Verilog Tutorial

25-Oct-2003

Deepak Kumar Tala

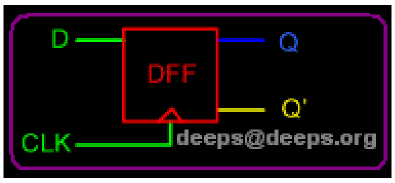
Comments :deeps@deeps.org
Website : http://www.deeps.org

Index

INTRODUCTION

Introduction.

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



```
module d_ff ( d, clk, q, q_bar);
  input d ,clk;
  ouput q, q_bar;

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

One can describe a simple Flip flop as that in above figure as well as one can describe a complicated designs having 1 million gates. Verilog is one of the HDL languages available in the industry for designing the Hardware. Verilog 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, deterring the details of implementation to a later stage of design in the final design.

Many engineers who want to learn Verilog, most often ask this question, how much time it will take to learn Verilog?, Well my answer to them is " **It may not take more then one week, if you happen to know at least one programming language".**

Design Styles

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

Bottom-Up Design

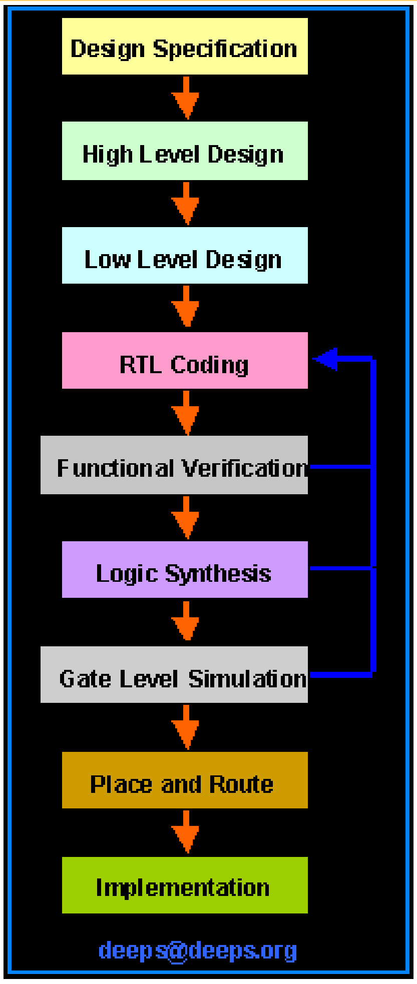
The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates (Refer to the Digital

Section for more details) With 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 design practices it would be impossible to handle the new complexity.

Top-Down Design

The desired design-style of all designers is the top-down design. 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 mix of both the methods, implementing some key elements of both design styles.

Figure shows a Top-Down design approach.



Abstraction Levels of Verilog

Verilog supports a design 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 possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code 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.*

History of Verilog

Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo as well as from traditional computer language such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990.

Verilog simulator was first used beginning in 1985 and was extended substantially through 1987.The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.

The time was late 1990. Cadence Design System, whose primary product at that time included Thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway product, Cadence now became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys was marketing the top-down design methodology, using Verilog. This was a powerful combination.

In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.

OVI did a considerable amount of work to improve the Language Reference Manual (LRM), clarifying things and making the language specification as vendor-independent as possible.In 1990.

Soon it was realized, that if there were too many companies in the market for Verilog, potentially everybody would like to do what Gateway did so far - changing the language for their own benefit. This would defeat the main purpose of releasing the language to public domain. As a result in 1994, the IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard. This effort was concluded with a successful ballot in 1995, and Verilog became an IEEE standard in December, 1995.

When Cadence gave OVI the LRM, several companies began working on Verilog simulators. In 1992, the first of these were announced, and by 1993 there were several Verilog simulators available from companies other than Cadence. The most successful of these was VCS, the Verilog Compiled Simulator, from Chronologic Simulation. This was a true compiler as opposed to an interpreter, which is what Verilog-XL was. As a result, compile time was substantial, but simulation execution speed was much faster.

In the meantime, the popularity of Verilog and PLI was rising exponentially. Verilog as a HDL found more admirers than well-formed and federally funded VHDL. It was only a matter of time before people in OVI realized the need of a more universally accepted standard. Accordingly, the board of directors of OVI requested IEEE to form a working committee for establishing Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993 and on October 14, 1993, it had its first meeting.

The standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed in May 1995 and now known as IEEE Std. 1364-1995.

After many years, new features have been added to Verilog, and new version is called Verilog 2001. This version seems to have fixed lot of problems that Verilog 1995 had. This version is called 1364-2000. Only waiting now is that all the tool vendors implementing it.

DESIGN AND TOOL FLOW

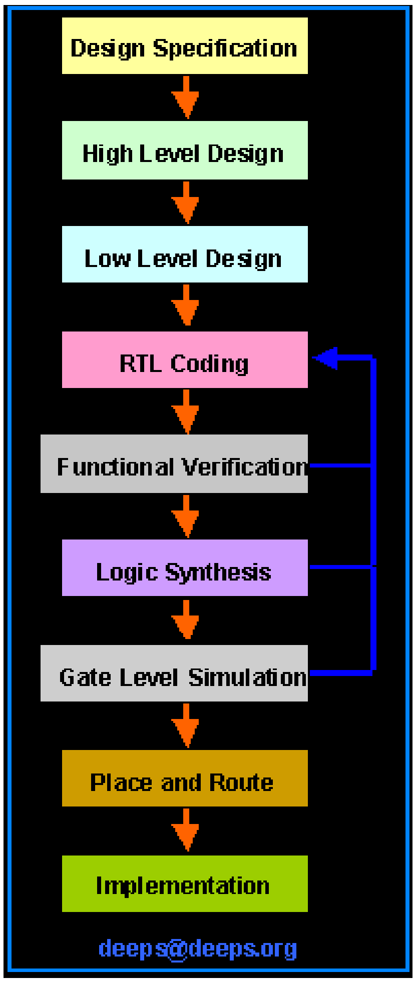
Introduction

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 front end design part of the tool flow and bit of FPGA design flow that can be done without any fat money spent on tools. If you have any suggestions or questions please don't hesitate to mail me. (Note : I have missed steps in P&R, Will add then shortly)

Various stages of ASIC/FPGA

- **Specification :** Word processor like Word, Kwriter, AbiWord
- **High Level Design :** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word.
- **Micro Design/Low level design:** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word. For FSM StateCAD or some similar tool.
- **RTL Coding :** Vim, Emacs, conTEXT, HDL TurboWriter
- **Simulation :** Modelsim, VCS, Verilog-XL, Veriwell, Finsim, iVerilog, VeriDOS.
- **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.

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. Simple example would be, like I want to design a counter, it should be 4 bit wide, should have synchronous reset, with active high enable and reset signal, When reset is active, counter output should go to "0". You can use Microsoft Word, or GNU Abiword or Openoffice for entering the specification.

High Level Design

This is the stage at which you define various blocks in the design and how they communicate. Lets assume that we need to design microprocessor, High level design means splitting the design into blocks based on their function, In our case various blocks are registers, ALU, Instruction Decode, Memory Interface, etc. You can use Microsoft Word, or KWriter or Abiword or Openoffice for entering high level design.

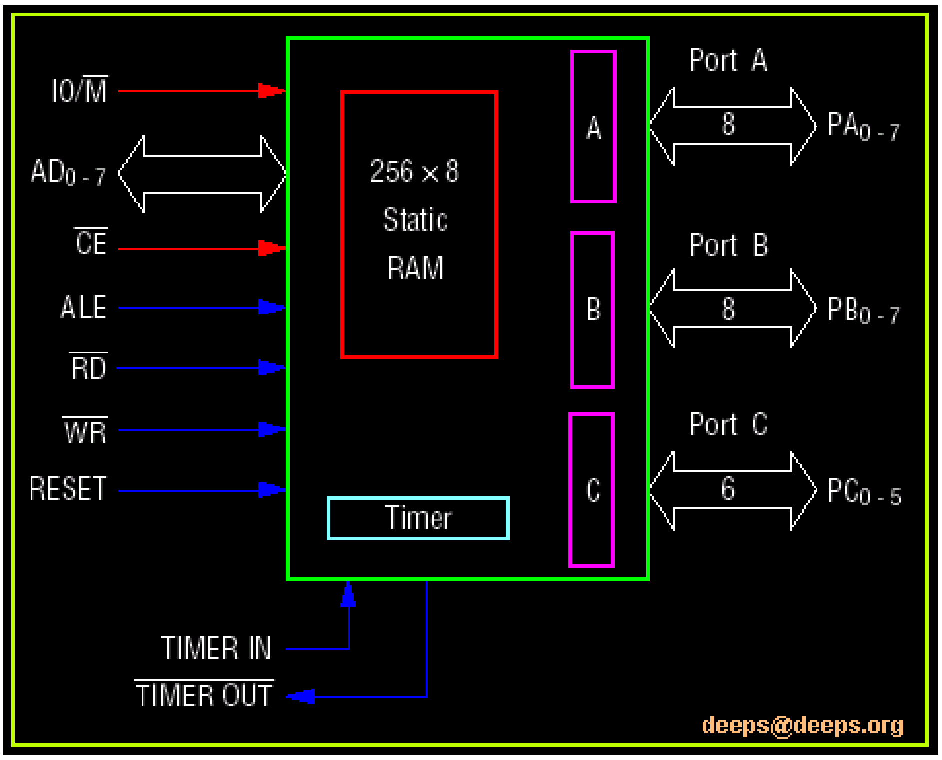
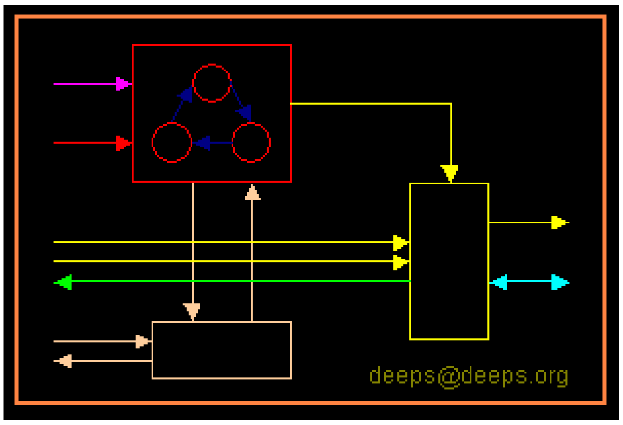


Figure : I8155 High Level Block Diagram

Micro Design/Low level design

Low level design or Micro design is the phase in which, designer describes how each block is implemented. It contains details of State machines, counters, Mux, decoders, internal registers. For state machine entry you can use either Word, or special tools like StateCAD. It is always a good idea if waveform is drawn at various interfaces.

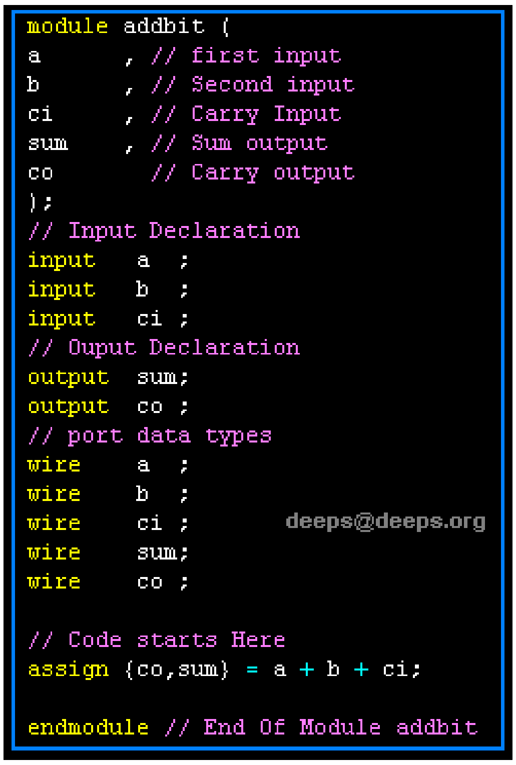
Figure : Sample Low level design



RTL Coding

In RTL coding, Micro Design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally we use vim editor, but I prefer conTEXT and Nedit editor, it all depends on which editor you like. Some use Emacs.

Figure : Sample RTL code



