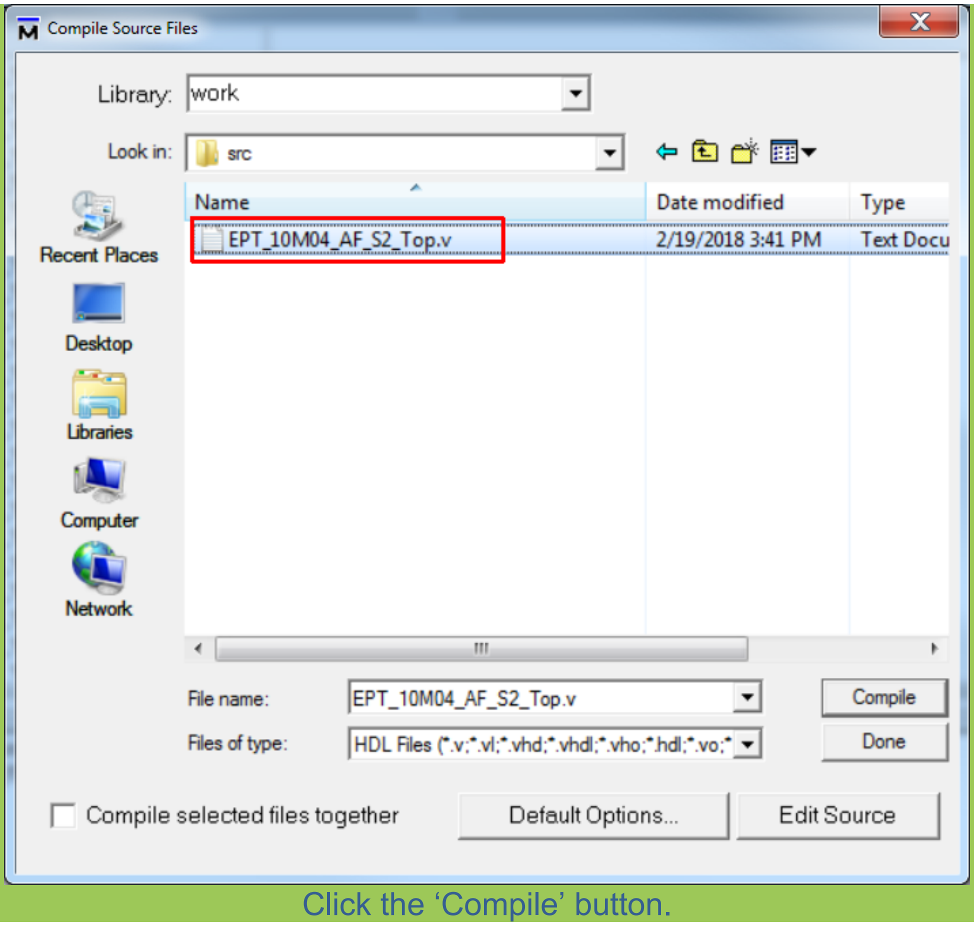
Type: File folder
Date modified: 2/15/2018 6:18 PM
Size: 1.19 KB
Files: EPT_10M04_AF_S2_Top.v

File name:

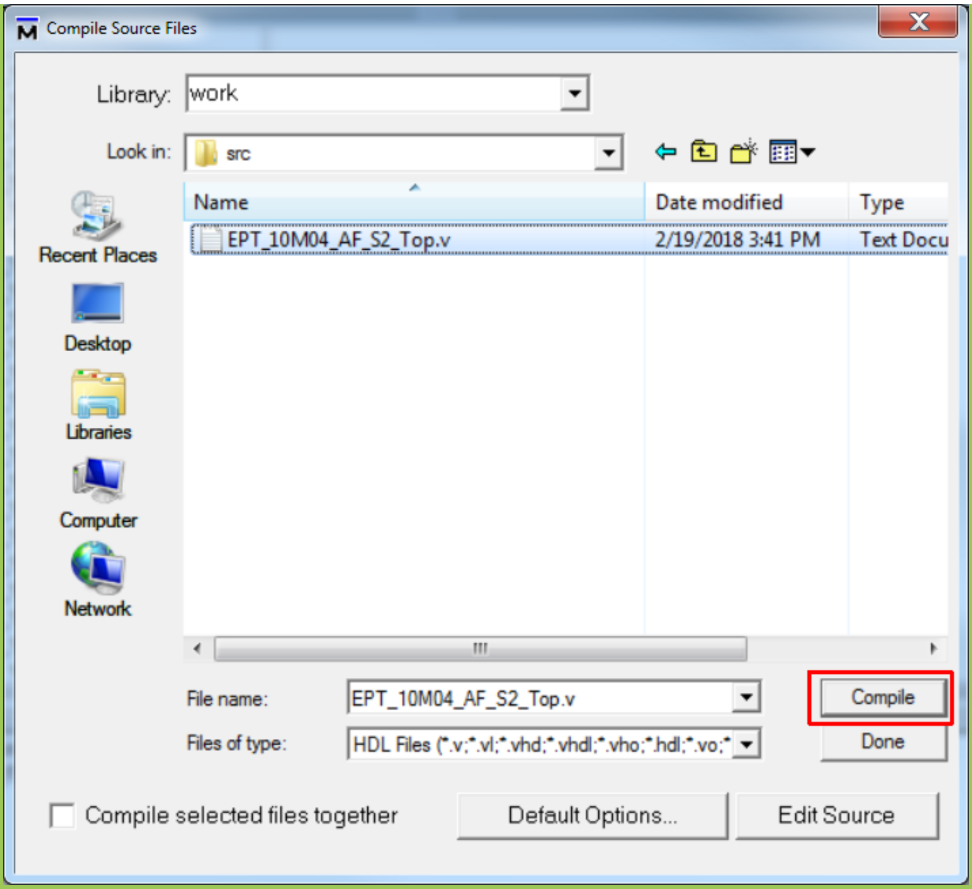
Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

☐ Compile selected files together

Select the ‘EPT_10M04)AF_S2_Top.v’ file.



Click the ‘Compile’ button.



Library: work

Look in: src

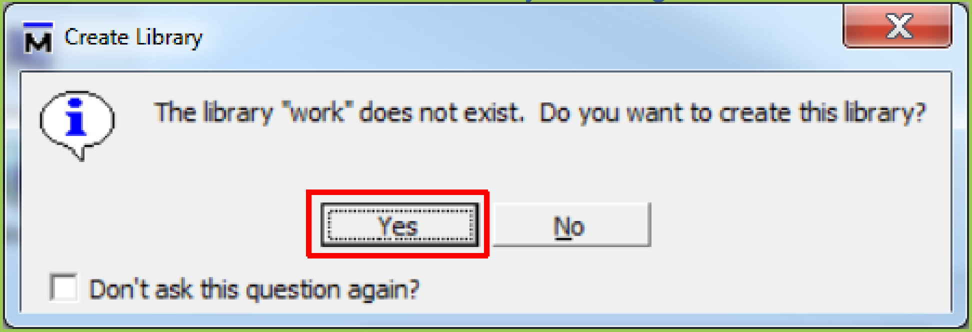
Name	Date modified	Type
EPT_10M04_AF_S2_Top.v	2/19/2018 3:41 PM	Text Docu

File name: EPT_10M04_AF_S2_Top.v

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

☐ Compile selected files together

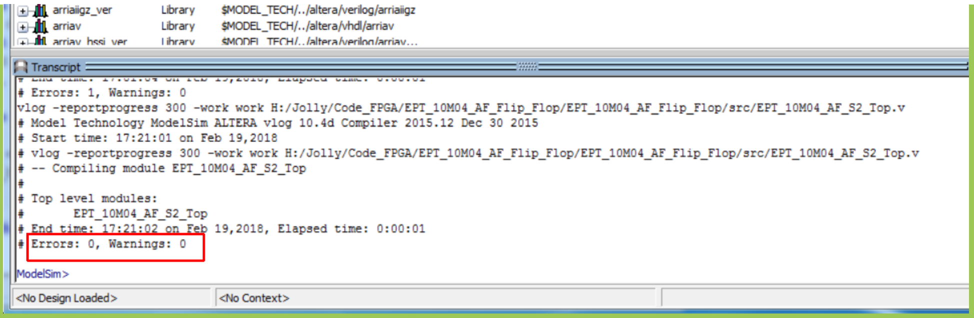
You will receive the ‘Create Library’ message box. Select Yes.



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

☐ Don't ask this question again?

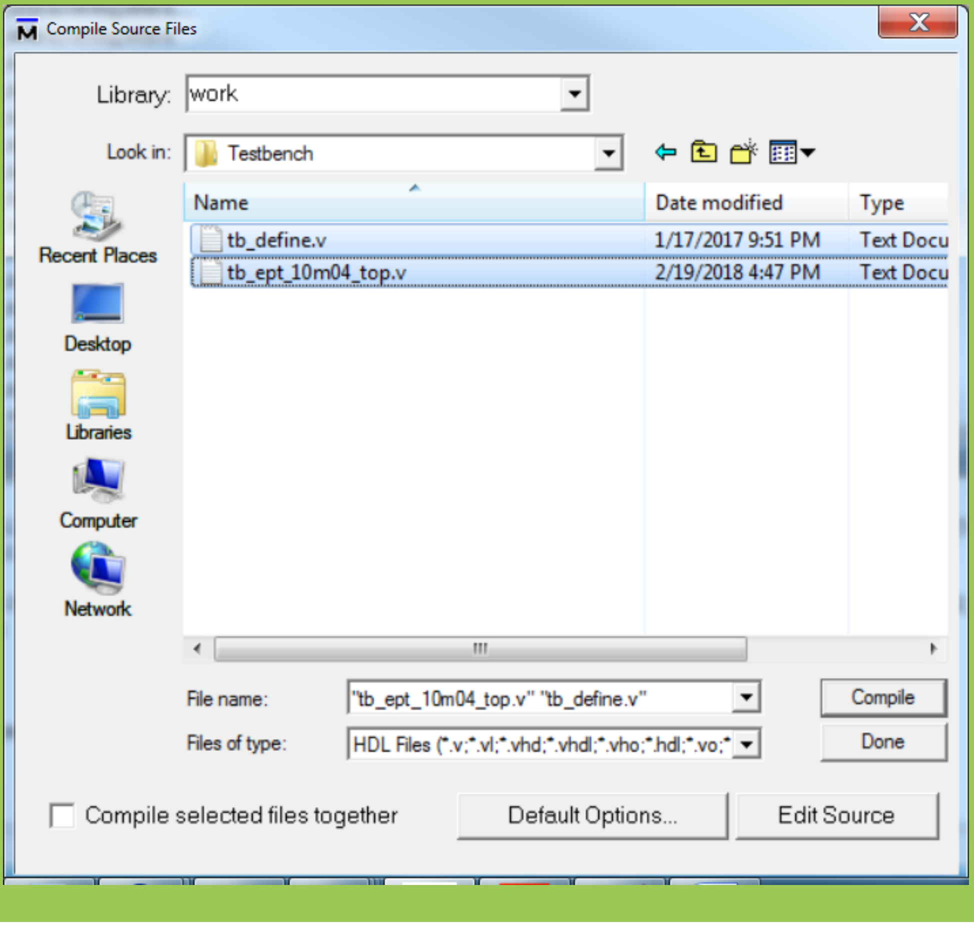
After the file is compiled, the log will indicate the status of compilation.


arriaiigz_ver Library $MODEL_TECH/../altera/verilog/arriaiigz
arriav Library $MODEL_TECH/../altera/vhdl/arriav
Transcript

```
# 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

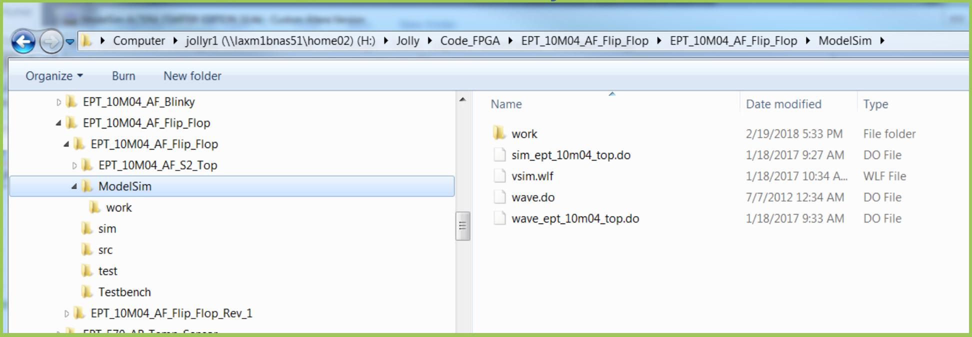
Name	Date modified	Type
tb_define.v	1/17/2017 9:51 PM	Text Docu
tb_ept_10m04_top.v	2/19/2018 4:47 PM	Text Docu

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

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

☐ Compile selected files together

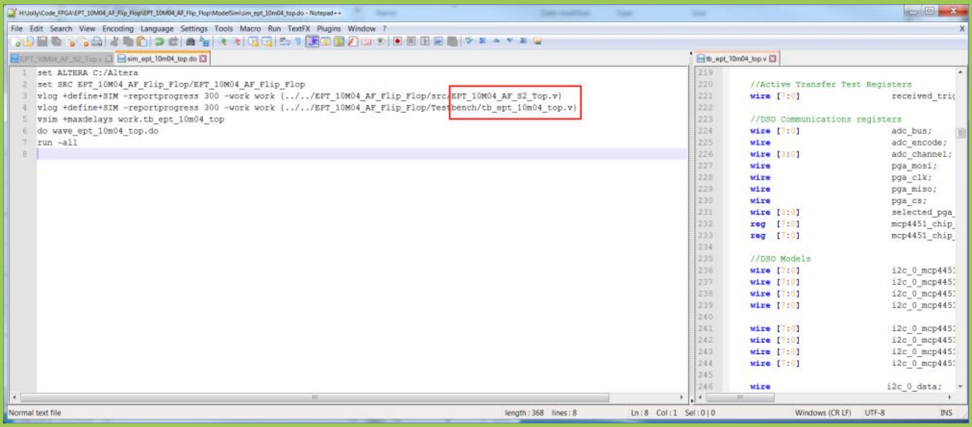
Note, you will not get receive the message box asking to create the 'work' folder this time. After the compilation has completed, we are ready to run the Simulation. The simulation enviroment is controlled by the *.do files. The *.do files act as makefiles to control which files get simulated and add 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 ▸
Organize
Burn
New folder
EPT_10M04_AF_Blinky
EPT_10M04_AF_Flip_Flop
EPT_10M04_AF_Flip_Flop
EPT_10M04_AF_S2_Top
ModelSim
work
sim
src
test
Testbench
EPT_10M04_AF_Flip_Flop_Rev_1

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.

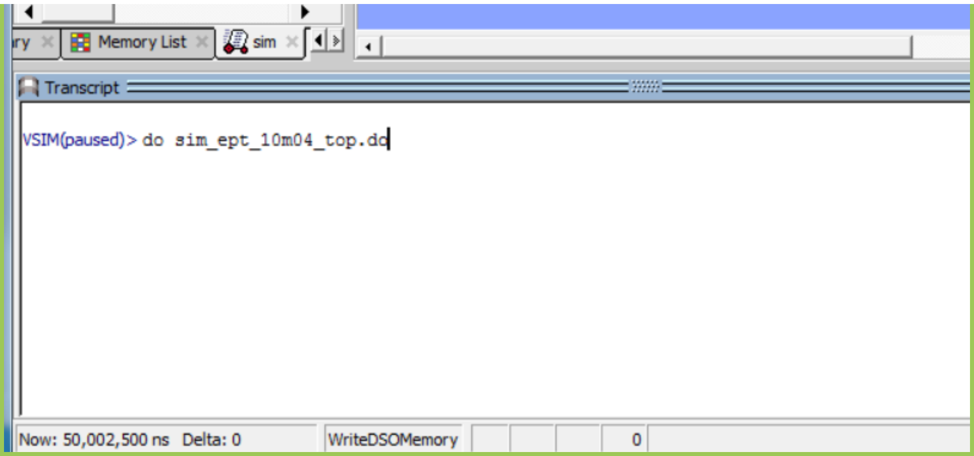
When we look inside the sim_ept_10m04_top.do file we see the two files to simulate.


H:\Jolly\Code_FPGA\EPT_10M04_AF_Flip_Flop\EPT_10M04_AF_Flip_Flop\ModelSim\sim_ept_10m04_top.do - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run TextFX Plugins Window ?
EPT_10M04_AF_S2_Top.v
sim_ept_10m04_top.do
tb_ept_10m04_top.v
Normal text file
length : 368 lines : 8
Ln : 8 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS

```
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
```

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.

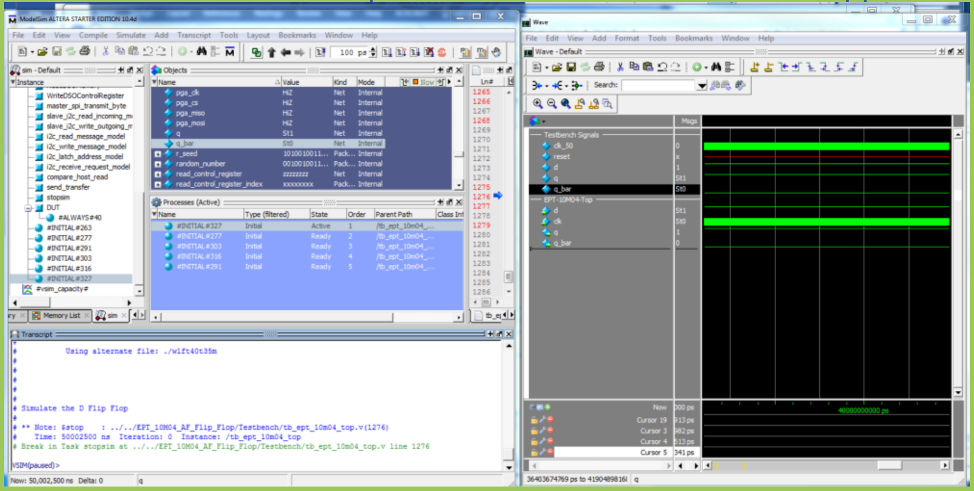
Start the simulation by typing do sim_ept_10m04_top.do into the command window.



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

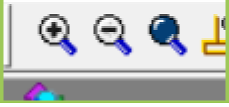
```
Now: 50,002,500 ns   Delta: 0
```

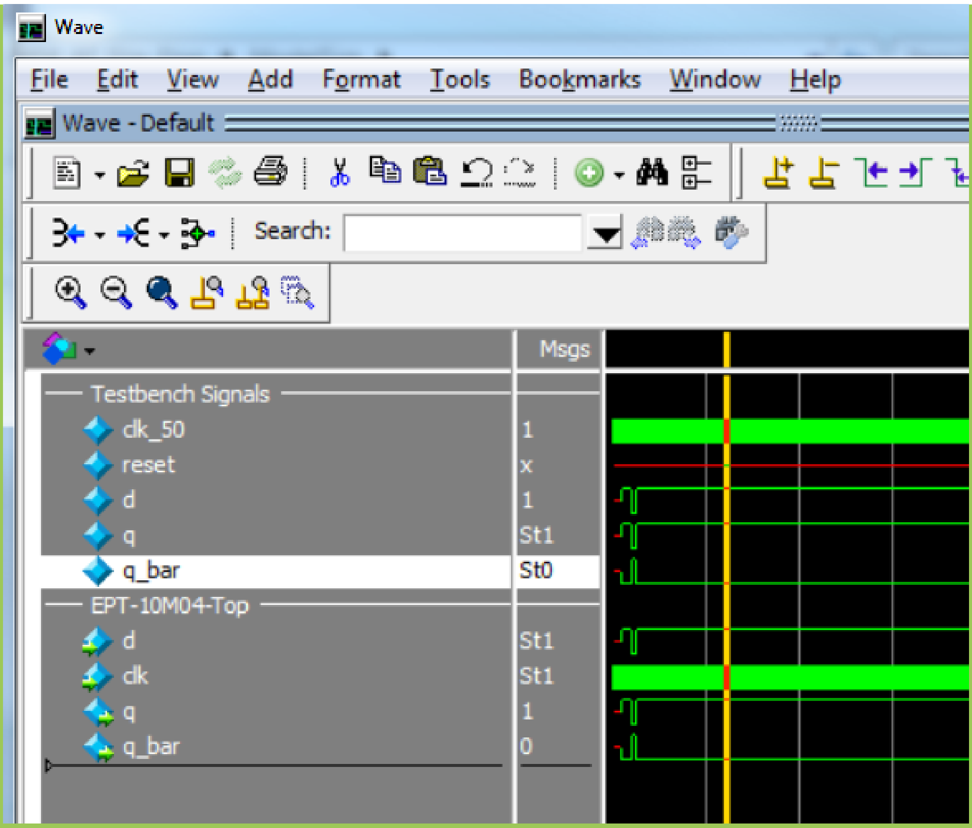
When the simulator completes its run, the Wave window appears.