```
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
```

Simulation

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

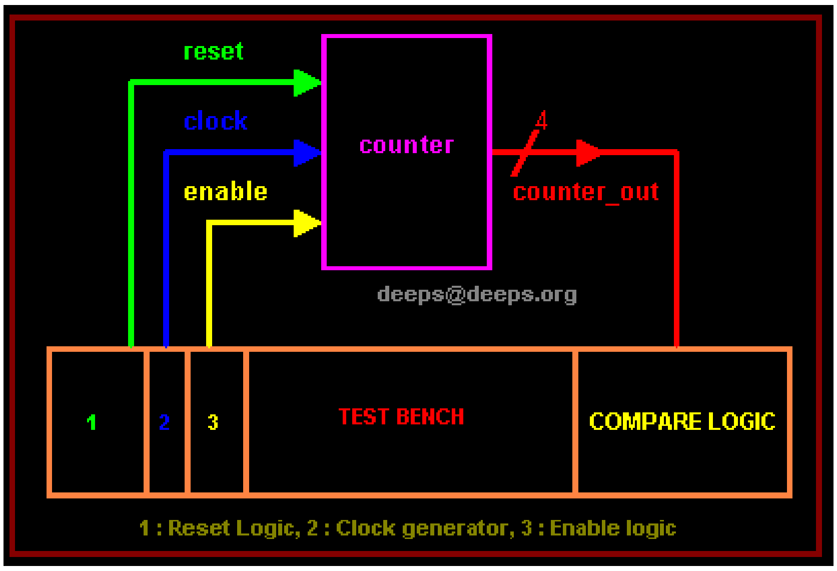


Figure : Sample Testbench Env

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

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 simulation***.

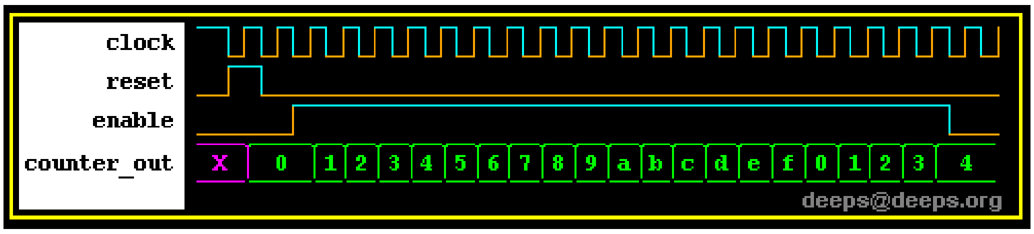


Figure : 4 bit Up Counter Waveform

Synthesis

Synthesis is process in which synthesis tool like design compiler or Synplify takes the 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 meeting the timing requirements. (Important thing to note is, synthesis tools are not aware of wire delays)

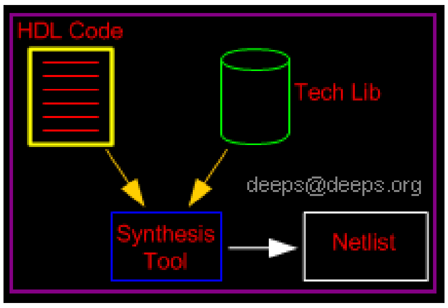


Figure : Synthesis Flow

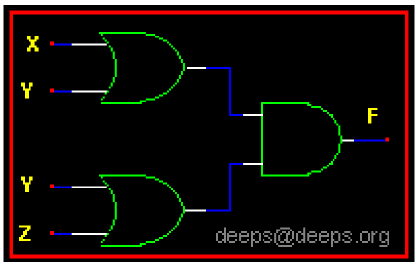


Figure : Synthesis output

Place & Route

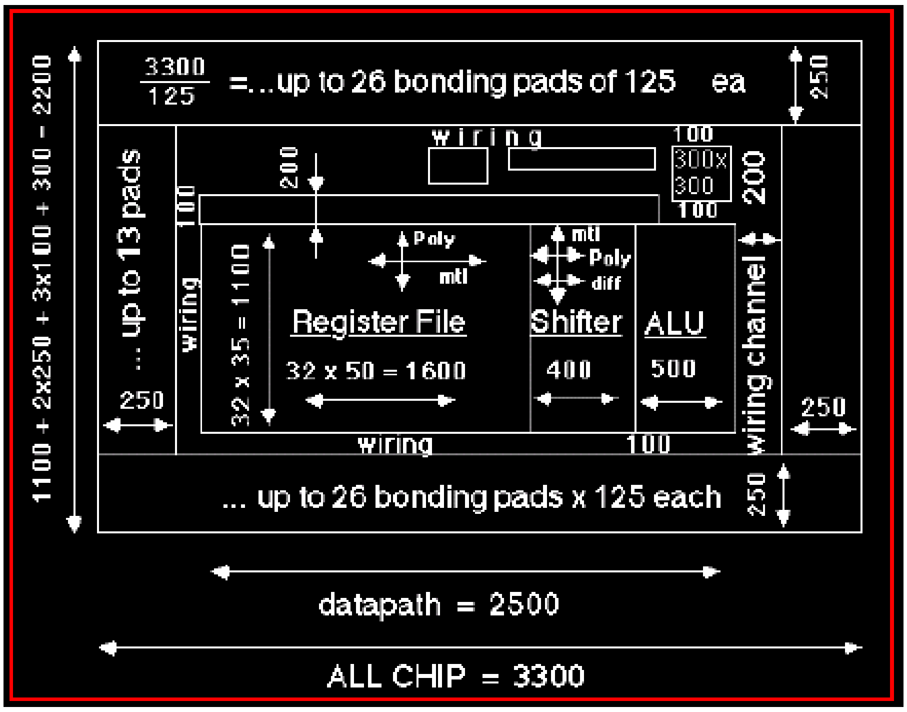


Figure : Sample micro-processor placement

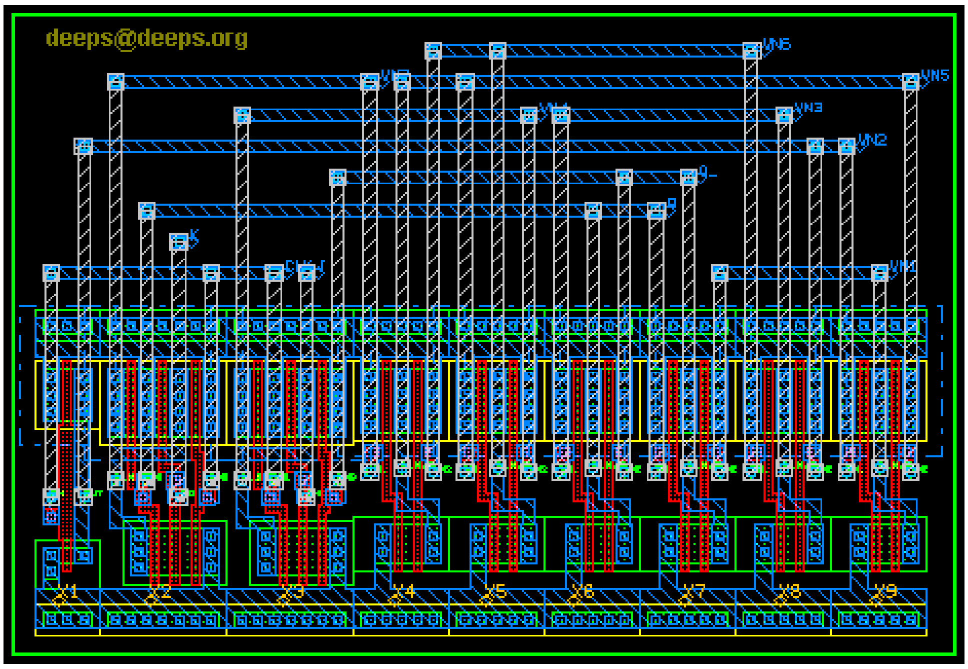


Figure : J-K Flip-Flop

Note : Under construction, please feel free to send your comments

My first program in Verilog

Introduction

If you refer to any book on programming language it starts with "hello World" program, once you have written the program, 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** in Verilog, followed by **"counter" design** in Verilog.

Hello World Program

```
01 -----------------------------------------------------
02 // This is my first Verilog Program
03 // Design Name : hello_world
04 // File Name      : hello_world.v
05 // Function        : This program will print "hello
   world
06 // Coder            : Deepak"
07 //-----------------------------------------------------
08 module hello_world ;
09 
10 initial begin
11   $display ("Hello World by Deepak");
12   #10 $finish;
13 end
14 
15 endmodule // End of Module hello_world
```

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 7 contains module hello_world;

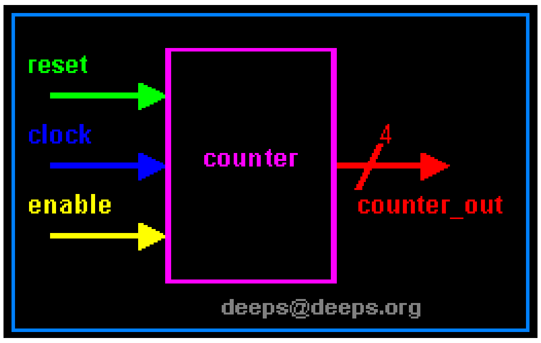
Line 9 contains the initial block, this block gets executed only once after the simulation starts and at time=0 (0ns). This block contains two statements, which are enclosed with in begin at line 7 and end at line 12. In Verilog if you have multiple lines within a block, you need to use begin and end.

Hello World Program Output

```
C:\www.deeps.org>veridos hello_world.v
VeriWell for Win32 HDL     <Version 2.1.4>    Sun Nov 04 17:52:38 2001
```

```
 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 : hello_world.v
The size of this model is [0%, 1%] of the capacity of the free version

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

Hello World by Deepak
Exiting VeriWell for Win32 at time 10
0 Errors, 0 Warnings, Memory Used: 0
Compile time = 0.0, Load time = 0.0, Simulation time = 0.0