```
#          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)>
```

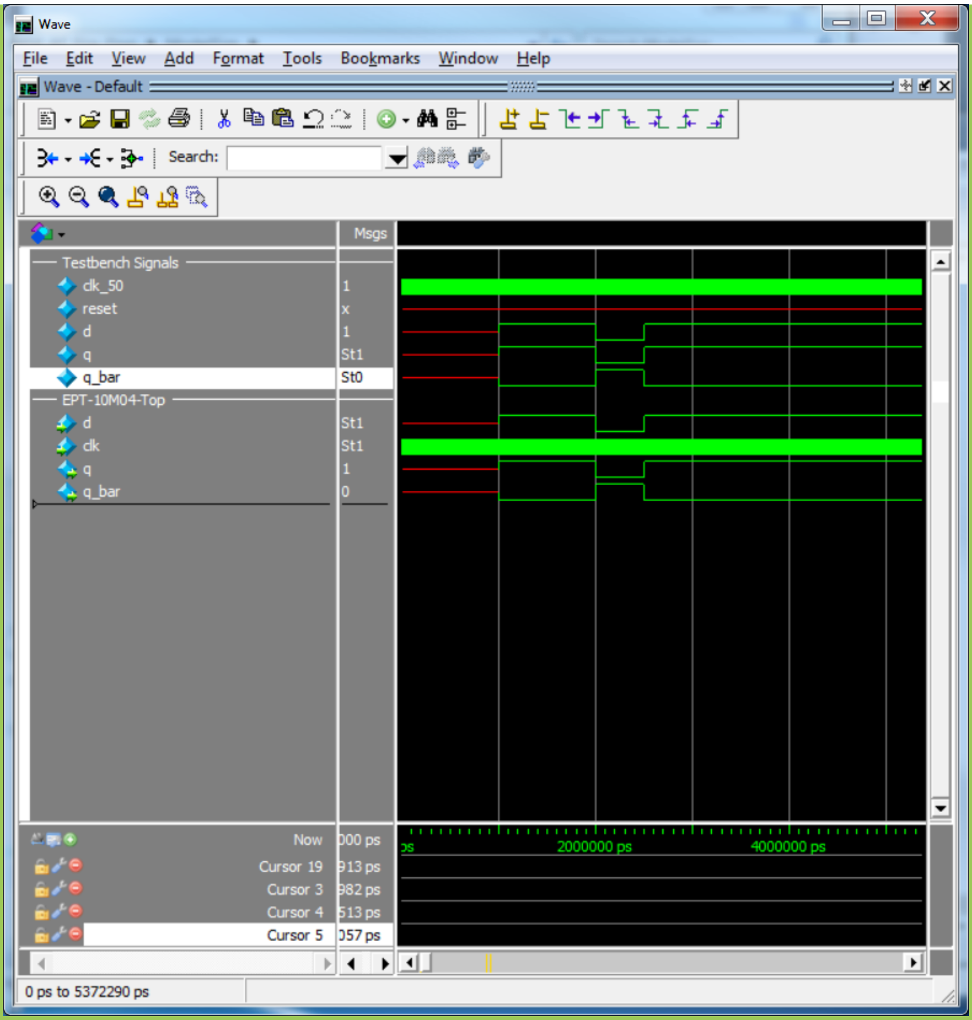
Zoom into the first 10 microseconds of the simulation usin the Zoom buttons.





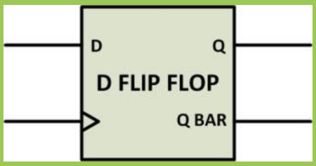
Signal	Msgs
Testbench Signals	
clk_50	1
reset	x
d	1
q	St1
q_bar	St0
EPT-10M04-Top	
d	St1
clk	St1
q	1
q_bar	0

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

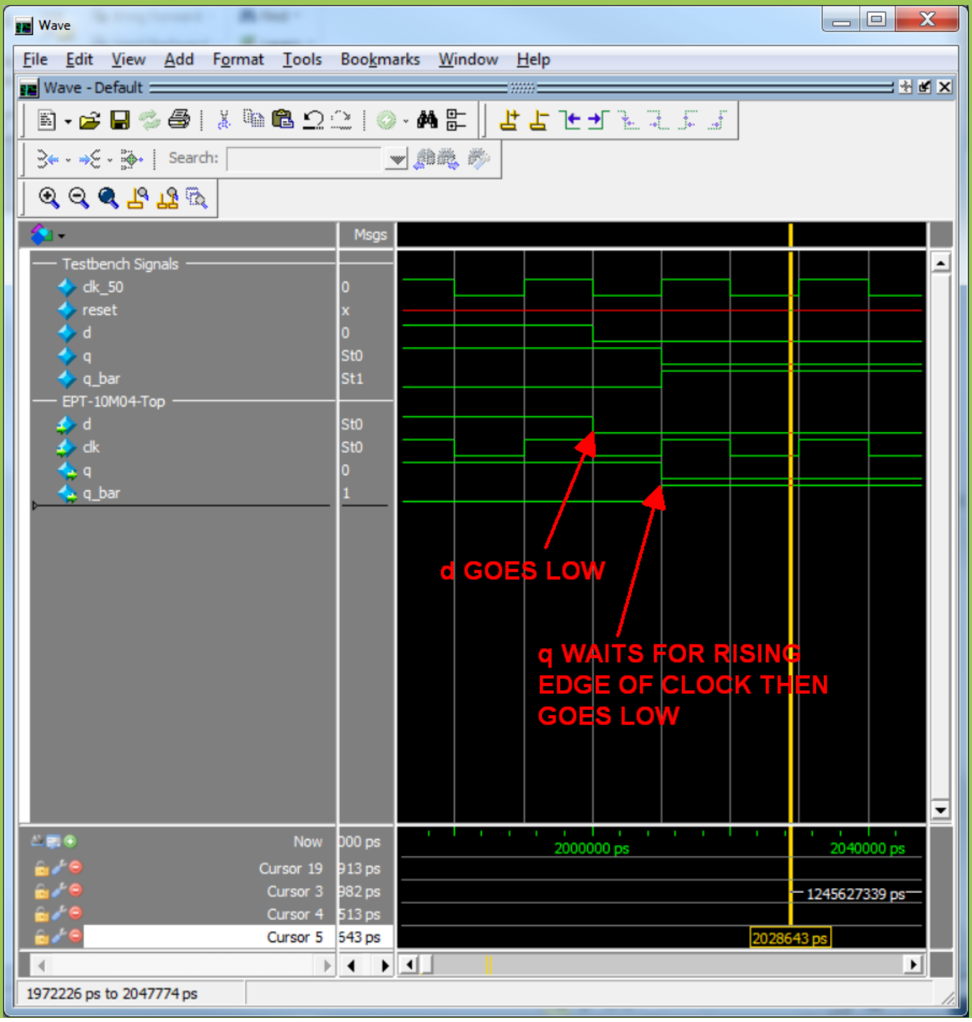


We can see after first delay that 'd' signal is asserted high. Before this assertion the signal is not defined in the simulation, so technically it is a Don't Care (indicated by the red 'floating' line). Once the 'd' is asserted, the delay is added by the simulator. After this delay the 'd' signal is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

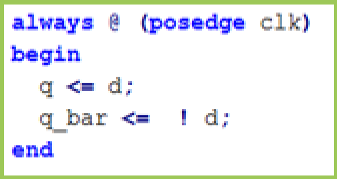
Now that we know what the simulator is doing, we can exam the user code, the D Flip Flop.



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.



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.

RTL DESCRIPTION

Many engineers who want to learn this language, very often ask this question, how much time will it take to learn Verilog? Well my answer to them is **"It may take no more than one week, if you happen to know at least one programming language"**.

Design Styles

Verilog, like any other hardware description language, permits a design in either Bottom-up or Top-down methodology.

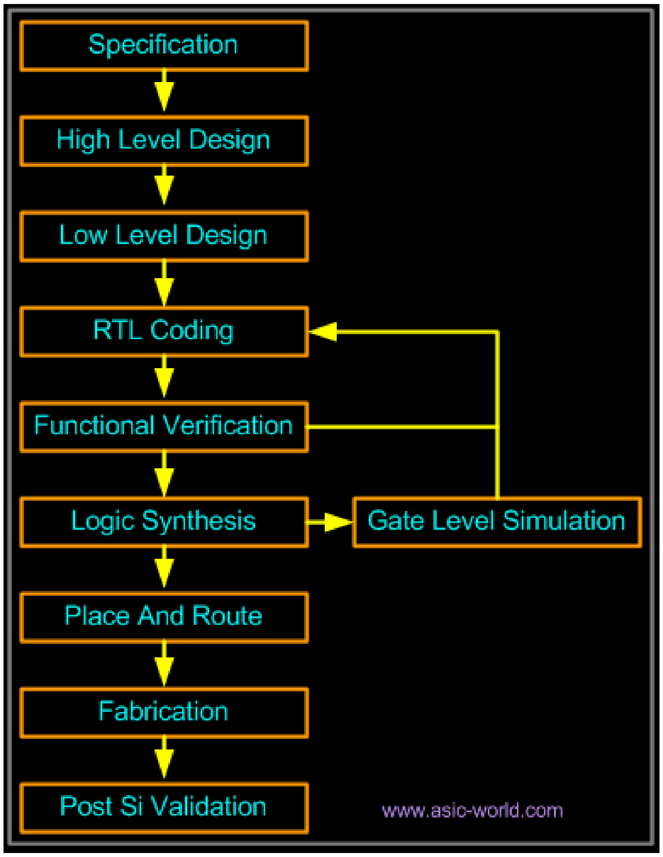
Bottom-Up Design

The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates (refer to the Digital Section for more details). With the increasing complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom-up designs have to give way to new structural, hierarchical design methods. Without these new practices it would be impossible to handle the new complexity.

Top-Down Design

The desired design-style of all designers is the top-down one. A real top-down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are a mix of both methods, implementing some key elements of both design styles.

Figure shows a Top-Down design approach.



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

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".

Gate Level

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). *Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.*

MODULES, PORTS, DATA TYPES AND OPERATORS

Every new learner's dream is to understand Verilog in one day, at least enough to use it. The next few pages are my attempt to make this dream a reality. There will be some theory and examples followed by some exercises. This tutorial will not teach you how to program; it is designed for those with some programming experience. Even though Verilog executes different code blocks concurrently as opposed to the sequential execution of most programming languages, there are still many parallels. Some background in digital design is also helpful.

Life before Verilog was a life full of schematics. Every design, regardless of complexity, was designed through schematics. They were difficult to verify and error-prone, resulting in long, tedious development cycles of design, verification... design, verification... design, verification...

When Verilog arrived, we suddenly had a different way of thinking about logic circuits. The Verilog design cycle is more like a traditional programming one, and it is what this tutorial will walk you through. Here's how it goes:

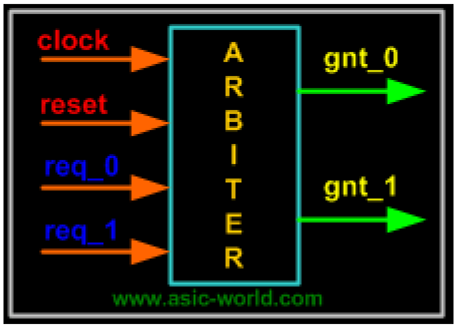
- Specifications (specs)
- High level design
- Low level (micro) design
- RTL coding
- Verification
- Synthesis.

First on the list is **specifications** - what are the restrictions and requirements we will place on our design? What are we trying to build? For this tutorial, we'll be building a two agent arbiter: a device that selects among two agents competing for mastership. Here are some specs we might write up.

- Two agent arbiter.
- Active high asynchronous reset.
- Fixed priority, with agent 0 having priority over agent 1
- Grant will be asserted as long as request is asserted.

Once we have the specs, we can draw the block diagram, which is basically an abstraction of the data flow through a system (what goes into or comes out of the black boxes?). Since the example that we have taken is a simple one, we can have a block diagram as shown below. We don't worry about what's inside the magical black boxes just yet.

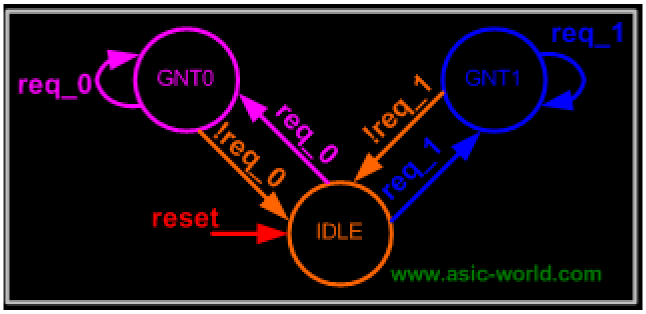
Block diagram of arbiter



Now, if we were designing this machine without Verilog, the standard procedure would dictate that we draw a state machine. From there, we'd make a truth table with state transitions for each flip-flop. And after that we'd draw Karnaugh maps, and from K-maps we could get the optimized circuit. This method works just fine for small designs, but with large designs this flow becomes complicated and error prone. This is where Verilog comes in and shows us another way.

Low level design

To see how Verilog helps us design our arbiter, let's go on to our state machine - now we're getting into the low-level design and peeling away the cover of the previous diagram's black box to see how our inputs affect the machine.



Each of the circles represents a **state** that the machine can be in. Each state corresponds to an output. The arrows between the states are state transitions, labeled by the event that causes the transition. For instance, the leftmost orange arrow means that if the machine is in state GNT0 (outputting the signal that corresponds to GNT0) and receives an input of !req_0, the machine moves to state IDLE and outputs the signal that corresponds to that. This state machine describes all the logic of the system that you'll need. The next step is to put it all in Verilog.

Modules

We'll need to backtrack a bit to do this. If you look at the arbiter block in the first picture, we can see that it has got a name ("arbiter") and input/output ports (req_0, req_1, gnt_0, and gnt_1).

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.

Code of module "arbiter"

If you look closely at the arbiter block we see that there are arrow marks, (incoming for inputs and outgoing for outputs). In Verilog, after we have declared the module name and port names, we can define the direction of each port. (version note: In Verilog 2001 we can define ports and port directions at the same time) 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                    ;
```

You could download file one_day1.v here

Here we have only two types of ports, input and output. In real life, we can have bi-directional ports as well. Verilog allows us to define bi-directional ports as "inout."

Bi-Directional Ports Example -

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

How do you 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 Type

What do data types have to do with hardware? Nothing, actually. People just wanted to write one more language that had data types in it. It's completely gratuitous; there's no point.

But wait... hardware does have two kinds of drivers.

(Drivers? What are those?)

A **driver** is a data type which can drive a load. Basically, in a physical circuit, a driver would be anything that electrons can move through/into.

  - Driver that can store a value (example: flip-flop).
- Driver that can not store value, but connects two points (example: wire).

The first type of driver is called a reg in Verilog (short for "register"). The second data type is called a wire (for... well, "wire"). You can refer to tidbits section to understand it better.

There are lots of other data types - for instance, registers can be signed, unsigned, floating point... as a newbie, don't worry about them right now.

Examples :

wire and_gate_output; // "and_gate_output" is a wire that only outputs

reg d_flip_flop_output; // "d_flip_flop_output" is a register; it stores and outputs a value

reg [7:0] address_bus; // "address_bus" is a little-endian 8-bit register

Summary

  - Wire data type is used for connecting two points.
  - Reg data type is used for storing values.
- May god bless the rest of data types. You'll see them someday.

Operators

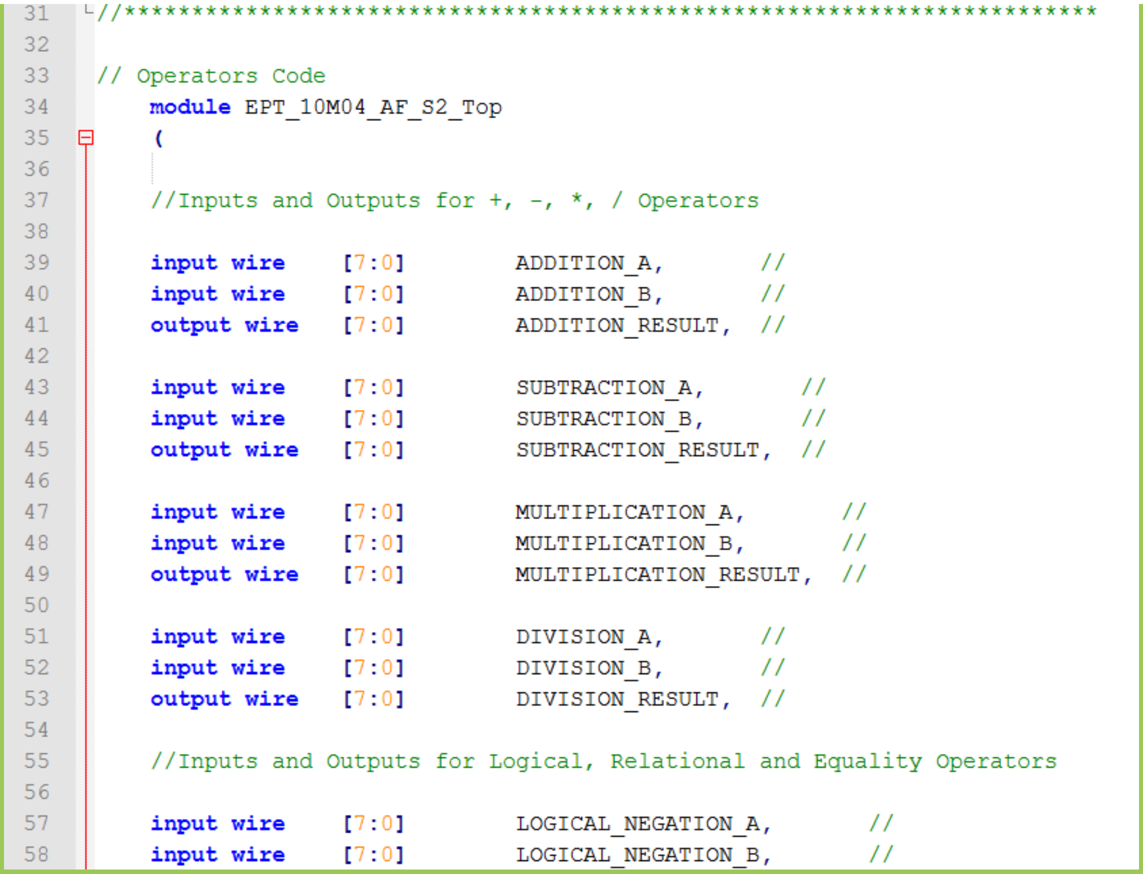
Operators, thankfully, are the same things here as they are in other programming languages. They take two values and compare (or otherwise operate on) them to yield a third result - common examples are addition, equals, logical-and... To make life easier for us, nearly all operators (at least the ones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

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

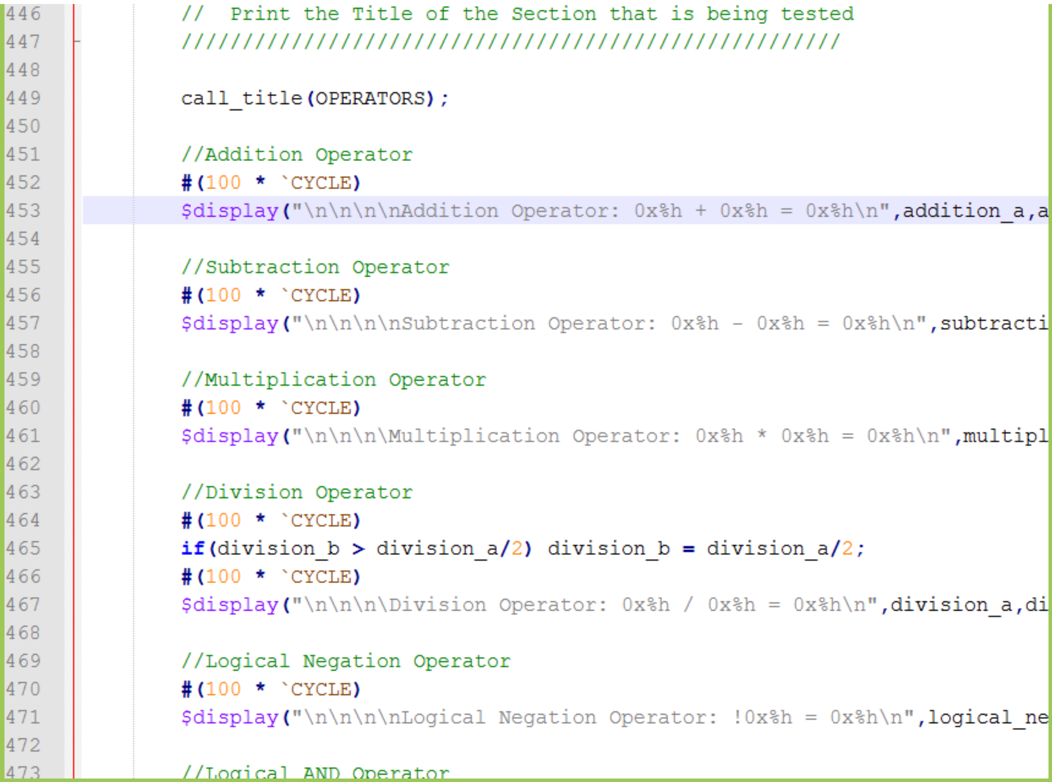
//////////////////////////////////////Lesson #2///

In this lesson, we will use some operators in Verilog code.
First, lets write some user code exercise the operators. Locate the EPT_10M04_AF_Operators folder in the DVD. We will create a 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,       //
```

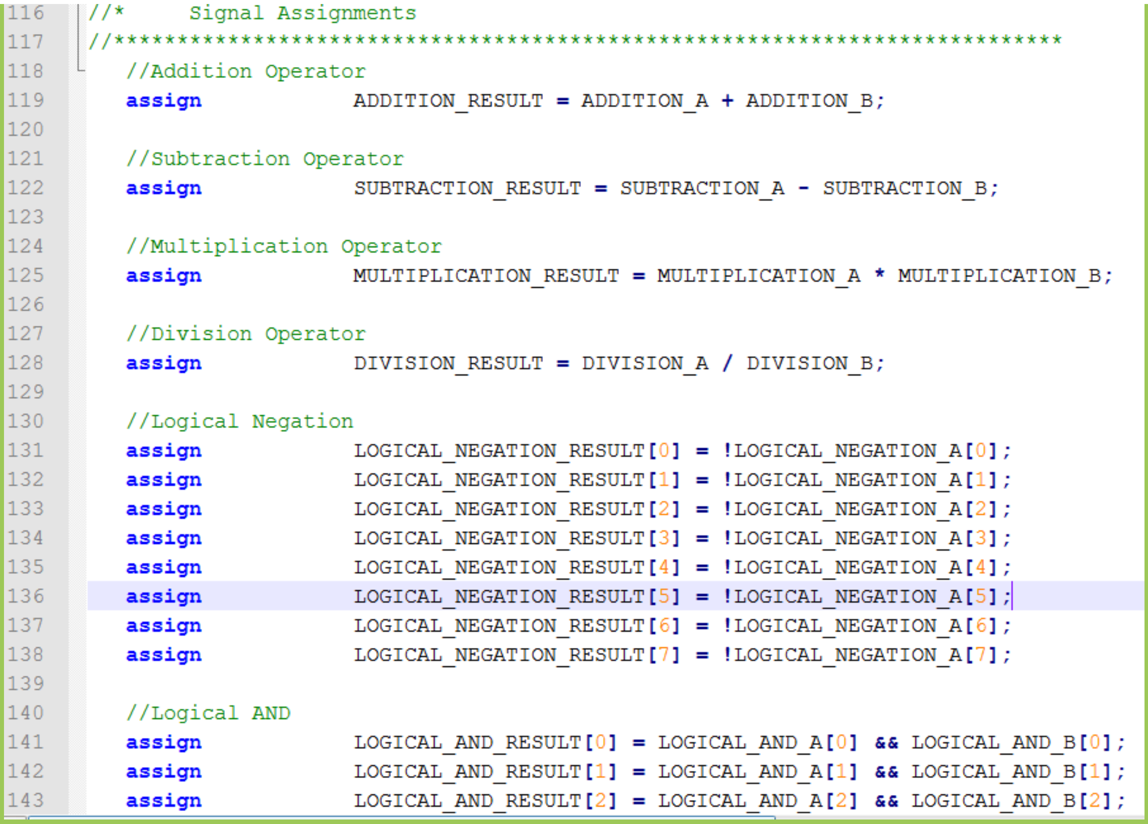
Then we will create a test bench to exercise each operator and display the results.



```
//  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
```

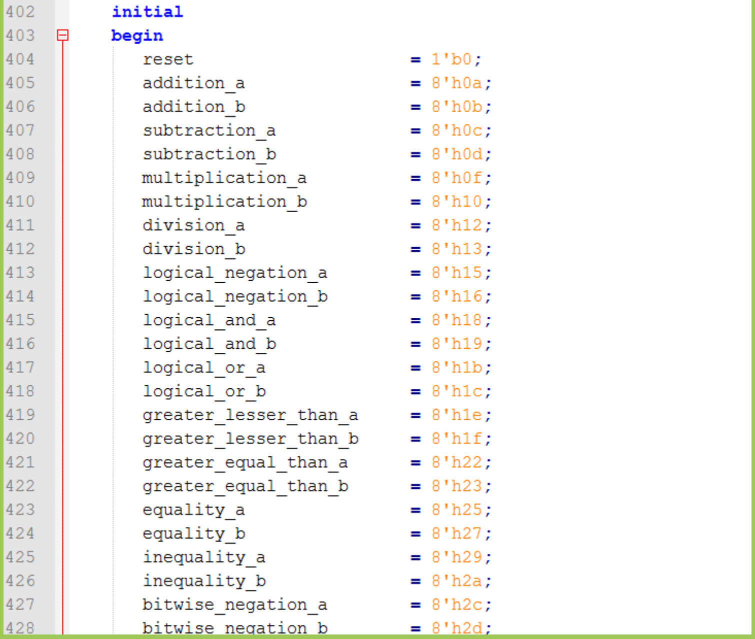
The top level module is created following the syntax rules of Verilog. Use the keyword “module” followed by the name of the module. Then, add an opening parenthesis and define the inputs and outputs. This tutorial will cover in greater detail the keywords “module”, “input”, and “output”, signals, registers 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,

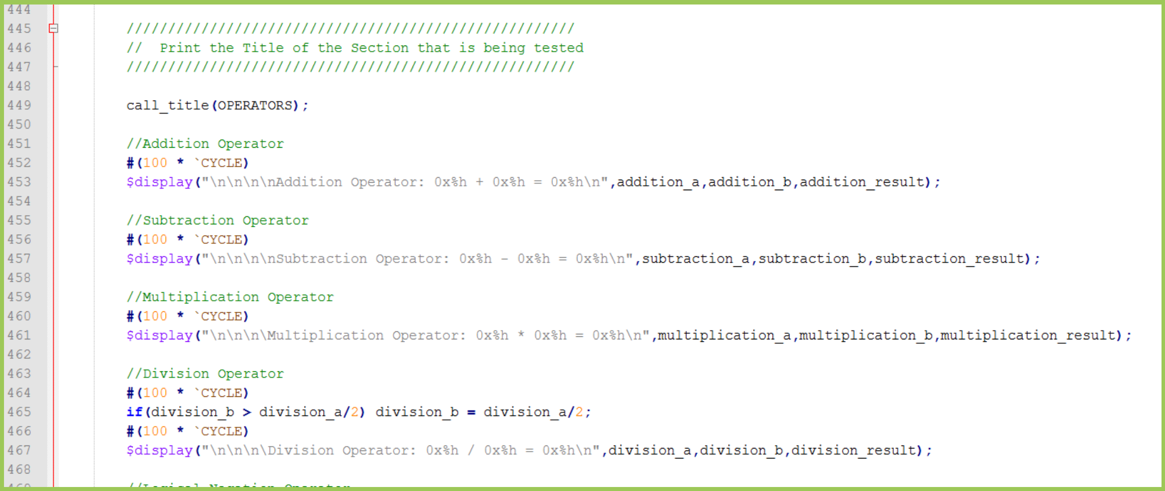


```
//*     Signal Assignments
//****************************************************************************
   //Addition Operator
   assign            ADDITION_RESULT = ADDITION_A + ADDITION_B;

   //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.