Normal exit
```

Counter Design Block



Counter Design Specs

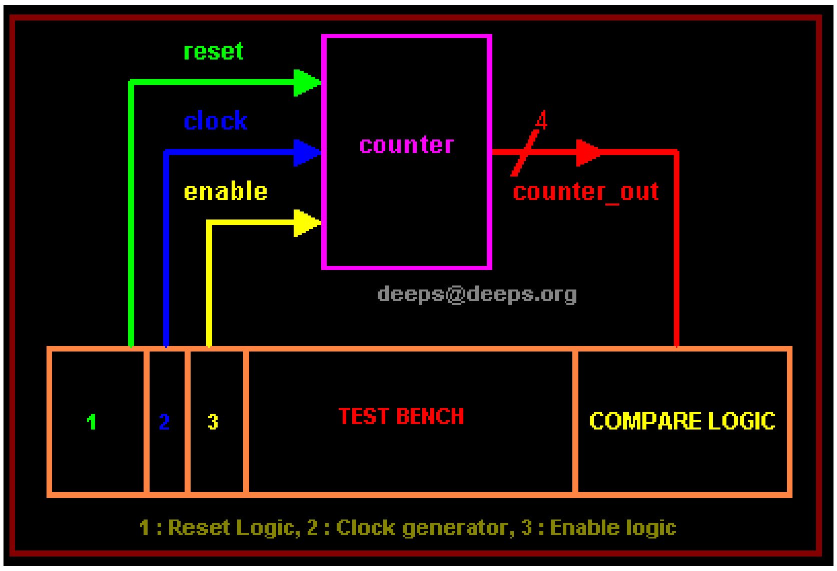
? 4-bit synchronous up counter.
? active high, synchronous reset.
? Active high enable.

Counter Design

```
//-----------------------------------------------------
// This is my second Verilog Design
// Design Name : counter
// File Name      : counter.v
// Function         : This is a 4 bit up-counter with
//                Synchronous active high reset and
//                with active high enable signal
//-----------------------------------------------------

module counter (
clock              ,  // Clock input ot the design
reset              ,  // active high, synchronous Reset
input
enable             ,  // Active high enabel 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 "0000"
 if (reset == 1'b1) begin
   counter_out <= #1 4'b000;
 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
```

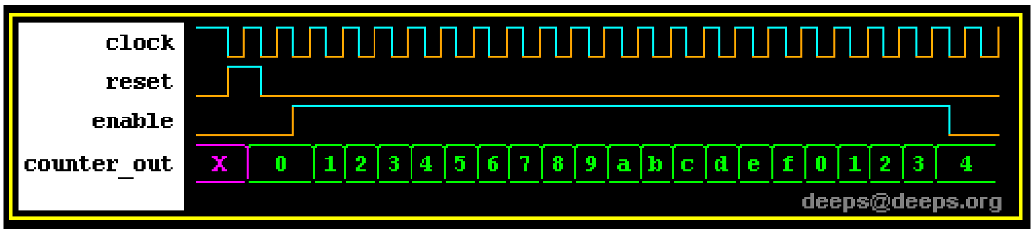
Counter Test Bench



Counter testbench consists of clock generator, reset control, enable control and compare logic. Below is the simple code of testbench without the compare logic.

```
module counter_tb();
// Declare inputs as regs and outputs as wires
reg clock, reset, enable
wire [3:0] counter_out;
// Initialize all variables
initial begin
  clock = 1;          // initial value of clock
  reset = 0;          // initial value of reset
  enable = 0;         // initial value of enable
  #5 reset = 1;       // Assert the reset
  #5 reset = 0;        // De-assert the reset
  #5 enable = 1;       // Assert enable
  #100 enable = 0;     // De-assert enable
  #10 $finish;         // Terminate simulation
end
// Clock generator
always begin
  #5 clock = ~clock;   // Toggle clock every 5 ticks
end
// Connect DUT to test bench
counter U_counter (
  clock,
  reset,
  enable,
  counter_out
);

endmodule
```

Counter Waveform



Note : Simulator used for this exercise can be got from here, If this is illegal, please let me know, I will remove it from my web page.

Verilog HDL Syntax and Semantics

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 formfeeds. 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

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

Never write code like this.

```
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;
```

Nice way to write code.

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

```
/* 1-bit adder example for showing
few verilog */  Multi line comment
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;
```

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)
```

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

Identifiers

Identifiers are names used to give an object, such as a register 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:

\486_up	**\Q~**	**\1,2,3**	**\reset***

```
module \486 (q,\q~,d,clk,\reset*);
```

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 pf binary, octal, decimal, or hexadecimal
- Radix is case and hex digits (a,b,c,d,e,f) are insensitive
- Spaces are allowed between the size, radix and value

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

Example of Integer Numbers

Integer	Stored as	Description
1	00000000000000000000000000000001	unsized 32 bits
8'hAA	10101010	sized hex
6'b10_0011	100011	sized binary
'hF	00000000000000000000000000001111	unsized hex 32 bits

Verilog expands **<value>** to be fill the specified **<size>** by working from *right-to-left*

- When **<size>** is *smaller* than **<value>**, then left-most bits of **<value>** are truncated

- When **<size>** is *larger* than **<value>**, then left-most bits are filled, based on the value of the left-most bit in **<value>.**
  - Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'

Example of Integer Numbers

Integer	Stored as	Description
6'hCA	001010	truncated, not 11001010
6'hA	001010	filled with two '0' on left
16'bZ	Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z	filled with 16 Z's
8'bx	x x x x x x x x	filled with 8 X's

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.

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 the type 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 become signed numbers. Verilog internally represents negative numbers in 2's compliment format. An optional signed specifier can be added for signed arithmetic.

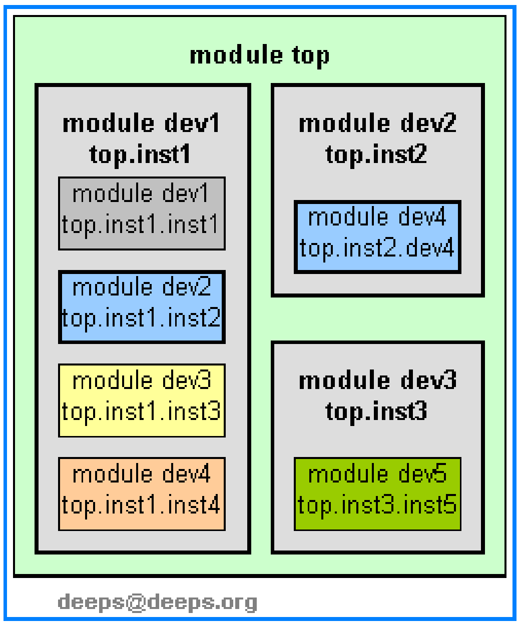
Examples

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

Below example file show how Verilog treats signed and unsigned numbers.

Modules

- Module are the building blocks of Verilog designs
- You create design hierarchy by *instantiating* modules in other modules.
- An *instant* of a module is a use of that module in another, higher-level module.



Hierarchical Identifiers

- Hierarchical path names are based on the **top module identifier** followed by **module instant identifiers**, separated by **periods**.

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 :

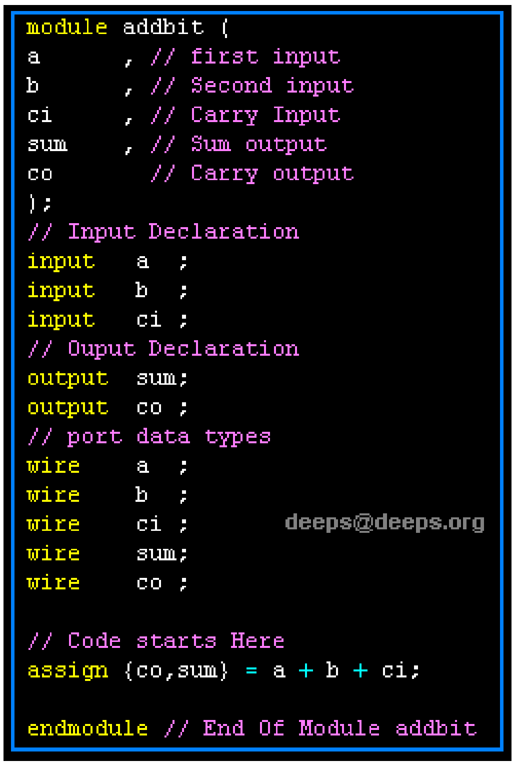
```
input   [range_val:range_var] list_of_identifiers;
output  [range_val:range_var] list_of_identifiers;
inout   [range_val:range_var] list_of_identifiers;
```

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
```

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
```

Modules connected by port order (implicit):

Here order should match correctly. Normally it not a good idea to connect ports implicit. Could cause problem in debug, when any new port is added or deleted.

```
module adder (
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]          ,
```

Modules connect by name (explicit) : Here name should match correctly.

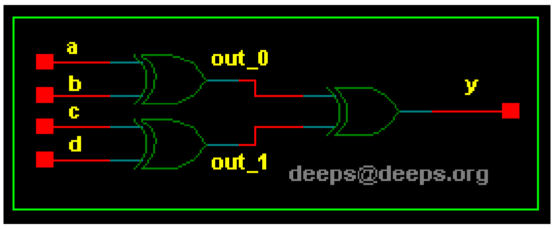
```
module adder (
        result      , // Output of the adder
        carry       , // Carry output of adder
        r1            , // first input
        r2            , // second input
        ci               // carry input
        );
        // Input Port Declarations
        input    [3:0]   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])          ,
```

Instantiating a module

```
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
```

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

Schematic



Data Types

Verilog Language has two primary data types

- **Nets -** represents 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 but used to connect structural building blocks in your code.
- **Implicit declaration** is always a **net** of type wire and is one bit wide.

Types of Nets

Each net type has 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		

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

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 arrays of the regs 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 the register types, reg is the one which is most widely used

Question : What is the difference between wire and reg data type Answer

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 with the escape sequence that represents the character in the left-hand column.

Special Characters in Strings

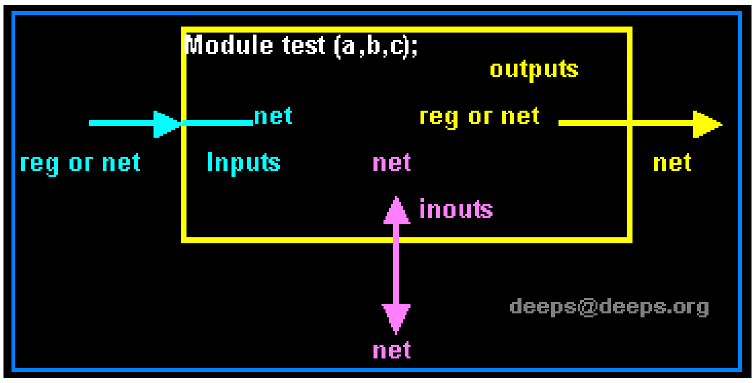
\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Example

```
reg [8*17:0] version   ; // Declare a register variable that is 18 bytes

initial
  version = "model version 1.0";
```

Port Connection Rules

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

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

Example : Explicit

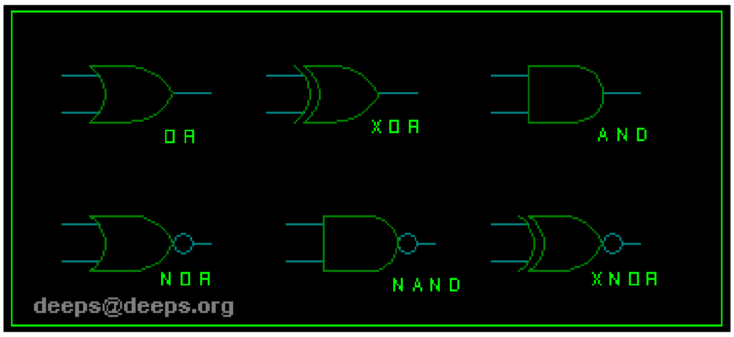
```
dff u0 (
   .q              (q_out),
   .q_bar          (),
   .clk            (clk_in),
   .d              (d_in),
   .rst            (rst_in),
   .pre            (pre_in)
); // Here second port is not connected
```

Gate Level Modeling

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. This are rarely used for in design work, but are used in post synthesis world for modeling the ASIC/FPGA cells, this cells are then used for gate level simulation or what is called as SDF simulation.

Gate Primitives



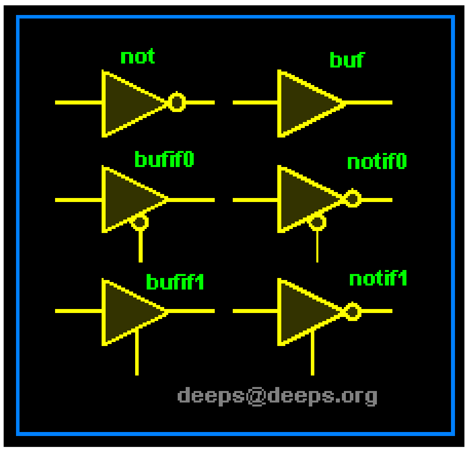
The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

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

```
and U1(out,in);
and U2(out,in1,in2,in3,in4);
xor U3(out,in1,in2,in3);
```

Transmission Gate Primitives

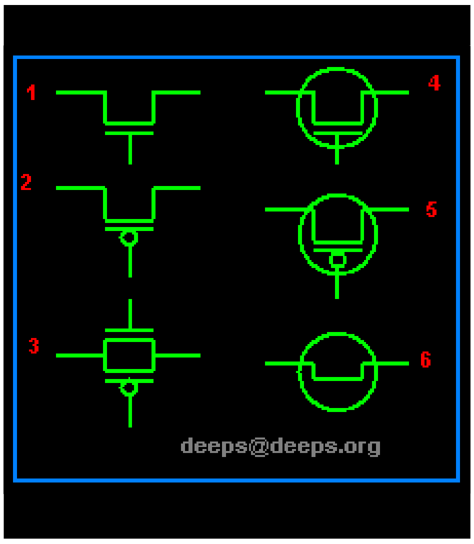


not	N-output invertor.
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

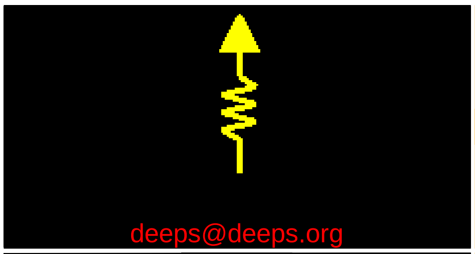
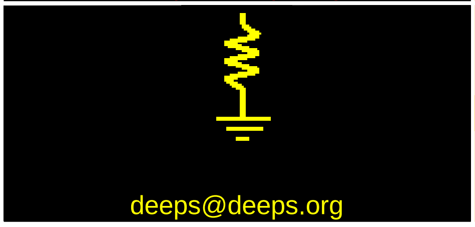
Examples

```
bufif0 U1(data_bus,data_drive, data_enable_low);
buf  U2(out,in);
not U3(out,in);
```

Switch Primitives



1	pmos	Uni-directional PMOS switch
	rpmos	Resistive PMOS switch
2	nmos	Uni-directional NMOS switch
	rnmos	Resistive NMOS switch
3	cmos	Uni-directional CMOS switch
	rcmos	Resistive CMOS switch
4	tranif1	Bi-directional transistor (High)
	tranif1	Resistive transistor (High)
5	tranif0	Bi-directional transistor (Low)
	rtranif1	Resistive Transistor (Low)
6	tran	Bi-directional pass transistor
	rtran	Resistive pass transistor

		pullup	Pull up resistor.
		pulldown	Pull down resistor.

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

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

tranif0 my_gate1 (net5, net8, cnt);
rtranif1 my_gate2 (net5, net12, cnt);

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

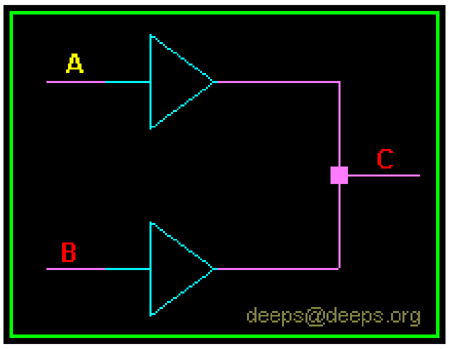
0	zero, low, false
1	one, high, true
z or **Z**	high impendence, floating
x or **X**	unknown, uninitialized, contention

Verilog Strength Levels

Strength Level	Strength	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

Examples

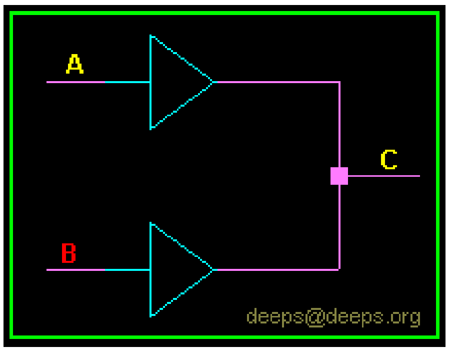


Two buffers that has output

A : Pull 1

B : Supply 0

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



Two buffers that has output

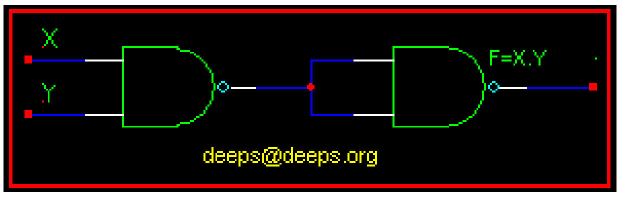
A : Supply 1

B : Large 1

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

Designing Using Primitives

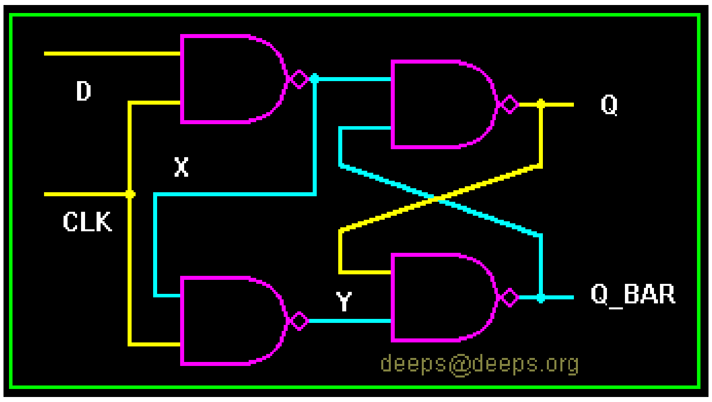
AND Gate from NAND Gate



Verilog Code

```
// Structural model of AND gate from two NANDS
module and_from_nand(X, Y, F);

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

endmodule
```

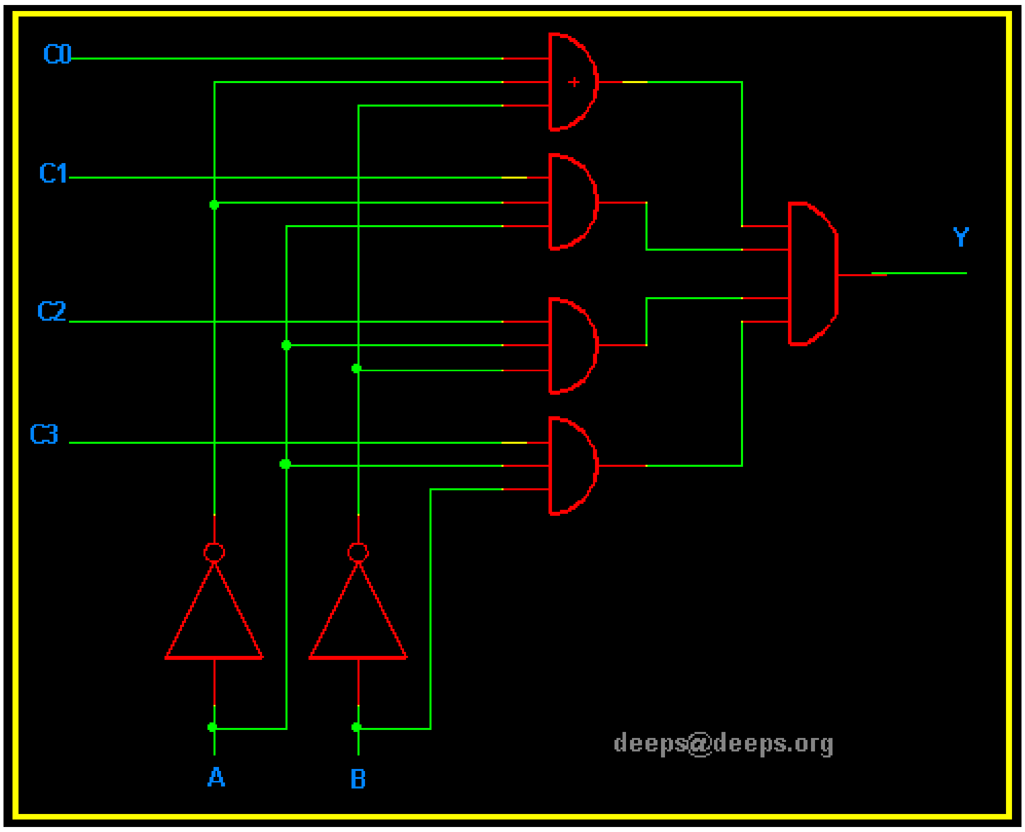
D-Flip flop from NAND Gate



Verilog Code

```
module dff(Q,Q_BAR,D,CLK);
output Q,Q_BAR;
input D,CLK;

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

endmodule
```

Multiplexer from primitives



Verilog Code

```
//Module 4-2 Mux
module mux (c0,c1,c2,c3,A,B,Y);
   input c0,c1,c2,c3,A,B;
   ouput 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);

endmodule
```

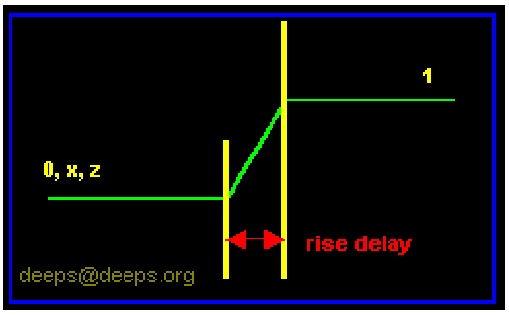
Gate and Switch delays

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

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

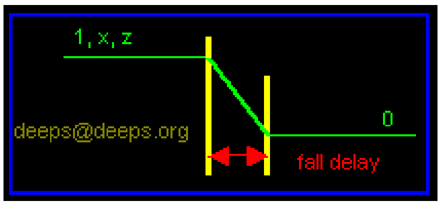
Rise Delay

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 fall 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

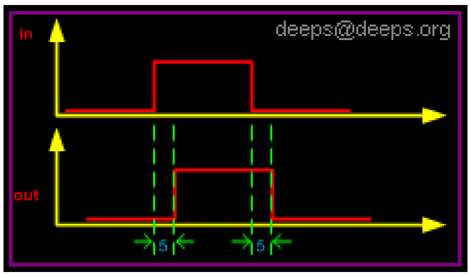
The max value is the maximum delay value that the gate is expected to have.