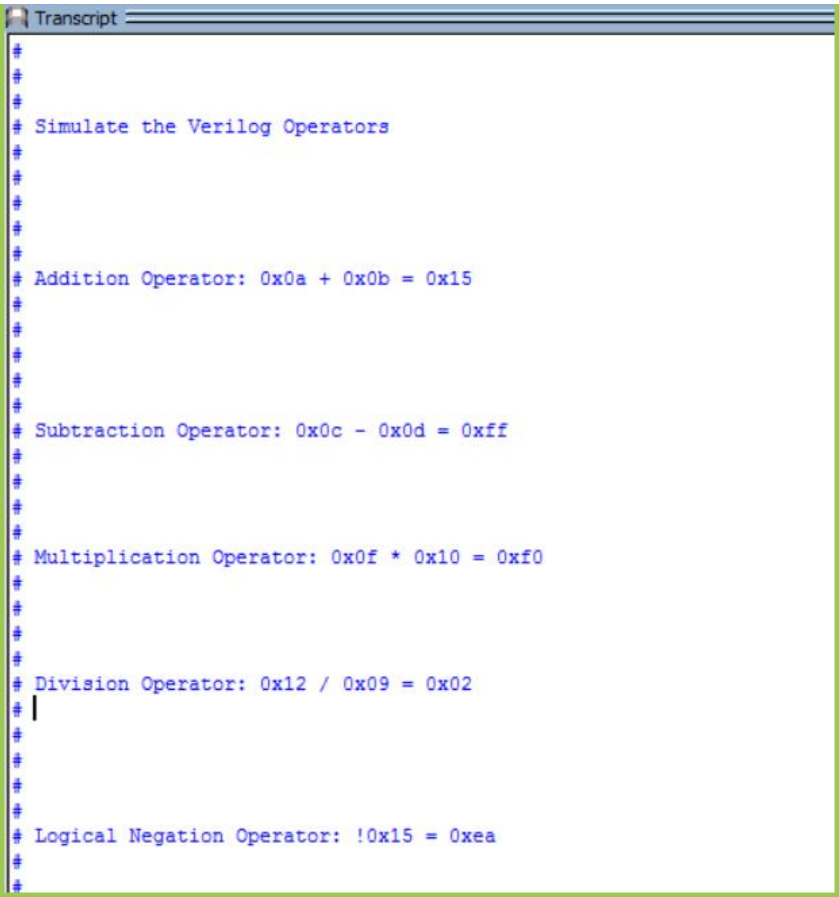


```
initial
begin
    reset                       = 1'b0;
    addition_a                  = 8'h0a;
    addition_b                  = 8'h0b;
    subtraction_a               = 8'h0c;
    subtraction_b               = 8'h0d;
    multiplication_a            = 8'h0f;
    multiplication_b            = 8'h10;
    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;
```



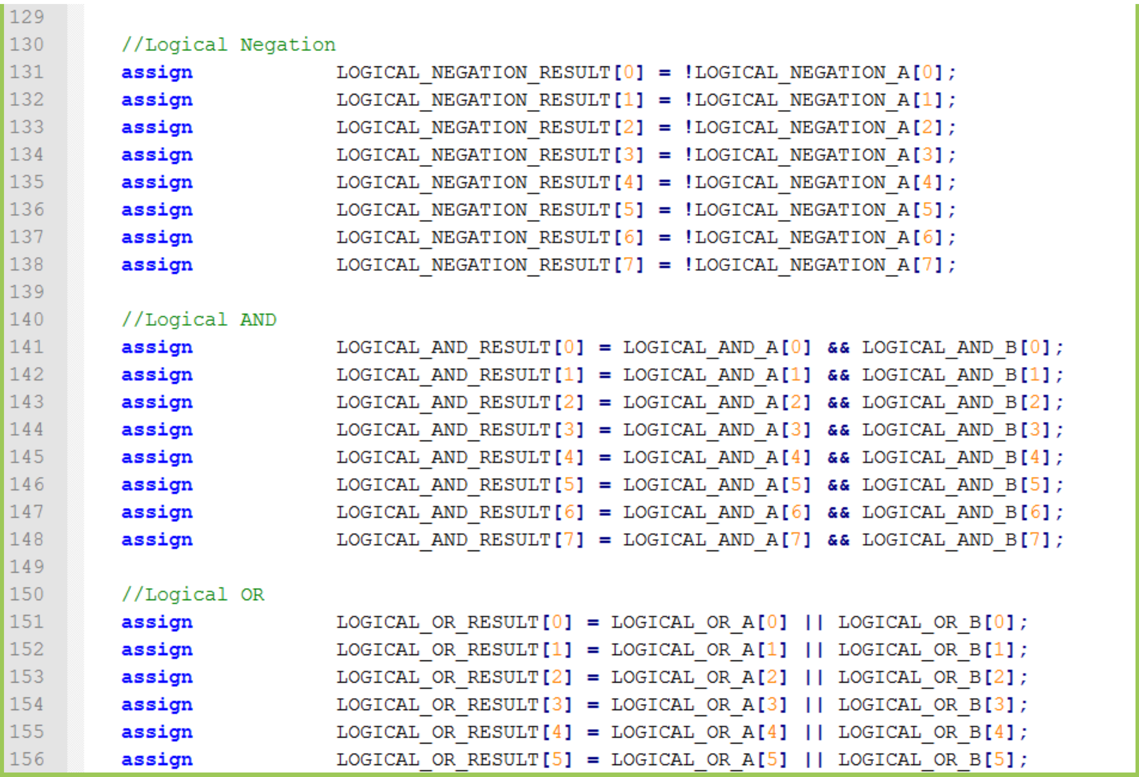
```
    ////////////////////////////////////////////////////////
    //  Print the Title of the Section that is being tested
    ////////////////////////////////////////////////////////

    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:



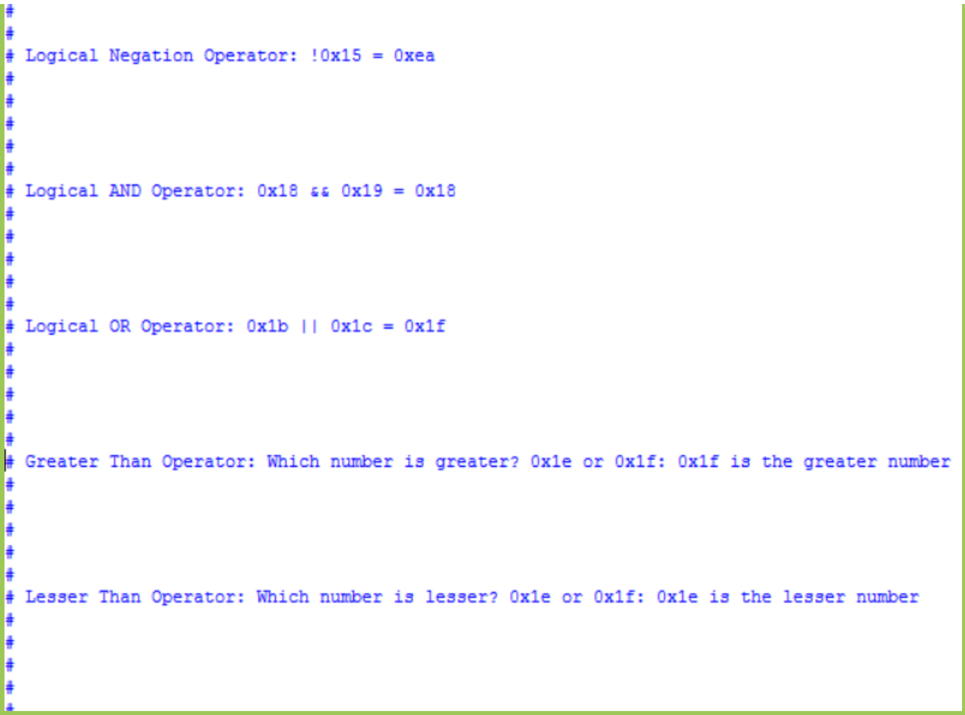
```
#
#
#
# Simulate the Verilog Operators
#
#
#
#
#
# Addition Operator: 0x0a + 0x0b = 0x15
#
#
#
#
#
# Subtraction Operator: 0x0c - 0x0d = 0xff
#
#
#
#
# Multiplication Operator: 0x0f * 0x10 = 0xf0
#
#
#
#
# Division Operator: 0x12 / 0x09 = 0x02
#
#
#
#
#
# Logical Negation Operator: !0x15 = 0xea
#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

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.



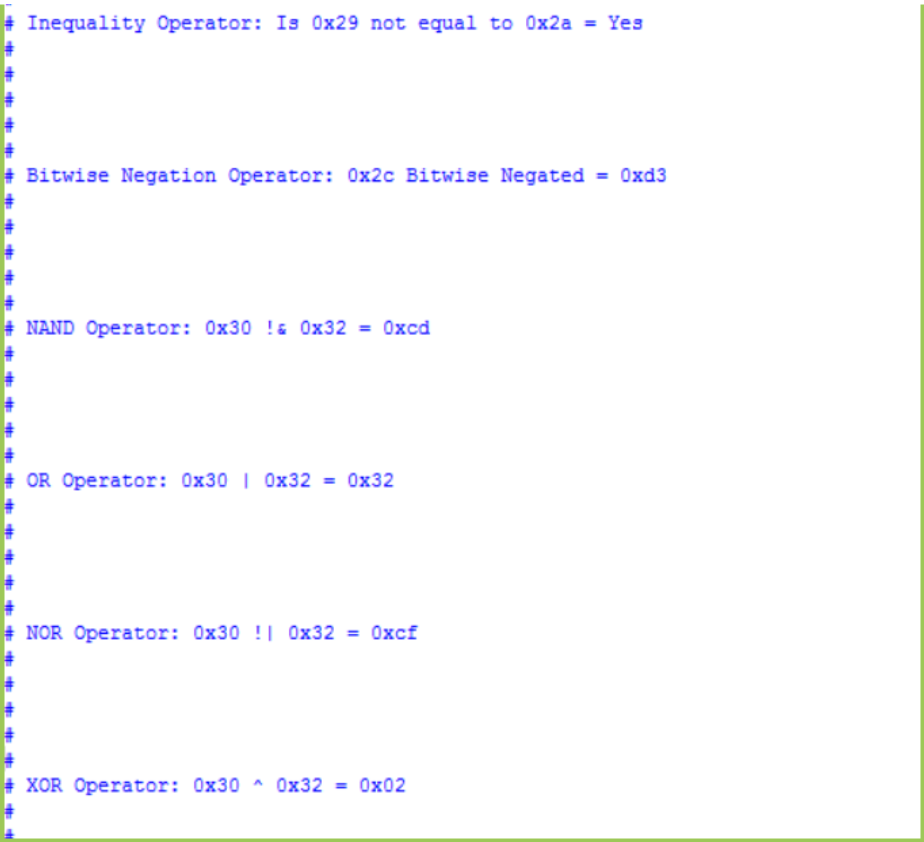
```

//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

Wait, what's this? **if, else, repeat, while, for, case** - it's Verilog that looks exactly like C (and probably whatever other language you're used to program in)! 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.

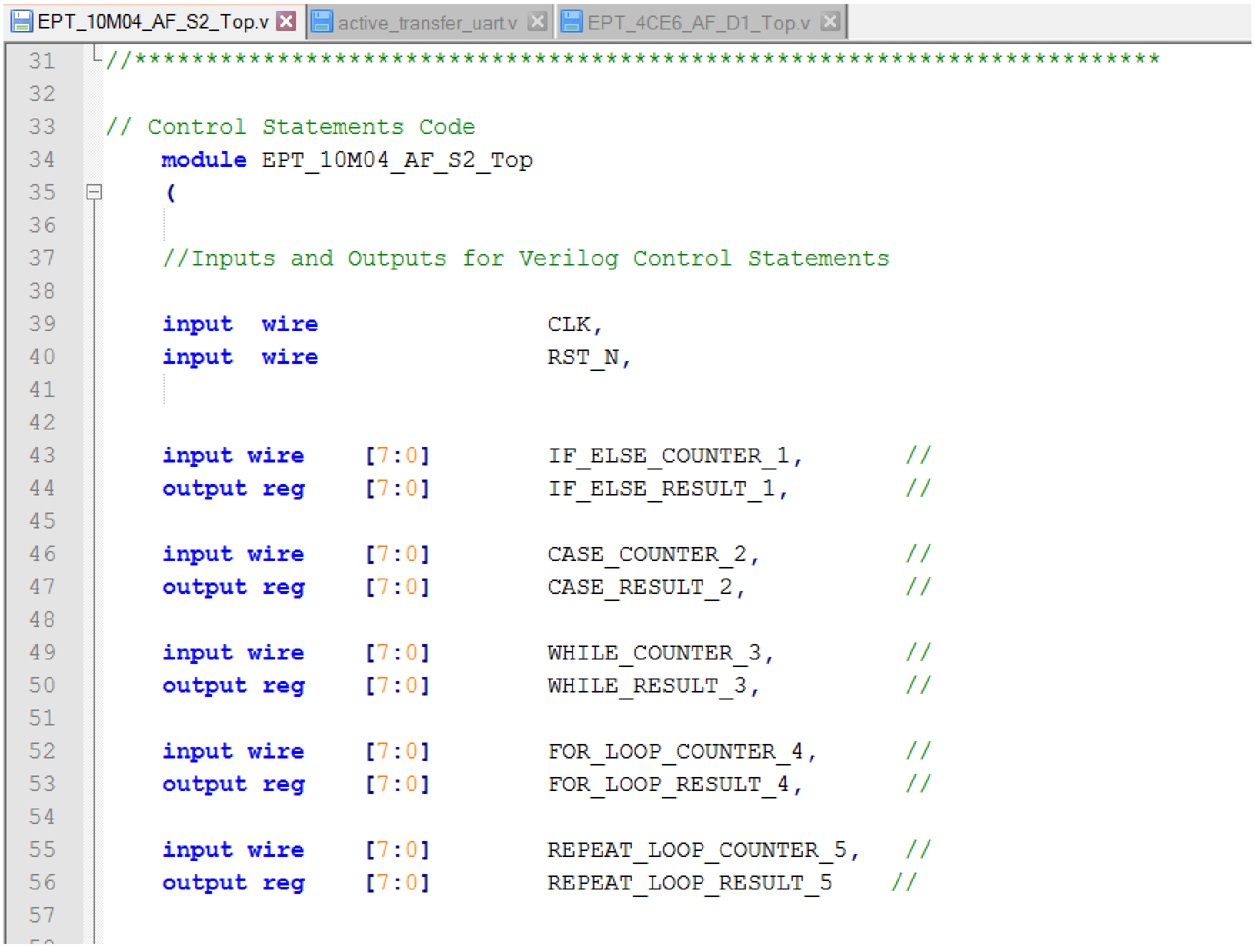
Summary

- 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///

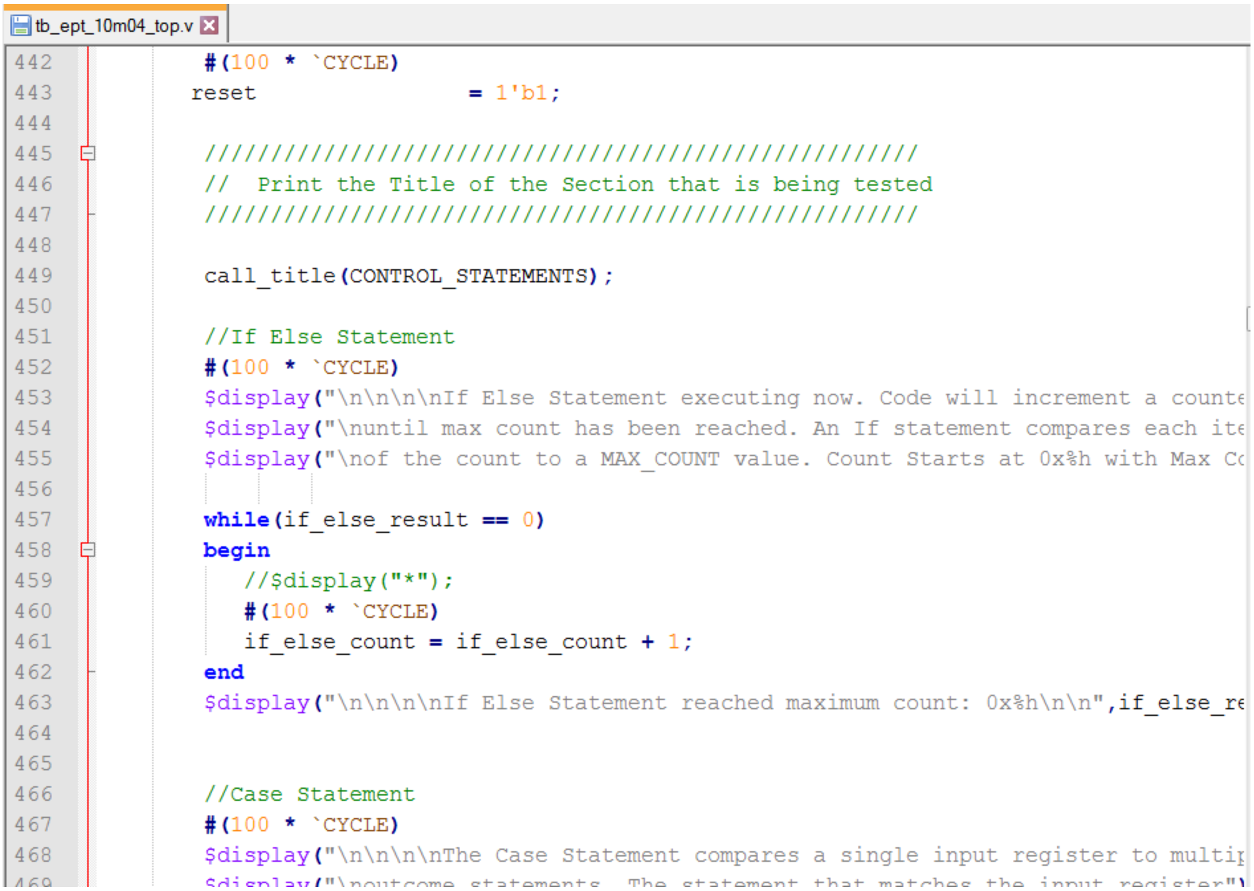
In this lesson, let's explore the Verilog control statements. The control statements are a big contributor to a lot of Verilog code out there in the world. The control statements encompass both synthesizable and non-synthesizable Verilog code.

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:



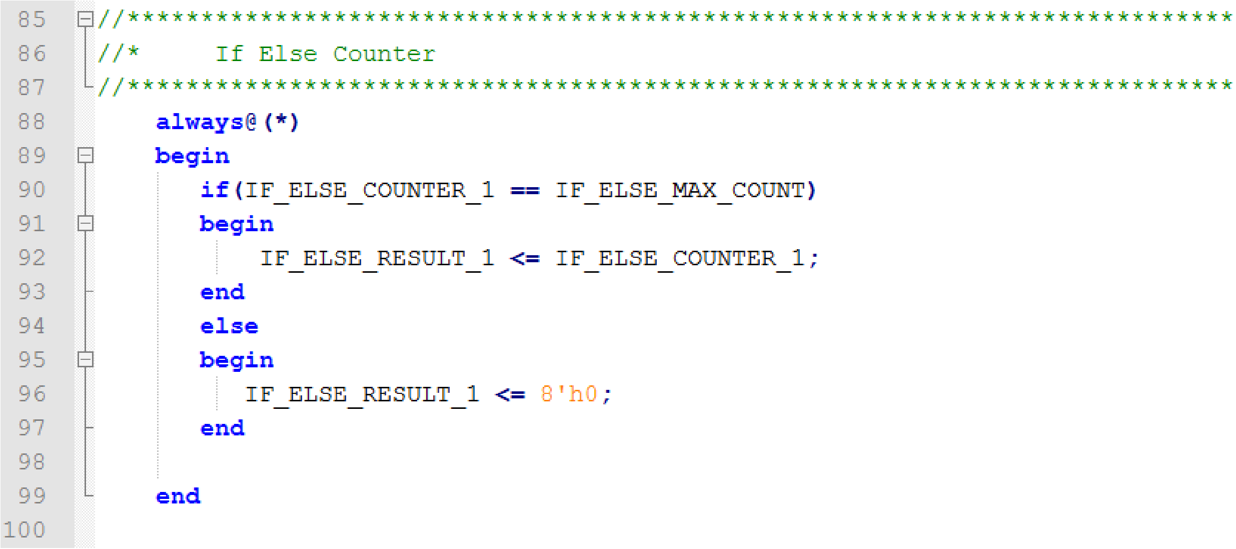
```
//************************************************************************

// 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:



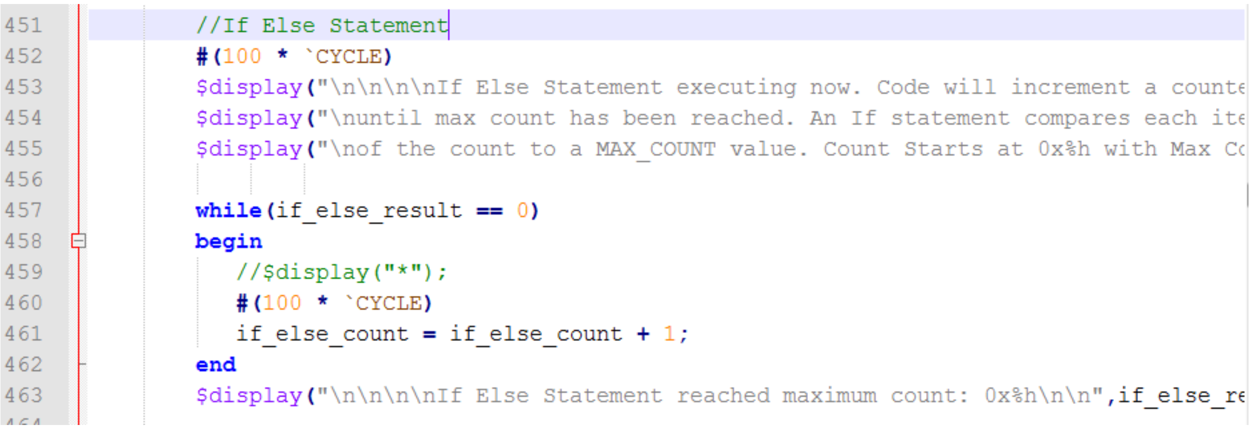
```
         #(100 * `CYCLE)
        reset                   = 1'b1;

         //////////////////////////////////////////////////////
         //  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 counte
         $display("\nuntil max count has been reached. An If statement compares each ite
         $display("\nof the count to a MAX_COUNT value. Count Starts at 0x%h with Max Co

         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_re


         //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.



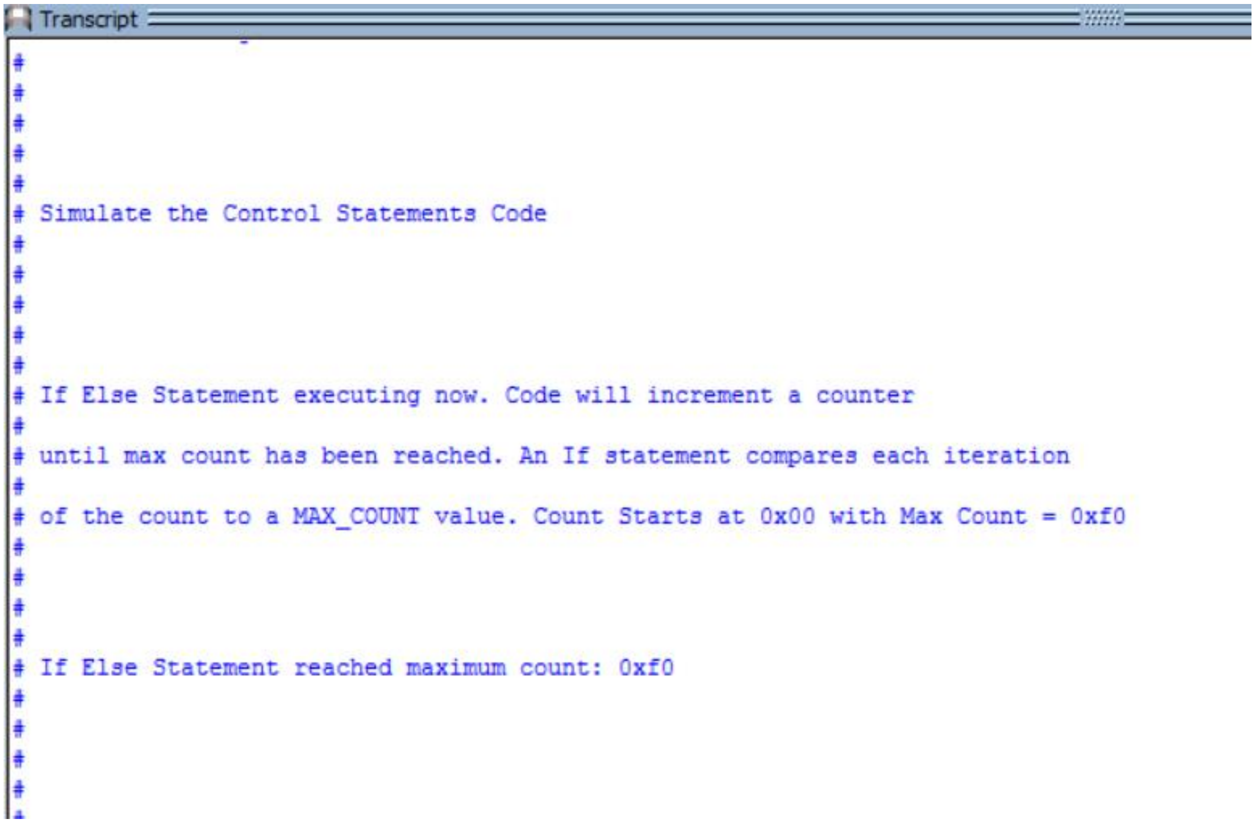
```
//***************************************************************************
//*      If Else Counter
//***************************************************************************
    always@(*)
    begin
       if(IF_ELSE_COUNTER_1 == IF_ELSE_MAX_COUNT)
       begin
           IF_ELSE_RESULT_1 <= 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 counte
       $display("\nuntil max count has been reached. An If statement compares each ite
       $display("\nof the count to a MAX_COUNT value. Count Starts at 0x%h with Max Co