Examples

```
// Delay for all transitions
or #5  u_or (a,b,c);

// Rise and fall delay
and #(1,2) u_and (a,b,c);

// Rise, fall and turn off delay
nor # (1,2,3) u_nor (a,b,c);

//One Delay, min, typ and max
nand #(1:2:3) u_nand (a,b,c);

//Two delays, min,typ and max
buf #(1:4:8,4:5:6) u_buf (a,b);

//Three delays, min, typ, and max
notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (a,b,c);
```

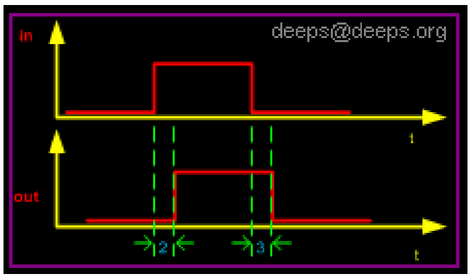
Gate Delay Code Example

```
module not_gate (in,out);
  input in;
  output out;

  not #(5) (out,in);

endmodule
```



Gate Delay Code Example

```
module not_gate (in,out);
  input in;
  output out;

  not #(2,3) (out,in);

endmodule
```



Normally we can have three models of delays, typical, minimum and maximum delay. During compilation of a modules one needs to specify the delay models to use, else Simulator will use the typical model.

```
Verilog +minterms myfile.v
```

I have assumed Verilog-XL as simulator

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

```
// Two input AND gate
and u_and (out, in1, in2);
```

```
// four input AND gate
and u_and (out, in1, in2, in3, in4);
```

```
// three input XNOR gate
xnor u_xnor (out, in_1, in_2, in_3);
```

N-Output Primitives

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

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

Examples

```
// one output Buffer gate
buf u_buf (out,in);

// four output Buffer gate
buf u_buf (out_0, out_1, out_2, out_3, in);

// three output Invertor gate
not u_not (out_a, out_b, out_c, in);
```

Verilog Operators

Arithmetic Operators

- Binary: +, -, *, /, % (the modulus operator)
- Unary: +, -
- 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

Relational Operators

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:
- 0 if the relation is false
- 1 if the relation is true
- x if any of the operands has unknown x bits
- Note: If a value is x or z, then the result of that test is false

Equality Operators

a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, resulting may be unknown
a != b	a not equal to b, result may be unknown

- 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
  - the result is always 0 or 1

Logical Operators

!	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 unknown x bits

Bit-wise Operators

~	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

Reduction Operators

&	**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.

Shift Operators

<<	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.

Concatenation Operator

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within
    - Examples
        - {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
- Repetition multipliers that must be constants can be used:
    - {3{a}} // this is equivalent to {a, a, a}
- Nested concatenations are possible:
    - {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

Conditional Operator

- 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 whether cond_expr evaluates to true or false

Example

out = (enable) ? data : 8'bz; // Tri state buffer

Operator Precedence

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

Verilog Behavioral Modeling

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 procedures blocks, but there is a exception, some behavioral code also exist outside procedures 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 name means, it executes always.

Example : initial and always

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

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

Procedural Assignment Statements

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

Example : Bad and Good procedural assignment

```
wire clk, reset;
```

```
reg clk, reset;
reg enable, data;

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

```
reg enable, data;

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

Procedural Assignment Groups

If a procedure block contains more then 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" and "fork - join"

```
initial
   begin
      #1 clk = 0;
      #5 reset = 0;
      #5 enable = 0;
      #2 data = 0;
   end
```

```
initial
   fork
     #1 clk = 0;
     #5 reset = 0;
     #5 enable = 0;
     #2 data = 0;
   join
```

Begin : clk gets 0 after 1 time unit, reset gets 0 after 6 time units, enable after 11 time units, data after 13 units. All the statements are executed in sequentially.

Fork : clk gets value after 1 time unit, reset after 5 time units, enable after 5 time units, data after 2 time units. All the statements are executed in parallel.

Sequential Statement Groups

The **begin** - **end** keywords:

- Group several statements together.
- Cause the statements to be evaluated in 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.

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 high delay, it can be the first statement in the block).

The Conditional Statement if-else

The **if - else** statement controls the execution of other statements, In programming language like c, if - else controls the flow of program.

```
if (condition)
  statements;

if (condition)
  statements;
else
  statements;

if (condition)
  statements;
else if (condition)
  statements;
................
................
else
  statements;
```

Example

```
// Simple if statement
if (enable)
  q <= d;
// One else statement
if (reset == 1'b1)
  q <= 0;;
else
  q <= d;
// Nested if-else-if statements
if (reset == 1'b0)
  counter <= 4'b0000;
else if (enable == 1'b1 && up_en == 1'b1)
  counter <= counter + 1'b1;
else if (enable == 1'b1 && down_en == 1'b1);
  counter <= counter - 1'b0;
else
  counter <= counter; // Redundant code
```

Note : More to be added on if-else, as this is the one which is most widely used.

The Case Statement

The **case** statement compares a 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.

```
case (<expression>)
   <case1> : <statement>
   <case2> : <statement>
   .....
   default : <statement>
endcase
```

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
```

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

Example with z and x

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

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** uses the **z** as the don't care instead of as a logic value
- **casex** uses either the **x** or the **z** as don't care instead of as logic values

Example casez

```
casez(opcode)
   4'b1zzz : out = a; // don't care about lower 3 bits
   4'b01??: out = b; //the ? is same as z in a number
   4'b001?: out = c;
   default : out = $display ("Error xxxx does matches 0000");
endcase
```

Looping Statements

Looping statements appear inside a 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

syntax : **forever** <statement>

Example : Free running clock generator

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

The repeat statement

The **repeat** loop executes statement fixed <number> of times

syntax : **repeat** (<number>) <statement>

Example:

```
 if (opcode == 10) //perform rotate
    repeat (8) begin
       temp = data[7];
       data = {data<<1,temp};
    end
```

The while loop statement

The **while** loop executes as long as an <expression> evaluates as true

syntax : **while** (<expression>) <statement>

Example :

```
loc = 0;
if (data = 0) // example  of a 1 detect shift value
   loc = 32;
else while (data[0] == 0); //find the first set bit
begin
   loc = loc + 1;
   data = data << 1;
end
```

The for loop statement

The for loop is 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>

Example :

```
for (i=0;i<=63;i=i+1)
   ram[i] <= 0; // Inialize the RAM with 0
```

Continuous Assignment Statements

Continuous assignment statements drives 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 and procedural assignments.
- The left-hand side of a continuous assignment must be net data type.

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

Example: 1-bit Adder

```
module adder (a,b,sum,carry);
    input a, b;
    output sum, carry;
    assign #5 {carry,sum} = a+b;
endmodule
```

Example: Tri-State Buffer

```
module tri_buf(a,b,enable);
    input a, enable;
    output b;
    assign b = (enable) ? a : 1'bz;
endmodule
```

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(a,b,enable);
    input a, enable;
    output b;
    assign #(1:2:3) b = (enable) ? a : 1'bz;
endmodule
```

Procedural Block Control

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

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

An event sensitive delay at the begining of a procedure, any change in either d or enable satisfies the even 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.

Example : 1-bit Adder

```
module adder (a,b,sum,carry);
   input a, b;
   output sum, carry;
   reg sum, carry;
   always @ (a or b)
   begin
     {carry} = a + b;
   end

endmodule
```

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

Example : 4-bit Adder

```
module adder (a,b,sum,carry);
   input [3:0] a, b;
   output [3:0] sum;
   output carry;
   reg [3:0] sum;
   reg carry;
   always @ (a or b)
   begin
     {carry} = a + b;
   end

endmodule
```

A procedure can't trigger itself

Once cannot trigger the block with the variable that block assigns value or drive's.

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

Procedural Block Concurrency

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

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

  always @ ( c)
    a = c;

  always @ (d or a)
    b = a &d;

endmodule
```

Race condition

```
initial
  b = 0;

initial
  b = 1;
```

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

Named Blocks

Blocks can be named by adding : block_name after the keyword begin. This block can be disabled using disable statement.

Example

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

  always @ ( c)
    a = c;

  always @ (d or a)
  begin : my_block
    b = a &d;
  end

endmodule
```

In above example, my_block is the named block. **(Need to add more practical example)**

Procedural Timing Control

Procedural blocks and timing controls.

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

Delay Controls

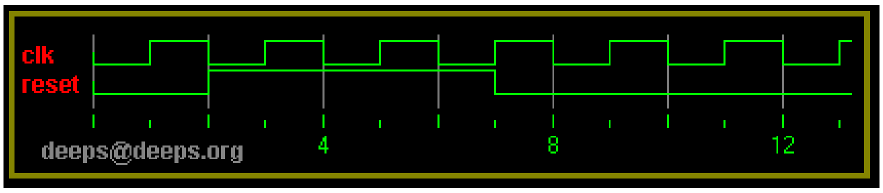
Delays the execution of a procedural statement by specific simulation time.

#<time> <statement>;

Example :

```
module clk_gen (clk,reset);
   output clk,reset;
   reg clk, reset;
   initial begin
       clk = 0;
       reset = 0;
       #2 reset = 1;
       #5 reset = 0;
   end
   always
   #1 clk = !clk;
endmodule
```

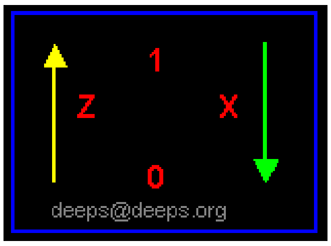
Waveform



Edge sensitive Event Controls

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

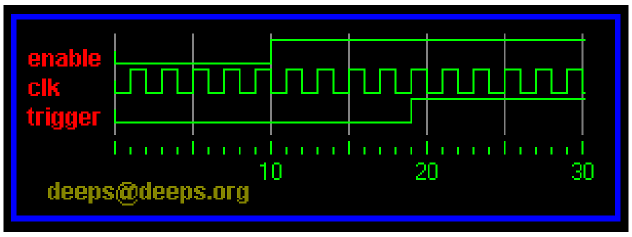
@ (<posedge>|<negedge> signal) <statement>;



Example :

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

Waveform



Level-Sensitive Even Controls (Wait statements)

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

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

Example :

```
while (mem_read == 1'b1) begin
  wait (data_ready) data = data_bus;
  read_ack = 1;
end
```

Intra-Assignment Timing Controls

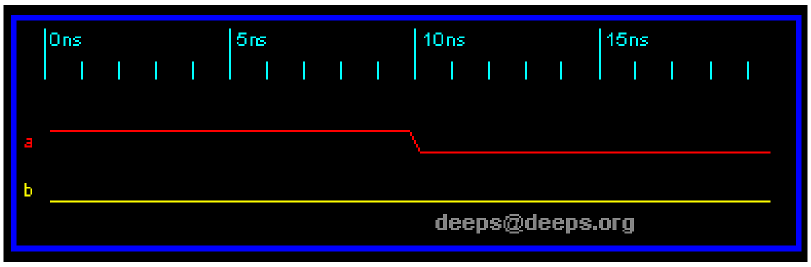
Intra-assignment controls evaluate the right side expression right always and assigns the result after the delay or event control.

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

Example :

```
initial begin
  a = 1;
  b = 0;
  a = #10 0;
  b = a;
end
```

Waveform



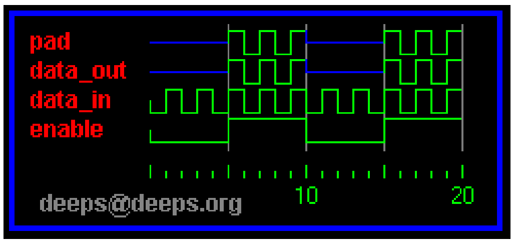
Modeling Combinational Logic with Continuous Assignments

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

Example : Tri-state buffer

```
module tri_buf (data_in,data_out, pad,enable);
  input data_in, enable;
  output data_out;
  inout pad;
  wire pad, data_out;
  assign pad = (enable) ? data_in : 1'bz;
  assign data_out = pad;
endmodule
```

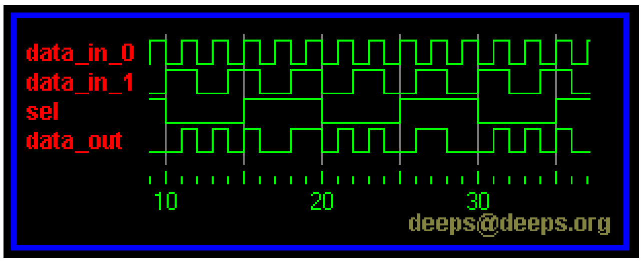
Waveform



Example : 2:1 Mux

```
module mux2x1 (data_in_0,data_in_1, sel, data_out);
   input data_in_0, data_in_1;
   output data_out;
   input  sel;
   wire data_out;
   assign data_out = (sel) ? data_in_1 : data_in_0;
endmodule
```

Waveform



Task and Function

Task

Tasks are used in all programming languages, generally known as Procedures or sub routines. Many lines of code are enclosed in task....end task brackets. Data is passed to the task, the processing done, and the result returned to a specified value. They have to be specifically called, with data in 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.

- task are defined in the module in which they are used. it is possible to define task in separate file and use compile directive 'include to include the task in the file which instantiates the task.
- task can include timing delays, like posedge, negedge, # delay.
- task 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.
- task can take drive and source global variables, when no local variables are used. When local variables are used, it basically assigned output only at the end of task execution.
- task can call another task or function.
- task 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

- task begins with keyword task and end's with keyword endtask
- input and output are declared after the keyword task.
- local variables are declared after input and output declaration.

Example : Simple Task

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

Example : Task using Global Variables

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

Calling a Task

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

```
module temp_cal (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)
    convert (temp_a, temp_b);

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

endmodule
```

Function

A Verilog HDL function is same as task, with very little difference, like function cannot drive more then one output, can not contain delays.

- function are defined in the module in which they are used. it is possible to define function in separate file and use compile directive 'include to include the function in the file which instantiates the task.
- function can not include timing delays, like posedge, negedge, # delay. Which means that function should be executed in "zero" time delay.
- function can have any number of inputs and 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.

- function can take drive and source global variables, when no local variables are used. When local variables are used, it basically assigned output only at the end of function execution.
- function can be used for modeling combinational logic.
- function can call other functions, but can not call task.

Syntax

- function begins with keyword function and end's with keyword endfunction
- input are declared after the keyword function. Ouputs are delcared.

Example : Simple Function

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

Calling a Function

Lets assume that function in above example is stored in a file called myfunction.v. Advantage of coding function in separate file is that, it can be used in multiple module's.

```
module func_test(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
```

System Task and Function

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*. $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 where 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 the 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 give it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

Syntax

data_out = $random (seed);

$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.dmp")
- $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 one 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.

Art of Writing TestBenches

Introduction

Writing testbench is as complex as writing the RTL code itself. This days ASIC's are getting more and more complex and thus the challenge to verify this complex ASIC. Typically 60-70% of time in any ASIC is spent on verification/validation/testing. Even though above facts are well know to most of the ASIC engineers, but still engineers think that there is no glory in verification.

I have picked few 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 below tutorial.

Before you Start

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

Example : Counter

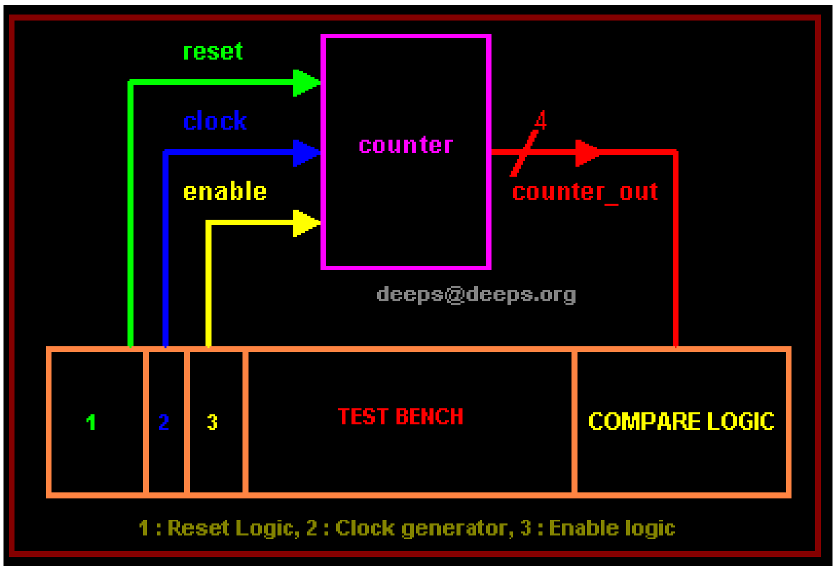
Lets assume that we have to verify a simple 4-bit up counter, which increments its count when ever enable is high and resets to zero, when reset is asserted high. Reset is synchronous to clock.

Code for Counter

```
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)
      count <= 0;
   else if ( enable == 1'b1)
      count <= count + 1;

endmodule
```

Test Plan

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



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

Test Cases

- Reset Test : We can start with reset deasserted, 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 then we are not here to test the counter, but to learn how to write test bench.

Writing TestBench

First step of any testbench creation is to creating a dummy template which basically declares inputs to DUT as reg and outputs from DUT as wire, instantiate the DUT as shown in code below. Note 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
```

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

Test Bench with Clock gen

```
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
```

Initial block in verilog is executed only once, thus simulator sets the value of clk, reset and enable to 0, which by looking at the counter code (of course you will be refering to the the DUT specs) could be found that driving 0 makes all this signals disabled.

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

At this point, you would like test if the testbench is generating the clock correctly, well you can compile with the Veriwell command line compiler found here. You need to give command line option as shown below. (Please let me know if this is illegal to have this compiler local to this website).

```
C:\www.deeps.org\veridos counter.v counter_tb.v
```

Of course it is a very good idea to keep file names same as module name. Ok, coming back to compiling, you will see that simulator does not come out, or print anything on screen or does it dump any waveform. Thus we need to add support for all the above as shown in 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
```

$dumpfile is used for specifying the file that simulator will use to store the waveform, that can be used later to view using waveform viewer. (Please refer to tools section for freeware version 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 tab. Syntax is same as printf. Second line $monitor is bit different, $monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). When ever anyone of them changes, it prints their value, in the respective radix specified.

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

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

```
C:\www.deeps.org>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 a event has occurred.

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

Code of reset logic

```
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
```

Adding test case logic

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

- Reset Test : We can start with reset deasserted, 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.

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

Test Case # 1 : Asserting/ Deasserting 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
```

Test Case # 2 : Asserting/ Deasserting 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 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
```

Test Case # 3 : Asserting/Deasserting 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 begin
      repeat (10) begin
        @ (negedge clk);
        enable = $random;
      repeat (10) begin
        @ (negedge clk);
        reset = $random;
      end
   end
 end
```

Well you might ask, are 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 condition 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 as `include directive 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 all the three test cases, you will find that, even through test case execution is not complete, simulation terminates. To have better control, what we can do is, add a 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 below.

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

and the modified test case #2 would 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
```

Second problem with the approach that we have taken till now it that, we need to manually check the waveform and also the output of simulator 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 model that mimics the DUT in functionality. In our example, to mimic DUT, it going to be very easy, but at times if DUT is complex, then to mimic the DUT will be a very complex and requires lot of innovative techniques to make self checking work.

```
reg [3:0] count_compare;
always @ (posedge clk)
if (reset == 1'b1)
  count_compare <= 0;
else if ( enable == 1'b1)
  count_compare <= count_compare + 1;
```

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 print's 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
```

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 env more friendly.

```
C:\Download\work>veridos counter.v counter_tb.v
VeriWell for Win32 HDL <Version 2.1.4> 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 above output, well you can find it here and counter code here.
There are lot of things that I have not covered, may be when I find time, I may add some more details on this subject.
As of books, I am yet to find a good book on writing test benches.

Modeling Memories and FSM

Memory Modeling

To help modeling of memory, Verilog provides support of two dimension arrays. Behavioral models of memories are modeled by declaring an array of register variables, any word in the array may be accessed by 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 width of memory and [0:255] is depth of memory with following parameters

- Width : 8 bits, little endian
- Depth : 256, address 0 corresponds to location 0 in array.

Storing Values

my_memory[address] = data_in;

Reading Values

data_out = my_memory[address];

Bit Read

Sometime there may be need to just read only one bit. Unfortunately Verilog does not allow to read only or write only one bit, the work around 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 task $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
```