       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_re
```

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’)

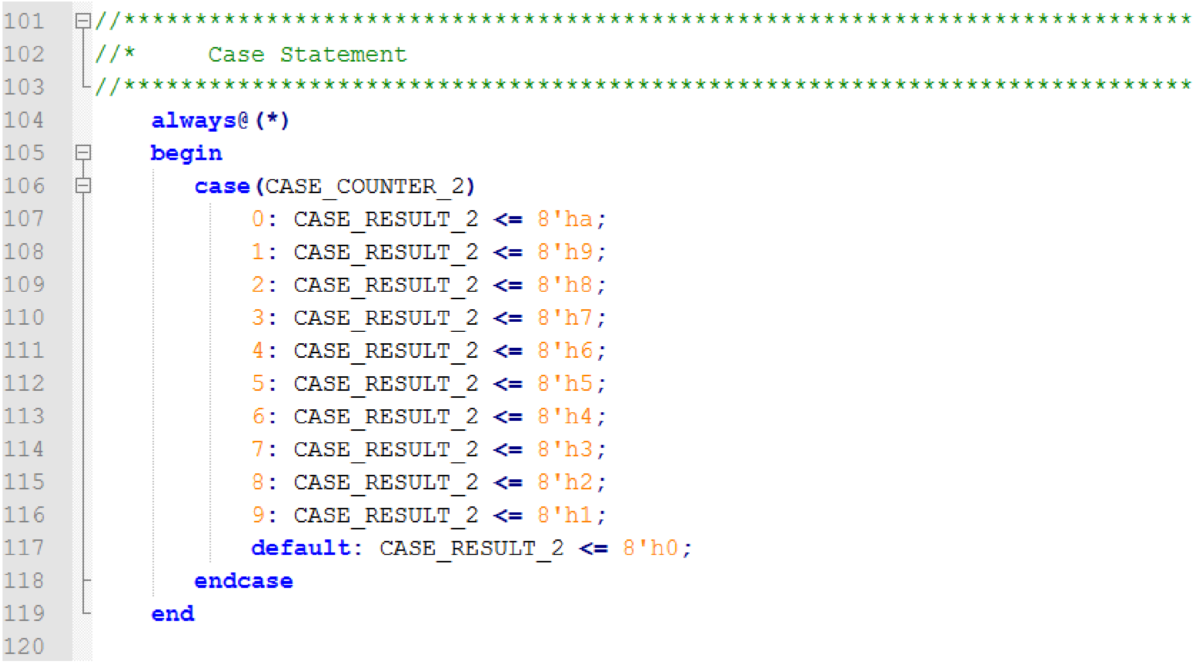
1: ‘statement’;

2: ‘statement’:

.

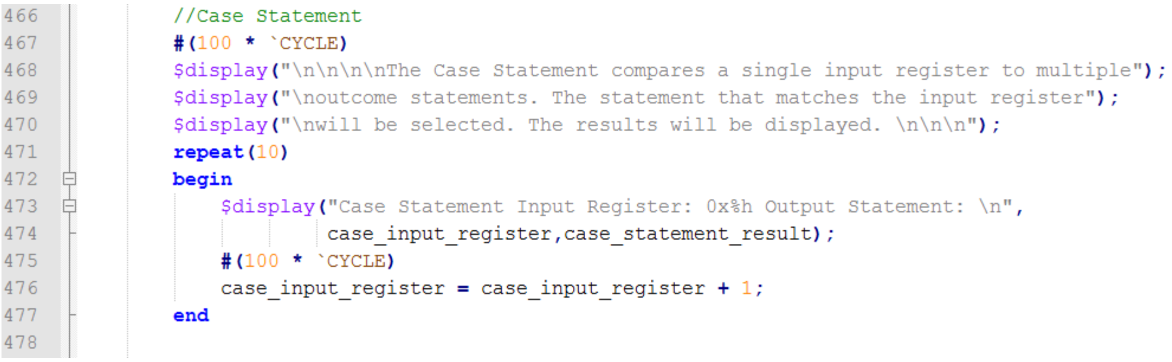
.

endcase



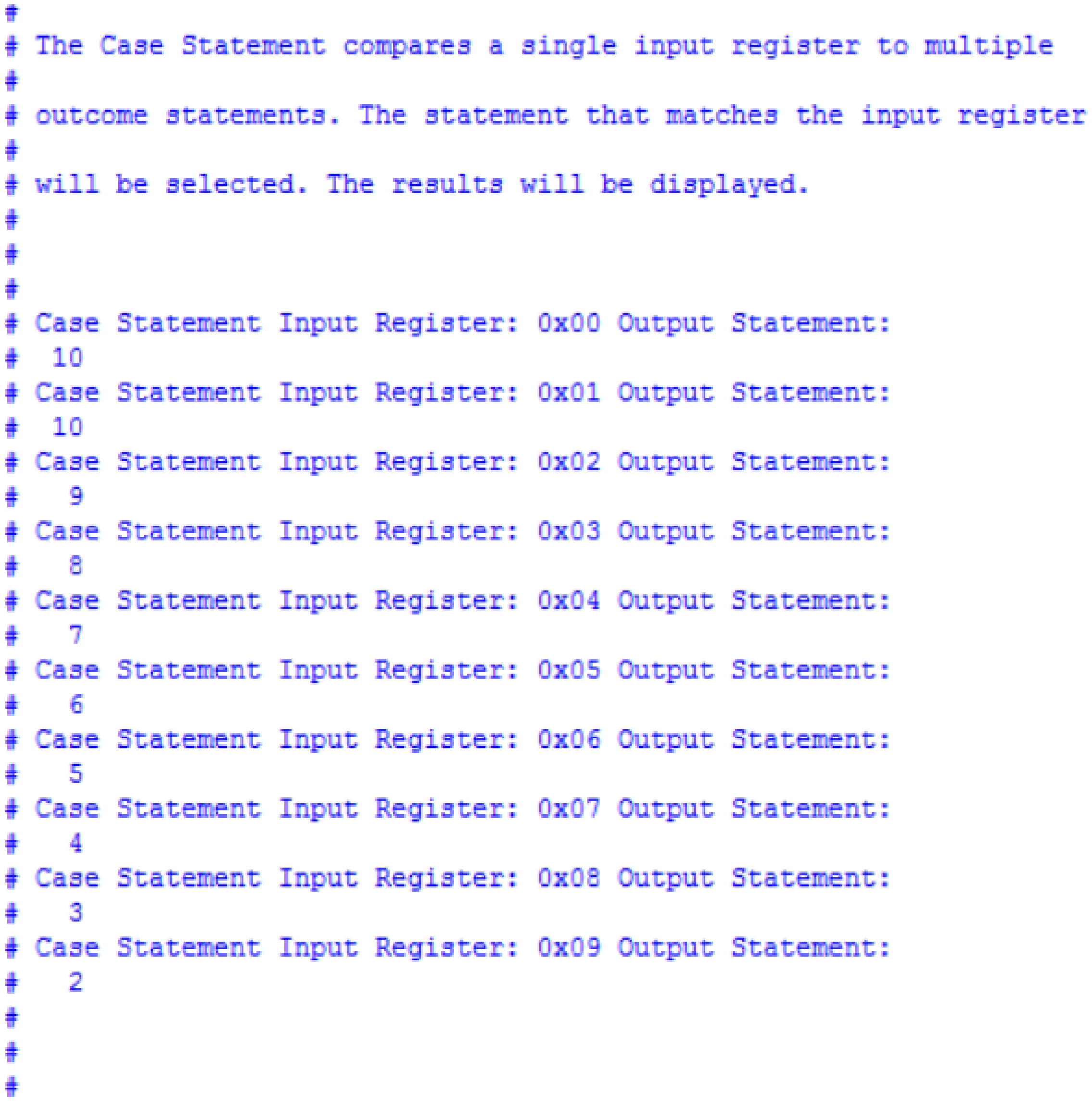
```
//****************************************************************************
//*      Case Statement
//****************************************************************************
    always@(*)
    begin
        case(CASE_COUNTER_2)
            0: CASE_RESULT_2 <= 8'ha;
            1: CASE_RESULT_2 <= 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.



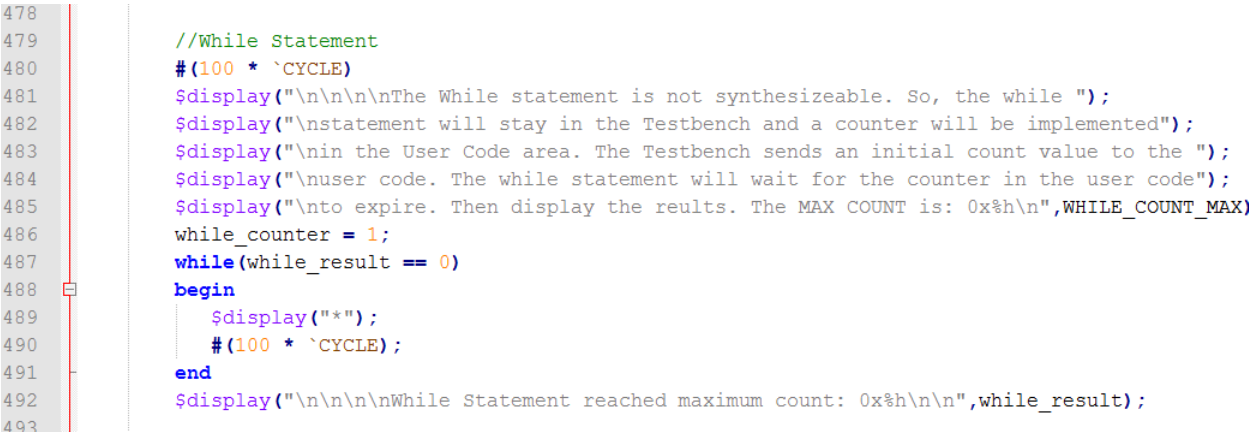
```
        //Case Statement
        #(100 * `CYCLE)
        $display("\n\n\n\nThe Case Statement compares a single input register to multiple");
        $display("\noutcome statements. The statement that matches the input register");
        $display("\nwill be selected. The results will be displayed. \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:



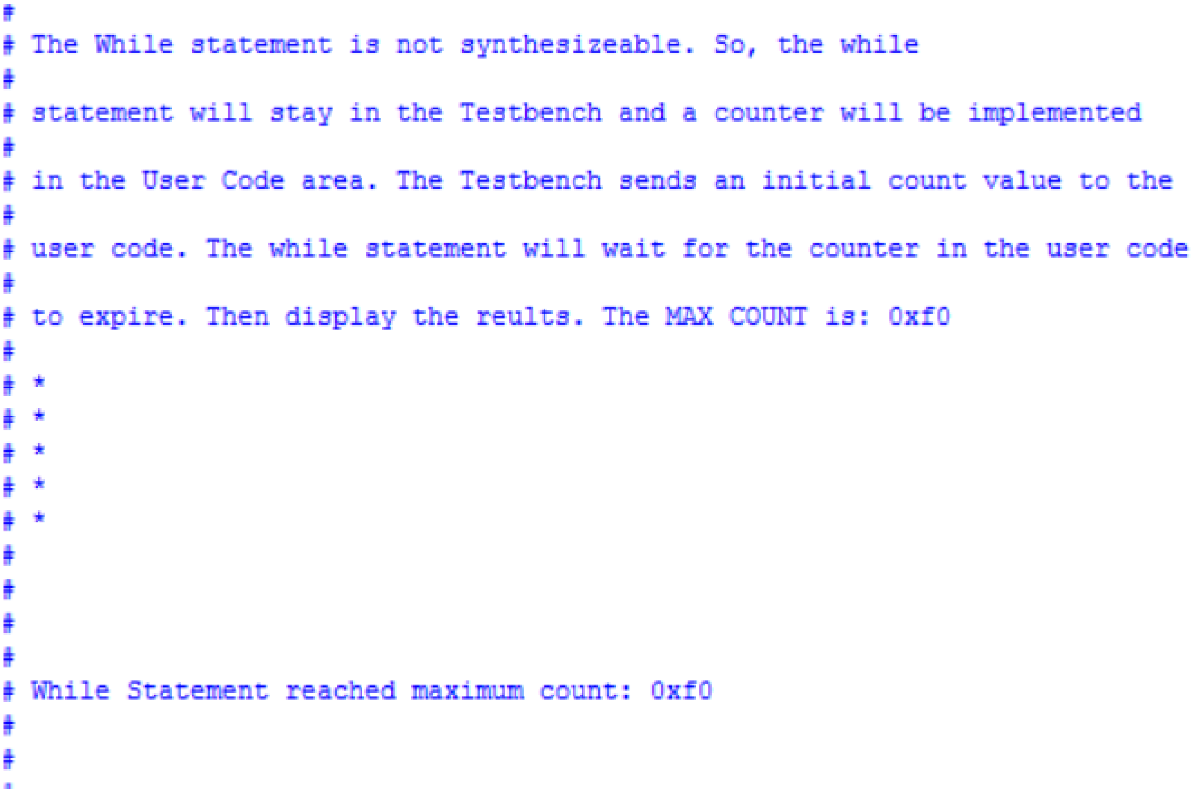
```
#
# The Case Statement compares a single input register to multiple
#
# outcome statements. The statement that matches the input register
#
# will be selected. The results will be displayed.
#
#
#
# Case Statement Input Register: 0x00 Output Statement:
#  10
# Case Statement Input Register: 0x01 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_MAX)
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

This example is a simple buffer.

Task and Function

When repeating the same old things again and again, Verilog, like any other programming language, provides means to address repeated used code, these are called Tasks and Functions. I wish I had something similar for webpages, just call it to print this programming language stuff again and again.

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.

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

TEST BENCHES

Ok, we have code written according to the design document, now what?

Well we need to test it to see if it works according to specs. Most of the time, it's the same we use to do in digital labs in college days: drive the inputs, match the outputs with expected values. Let's look at the arbiter testbench.

```
module arbiter (
clock,
reset,
req_0,
req_1,
gnt_0,
gnt_1
);

input clock, reset, req_0, req_1;
output gnt_0, gnt_1;

reg gnt_0, gnt_1;

always @ (posedge clock or posedge reset)
if (reset) begin
 gnt_0 <= 0;
 gnt_1 <= 0;
end else if (req_0) begin
  gnt_0 <= 1;
  gnt_1 <= 0;
end else if (req_1) begin
  gnt_0 <= 0;
  gnt_1 <= 1;
end

endmodule
// Testbench Code Goes here
module arbiter_tb;

reg clock, reset, req0,req1;
wire gnt0,gnt1;

initial begin
  $monitor ("req0=%b,req1=%b,gnt0=%b,gnt1=%b", req0,req1,gnt0,gnt1);
  clock = 0;
  reset = 0;
  req0 = 0;
  req1 = 0;
  #5  reset = 1;
  #15  reset = 0;
  #10  req0 = 1;
  #10  req0 = 0;
  #10  req1 = 1;
  #10  req1 = 0;
  #10  {req0,req1} = 2'b11;
  #10  {req0,req1} = 2'b00;
  #10  $finish;
end

always  begin
 #5  clock = !clock;
end

arbiter U0 (
.clock (clock),
.reset (reset),
.req_0 (req0),
.req_1 (req1),
.gnt_0 (gnt0),
.gnt_1 (gnt1)
);

endmodule
```

You could download file arbiter.v here

It looks like we have declared all the arbiter inputs as reg and outputs as wire; well, that's true. We are doing this as test bench needs to drive inputs and needs to monitor outputs.

After we have declared all needed variables, we initialize all the inputs to known state: we do that in the initial block. After initialization, we assert/de-assert reset, req0, req1 in the sequence we want to test the arbiter. Clock is generated with an always block.

After we are done with the testing, we need to stop the simulator. Well, we use $finish to terminate simulation. $monitor is used to monitor the changes in the signal list and print them in the format we want.

```
                      req0=0,req1=0,gnt0=x,gnt1=x
req0=0,req1=0,gnt0=0,gnt1=0
req0=1,req1=0,gnt0=0,gnt1=0
req0=1,req1=0,gnt0=1,gnt1=0
req0=0,req1=0,gnt0=1,gnt1=0
req0=0,req1=1,gnt0=1,gnt1=0
req0=0,req1=1,gnt0=0,gnt1=1
req0=0,req1=0,gnt0=0,gnt1=1
req0=1,req1=1,gnt0=0,gnt1=1
req0=1,req1=1,gnt0=1,gnt1=0
req0=0,req1=0,gnt0=1,gnt1=0
```

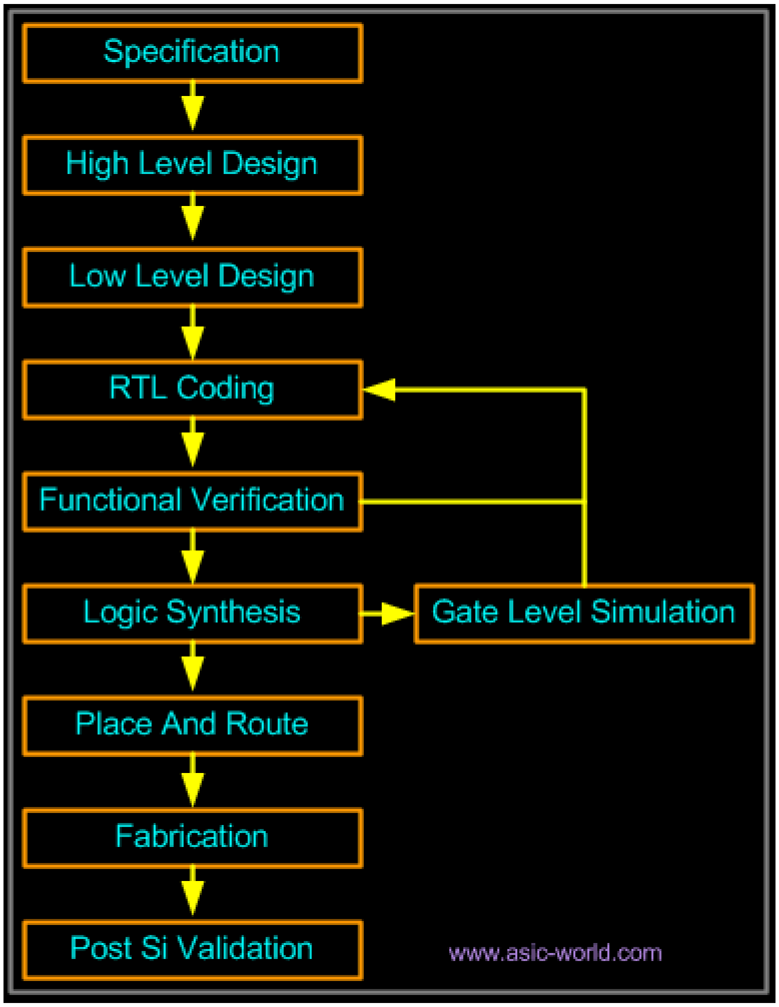
Introduction

Being new to Verilog you might want to try some examples and try designing something new. I have listed the tool flow that could be used to achieve this. I have personally tried this flow and found this to be working just fine for me. Here I have taken only the front end design part and bits of FPGA design of the tool flow, that can be done without any fat money spent on tools.

Various stages of ASIC/FPGA

- **Specification :** Word processor like Word, Kwriter, AbiWord, Open Office.
- **High Level Design :** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.
- **Micro Design/Low level design:** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word.
- **RTL Coding :** Vim, Emacs, conTEXT, HDL TurboWriter
- **Simulation :** Modelsim, VCS, Verilog-XL, Veriwell, Finsim, Icarus.
- **Synthesis :** Design Compiler, FPGA Compiler, Synplify, Leonardo Spectrum. You can download this from FPGA vendors like Altera and Xilinx for free.
- **Place & Route :** For FPGA use FPGA' vendors P&R tool. ASIC tools require expensive P&R tools like Apollo. Students can use LASI, Magic.
- **Post Si Validation :** For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place.

Figure : Typical Design flow



Specification

This is the stage at which we define what are the important parameters of the system/design that you are planning to design. A simple example would be: I want to design a counter; it should be 4 bit wide, should have synchronous reset, with active high enable; when reset is active, counter output should go to "0".

High Level Design

This is the stage at which you define various blocks in the design and how they communicate. Let's assume that we need to design a microprocessor: high level design means splitting the design into blocks based on their function; in our case the blocks are registers, ALU, Instruction Decode, Memory Interface, etc.

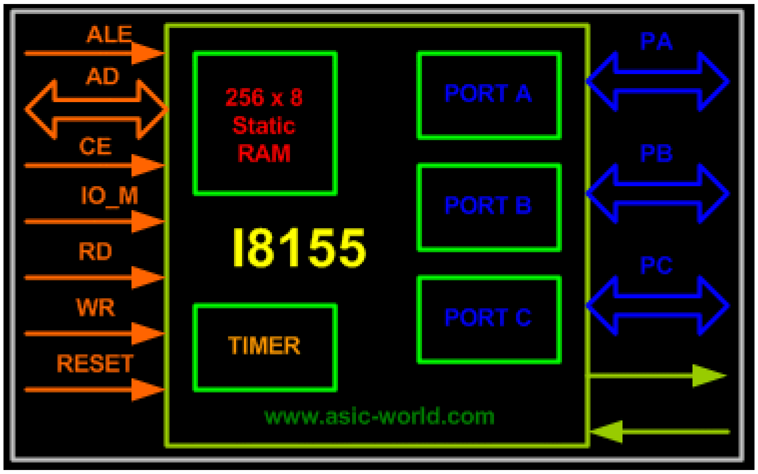
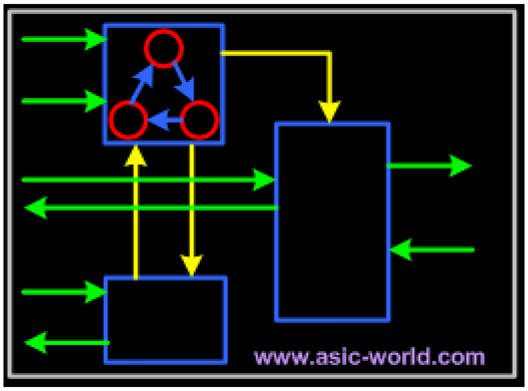


Figure : I8155 High Level Block Diagram

Micro Design/Low level design

Low level design or Micro design is the phase in which the designer describes how each block is implemented. It contains details of State machines, counters, Mux, decoders, internal registers. It is always a good idea to draw waveforms at various interfaces. This is the phase where one spends lot of time.



www.asic-world.com

Figure : Sample Low level design

RTL Coding

In RTL coding, Micro design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally we like to lint the code, before starting verification or synthesis.

```
module addbit (
a        , // first input
b        , // Second input
ci       , // Carry input
sum      , // sum output
co         // carry output
);
//Input declaration
input a;
input b;
input ci;
//Ouput declaration
output sum;
output co;
//Port Data types
wire  a;
wire  b;
wire  ci;
wire  sum;
wire  co;
//Code starts here
```

```
assign {co,sum} = a + b + ci;

endmodule // End of Module addbit
```

You could download file addbit.v here

Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the Hardware models. To test if the RTL code meets the functional requirements of the specification, we must see if all the RTL blocks are functionally correct. To achieve this we need to write a testbench, which generates clk, reset and the required test vectors. A sample testbench for a counter is shown below. Normally we spend 60-70% of time in design verification.

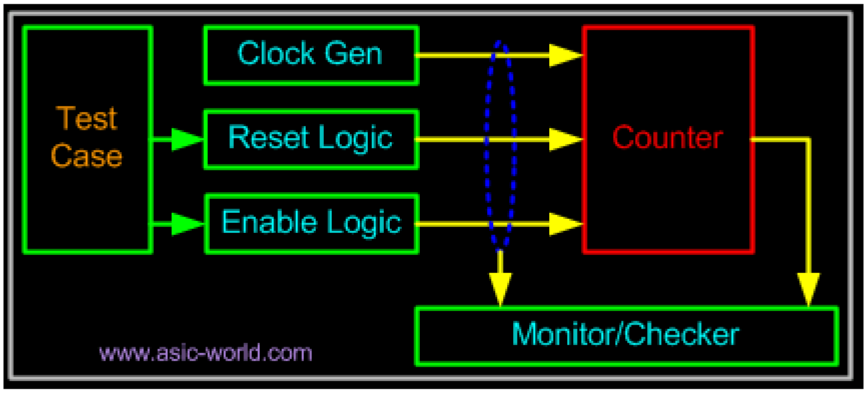


Figure : Sample Testbench Env

We use the waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators come with a waveform viewer. As design becomes complex, we write self checking testbench, where testbench applies the test vector, then compares the output of DUT with expected values.

There is another kind of simulation, called **timing simulation**, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wire delays and see if DUT works at rated clock speed. This is also called as **SDF simulation** or **gate level simulation**.

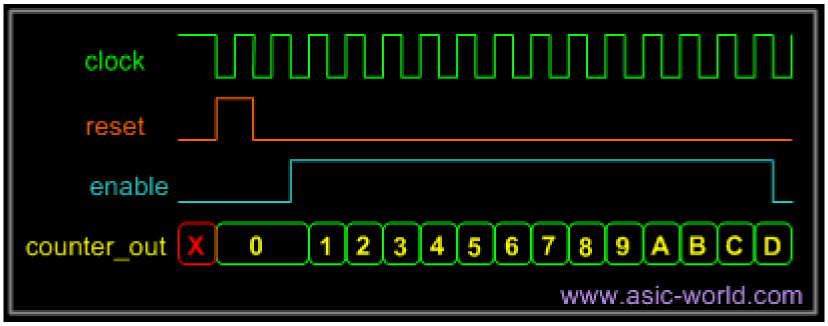


Figure : 4 bit Up Counter Waveform

Synthesis

Synthesis is the process in which synthesis tools like design compiler or Synplify take RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives. Synthesis tool, after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements. (Important thing to note is, synthesis tools are not aware of wire delays, they only know of gate delays). After the synthesis there are a couple of things that are normally done before passing the netlist to backend (Place and Route)

- **Formal Verification :** Check if the RTL to gate mapping is correct.
- **Scan insertion :** Insert the scan chain in the case of ASIC.

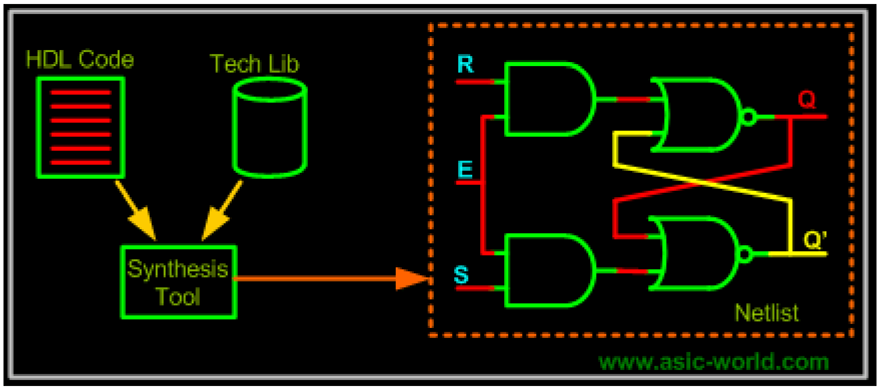


Figure : Synthesis Flow

Place & Route

The gatelevel netlist from the synthesis tool is taken and imported into place and route tool in Verilog netlist format. All the gates and flip-flops are placed; clock tree synthesis and reset is routed. After this each block is routed. The P&R tool output is a GDS file, used by foundry for fabricating the ASIC. Backend team normally dumps out SPEF (standard parasitic exchange format) /RSPF (reduced parasitic exchange format)/DSPF (detailed parasitic exchange format) from layout tools like ASTRO to the frontend team, who then use the read_parasitic command in tools like Prime Time to write out SDF (standard delay format) for gate level simulation purposes.

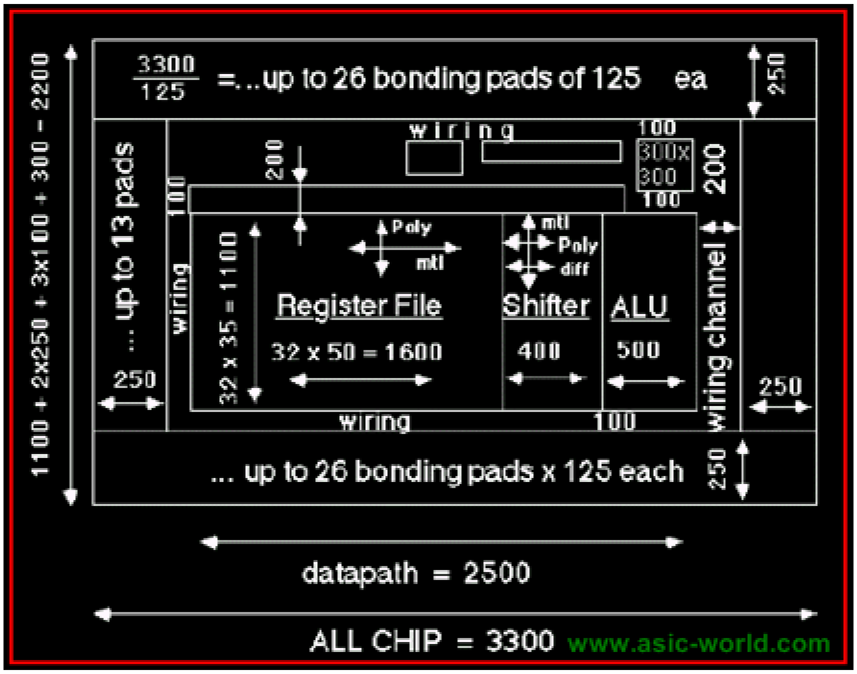


Figure : Sample micro-processor placement

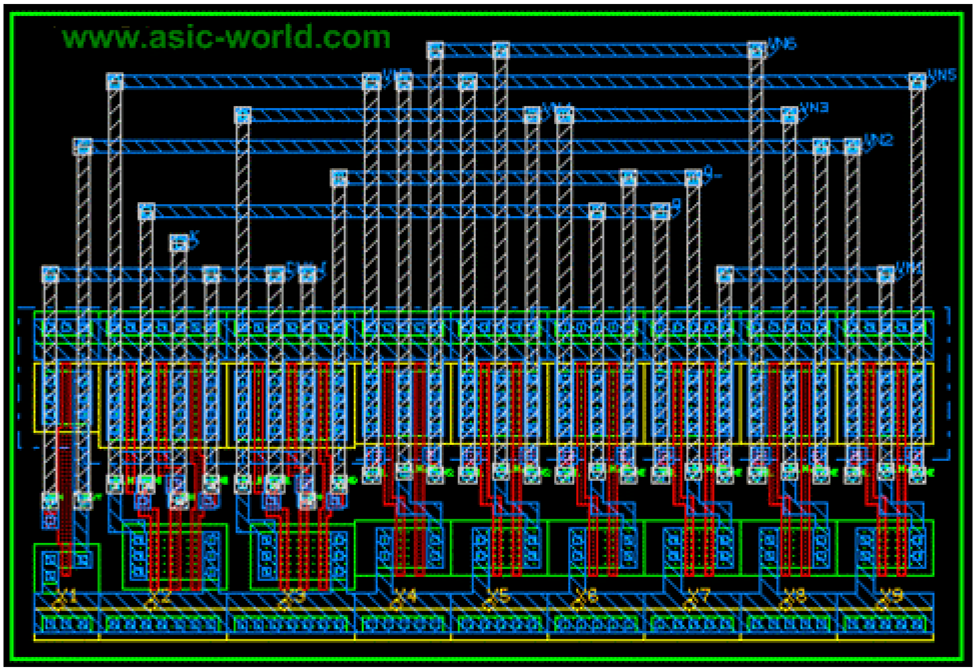


Figure : J-K Flip-Flop

Post Silicon Validation

Once the chip (silicon) is back from fab, it needs to be put in a real environment and tested before it can be released into Market. Since the simulation speed (number of clocks per second) with RTL is very slow, there is always the possibility to find a bug in Post silicon validation.

Introduction

If you refer to any book on programming languages, it starts with an "Hello World" program; once you have written it, you can be sure that you can do something in that language ☺.

Well I am also going to show how to write a **"hello world"** program, followed by a **"counter"** design, in Verilog.

Hello World Program

```verilog
//-----------------------------------------------------
// This is my first Verilog Program
// Design Name : hello_world
// File Name : hello_world.v
// Function : This program will print 'hello world'
// Coder    : Deepak
//-----------------------------------------------------
module hello_world ;

initial begin
   $display ("Hello World by Deepak");
   #10 $finish;
end

endmodule // End of Module hello_world
```

You could download file hello_world.v here

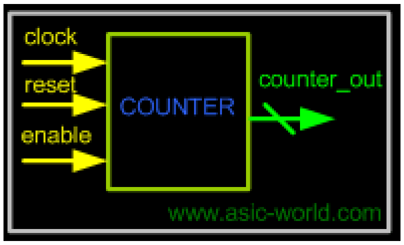
Words in green are comments, blue are reserved words. Any program in Verilog starts with reserved word 'module' <module_name>. In the above example line 8 contains module hello_world. (Note: We can have compiler pre-processor statements like `include', `define' before module declaration)

Line 10 contains the initial block: this block gets executed only once after the simulation starts, at time=0 (0ns). This block contains two statements which are enclosed within begin, at line 10, and end, at line 13. In Verilog, if you have multiple lines within a block, you need to use begin and end. Module ends with 'endmodule' reserved word, in this case at line 15.

Hello World Program Output

```
Hello World by Deepak
```

Counter Design Block



Counter Design Specs

- 4-bit synchronous up counter.
- active high, synchronous reset.
- Active high enable.

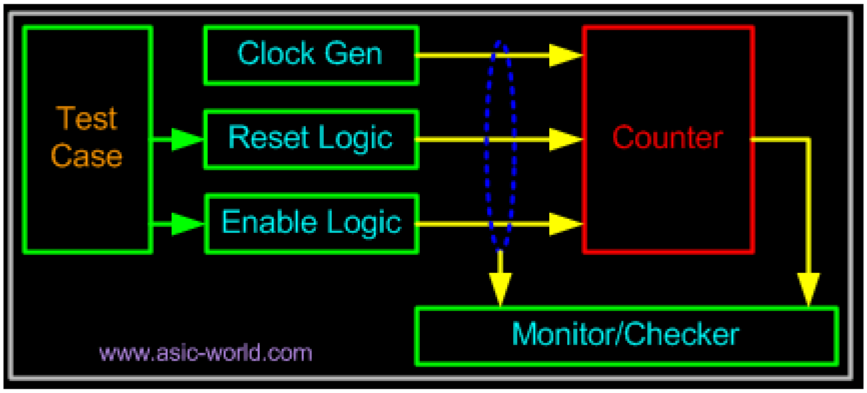
Counter Design

```
//-----------------------------------------------------
// This is my second Verilog Design
// Design Name : first_counter
// File Name : first_counter.v
// Function : This is a 4 bit up-counter with
// Synchronous active high reset and
// with active high enable signal
//-----------------------------------------------------
module first_counter (
clock , // Clock input of the design
reset , // active high, synchronous Reset input
enable , // Active high enable signal for counter
counter_out // 4 bit vector output of the counter
); // End of port list
//-------------Input Ports-----------------------------
input clock ;
input reset ;
input enable ;
//-------------Output Ports----------------------------
output [3:0] counter_out ;
//-------------Input ports Data Type-------------------
// By rule all the input ports should be wires
wire clock ;
wire reset ;
wire enable ;
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [3:0] counter_out ;

//------------Code Starts Here-------------------------
// Since this counter is a positive edge trigged one,
// We trigger the below block with respect to positive
// edge of the clock.
always @ (posedge clock)
begin : COUNTER // Block Name
  // At every rising edge of clock we check if reset is active
  // If active, we load the counter output with 4'b0000
  if (reset == 1'b1) begin
    counter_out <= #1 4'b0000;
  end
  // If enable is active, then we increment the counter
  else if (enable == 1'b1) begin
    counter_out <= #1 counter_out + 1;
  end
end // End of Block COUNTER

endmodule // End of Module counter
```

You could download file first_counter.v here

Counter Test Bench

Any digital circuit, no matter how complex, needs to be tested. For the counter logic, we need to provide clock and reset logic. Once the counter is out of reset, we toggle the enable input to the counter, and check the waveform to see if the counter is counting correctly. This is done in Verilog.



The counter testbench consists of clock generator, reset control, enable control and monitor/checker logic. Below is the simple code of testbench without the monitor/checker logic.

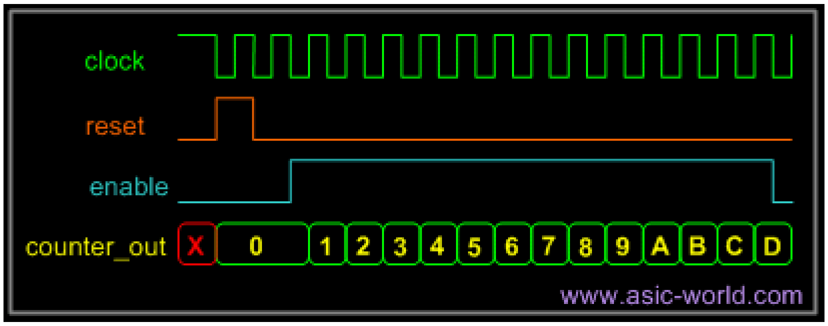
```
`include  "first_counter.v"
module first_counter_tb();
// Declare inputs as regs and outputs as wires
reg clock, reset, enable;
wire [3:0] counter_out;

// Initialize all variables
initial begin
  $display ("time\t clk reset enable counter");
  $monitor ("%g\t %b   %b     %b      %b",
      $time, clock, reset, enable, counter_out);
  clock = 1;          // initial value of clock
  reset = 0;          // initial value of reset
  enable = 0;         // initial value of enable
  #5  reset = 1;      // Assert the reset
  #10  reset = 0;     // De-assert the reset
  #10  enable = 1;    // Assert enable
  #100  enable = 0;   // De-assert enable
  #5  $finish;        // Terminate simulation
end

// Clock generator
always begin
  #5  clock = ~clock; // Toggle clock every 5 ticks
end

// Connect DUT to test bench
first_counter U_counter (
clock,
reset,
enable,
counter_out
);

endmodule
```

You could download file first_counter_tb.v here

```
                    time     clk reset enable counter
0      1  0     0      xxxx
```

```
5      0  1    0     xxxx
10      1  1    0     xxxx
11      1  1    0     0000
15      0  0    0     0000
20      1  0    0     0000
25      0  0    1     0000
30      1  0    1     0000
31      1  0    1     0001
35      0  0    1     0001
40      1  0    1     0001
41      1  0    1     0010
45      0  0    1     0010
50      1  0    1     0010
51      1  0    1     0011
55      0  0    1     0011
60      1  0    1     0011
61      1  0    1     0100
65      0  0    1     0100
70      1  0    1     0100
71      1  0    1     0101
75      0  0    1     0101
80      1  0    1     0101
81      1  0    1     0110
85      0  0    1     0110
90      1  0    1     0110
91      1  0    1     0111
95      0  0    1     0111
100      1  0    1     0111
101      1  0    1     1000
105      0  0    1     1000
110      1  0    1     1000
111      1  0    1     1001
115      0  0    1     1001
120      1  0    1     1001
121      1  0    1     1010
125      0  0    0     1010
```

Counter Waveform



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

Examples of White Spaces

Functional Equivalent Code

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.

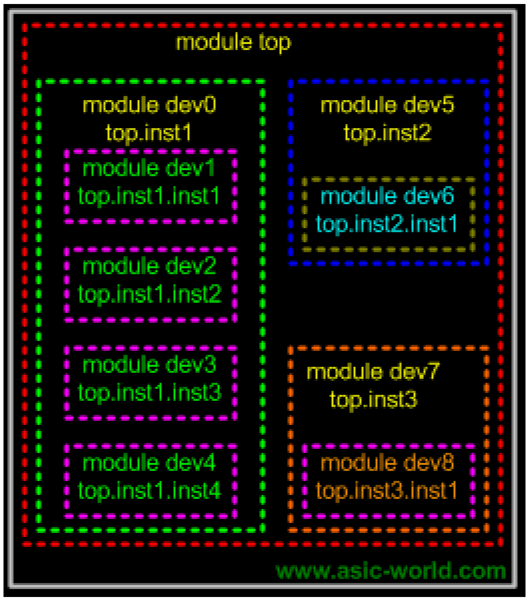
```verilog
module signed_number;

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
```

Modules

- Modules are the building blocks of Verilog designs
- You create the design hierarchy by instantiating modules in other modules.
- You instance a module when you use that module in another, higher-level module.



Ports

- Ports allow communication between a module and its environment.
- All but the top-level modules in a hierarchy have ports.
- Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is :

input [range_val:range_var] list_of_identifiers;

output [range_val:range_var] list_of_identifiers;

inout [range_val:range_var] list_of_identifiers;

NOTE : As a good coding practice, there should be only one port identifier per line, as shown below

Examples : Port Declaration

```
input           clk       ; // clock input
input  [15:0]   data_in   ; // 16 bit data input bus
output [7:0]    count     ; // 8 bit counter output
inout           data_bi   ; // Bi-Directional data bus
```

You could download file port_declare.v here

Examples : A complete Example in Verilog

```
module addbit (
a       , // first input
b       , // Second input
ci      , // Carry input
sum     , // sum output
co        // carry output
);
//Input declaration
input a;
input b;
input ci;
//Ouput declaration
output sum;
output co;
//Port Data types
wire  a;
wire  b;
wire  ci;
wire  sum;
wire   co;
//Code starts here
assign {co,sum} = a + b + ci;

endmodule  // End of Module addbit
```

You could download file addbit.v here

Modules connected by port order (implicit)

Here order should match correctly. Normally it's not a good idea to connect ports implicitly. It could cause problem in debug (for example: locating the port which is causing a compile error), when any port is added or deleted.

```
//-----------------------------------------------------
// This is simple adder Program
// Design Name : adder_implicit
// File Name   : adder_implicit.v
// Function    : This program shows how implicit
//               port connection are done
// Coder       : Deepak Kumar Tala
//-----------------------------------------------------
module adder_implicit (
result         , // Output of the adder
carry          , // Carry output of adder
r1             , // first input
r2             , // second input
ci               // carry input
);

// Input Port Declarations
input    [3:0]   r1         ;
input    [3:0]   r2         ;
input            ci         ;

// Output Port Declarations
output   [3:0]  result      ;
output          carry       ;

// Port Wires
wire     [3:0]    r1        ;
wire     [3:0]    r2        ;
wire              ci        ;
wire        [3:0]    result    ;
wire                 carry     ;

// Internal variables
wire                 c1        ;
wire                 c2        ;
wire                 c3        ;

// Code Starts Here
addbit u0 (
r1[0]           ,
r2[0]           ,
ci              ,
result[0]       ,
c1
);

addbit u1 (
r1[1]          ,
r2[1]          ,
c1             ,
result[1]      ,
c2
);

addbit u2 (
r1[2]          ,
r2[2]          ,
c2             ,
result[2]      ,
c3
);

addbit u3 (
r1[3]          ,
r2[3]          ,
c3             ,
result[3]      ,
carry
);

endmodule  // End Of Module adder
```

You could download file adder_implicit.v here

Modules connected by name

Here the name should match with the leaf module, the order is not important.

```
//-----------------------------------------------------
// This is simple adder Program
// Design Name : adder_explicit
// File Name   : adder_explicit.v
// Function    : Here the name should match
// with the leaf module, the order is not important.
// Coder       : Deepak Kumar Tala
//-----------------------------------------------------
module adder_explicit (
result          , // Output of the adder
carry           , // Carry output of adder
r1              , // first input
r2              , // second input
ci                // carry input
);

// Input Port Declarations
input      [3:0]    r1          ;
input      [3:0]    r2          ;
input               ci          ;

// Output Port Declarations
output     [3:0]  result        ;
output            carry         ;

// Port Wires
wire       [3:0]     r1         ;
wire       [3:0]     r2         ;
wire                 ci         ;
wire       [3:0]     result     ;
wire                 carry      ;

// Internal variables
wire                 c1         ;
wire                 c2         ;
wire                 c3         ;

// Code Starts Here
addbit u0 (
.a             (r1[0])         ,
.b              (r2[0])         ,
.ci             (ci)            ,
.sum            (result[0])     ,
.co             (c1)
);

addbit u1 (
.a              (r1[1])         ,
.b              (r2[1])         ,
.ci             (c1)            ,
.sum            (result[1])     ,
.co             (c2)
);

addbit u2 (
.a              (r1[2])         ,
.b              (r2[2])         ,
.ci             (c2)            ,
.sum            (result[2])     ,
.co             (c3)
);

addbit u3 (
.a              (r1[3])         ,
.b              (r2[3])         ,
.ci             (c3)            ,
.sum            (result[3])     ,
.co             (carry)
);

endmodule  // End Of Module adder
```

You could download file adder_explicit.v here

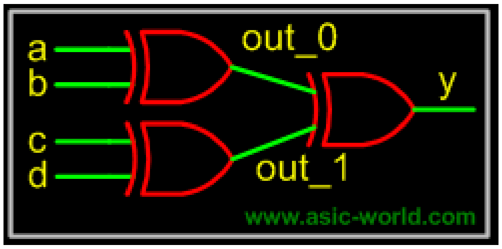
Instantiating a module

```
//-----------------------------------------------------
// This is simple parity Program
// Design Name : parity
// File Name   : parity.v
// Function    : This program shows how a verilog
//               primitive/module port connection are done
// Coder       : Deepak
//-----------------------------------------------------
module parity (
a       , // First input
b          , // Second input
c          , // Third Input
d          , // Fourth Input
y            // Parity  output
);

// Input Declaration
input          a        ;
input          b        ;
input          c        ;
input          d        ;
// Ouput Declaration
output         y       ;
// port data types
wire           a          ;
wire           b          ;
wire           c          ;
wire           d          ;
wire           y          ;
// Internal variables
wire           out_0 ;
wire           out_1 ;

// Code starts Here
xor u0 (out_0,a,b);

xor u1 (out_1,c,d);

xor u2 (y,out_0,out_1);

endmodule // End Of Module parity
```

You could download file parity.v here

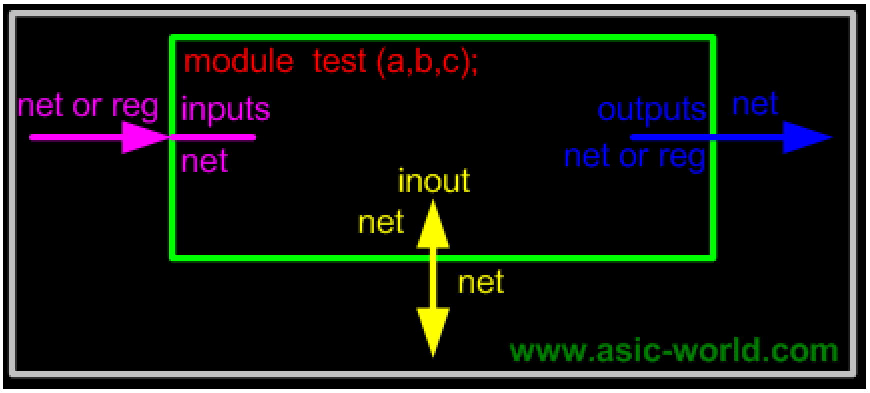
Question : What is the difference between u0 in module adder and u0 in module parity?

✦ Schematic



Port Connection Rules

- Inputs : internally must always be of type net, externally the inputs can be connected to a variable of type reg or net.
- Outputs : internally can be of type net or reg, externally the outputs must be connected to a variable of type net.
- Inouts : internally or externally must always be type net, can only be connected to a variable net type.



- Width matching : It is legal to connect internal and external ports of different sizes. But beware, synthesis tools could report problems.
- Unconnected ports : unconnected ports are allowed by using a ",".
- The net data types are used to connect structure.
- A net data type is required if a signal can be driven a structural connection.

Example - Implicit Unconnected Port

```verilog
module implicit();
reg clk,d,rst,pre;
wire q;

// Here second port is not connected
dff u0 ( q,,clk,d,rst,pre);

endmodule

// D fli-flop
module dff (q, q_bar, clk, d, rst, pre);
input clk, d, rst, pre;
output q, q_bar;
reg q;

assign q_bar = ~q;

always @ (posedge clk)
if (rst == 1'b1) begin
  q <= 0;
end else if (pre == 1'b1) begin
  q <= 1;
end else begin
  q <= d;
end

endmodule
```

You could download file implicit.v here

Example - Explicit Unconnected Port

```verilog
module explicit();
reg clk,d,rst,pre;
wire q;

// Here q_bar is not connected
// We can connect ports in any order
dff u0 (
.q           (q),
.d  (d),
.clk         (clk),
.q_bar       (),
.rst         (rst),
.pre         (pre)
);

endmodule

// D fli-flop
module dff (q, q_bar, clk, d, rst, pre);
input clk, d, rst, pre;
output q, q_bar;
reg q;

assign q_bar = ~q;

always @ (posedge clk)
if (rst == 1'b1) begin
  q <= 0;
end else if (pre == 1'b1) begin
  q <= 1;
end else begin
  q <= d;
end

endmodule
```

You could download file explicit.v here

Hierarchical Identifiers

e top module iden

tifier followed by module instant identifiers, separated by periods.

This is useful basically when we want to see the signal inside a lower module, or want to force a value inside an internal module. The example below shows how to monitor the value of an internal module signal.