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
```

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

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

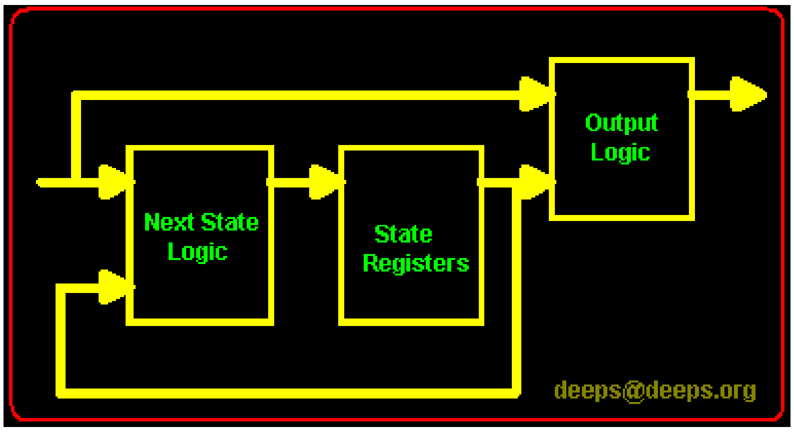
Introduction to FSM

State machine or FSM are the heart of any digital design, of course counter is a simple form of FSM. When I was learning Verilog, I use 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 could be found in the tidbits section.

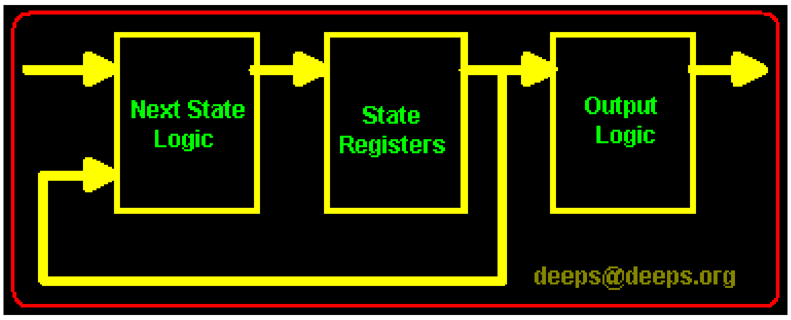
State machine Types

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



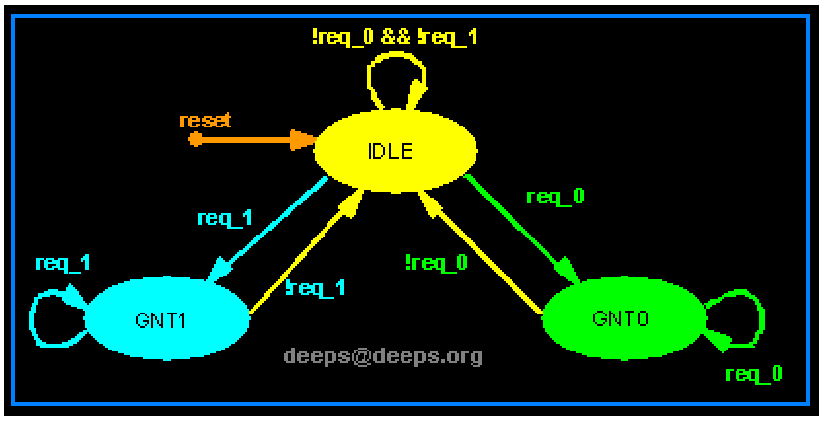
State machines can also be classified based on type state encoding used. Encoding style is also a critical factor which decides speed, and gate complexity of the FSM. Binary, gray, one hot, one cold, and almost one hot are the different types of encoding styles used in coding FSM states.

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 should be synchronized before feeding to FSM. Always try to keep FSM in separate Verilog file.

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

State Diagram.



Verilog Code

FSM code should have three sections,

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

Encoding Style

One Hot Encoding

```
parameter [1:0]  IDLE  = 3'b001,
                 GNT0 = 3'b010,
                 GNT1 = 3'b100;
```

Binary Encoding

```
parameter [1:0]  IDLE  = 2'b00,
                 GNT0 = 2'b01,
                 GNT1 = 2'b10;
```

Gray Encoding

```
parameter [1:0]  IDLE  = 2'b00,
                 GNT0 = 2'b10,
                 GNT1 = 2'b01;
```

Combinational Section

This section can be modeled using function, assign statement or using always block with case statement. For time being lets see always block version

```

 next_state = 3'b000
 case(state)
   IDLE : if (req_0 == 1'b1)
            next_state = GNT0;
            else if (req_1 == 1'b1)
             next_state= GNT1;
            else
            next_state = IDLE;
   GNT0 : if (req_0 == 1'b1)
            next_state = GNT0;
             else
            next_state = IDLE;
   GNT1 : if (req_1 == 1'b1) begin
             next_state = GNT1;
            else
            next_state =1 IDLE;
   default : next_state = IDLE
 endcase
end
```

Sequential Section.

This section has 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;
    state <= #1 IDLE;
  end
  else begin
```

```
    state <= #1 next_state;
    case(state)
      IDLE : begin
                gnt_0 <= #1 1'b0;
                gnt_1 <= #1 1'b0;
                end
      GNT0 : begin
                gnt_0 <= #1 1'b1;
                gnt_1 <= #1 1'b0;
                end
       GNT1 : begin
                gnt_0 <= #1 1'b0;
                gnt_1 <= #1 1'b1;
                end
      default : begin
                gnt_0 <= #1 1'b0;
                gnt_1 <= #1 1'b0;
                end
    endcase
  end
end
```

Parameterized Modules

Introduction

Lets assume that we have a design, which requires us to have counters of various width, but of same functionality. May be we can assume that we have a design which requires lot of instants of different depth and width of RAM's of same functionality. Normally what we do is, create counters of different widths and then use them. Same rule applies to RAM that we talked about.

But Verilog provides a powerful way to work around 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 with either using defparam or by passing new set of parameters during instantiating. We call this as parameter over riding.

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.
- May be 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.
- May be changed using the defparam statement

Parameter Override using defparam

```
module secret_number;
  parameter my_secret = 0;
  initial
  $display("My secret number is %d", my_secret);

endmodule

module top;

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

  secret_number U0();
  secret_number U1();

endmodule
```

Parameter Override during instantiating.

```
module secret_number;
  parameter my_secret = 0;
  initial
  $display("My secret number in module is %d",
  my_secret);

endmodule

module top;

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

endmodule
```

Passing more then 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;
```

for complete code refer to models section.

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

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

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

endmodule
```

Verilog 2001

In Verilog 2001, above code 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),
.ADDRE_WIDTH(8), .RAM_DEPTH(256))
ram(clk,address,data,cs,we,oe);

endmodule
```

Was this copied from VHDL?

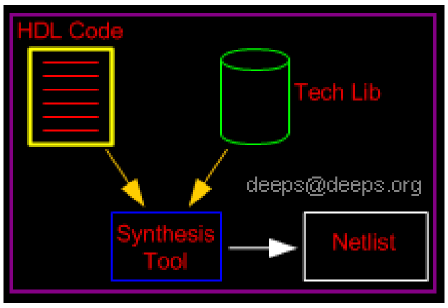
Verilog Synthesis Tutorial

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 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 is called technology library. Normally technology library is know 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 last chapter of Verilog tutorial.



Life before HDL (Logic synthesis)

As you must have experience in college, that let be counter or any other fancy logic. In college every thing has to be 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 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, while doing so it used 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 same as rest of the languages. What we intent to look in next few pages is how particular code gets translated to gates. As you must have wondered whiled reading earlier chapters, how could this be represented in Hardware. Example would be "delays", There is no way we could synthesize delays, but ofcourse we can add delay to particular signal by adding buffers. But then this becomes too dependent on synthesis target technology. (More on this in VLSI section).

First we will look at the constructs that are not supported by synthesis tools, Table below shows the constructs that are 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 non-blocking 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 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 then one always blocks. Well it will surely cause synthesis error.

initial Statement

```
module clk_gen (clk);
 output clk;
 reg clk;

 initial begin
   clk = 0;
 end

 always begin
   #10 clk = !clk;
 end

endmodule
```

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 below.

a = b;

Comparison to X and Z are always ignored

```
module compare (a,b);
  output a;
  input b;
  reg a;

  always @ (b)
    if (b == 1'bz)
      a = 1;
    else
      a = 0;

endmodule
```

There seems to a common problem with all the new to hardware design engineers. They normally tend to compare variables with X and Z. In practice it is worst thing to do. So please avoid comparing with X and Z. Limit your design to two state's, 0 and 1. Use tri-state only at chip IO pads level. We will see this as a example in 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 are simple and cause little headache's 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.

Most common way to model any logic is to use either assign statement or always block. assign statement can be used for modeling only combinational logic and always can be used for modeling both combinational and Sequential logic.

Construct Type	Keyword or Description	Notes
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	Eg: nand (out,a,b) bad idea to code RTL this way.
function and tasks	function , task	Timing constructs ignored
procedural	always, if, then, 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 occure, getting confused with logical and Reduction operators. So watch out.

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Divide
	+	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
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 second is sequential circuits. There are very few rules that need to be followed to get good synthesis output and avoid surprises.

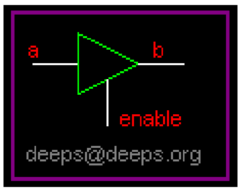
Combinational Circuit Modeling using assign

Combinational circuits modeling in Verilog can be done using assign and always blocks. Writing simple combination circuit in Verilog using assign statement is very straight forward. Like in example below

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

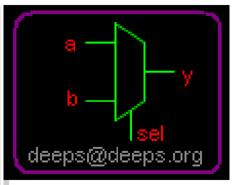
Tri-state buffer

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

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

endmodule
```



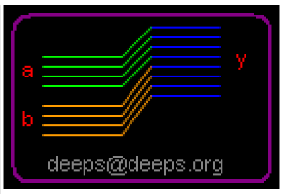
2:1 mux

```
module mux_21 (a,b,sel,y);
    input a, b;
    output y;
    input sel;
    wire y;

    assign y = (sel) ? b : a;

endmodule
```



Simple Concatenation

```
module bus_con (a,b);
    input [3:0] a, b;
    output [7:0] y;
    wire [15:0] y;

    assign y = {a,b};

endmodule
```



1 bit adder with carry

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

    assign {carry,sum} = (a + b);

endmodule
```

Multiply by 2

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

    assign product  = a << 1;

endmodule
```

3 is to 8 decoder

```
module decoder (in,out);
    input [2:0] in;
    output [7:0] out;
    wire [4:0] 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
```

Combinational Circuit Modeling using always

While modeling using always statement, there is chance of getting latch after synthesis if proper care is not taken care. (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 problem with latches and second reason being, glitch at enable pin of latch is another problem).

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

3 is to 8 decoder

```
module decoder (in,out);
    input [2:0] in;
    output [7:0] out;
    reg [4:0] 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
```

Sequential Circuit Modeling

Sequential logic circuits are modeled by use of edge sensitive elements in sensitive list of always blocks. Sequential logic can be modeled only by use of always blocks. Normally we use non-blocking assignments for sequential circuits.