Example

```
//-----------------------------------------------------
// This is simple adder Program
// Design Name : adder_hier
// File Name   : adder_hier.v
// Function    : This program shows verilog hier path works
// Coder       : Deepak
//-----------------------------------------------------
`include  "addbit.v"
module adder_hier (
result          , // Output of the adder
carry           , // Carry output of adder
r1              , // first input
r2              , // second input
ci                // carry input
);

// Input Port Declarations
input     [3:0]   r1         ;
input     [3:0]   r2         ;
input             ci         ;

// Output Port Declarations
output    [3:0]  result      ;
output           carry       ;

// Port Wires
wire      [3:0]     r1       ;
wire      [3:0]     r2       ;
wire                ci       ;
wire      [3:0]     result   ;
wire                carry    ;

// Internal variables
wire                c1       ;
wire                c2       ;
wire                c3       ;

// Code Starts Here
addbit u0 (r1[0],r2[0],ci,result[0],c1);
addbit u1 (r1[1],r2[1],c1,result[1],c2);
addbit u2 (r1[2],r2[2],c2,result[2],c3);
addbit u3 (r1[3],r2[3],c3,result[3],carry);

endmodule  // End Of Module adder

module tb();

reg [3:0] r1,r2;
reg  ci;
wire [3:0] result;
wire  carry;

// Drive the inputs
initial begin
   r1 = 0;
   r2 = 0;
   ci = 0;
   #10  r1 = 10;
   #10  r2 = 2;
   #10  ci = 1;
   #10  $display("+--------------------------------------------------------+");
   $finish;
end

// Connect the lower module
adder_hier U (result,carry,r1,r2,ci);

// Hier demo here
initial begin
   $display("+--------------------------------------------------------+");
   $display("|  r1  |  r2  |  ci  | u0.sum | u1.sum | u2.sum | u3.sum |");
   $display("+--------------------------------------------------------+");
   $monitor("|  %h  |  %h  |  %h  |    %h   |   %h   |   %h   |   %h   |",
   r1,r2,ci, tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum, tb.U.u3.sum);
end

endmodule
```

You could download file adder_hier.v here

```
                         +--------------------------------------------------------+
|  r1  |  r2  |  ci  | u0.sum | u1.sum | u2.sum | u3.sum |
+--------------------------------------------------------+
|  0   |  0   |  0   |    0   |    0   |    0   |    0   |
|  a   |  0   |  0   |    0   |    1   |    0   |    1   |
|  a   |  2   |  0   |    0   |    0   |    1   |    1   |
|  a   |  2   |  1   |    1   |    0   |    1   |    1   |
+--------------------------------------------------------+
```

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 wi

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).

Note : Of all net types, wire is the one which is most widely used.

Example - wor

```
module test_wor();

wor a;
reg b, c;

assign a =  b;
assign a =  c;

initial begin
  $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
  #1 b  = 0;
  #1 c  = 0;
  #1 b = 1;
  #1 b = 0;
  #1 c = 1;
  #1 b = 1;
  #1 b = 0;
  #1 $finish;
end

endmodule
```

You could download file test_wor.v here

Simulator Output

```
0 a = x b = x c = x
```

```
1 a = x b = 0 c = x
2 a = 0 b = 0 c = 0
3 a = 1 b = 1 c = 0
4 a = 0 b = 0 c = 0
5 a = 1 b = 0 c = 1
6 a = 1 b = 1 c = 1
7 a = 1 b = 0 c = 1
```

Example - wand

```
module test_wand();

wand a;
reg b, c;

assign a =  b;
assign a =  c;

initial begin
   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
   #1 b  = 0;
   #1 c  = 0;
   #1 b = 1;
   #1 b = 0;
   #1 c = 1;
   #1 b = 1;
   #1 b = 0;
   #1 $finish;
end

endmodule
```

You could download file test_wand.v here

Simulator Output

```
                    0 a = x b = x c = x
1 a = 0 b = 0 c = x
2 a = 0 b = 0 c = 0
3 a = 0 b = 1 c = 0
4 a = 0 b = 0 c = 0
5 a = 0 b = 0 c = 1
6 a = 1 b = 1 c = 1
7 a = 0 b = 0 c = 1
```

Example - tri

```verilog
module test_tri();

tri a;
reg b, c;

assign a = (b) ? c : 1'bz;

initial begin
   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
   b  = 0;
   c  = 0;
   #1 b = 1;
   #1 b = 0;
   #1 c = 1;
   #1 b = 1;
   #1 b = 0;
   #1 $finish;
end

endmodule
```

You could download file test_tri.v here

Simulator Output

```
                    0 a = z b = 0 c = 0
1 a = 0 b = 1 c = 0
2 a = z b = 0 c = 0
3 a = z b = 0 c = 1
4 a = 1 b = 1 c = 1
```

```
5 a = z b = 0 c = 1
```

Example - trireg

```
module test_trireg();

trireg a;
reg b, c;

assign a = (b) ? c : 1'bz;

initial begin
   $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
   b  = 0;
   c  = 0;
   #1  b = 1;
   #1  b = 0;
   #1  c = 1;
   #1  b = 1;
   #1  b = 0;
   #1  $finish;
end

endmodule
```

You could download file test_trireg.v here

Simulator Output

```
 0 a = x b = 0 c = 0
 1 a = 0 b = 1 c = 0
 2 a = 0 b = 0 c = 0
 3 a = 0 b = 0 c = 1
 4 a = 1 b = 1 c = 1
 5 a = 1 b = 0 c = 1
```

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

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

```verilog
//-----------------------------------------------------
// 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

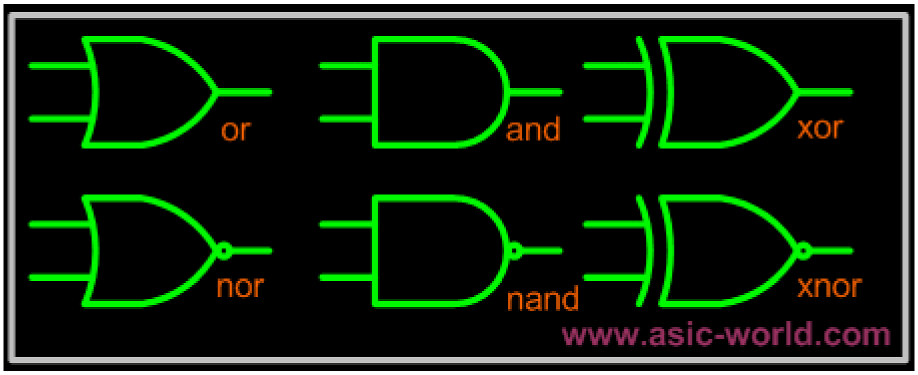
```
This is sample string
```

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. These are rarely used in design (RTL Coding), but are used in post synthesis world for modeling the ASIC/FPGA cells; these cells are then used for gate level simulation, or what is called as SDF simulation. Also the output netlist format from the synthesis tool, which is imported into the place and route tool, is also in Verilog gate level primitives.

Note : RTL engineers still may use gate level primitivies or ASIC library cells in RTL when using IO CELLS, Cross domain synch cells.

Gate Primitives



The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

Gate	Description
and	N-input AND gate
nand	N-input NAND gate
or	N-input OR gate
nor	N-input NOR gate
xor	N-input XOR gate
xnor	N-input XNOR gate

Examples

```
module gates();

wire out0;
wire out1;
wire out2;
reg  in1,in2,in3,in4;

not U1(out0,in1);
and U2(out1,in1,in2,in3,in4);
xor U3(out2,in1,in2,in3);

initial begin
   $monitor(
   "in1=%b in2=%b in3=%b in4=%b out0=%b out1=%b out2=%b",
    in1,in2,in3,in4,out0,out1,out2);
   in1 = 0;
   in2 = 0;
   in3 = 0;
   in4 = 0;
   #1  in1 = 1;
   #1  in2 = 1;
   #1  in3 = 1;
   #1  in4 = 1;
   #1  $finish;
end

endmodule
```

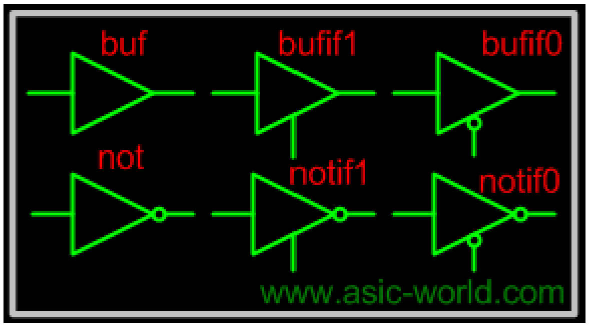
You could download file gates.v here

```
                        in1 = 0 in2 = 0 in3 = 0 in4 = 0 out0 = 1 out1 = 0 out2 = 0
in1 = 1 in2 = 0 in3 = 0 in4 = 0 out0 = 0 out1 = 0 out2 = 1
in1 = 1 in2 = 1 in3 = 0 in4 = 0 out0 = 0 out1 = 0 out2 = 0
in1 = 1 in2 = 1 in3 = 1 in4 = 0 out0 = 0 out1 = 0 out2 = 1
in1 = 1 in2 = 1 in3 = 1 in4 = 1 out0 = 0 out1 = 1 out2 = 1
```

Transmission Gate Primitives

Transmission gates are bi-directional and can be resistive or non-resistive.

Syntax: keyword unique_name (inout1, inout2, control);



Gate	Description
not	N-output inverter
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Transmission gates tran and rtran are permanently on and do not have a control line. Tran can be used to interface two wires with seperate drives, and rtran can be used to weaken signals.

Examples

```
module transmission_gates();

reg data_enable_low, in;
wire data_bus, out1, out2;

bufif0 U1(data_bus,in, data_enable_low);
buf  U2(out1,in);
not U3(out2,in);

initial begin
  $monitor(
    "@%g in=%b data_enable_low=%b out1=%b out2= b data_bus=%b",
    $time, in, data_enable_low, out1, out2, data_bus);
  data_enable_low = 0;
  in = 0;
  #4 data_enable_low = 1;
  #8 $finish;
end

always #2 in = ~in;

endmodule
```

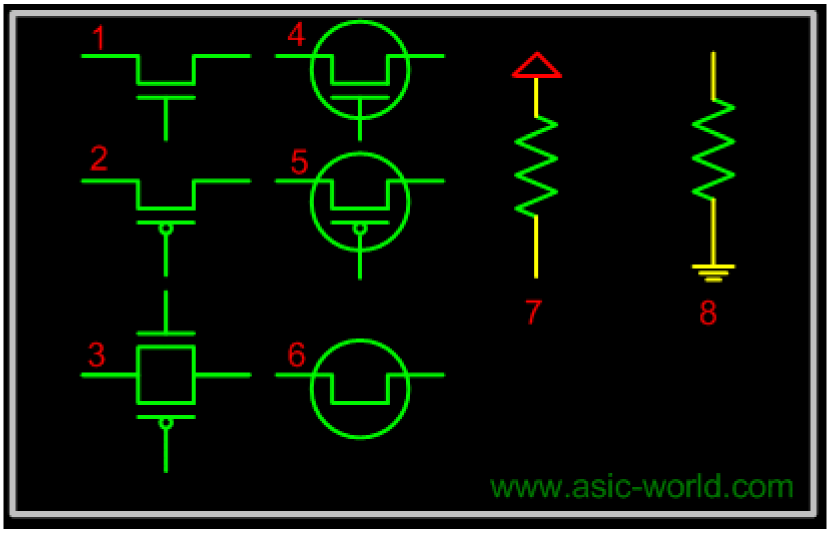
You could download file transmission_gates.v here

```
@0 in = 0 data_enable_low = 0 out1 = 0 out2 = 1 data_bus = 0
@2 in = 1 data_enable_low = 0 out1 = 1 out2 = 0 data_bus = 1
@4 in = 0 data_enable_low = 1 out1 = 0 out2 = 1 data_bus = z
@6 in = 1 data_enable_low = 1 out1 = 1 out2 = 0 data_bus = z
@8 in = 0 data_enable_low = 1 out1 = 0 out2 = 1 data_bus = z
@10 in = 1 data_enable_low = 1 out1 = 1 out2 = 0 data_bus = z
```

Switch Primitives

There are six different switch primitives (transistor models) used in Verilog, nmos, pmos and cmos and the corresponding three resistive versions rnmos, rpmos and rcmos. The cmos type of switches have two gates and so have two control signals.

Syntax: keyword unique_name (drain. source, gate)



Gate	Description
1. pmos	Uni-directional PMOS switch
1. rpmos	Resistive PMOS switch
2. nmos	Uni-directional NMOS switch
2. rnmos	Resistive NMOS switch
3. cmos	Uni-directional CMOS switch
3. rcmos	Resistive CMOS switch

4. tranif1	Bi-directional transistor (High)
4. tranif0	Bi-directional transistor (Low)
5. rtranif1	Resistive Transistor (High)
5. rtranif0	Resistive Transistor (Low)
6. tran	Bi-directional pass transistor
6. rtran	Resistive pass transistor
7. pullup	Pull up resistor
8. pulldown	Pull down resistor

Transmission gates are bi-directional and can be resistive or non-resistive. Resistive devices reduce the signal strength which appears on the output by one level. All the switches only pass signals from source to drain, incorrect wiring of the devices will result in high impedance outputs.

Examples

```
module switch_primitives();

wire  net1, net2, net3;
wire  net4, net5, net6;

tranif0 my_gate1 (net1, net2, net3);
rtranif1 my_gate2 (net4, net5, net6);

endmodule
```

You could download file switch_primitives.v here

Transmission gates tran and rtran are permanently on and do not have a control line. Tran can be used to interface two wires with separate drives, and rtran can be used to weaken signals. Resistive devices reduce the signal strength which appears on the output by one level. All the switches only pass signals from source to drain, incorrect wiring of the devices will result in high impedance outputs.

Logic Values and signal Strengths

The Verilog HDL has got four logic values

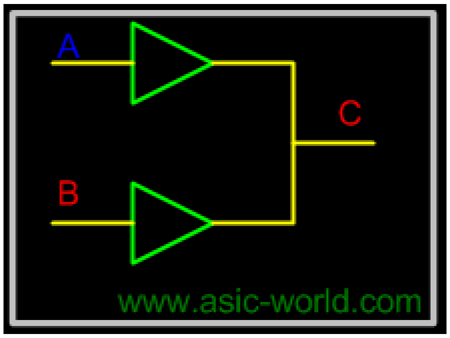
Logic Value	Description
0	zero, low, false
1	one, high, true
z or **Z**	high impedance, floating
x or **X**	unknown, uninitialized, contention

Verilog Strength Levels

Strength Level	Specification Keyword
7 Supply Drive	supply0 supply1
6 Strong Pull	strong0 strong1
5 Pull Drive	pull0 pull1
4 Large Capacitance	large

3 Weak Drive	weak0 weak1
2 Medium Capacitance	medium
1 Small Capacitance	small
0 Hi Impedance	highz0 highz1

Example : Strength Level



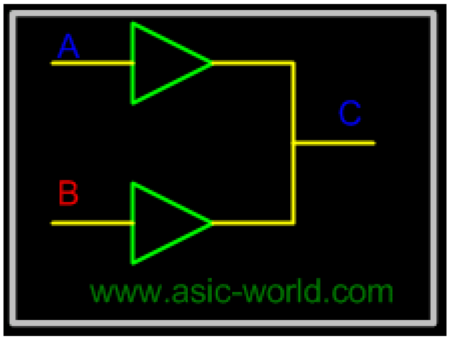
Two buffers that has output

A : Pull 1

B : Supply 0

Since supply 0 is stronger then pull 1, Output C takes value of B.

Example 2 : Strength Level



Two buffers that has output

A : Supply 1

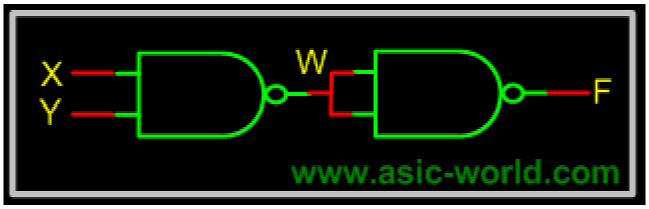
B : Large 1

Since Supply 1 is stronger then Large 1, Output C takes the value of A

Designing Using Primitives

Designing using primitives is used only in library development, where the ASIC vendor provides the ASIC library Verilog description, using Verilog primitives and user defined primitives (UDP).

AND Gate from NAND Gate



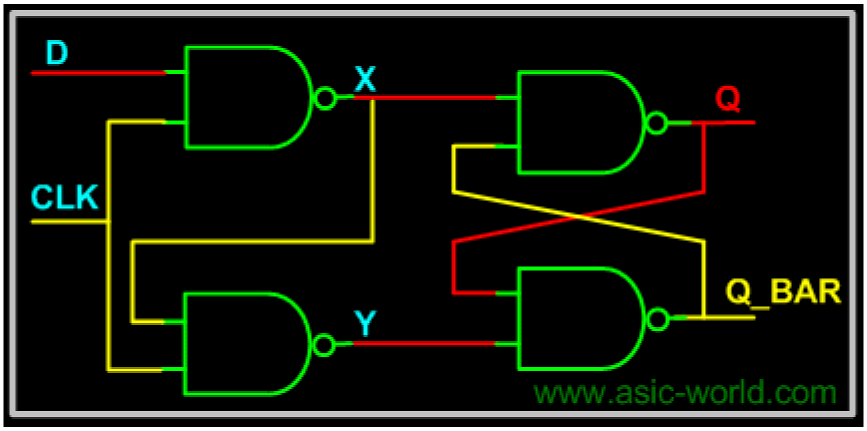
Code

```
// Structural model of AND gate from two NANDS
module and_from_nand();

reg X, Y;
wire F, W;
// Two instantiations of the module NAND
nand U1(W,X, Y);
nand U2(F, W, W);

// Testbench Code
initial  begin
   $monitor ("X = %b Y = %b F = %b", X, Y, F);
   X = 0;
   Y = 0;
   #1  X = 1;
   #1  Y = 1;
   #1  X = 0;
   #1  $finish;
end

endmodule
```

You could download file and_from_nand.v here

```
                    X = 0 Y = 0 F = 0
X = 1 Y = 0 F = 0
X = 1 Y = 1 F = 1
X = 0 Y = 1 F = 0
```

D-Flip flop from NAND Gate



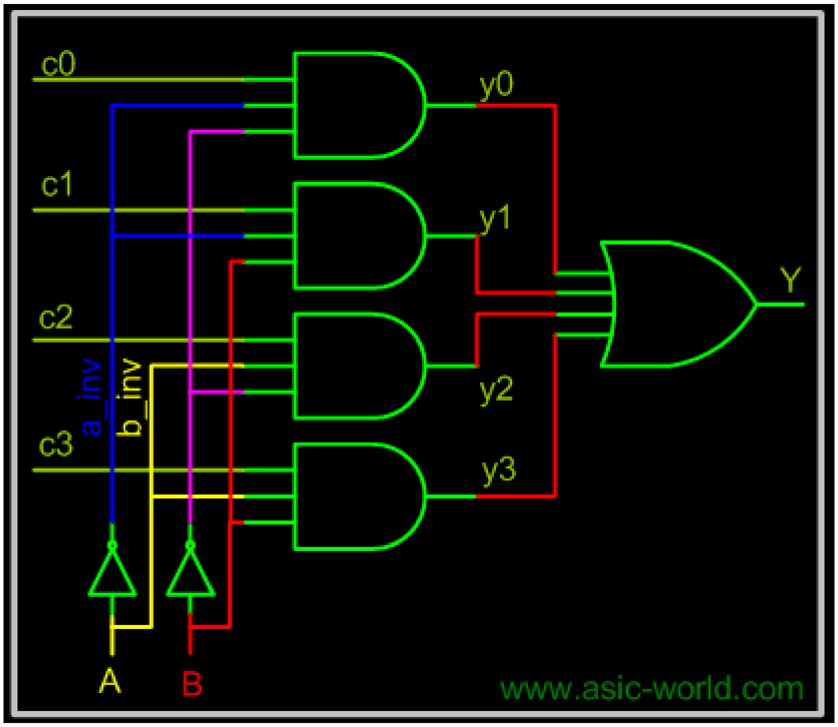
Verilog Code

```
module dff_from_nand();
wire Q,Q_BAR;
reg D,CLK;

nand U1 (X,D,CLK) ;
nand U2 (Y,X,CLK) ;
nand U3 (Q,Q_BAR,X);
nand U4 (Q_BAR,Q,Y);

// Testbench of above code
initial begin
  $monitor("CLK = %b D = %b Q = %b Q_BAR = %b",CLK, D, Q, Q_BAR);
  CLK = 0;
  D = 0;
  #3 D = 1;
  #3 D = 0;
  #3 $finish;
end

always #2 CLK = ~CLK;

endmodule
```

You could download file dff_from_nand.v here

```
                    CLK = 0 D = 0 Q = x Q_BAR = x
CLK = 1 D = 0 Q = 0 Q_BAR = 1
CLK = 1 D = 1 Q = 1 Q_BAR = 0
CLK = 0 D = 1 Q = 1 Q_BAR = 0
CLK = 1 D = 0 Q = 0 Q_BAR = 1
CLK = 0 D = 0 Q = 0 Q_BAR = 1
```

Multiplexer from primitives



Verilog Code

```
module mux_from_gates ();
reg c0,c1,c2,c3,A,B;
```

```
wire Y;
//Invert the sel signals
not (a_inv, A);
not (b_inv, B);
// 3-input AND gate
and (y0,c0,a_inv,b_inv);
and (y1,c1,a_inv,B);
and (y2,c2,A,b_inv);
and (y3,c3,A,B);
// 4-input OR gate
or (Y, y0,y1,y2,y3);

// Testbench Code goes here
initial begin
   $monitor (
     "c0 = %b c1 = %b c2 = %b c3 = %b A = %b B = %b Y = %b",
     c0, c1, c2, c3, A, B, Y);
   c0 = 0;
   c1 = 0;
   c2 = 0;
   c3 = 0;
   A = 0;
   B = 0;
   #1 A = 1;
   #2 B = 1;
   #4 A = 0;
   #8 $finish;
end

always #1 c0 = ~c0;
always #2 c1 = ~c1;
always #3 c2 = ~c2;
always #4 c3 = ~c3;

endmodule
```

You could download file mux_from_gates.v here

```
                    c0 = 0 c1 = 0 c2 = 0 c3 = 0 A = 0 B = 0 Y = 0
c0 = 1 c1 = 0 c2 = 0 c3 = 0 A = 1 B = 0 Y = 0
c0 = 0 c1 = 1 c2 = 0 c3 = 0 A = 1 B = 0 Y = 0
c0 = 1 c1 = 1 c2 = 1 c3 = 0 A = 1 B = 1 Y = 0
c0 = 0 c1 = 0 c2 = 1 c3 = 1 A = 1 B = 1 Y = 1
c0 = 1 c1 = 0 c2 = 1 c3 = 1 A = 1 B = 1 Y = 1
c0 = 0 c1 = 1 c2 = 0 c3 = 1 A = 1 B = 1 Y = 1
c0 = 1 c1 = 1 c2 = 0 c3 = 1 A = 0 B = 1 Y = 1
c0 = 0 c1 = 0 c2 = 0 c3 = 0 A = 0 B = 1 Y = 0
```

```
    c0 = 1 c1 = 0 c2 = 1 c3 = 0 A = 0 B = 1 Y = 0
    c0 = 0 c1 = 1 c2 = 1 c3 = 0 A = 0 B = 1 Y = 1
    c0 = 1 c1 = 1 c2 = 1 c3 = 0 A = 0 B = 1 Y = 1
    c0 = 0 c1 = 0 c2 = 0 c3 = 1 A = 0 B = 1 Y = 0
    c0 = 1 c1 = 0 c2 = 0 c3 = 1 A = 0 B = 1 Y = 0
 c0 = 0 c1 = 1 c2 = 0 c3 = 1 A = 0 B = 1 Y = 1
```

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.

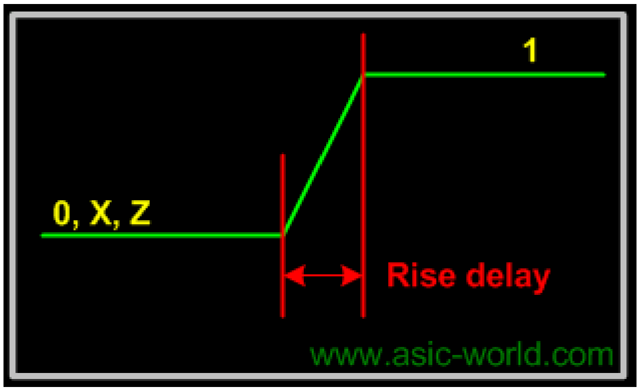
Syntax: keyword #(delay{s}) unique_name (node specifications);

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

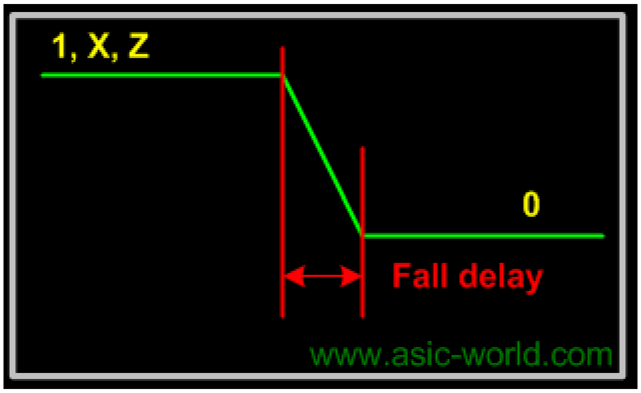
Rise Delay

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

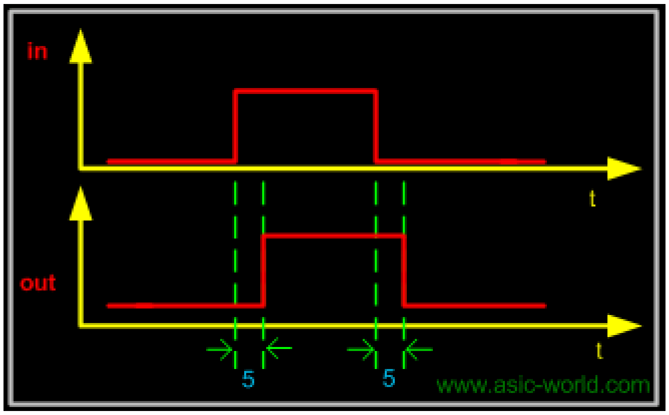
Below are some examples to show the usage of delays.

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
```

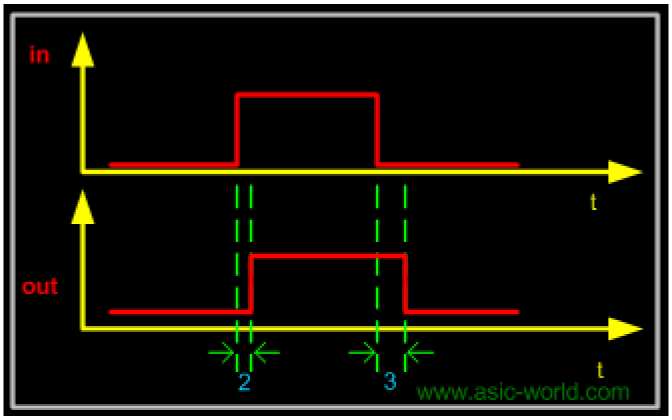


Example - Two Delays

```
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
```

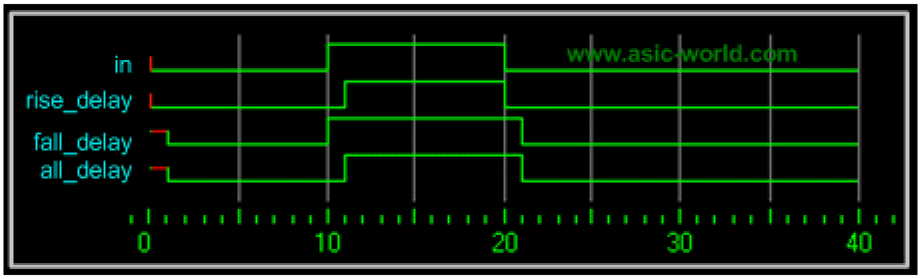


Example - All Delays

```
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
```

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

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));
```

```verilog
  $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	? :

Verilog HDL Abstraction Levels

- Behavioral Models : Higher level of modeling where behavior of logic is modeled.
- RTL Models : Logic is modeled at register level
- Structural Models : Logic is modeled at both register level and gate level.

Procedural Blocks

Verilog behavioral code is inside procedure blocks, but there is an exception: some behavioral code also exist outside procedure blocks. We can see this in detail as we make progress.

There are two types of procedural blocks in Verilog:

- **initial :** initial blocks execute only once at time zero (start execution at time zero).
- **always :** always blocks loop to execute over and over again; in other words, as the name suggests, it executes always.

Example - initial

```verilog
module initial_example();
reg clk,reset,enable,data;

initial begin
  clk = 0;
  reset = 0;
  enable = 0;
  data = 0;
end

endmodule
```

You could download file initial_example.v here

In the above example, the initial block execution and always block execution starts at time 0. Always block waits for the event, here positive edge of clock, whereas initial block just executed all the statements within begin and end statement, without waiting.

Example - always

```
module always_example();
reg clk,reset,enable,q_in,data;

always @ (posedge clk)
if (reset)  begin
   data <= 0;
end else if (enable) begin
   data <= q_in;
end

endmodule
```

You could download file always_example.v here

In an always block, when the trigger event occurs, the code inside begin and end is executed; then once again the always block waits for next event triggering. This process of waiting and executing on event is repeated till simulation stops.

Procedural Assignment Statements

- Procedural assignment statements assign values to reg, integer, real, or time variables and can not assign values to nets (wire data types)
- You can assign to a register (reg data type) the value of a net (wire), constant, another register, or a specific value.

Example - Bad procedural assignment

```
module initial_bad();
reg clk,reset;
wire enable,data;

initial begin
 clk = 0;
```

```
  reset = 0;
  enable = 0;
  data = 0;
end

endmodule
```

You could download file initial_bad.v here

Example - Good procedural assignment

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

initial begin
  clk = 0;
  reset = 0;
  enable = 0;
  data = 0;
end

endmodule
```

You could download file initial_good.v here

Procedural Assignment Groups

If a procedure block contains more than one statement, those statements must be enclosed within

- Sequential **begin - end** block
- Parallel **fork - join** block

When using begin-end, we can give name to that group. This is called **named blocks**.

Example - "begin-end"

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

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

endmodule
```

You could download file initial_begin_end.v here

Begin : clk gets 0 after 1 time unit, reset gets 0 after 11 time units, enable after 16 time units, data after 19 units. All the statements are executed sequentially.

Simulator Output

```
                 0 clk=x reset=x enable=x data=x
 1 clk=0 reset=x enable=x data=x
 11 clk=0 reset=0 enable=x data=x
 16 clk=0 reset=0 enable=0 data=x
 19 clk=0 reset=0 enable=0 data=0
```

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
```

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 statment, 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;
```

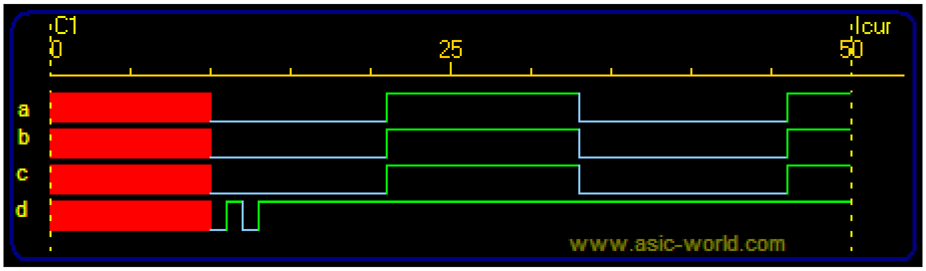
```
end

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



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

```verilog
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
```

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

Continuous Assignment Statements

Continuous assignment statements drive nets (wire data type). They represent structural connections.

- They are used for modeling Tri-State buffers.
- They can be used for modeling combinational logic.
- They are outside the procedural blocks (always and initial blocks).
- The continuous assign overrides any procedural assignments.
- The left-hand side of a continuous assignment must be net data type.

syntax : assign (strength, strength) **#(delay)** net = expression;

Example - One bit Adder

```
module adder_using_assign ();
reg a, b;
wire sum, carry;

assign #5  {carry,sum} = a+b;

initial begin
   $monitor (" A = %b  B = %b CARRY = %b SUM = %b",a,b,carry,sum);
   #10  a = 0;
   b = 0;
   #10   a = 1;
   #10   b = 1;
   #10   a = 0;
   #10   b = 0;
   #10  $finish;
end

endmodule
```

You could download file adder_using_assign.v here

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

Propagation Delay

Continuous Assignments may have a delay specified; only one delay for all transitions may be specified. A minimum:typical:maximum delay range may be specified.

Example - Tri-state buffer

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

assign #(1:2:3) pad = (enable) ? data_in : 1'bz;

initial begin
  $monitor ("ENABLE = %b DATA : %b PAD %b",enable,
data_in,pad);
  #10 enable = 0;
  #10 data_in = 1;
  #10 enable = 1;
  #10 data_in = 0;
  #10 enable = 0;
```

```
   #10  $finish;
end

endmodule
```

You could download file tri_buf_using_assign_delays.v here

Procedural Block Control

Procedural blocks become active at simulation time zero. Use level sensitive event controls to control the execution of a procedure.

```
module dlatch_using_always();
reg q;

reg d, enable;

always @ (d or enable)
if (enable) begin
  q = d;
end

initial begin
  $monitor (" ENABLE = %b D = %b  Q = %b",enable,d,q);
  #1 enable = 0;
  #1 d = 1;
  #1 enable = 1;
  #1 d = 0;
  #1 d = 1;
  #1 d = 0;
  #1 enable = 0;
  #10 $finish;
end

endmodule
```

You could download file dlatch_using_always.v here

Any change in either d or enable satisfies the event control and allows the execution of the statements in the procedure. The procedure is sensitive to any change in d or enable.

Combo Logic using Procedural Coding

To model combinational logic, a procedure block must be sensitive to any change on the input. There is one important rule that needs to be followed while modeling combinational logic. If you use conditional checking using "if", then you need to mention the "else" part. Missing the else part results in a latch. If you don't like typing the else part, then you must initialize all the variables of that combo block as soon as it enters.

Example - One bit Adder

```
module adder_using_always ();
reg a, b;
reg sum, carry;

always @ (a or b)
begin
  {carry,sum} = a + b;
end

initial begin
  $monitor (" A = %b B = %b CARRY = %b SUM = %b",a,b,carry,sum);
  #10 a = 0;
   b = 0;
   #10 a = 1;
   #10 b = 1;
   #10 a = 0;
   #10 b = 0;
   #10 $finish;
end

endmodule
```

You could download file adder_using_always.v here

The statements within the procedural block work with entire vectors at a time.

Example - 4-bit Adder

```
module adder_4_bit_using_always ();
reg[3:0] a, b;
reg [3:0] sum;
reg carry;

always @ (a or b)
begin
  {carry,sum} = a + b;
end

initial begin
  $monitor (" A = %b B = %b CARRY = %b SUM = %b",a,b,carry,sum);
  #10 a = 8;
   b = 7;
   #10 a = 10;
   #10 b = 15;
   #10 a = 0;
   #10 b = 0;
   #10 $finish;
end

endmodule
```

You could download file adder_4_bit_using_always.v here

Example - Ways to avoid Latches - Cover all conditions

```
module avoid_latch_else ();

reg q;
reg enable, d;

always @ (enable or d)
if (enable) begin
  q = d;
end else begin
```

```
  q = 0;
end

initial begin
  $monitor (" ENABLE = %b  D = %b Q = %b",enable,d,q);
  #1 enable = 0;
  #1 d = 0;
  #1 enable = 1;
  #1 d = 1;
  #1 d = 0;
  #1 d = 1;
  #1 d = 0;
  #1 d = 1;
  #1 enable = 0;
  #1 $finish;
end

endmodule
```

You could download file avoid_latch_else.v here

Example - Ways to avoid Latches - Snit the variables to zero

```
module avoid_latch_init ();

reg q;
reg enable, d;

always @ (enable or d)
begin
  q = 0;
  if (enable) begin
    q = d;
  end
end

initial begin
  $monitor (" ENABLE = %b  D = %b Q = %b",enable,d,q);
  #1 enable = 0;
  #1 d = 0;
  #1 enable = 1;
  #1 d = 1;
  #1 d = 0;
  #1 d = 1;
  #1 d = 0;
  #1 d = 1;
  #1 enable = 0;
  #1 $finish;
end

endmodule
```

You could download file avoid_latch_init.v here

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

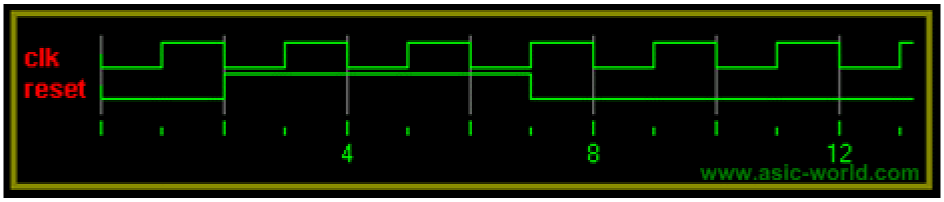
```
module clk_gen ();

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
```

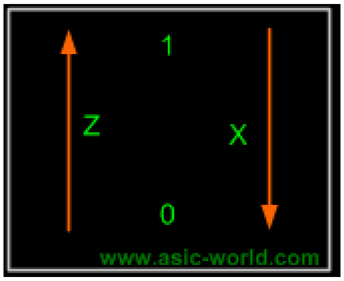
Waveform



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

```
module intra_assign();

reg a, b;
```

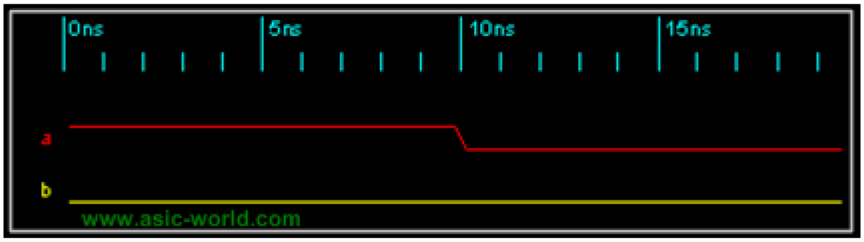
```

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

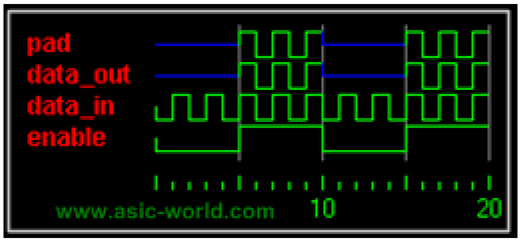
```verilog
module tri_buf_using_assign();
reg data_in, enable;
wire pad;

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

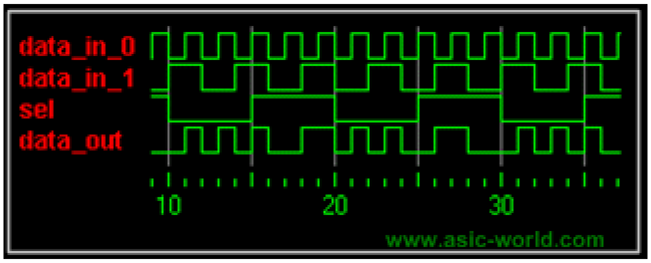
```
module mux_using_assign();
reg data_in_0, data_in_1;
wire data_out;
reg  sel;

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



Task

Tasks are used in all programming languages, generally known as procedures or subroutines. The lines of code are enclosed in task....end task brackets. Data is passed to the task, the processing done, and the result returned. They have to be specifically called, with data ins and outs, rather than just wired in to the general netlist. Included in the main body of code, they can be called many times, reducing code repetition.

- tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive 'include to include the task in the file which instantiates the task.
- tasks can include timing delays, like posedge, negedge, # delay and wait.
- tasks can have any number of inputs and outputs.
- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- tasks can take, drive and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of task execution.
- tasks can call another task or function.
- tasks can be used for modeling both combinational and sequential logic.
- A task must be specifically called with a statement, it cannot be used within an expression as a function can.

Syntax

- A task begins with keyword task and ends with keyword endtask
- Inputs and outputs are declared after the keyword task.
- Local variables are declared after input and output declaration.

Example - Simple Task

```
module simple_task();

task convert;
```

```
input [7:0] temp_in;
output [7:0] temp_out;
begin
  temp_out = (9/5) *( temp_in + 32)
end
endtask

endmodule
```

You could download file simple_task.v here

Example - Task using Global Variables

```
module task_global();

reg [7:0] temp_out;
reg [7:0] temp_in;

task convert;
begin
  temp_out = (9/5) *( temp_in + 32);
end
endtask

endmodule
```

You could download file task_global.v here

Calling a Task

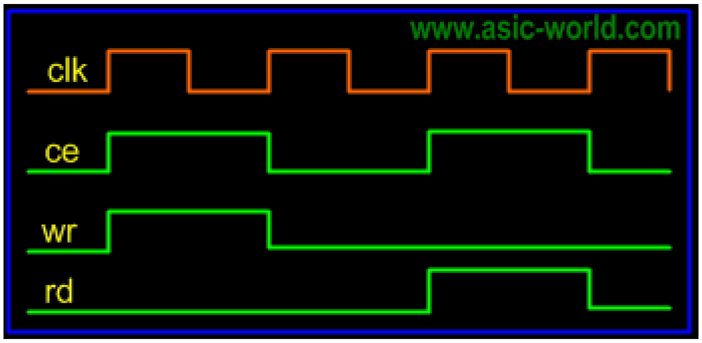
Let's assume that the task in example 1 is stored in a file called mytask.v. Advantage of coding a task in a separate file, is that it can be used in multiple modules.

```
module  task_calling (temp_a, temp_b, temp_c, temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
reg [7:0] temp_b, temp_d;
`include "mytask.v"

always @ (temp_a)
begin
  convert (temp_a, temp_b);
end

always @ (temp_c)
begin
  convert (temp_c, temp_d);
end

endmodule
```

You could download file task_calling.v here

Example - CPU Write / Read Task

Below is the waveform used for writing into memory and reading from memory. We make the assumption that there is a need to use this interface from multiple agents. So we write the read/write as tasks.



```
module bus_wr_rd_task();

```

```
reg clk,rd,wr,ce;
reg [7:0]  addr,data_wr,data_rd;
reg [7:0]  read_data;

initial  begin
  clk = 0;
  read_data = 0;
  rd = 0;
  wr = 0;
  ce = 0;
  addr = 0;
  data_wr = 0;
  data_rd = 0;
  // Call the write and read tasks here
  #1  cpu_write(8'h11,8'hAA);
  #1  cpu_read(8'h11,read_data);
  #1  cpu_write(8'h12,8'hAB);
  #1  cpu_read(8'h12,read_data);
  #1  cpu_write(8'h13,8'h0A);
  #1  cpu_read(8'h13,read_data);
  #100  $finish;
end
// Clock Generator
always
  #1  clk = ~clk;
// CPU Read Task
task cpu_read;
  input  [7:0]  address;
  output  [7:0] data;
  begin
    $display  ("%g CPU Read  task with address : %h", $time, address);
    $display  ("%g  -> Driving CE, RD and ADDRESS on to bus", $time);
    @  (posedge  clk);
    addr = address;
    ce = 1;
    rd = 1;
    @  (negedge  clk);
    data = data_rd;
    @  (posedge  clk);
    addr = 0;
    ce = 0;
    rd = 0;
    $display  ("%g CPU Read  data          : %h", $time, data);
    $display  ("======================");
  end
endtask
// CU Write Task
task cpu_write;
```

```
  input [7:0]  address;
  input [7:0] data;
  begin
    $display ("%g CPU Write task with address : %h Data : %h",
      $time, address,data);
    $display ("%g  -> Driving CE, WR, WR data and ADDRESS on to bus",
      $time);
    @ (posedge clk);
    addr = address;
    ce = 1;
    wr = 1;
    data_wr = data;
    @ (posedge clk);
    addr = 0;
    ce = 0;
    wr = 0;
    $display ("=====================");
  end
endtask

// Memory model for checking tasks
reg [7:0] mem [0:255];

always @ (addr or ce or rd or wr or data_wr)
if (ce) begin
  if (wr) begin
    mem[addr] = data_wr;
  end
  if (rd) begin
    data_rd = mem[addr];
  end
end

endmodule
```

You could download file bus_wr_rd_task.v here

Simulation Output

```
                   1 CPU Write task with address : 11 Data : aa
1  -> Driving CE, WR, WR data and ADDRESS on to bus
=====================
4 CPU Read  task with address : 11
4  -> Driving CE, RD and ADDRESS on to bus
7 CPU Read  data              : aa
```

```
======================
8 CPU Write task with address : 12 Data : ab
8  -> Driving CE, WR, WR data and ADDRESS on to bus
======================
12 CPU Read  task with address : 12
12  -> Driving CE, RD and ADDRESS on to bus
15 CPU Read  data              : ab
======================
16 CPU Write task with address : 13 Data : 0a
16  -> Driving CE, WR, WR data and ADDRESS on to bus
======================
20 CPU Read  task with address : 13
20  -> Driving CE, RD and ADDRESS on to bus
23 CPU Read  data              : 0a
======================
```

Function

A Verilog HDL function is the same as a task, with very little differences, like function cannot drive more than one output, can not contain delays.

- functions are defined in the module in which they are used. It is possible to define functions in separate files and use compile directive 'include to include the function in the file which instantiates the task.
- functions **can not include timing delays**, like posedge, negedge, # delay, which means that functions should be executed in "zero" time delay.
- functions can have any number of inputs but only one output.
- The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.
- functions can take, drive, and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution.
- functions can be used for **modeling combinational logic**.
- functions can **call other functions, but can not call tasks**.

Syntax

- A function begins with keyword **function** and ends with keyword **endfunction**
- **inputs** are declared after the keyword function.

Example - Simple Function

```
module simple_function();

function  myfunction;
input a, b, c, d;
begin
  myfunction = ((a+b) + (c-d));
end
endfunction

endmodule
```

You could download file simple_function.v here

Example - Calling a Function

```
module  function_calling(a, b, c, d, e, f);

input a, b, c, d, e ;
output f;
wire f;
`include "myfunction.v"

assign f =  (myfunction (a,b,c,d)) ? e :0;

endmodule
```

You could download file function_calling.v here

Introduction

There are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign ($). The synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models.

$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.

I have picked up some examples from the VLSI classes that I used to teach during 1999-2001, when I was in Chennai. Please feel free to give your feedback on how to improve the tutorial below.

Before you Start

For writing testbenches it is important to have the design specification of "design under test" or simply DUT. Specs need to be understood clearly and a test plan, which basically documents the test bench architecture and the test scenarios (test cases) in detail, needs to be made.

Example - Counter

Let's assume that we have to verify a simple 4-bit up counter, which increments its count whenever enable is high, and resets to zero when reset is asserted high. Reset is synchronous to clock.

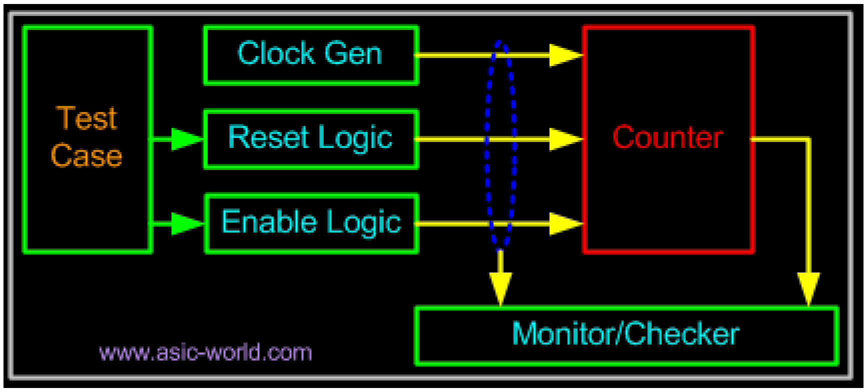
Code for Counter

```
//-----------------------------------------------------
// Design Name : counter
// File Name   : counter.v
// Function    : 4 bit up counter
// Coder       : Deepak
//-----------------------------------------------------
module counter (clk, reset, enable, count);
input clk, reset, enable;
output [3:0] count;
reg [3:0] count;

always @ (posedge clk)
if (reset == 1'b1) begin
  count <= 0;
end else if ( enable == 1'b1) begin
  count <= count + 1;
end

endmodule
```

You could download file counter.v here

Test Plan

We will write a self-checking test bench, but we will do this in steps to help you understand the concept of writing automated test benches. Our testbench environment will look something like the figure below.



DUT is instantiated in the testbench, and the testbench will contain a clock generator, reset generator, enable logic generator and compare logic, which basically calculates the expected count value of counter and compares it with the output of counter.

Test Cases

- Reset Test : We can start with reset de-asserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.
- Enable Test : Assert/deassert enable after reset is applied.
- Random Assert/deassert of enable and reset.

We can add some more test cases; but we are not here to test the counter, rather to learn how to write test benches.

Writing a TestBench

First step of any testbench creation is building a dummy template which basically declares inputs to DUT as reg and outputs from DUT as wire, then instantiates the DUT as shown in the code below. Note that there is no port list for the test bench.

Test Bench

```
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;

  counter U0 (
  .clk    (clk),
  .reset  (reset),
  .enable (enable),
  .count  (count)
  );

endmodule
```

You could download file counter_tb1.v here

Next step would be to add clock generator logic: this is straight forward, as we know how to generate a clock. Before we add a clock generator we need to drive all the inputs to DUT to some known state as shown in the code below.

Test Bench with Clock generator

```
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;

  counter U0 (
  .clk    (clk),
  .reset  (reset),
  .enable (enable),
  .count  (count)
  );

  initial
  begin
    clk = 0;
    reset = 0;
    enable = 0;
  end

  always
    #5  clk = ! clk;

endmodule
```

You could download file counter_tb2.v here

An initial block in Verilog is executed only once, thus simulator sets the value of clk, reset and enable to 0; by looking at the counter code (of course you will be referring to the DUT specs) could be found that driving 0 makes all these signals disabled.

There are many ways to generate a clock: one could use a forever loop inside an initial block as an alternative to the above code. You could a add parameter or use `define to control the clock frequency. You may write a complex clock generator, where we could introduce PPM (Parts per million, clock width drift), then control the duty cycle. All the above depends on the specs of the DUT and the creativity of a "Test Bench Designer".

At this point, you would like to test if the testbench is generating the clock correctly: well you can compile it with any Verilog simulator. You need to give command line options as shown below.

```
C:\www.asic-world.com\veridos counter.v counter_tb.v
```

Of course it is a very good idea to keep file names the same as the module name. Ok, coming back to compiling, you will see that the simulator does print anything on screen, or dump any waveform. Thus we need to add support for all the above as shown in the code below.

Test Bench continues...

```
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] count;

  counter U0 (
  .clk     (clk),
  .reset   (reset),
  .enable (enable),
  .count   (count)
  );

  initial begin
    clk = 0;
    reset = 0;
    enable = 0;
  end

  always
    #5 clk = ! clk;

  initial  begin
    $dumpfile ("counter.vcd");
    $dumpvars;
  end

  initial  begin
    $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
    $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,count);
  end

  initial
  #100  $finish;

  //Rest of testbench code after this line

endmodule
```

You could download file counter_tb3.v here

$dumpfile is used for specifying the file that the simulator will use to store the waveform, that can be used later using a waveform viewer. (Please refer to the tools section for freeware versions of viewers.) $dumpvars basically instructs the Verilog compiler to start dumping all the signals to "counter.vcd".

$display is used for printing text or variables to stdout (screen), \t is for inserting tabs. The syntax is the same as for printf C language. $monitor in the second line is a bit different: $monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). Whenever any of them changes, it prints their value, in the respective radix specified.

$finish is used for terminating the simulation after #100 time units (note: all the initial, always blocks start execution at time 0).

Now that we have written the basic skeleton, let's compile and see what we have just coded. Output of the simulator is shown below.

```
                C:\www.asic-world.com>veridos counter.v counter_tb.v
VeriWell for Win32 HDL Version 2.1.4 Fri Jan 17 21:33:25 2003

This is a free version of the VeriWell for Win32 Simulator
Distribute this freely; call 1-800-VERIWELL for ordering information
See the file "!readme.1st" for more information

Copyright (c) 1993-97 Wellspring Solutions, Inc.
All rights reserved


Memory Available: 0
Entering Phase I...
Compiling source file : counter.v
Compiling source file : counter_tb.v
The size of this model is [2%, 5%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
```

```
counter_tb

  time   clk,       reset,    enable, count
  0,                0,        0,         0,       x
  5,                1,        0,         0,       x
  10,    0,         0,        0,         x
  15,    1,         0,        0,         x
  20,    0,         0,        0,         x
  25,    1,         0,        0,         x
  30,    0,         0,        0,         x
  35,    1,         0,        0,         x
  40,    0,         0,        0,         x
  45,    1,         0,        0,         x
  50,    0,         0,        0,         x
  55,    1,         0,        0,         x
  60,    0,         0,        0,         x
  65,    1,         0,        0,         x
  70,    0,         0,        0,         x
  75,    1,         0,        0,         x
  80,    0,         0,        0,         x
  85,    1,         0,        0,         x
  90,    0,         0,        0,         x
  95,    1,         0,        0,         x

Exiting VeriWell for Win32 at time 100
0 Errors, 0 Warnings, Memory Used: 0
Compile time = 0.0 Load time = 0.0 Simulation time = 0.1

Normal exit
Thank you for using VeriWell for Win32
```

Adding Reset Logic

Once we have the basic logic to allow us to see what our testbench is doing, we can next add the reset logic. If we look at the testcases, we see that we had added a constraint that it should be possible to activate reset anytime during simulation. To achieve this we have many approaches, but I am going to teach something that will go long way. There is something called 'events' in Verilog: events can be triggered, and also monitored, to see if an event has occurred.

Let's code our reset logic in such a way that it waits for the trigger event "reset_trigger": when this event happens, reset logic asserts reset at negative edge of clock and de-asserts on next negative edge as shown in the code below. Also after de-asserting the reset, reset logic triggers another event called "reset_done_trigger". This trigger event can then be used somewhere else in the testbench to sync up.

Code of reset logic

```verilog
event reset_trigger;
  event  reset_done_trigger;

  initial  begin
    forever  begin
      @  (reset_trigger);
      @  (negedge  clk);
      reset = 1;
      @  (negedge  clk);
      reset = 0;
      -> reset_done_trigger;
    end
  end
```

You could download file counter_tb4.v here

Adding test case logic

Moving forward, let's add logic to generate the test cases, ok we have three testcases as in the first part of this tutorial. Let's list them again. ☺

- Reset Test : We can start with reset de-asserted, followed by asserting reset for few clock ticks and de-asserting the reset, See if counter sets its output to zero.
- Enable Test : Assert/de-assert enable after reset is applied.

- Random Assert/de-assert of enable and reset.

Repeating it again: "There are many ways" to code a test case, it all depends on the creativity of the Test bench designer. Let's take a simple approach and then slowly build upon it.

Test Case 1 - Asserting/ De-asserting reset

In this test case, we will just trigger the event reset_trigger after 10 simulation units.

```
initial
  begin : TEST_CASE
    #10 -> reset_trigger;
  end
```

You could download file counter_tb5.v here

Test Case 2 - Assert/ De-assert enable after reset is applied.

In this test case, we will trigger the reset logic and wait for the reset logic to complete its operation, before we start driving the enable signal to logic 1.

```
initial
 begin : TEST_CASE
   #10 -> reset_trigger;
   @ (reset_done_trigger);
   @ (negedge clk);
   enable = 1;
   repeat (10) begin
     @ (negedge clk);
   end
   enable = 0;
 end
```

You could download file counter_tb6.v here

Test Case 3 - Assert/De-assert enable and reset randomly.

In this testcase we assert the reset, and then randomly drive values on to enable and reset signal.

```
initial
 begin : TEST_CASE
   #10  -> reset_trigger;
   @ (reset_done_trigger);
   fork
     repeat (10) begin
        @ (negedge clk);
       enable = $random;
     end
     repeat (10) begin
       @ (negedge clk);
       reset = $random;
     end
   join
 end
```

You could download file counter_tb7.v here

Well you might ask, do all this three test case exist in same file? Well, the answer is no. If we try to have all three test cases on one file, then we end up having race conditions due to three initial blocks driving reset and enable signal. So normally, once test bench coding is done, test cases are coded separately and included in testbench with `include directives as shown below. (There are better ways to do this, but you have to think how you want to do it).

If you look closely at all the three test cases, you will find that even though test case execution is not complete, simulation terminates. To have better control, what we can do is adding an event like "terminate_sim" and execute $finish only when this event is triggered. We can trigger this event at the end of test case execution. The code for $finish now could look as shown below.

```
event terminate_sim;
initial begin
@ (terminate_sim);
  #5 $finish;
end
```

You could download file counter_tb8.v here

The modified test case #2 would be like:

```
initial
  begin: TEST_CASE
    #10 -> reset_trigger;
    @ (reset_done_trigger);
    @ (negedge clk);
    enable = 1;
    repeat (10) begin
      @ (negedge clk);
    end
    enable = 0;
    #5 -> terminate_sim;
  end

```

You could download file counter_tb9.v here

Second problem with the approach that we have taken till now is that we need to manually check the waveform and also the simulator output on the screen to see if the DUT is working correctly. Part IV shows how to automate this.

Adding compare Logic

To make any testbench self checking/automated, first we need to develop a model that mimics the DUT in functionality. In our example, it's going to be very easy, but at times if the DUT is complex, then to mimic it will be very complex and will require a lot of innovative techniques to make self-checking work.

```
reg [3:0] count_compare;

always @ (posedge clk)
if (reset == 1'b1) begin
  count_compare <= 0;
end else if ( enable == 1'b1) begin
  count_compare <= count_compare + 1;
end
```

You could download file counter_tb10.v here

Once we have the logic to mimic the DUT functionality, we need to add the checker logic, which at any given point keeps checking the expected value with the actual value. Whenever there is any error, it prints out the expected and actual value, and also terminates the simulation by triggering the event "terminate_sim".

```
always @ (posedge clk)
  if (count_compare != count) begin
    $display ("DUT Error at time %d", $time);
    $display (" Expected value %d, Got Value %d", count_compare, count);
    #5 -> terminate_sim;
  end
```

You could download file counter_tb11.v here

Now that we have the all the logic in place, we can remove $display and $monitor, as our testbench have become fully automatic, and we don't require to manually verify the DUT input and output. Try changing the count_compare = count_compare +2, and see how compare logic works. This is just another way to see if our testbench is stable.

We could add some fancy printing as shown in the figure below to make our test environment more friendly.

```
C:\Download\work>veridos counter.v counter_tb.v
VeriWell for Win32 HDL  Sat Jan 18 20:10:35 2003

This is a free version of the VeriWell for Win32 Simulator
Distribute this freely; call 1-800-VERIWELL for ordering information
See the file "!readme.1st" for more information

Copyright (c) 1993-97 Wellspring Solutions, Inc.
All rights reserved


Memory Available: 0
Entering Phase I...
Compiling source file : counter.v
Compiling source file : counter_tb.v
The size of this model is [5%, 6%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
counter_tb

########################################################
Applying reset
Came out of Reset
Terminating simulation
Simulation Result : PASSED
#######################################################
Exiting VeriWell for Win32 at time 96
0 Errors, 0 Warnings, Memory Used: 0
Compile time = 0.0, Load time = 0.0, Simulation time = 0.0

Normal exit
Thank you for using VeriWell for Win32
```

I know, you would like to see the test bench code that I used to generate the above output, well you can find it here and counter code here.

There are a lot of things that I have not covered; maybe when I find time, I may add some more details on this subject.

As to books, I am yet to find a good book on writing test benches.

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.

Refer to the examples section for more details on different types of memories.

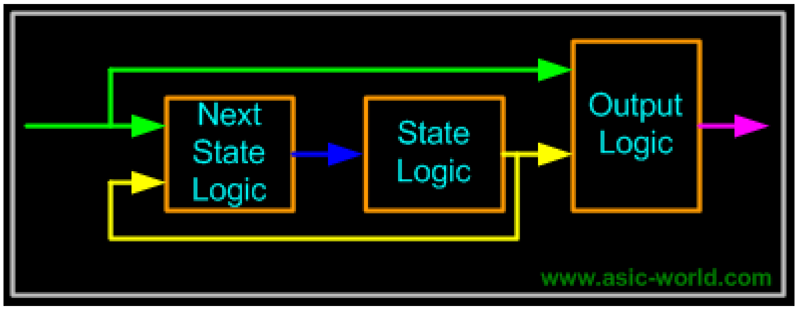
Introduction to FSM

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.

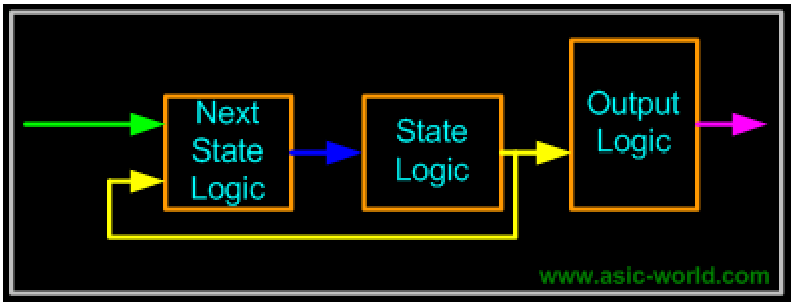
State machine Types

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.

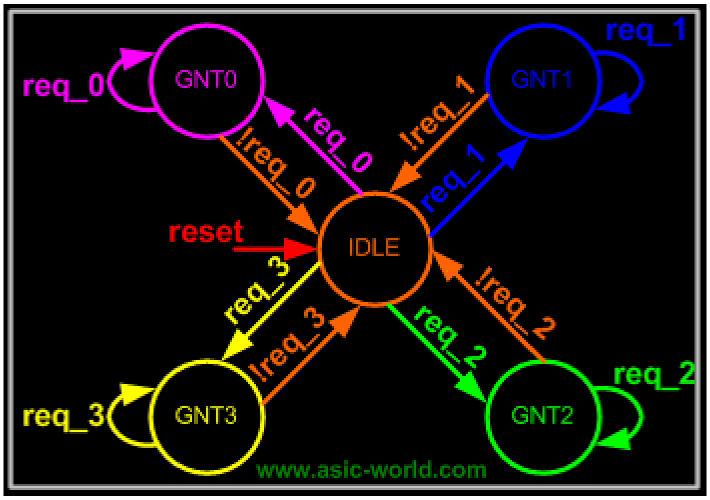
Modeling State machines.

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
```

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

Verilog 2001

In Verilog 2001, the code above will work, but the new feature makes the code more readable and error free.

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

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

endmodule
```

You could download file param_more_then_one2.v here

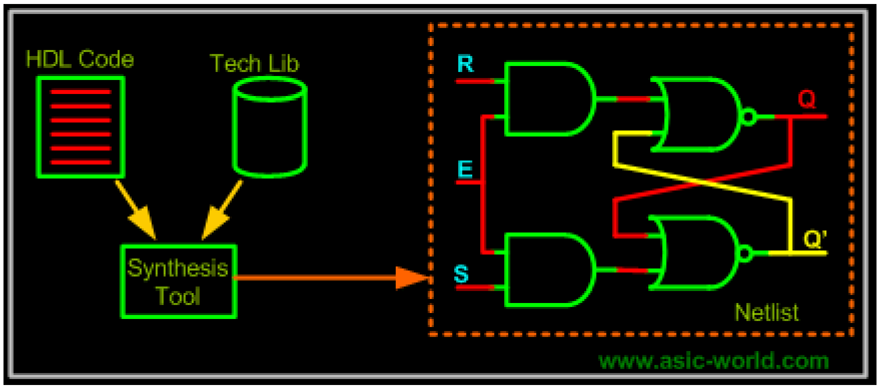
Was this copied from VHDL?

What is logic synthesis ?

Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation. Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops. Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).

A circuit description is written in Hardware Description Language (HDL) such as Verilog. The designer should first understand the architectural description. Then he should consider design constraints such as timing, area, testability, and power.

We will see a typical design flow with a large example in the last chapter of Verilog tutorial.



Life before HDL (Logic synthesis)

As you must have experienced in college, everything (all the digital circuits) is designed manually. Draw K-maps, optimize the logic, draw the schematic. This is how engineers used to design digital logic circuits in early days. Well this works fine as long as the design is a few hundred gates.

Impact of HDL and Logic synthesis.

High-level design is less prone to human error because designs are described at a higher level of abstraction. High-level design is done without significant concern about design constraints. Conversion from high-level design to gates is done by synthesis tools, using various algorithms to optimize the design as a whole. This removes the problem with varied designer styles for the different blocks in the design and suboptimal designs. Logic synthesis tools allow technology independent design. Design reuse is possible for technology-independent descriptions.

What do we discuss here ?

When it comes to Verilog, the synthesis flow is the same as for the rest of the languages. What we try to look in next few pages is how particular code gets translated to gates. As you must have wondered while reading earlier chapters, how could this be represented in Hardware ? An example would be "delays". There is no way we could synthesize delays, but of course we can add delay to particular signals by adding buffers. But then this becomes too dependent on synthesis target technology. (More on this in the VLSI section).

First we will look at the constructs that are not supported by synthesis tools; the table below shows the constructs that are not supported by the synthesis tool.

Constructs Not Supported in Synthesis

Construct Type	Notes
initial	Used only in test benches.
events	Events make more sense for syncing test bench components.
real	Real data type not supported.
time	Time data type not supported.
force and release	Force and release of data types not supported.
assign and deassign	assign and deassign of reg data types is not supported. But assign on wire data type is supported.
fork join	Use nonblocking assignments to get same effect.
primitives	Only gate level primitives are supported.
table	UDP and tables are not supported.

Example of Non-Synthesizable Verilog construct.

Any code that contains the above constructs are not synthesizable, but within synthesizable constructs, bad coding could cause synthesis issues. I have seen codes where engineers code a flip-flop with both posedge of clock and negedge of clock in sensitivity list.

Then we have another common type of code, where one reg variable is driven from more than one always block. Well it will surely cause synthesis error.

✦ Example - Initial Statement

```
module synthesis_initial(
clk,q,d);
input clk,d;
output q;
reg q;

initial begin
 q <= 0;
end

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

endmodule
```

You could download file synthesis_initial.v here

✦ Delays

a = #10 b; This code is useful only for simulation purpose.

Synthesis tool normally ignores such constructs, and just assumes that there is no #10 in above statement, thus treating above code as

a = b;

Comparison to X and Z are always ignored

```verilog
module synthesis_compare_xz (a,b);
output a;
input b;
reg a;

always @ (b)
begin
  if ((b == 1'bz) || (b == 1'bx)) begin
    a = 1;
  end else begin
    a = 0;
  end
end

endmodule
```

You could download file synthesis_compare_xz.v here

There seems to be a common problem with all the design engineers new to hardware. They normally tend to compare variables with X and Z. In practice it is the worst thing to do, so please avoid comparing with X and Z. Limit your design to two states, 0 and 1. Use tri-state only at chip IO pads level. We will see this as an example in the next few pages.

Constructs Supported in Synthesis

Verilog is such a simple language; you could easily write code which is easy to understand and easy to map to gates. Code which uses if, case statements is simple and cause little headaches with synthesis tools. But if you like fancy coding and like to have some trouble, ok don't be scared, you could use them after you get some experience with Verilog. Its great fun to use high level constructs, saves time.

The most common way to model any logic is to use either assign statements or always blocks. An assign statement can be used for modeling only combinational logic and always can be used for modeling both combinational and sequential logic.

Description	Notes	Construct Type Keyword or
ports	input, inout, output	Use inout only at IO level.
parameters	parameter	This makes design more generic
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances / primitive gate instances	E.g.- nand (out,a,b), bad idea to code RTL this way.
function and tasks	function , task	Timing constructs ignored
procedural	always, if, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
named Blocks	disable	Disabling of named block supported.
loops	for, while, forever	While and forever loops must contain @(posedge clk) or @(negedge clk)

Operators and their Effect.

One common problem that seems to occur is getting confused with logical and reduction operators. So watch out.

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 AND
	~&	Bitwise NAND
	|	Bitwise OR
	~|	Bitwise NOR

	^	Bitwise XOR
	^~ ~^	Bitwise XNOR
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

Logic Circuit Modeling

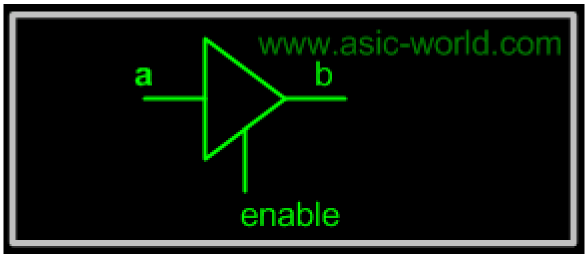
From what we have learnt in digital design, we know that there could be only two types of digital circuits. One is combinational circuits and the second is sequential circuits. There are very few rules that need to be followed to get good synthesis output and avoid surprises.

Combinational Circuit Modeling using assign

Combinational circuits modeling in Verilog can be done using assign and always blocks. Writing simple combinational circuits in Verilog using assign statements is very straightforward, like in the example below

```
assign y = (a&b) | (c^d);
```

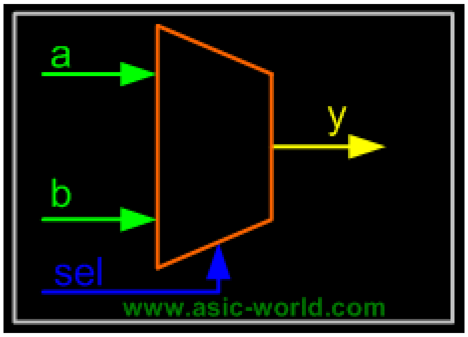
Tri-state buffer



```
module tri_buf (a,b,enable);
 input a;
 output b;
 input enable;
 wire a,enable;
 wire b;

assign b = (enable) ? a : 1'bz;

endmodule
```

You could download file tri_buf.v here

Mux

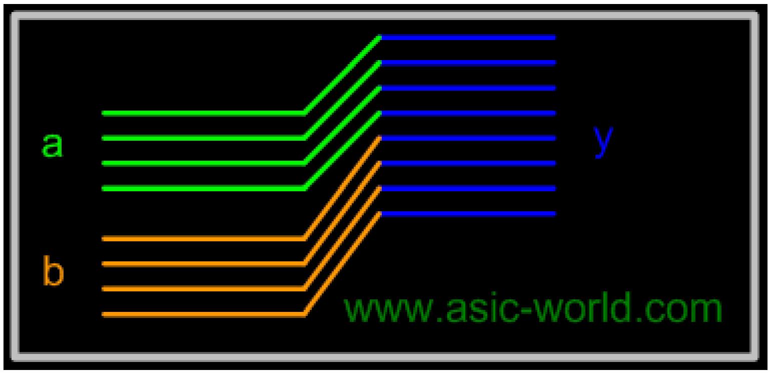


```
module mux_21 (a,b,sel,y);
```

```
         input a, b;
         output y;
         input sel;
         wire y;

         assign y = (sel) ? b : a;

endmodule
```

You could download file mux_21.v here

Simple Concatenation



```
module bus_con (a,b);
         input [3:0] a, b;
         output [7:0] y;
         wire [7:0] y;

         assign y = {a,b};

endmodule
```

You could download file bus_con.v here

1 bit adder with carry

```
module addbit (
a        , // first input
b        , // Second input
ci       , // Carry input
sum      , // sum output
co         // carry output
);
//Input declaration
input a;
input b;
input ci;
//Ouput declaration
output sum;
output co;
//Port Data types
wire  a;
wire  b;
wire  ci;
wire  sum;
wire  co;
//Code starts here
assign {co,sum} = a + b + ci;

endmodule // End of Module addbit
```

You could download file addbit.v here

Multiply by 2

```
module muliply (a,product);
            input [3:0] a;
            output [4:0] product;
            wire [4:0] product;

            assign product  = a << 1;

endmodule
```

You could download file multiply.v here

3 is to 8 decoder

```
module decoder (in,out);
input [2:0] in;
output [7:0] out;
wire [7:0] out;
assign out  =        (in == 3'b000 ) ? 8'b0000_0001 :
 (in == 3'b001 ) ? 8'b0000_0010 :
 (in == 3'b010 ) ? 8'b0000_0100 :
 (in == 3'b011 ) ? 8'b0000_1000 :
 (in == 3'b100 ) ? 8'b0001_0000 :
 (in == 3'b101 ) ? 8'b0010_0000 :
 (in == 3'b110 ) ? 8'b0100_0000 :
 (in == 3'b111 ) ? 8'b1000_0000 : 8'h00;

endmodule
```

You could download file decoder.v here

Combinational Circuit Modeling using always

While modeling using always statements, there is the chance of getting a latch after synthesis if care is not taken. (No one seems to like latches in design, though they are faster, and take lesser transistor. This is due to the fact that timing analysis tools always have problems with latches; glitch at enable pin of latch is another problem).

One simple way to eliminate the latch with always statement is to always drive 0 to the LHS variable in the beginning of always code as shown in the code below.

3 is to 8 decoder using always

```
module decoder_always (in,out);
input [2:0] in;
output [7:0] out;
reg [7:0] out;

always @ (in)
begin
  out = 0;
  case (in)
    3'b001 : out = 8'b0000_0001;
    3'b010 : out = 8'b0000_0010;
    3'b011 : out = 8'b0000_0100;
    3'b100 : out = 8'b0000_1000;
    3'b101 : out = 8'b0001_0000;
    3'b110 : out = 8'b0100_0000;
    3'b111 : out = 8'b1000_0000;
  endcase
end

endmodule
```

You could download file decoder_always.v here

Sequential Circuit Modeling

Sequential logic circuits are modeled using edge sensitive elements in the sensitive list of always blocks. Sequential logic can be modeled only using always blocks. Normally we use nonblocking assignments for sequential circuits.

Simple Flip-Flop

```
module flif_flop (clk,reset, q, d);
input clk, reset, d;
output q;
reg q;

always @ (posedge clk )
begin
  if (reset == 1) begin
```

```
      q <= 0;
    end else begin
      q <= d;
    end
  end

  endmodule
```

You could download file flip_flop.v here

Verilog Coding Style

If you look at the code above, you will see that I have imposed a coding style that looks cool. Every company has got its own coding guidelines and tools like linters to check for this coding guidelines. Below is a small list of guidelines.

- Use meaningful names for signals and variables
- Don't mix level and edge sensitive elements in the same always block
- Avoid mixing positive and negative edge-triggered flip-flops
- Use parentheses to optimize logic structure
- Use continuous assign statements for simple combo logic
- Use nonblocking for sequential and blocking for combo logic
- Don't mix blocking and nonblocking assignments in the same always block (even if Design compiler supports them!!).
- Be careful with multiple assignments to the same variable
- Define if-else or case statements explicitly

Note : Suggest if you want more details.