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
```

Verilog Coding Style

If you look at the above code, you will see that I have imposed 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 small list of guidelines.

- Use meaningful names for signals and variables
- Don't mix level and edge sensitive in one 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 non-blocking for sequential and blocking for combo logic
- Don't mix blocking and non-blocking assignments in one always block. (though 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.

Verilog PLI Tutorial

Introduction

Verilog PLI(Programming Language Interface) is a mechanism to invoke C or C++ functions from Verilog code.

The function invoked in Verilog code is called a system call. An example of a built-in system call is $display, $stop, $random. PLI allows the user to create custom system calls, Something that Verilog syntax does not allow us to do. Some of this are:-

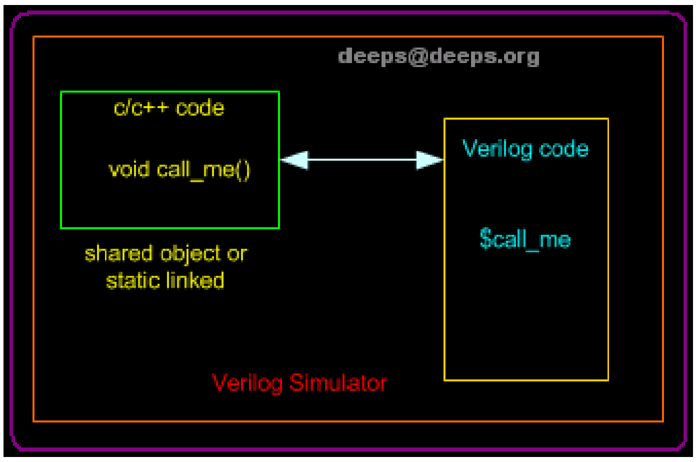
- Power analysis.
- Code coverage tools.
- Can modify the Verilog simulation data structure - more accurate delays.
- Custom output displays.
- Co-simulation.
- Design debug utilities.
- Simulation analysis.
- C-model interface to accelerate simulation.
- Testbench modeling.

To achieve above few application of PLI, C code should have the access to the internal data structure of the Verilog simulator. To facilitate this Verilog PLI provides with something called acc routines or simply access routines.

There is second set of routines, which are called tf routines, or simply task and function routines. The tf and acc are PLI 1.0 routines and is very vast and very old routines. The next set of routine, which was introduced with latest release of Verilog 2001 is called vpi routines. This is small and crystal clear PLI routines and thus the new version PLI 2.0.

How it Works

- Write the functions in C/C++ code.
- Compile them to generate shared lib (*.DLL in Windows and *.so in UNIX). Simulator like VCS allows static linking.
- Use this Functions in Verilog code (Mostly Verilog Testbench).
- Based on simulator, pass the C/C++ function details to simulator during compile process of Verilog Code.
- Once linked just run the simulator.



Example : Hello World

We will define a function hello, which when called will print "Hello Deepak". This example does not use any of the PLI standard functions (ACC, TF and VPI). For exact linking details, please refer to simulator manuals. Each simulator implements its own way for linking C/C++ functions to simulator.

C Code

```
#include <stdlib.h>   /* ANSI C standard library */
#include <stdio.h>   /* ANSI C standard input/output library */

void hello () {
    printf ("Hello Deepak");
}
```

Verilog Code

```
module hello_pli ()

initial begin
    $hello;
    #10 $finish;
end
endmodule
```

Running the Simulation

Need to add

PLI TF and ACC interface mechanism

Need to add

What's new in Verilog 2001

Introduction

Well most of the changes in Verilog 2001 are picked from other languages. Like generate, configuration, file operation was from VHDL. I am just adding a list of most commonly used Verilog 2001 changes. You could use the Icarus Verilog simulator for testing examples in this section.

Comma used in sensitive list

In earlier version of Verilog ,we use to use or to specify more then one sensitivity list elements. In the case of Verilog 2001, we use comma as shown in example below.

```
always @ (a, b, c, d, e )

always @ (posedge clk, posedge reset)
```

Combinational logic sensitive list

```
always @  *
a = ((b&c) || (c^d));
```

Wire Data type

In Verilog 1995, default data type is net and its width is always 1 bit. Where as in Verilog 2001. The width is adjusted automatically.

In Verilog 2001, we can disable default data type by `default net_type none, This basically helps in catching the undeclared wires.

Register Data type

Register data type is called as variable, as it created lot of confusion for beginners. Also it is possible to specify initial value to Register/variable data type. Reg data type can also be declared as signed.

```
reg [7:0] data = 0;
reg signed [7:0] data;
```

New operators

<<<, >>> : Shift left, shift right : To be used on signed data type
** : exponential power operator.

Port Declaration

```
module adder (
input [3:0] a,
input  [3:0] b,
output [3:0] sum
);

module adder (a,b,y);
input wire [3:0] a,
input wire [3:0] b,
output reg [3:0] sum
```

This is equivalent to Verilog 1995 as given below

```
module adder (a,b,y);
input a;
input b;
output y;
wire a;
wire b;
reg sum;
```

Random Generator

In Verilog 1995, each simulator used to implement its own version of $random. In Verilog 2001, $random is standardized, so that simulations runs across all the simulators with out any inconsistency.

Generate Blocks

This feature has been taken from VHDL with some modification. It is possible to use for loop to mimic multiple instants.

Multi Dimension Array.

More then two dimension supported.

There are lot of other changes, Which I plan to cover sometime later. Or may be I will mix this with the actual Verilog tutorial with reference to Verilog 2001, when ever necessary.

Verilog Quick Reference

MODULE	**module** MODID[({PORTID,})]; [**input | output | inout** [range] {PORTID,};] [{declaration}] [{parallel_statement}] [specify_block] **endmodule** range ::= [constexpr : constexpr]

DECLARATIONS	**parameter** {PARID = constexpr,};
	wire | wand | wor [range] {WIRID,};
	reg [range] {REGID [range],};
	integer {INTID [range],};
	time {TIMID [range],};
	real {REALID,};
	realtime {REALTIMID,};
	event {EVTID,};
	task TASKID;
	[{**input | output | inout** [range] {ARGID,};}]
	[{declaration}]
	begin
	[{sequential_statement}]
	end
	endtask
	function [range] FCTID;
	{**input [range**] {ARGID,};}
	[{declaration}]
	begin
	[{sequential_statement}]
	end
	endfunction

PARALLEL STATEMENTS	**assign** [(strength1, strength0)] WIRID = expr; **initial** sequential_statement **always** sequential_statement MODID [#({expr,})] INSTID ([{expr,} | {.PORTID(expr),}]); GATEID [(strength1, strength0)] [#delay] [INSTID] ({expr,}); defparam {HIERID = constexpr,}; strength ::= **supply | strong | pull | weak | highz** delay ::= number | PARID | (expr [, expr [, expr]])

GATE PRIMITIVES	and (out, in1, ..., inN);
	or (out, in1, ..., inN);
	xor (out, in1, ..., inN);
	buf (out1, ..., outN, in);
	bufif0 (out, in, ctl);
	notif0 (out, in, ctl);
	pullup (out);
	[r]pmos (out, in, ctl);
	[r]nmos (out, in, ctl);
	[r]cmos (out, in, nctl, pctl);
	[r]tran (inout, inout);
	[r]tranif1 (inout, inout, ctl);
	[r]tranif0 (inout, inout, ctl);
		nand (out, in1, ..., inN);
		nor (out, in1, ..., inN);
		xnor (out, in1, ..., inN);
		not (out1, ..., outN, in);
		notif1 (out, in, ctl);
		bufif1 (out, in, ctl);
		pulldown (out);

SEQUENTIAL STATEMENTS	**;** **begin**[: BLKID [{declaration}]] [{sequential_statement}] **end** **if** (expr) sequential_statement [else sequential_statement] **case | casex | casez** (expr) [{{expr,}: sequential_statement}] [default: sequential_statement] **endcase** **forever** sequential_statement **repeat** (expr) sequential_statement **while** (expr) sequential_statement **for** (lvalue = expr; expr; lvalue = expr) sequential_statement #(number | (expr)) sequential_statement @ (event [{or event}]) sequential_statement lvalue [<]= **[#(**number | (expr))] expr; lvalue [<]= [@ (event [{or event}])] expr;wait (expr) sequential_statement -> EVENTID; **fork**[: BLKID [{declaration}]] [{sequential_statement}] **join** **TASKID**[({expr,})]; disable BLKID | TASKID; **assign** lvalue = expr; **deassign** lvalue; lvalue ::= **ID**[range] | ID[expr] | {{lvalue,}} **event** ::= [posedge **|** negedge] expr	

SPECIFY BLOCK	specify_block ::= **specify**
	{specify_statement}
	endspecify

SPECIFY BLOCK STATEMENTS	**specparam** {ID = constexpr,}; (terminal => terminal) = path_delay; ((terminal,} *> {terminal,}) = path_delay; **if** (expr) (terminal [+|-]=> terminal) = path_delay; **if** (expr) ({terminal,} [+|-]*> {terminal,}) = path_delay; [**if** (expr)] ([**posedge|negedge**] terminal => (terminal [+|-]: expr)) = path_delay; [if (expr)] ([**posedge|negedge**] terminal *> ({terminal,} [+|-]: expr)) = path_delay; **$setup**(tevent, tevent, expr [, ID]); **$hold**(tevent, tevent, expr [, ID]); **$setuphold**(tevent, tevent, expr, expr [, ID]); **$period**(tevent, expr [, ID]); **$width**(tevent, expr, constexpr [, ID]); **$skew**(tevent, tevent, expr [, ID]); **$recovery**(tevent, tevent, expr [, ID]); **tevent** ::= [posedge | negedge] terminal [&&& scalar_expr] path_delay ::= expr **|** (expr, expr [, expr [, expr, expr, expr]]) terminal ::= ID[range] | ID[expr]
EXPRESSIONS	primary unop primary expr binop expr expr ? expr : expr primary ::= literal | lvalue | FCTID({expr,}) | (expr)
UNARY OPERATORS	**+, -** Positive, Negative **!** Logical negation **~** Bitwise negation **&, ~**& Bitwise and, nand **|, ~|** Bitwise or, nor **^, ~^, ^~** Bitwise xor, xnor

BINARY OPERATORS	Increasing precedence: **?:** if/else **||** Logical or **&&** Logical and **|** Bitwise or **^, ^~** Bitwise xor, xnor **&** Bitwise and **==, != , ===, !==** Equality **<, <=, >, >=** Inequality **<<, >>** Logical shift **+, -** Addition, Subtraction ***, /,** % Multiply, Divide, Modulo
SIZES OF EXPRESSIONS	unsized constant 32 sized constant as specified i op j +,-,*,/,%,&,|,^,^~ max(L(i), L(j)) op i +, -, ~ L(i) i op j ===, !==, ==, != &&, ||, >, >=, <, <= 1 op i &, ~&, |, ~|, ^, ~^ 1 i op j >>, << L(i) i ? j : k max(L(j), L(k)) {i,...,j} L(i) + ... + L(j) {i{j,...k}} i * (L(j)+...+L(k)) i = j L(i)
SYSTEM TASKS	* indicates tasks not part of the IEEE standard but mentioned in the informative appendix.
INPUT	**$readmemb**(“fname”, ID [, startadd [, stopadd]]); **$readmemh**(“fname”, ID [, startadd [, stopadd]]); **$sreadmemb**(ID, startadd, stopadd {, string}); **$sreadmemh**(ID, startadd, stopadd {, string});
OUTPUT	**$display**[defbase]([fmtstr,] {expr,}); **$write**[defbase] ([fmtstr,] {expr,}); **$strobe**[defbase] ([fmtstr,] {expr,}); **$monitor**[defbase] ([fmtstr,] {expr,}); **$fdisplay**[defbase] (fileno, [fmtstr,] {expr,}); **$fwrite**[defbase] (fileno, [fmtstr,] {expr,}); **$fstrobe**(fileno, [fmtstr,] {expr,}); **$fmonitor**(fileno, [fmtstr,] {expr,}); **fileno** = $fopen(“filename”); **$fclose**(fileno); **defbase ::= h | b | o**

TIME	**$time** “now” as TIME **$stime** “now” as INTEGER **$realtime** “now” as REAL **$scale**(hierid) Scale “foreign” time value **$printtimescale**[(path)] Display time unit & precision **$timeformat**(unit#, prec#, “unit”, minwidth) Set time %t display format
SIMULATION CONTROL	**$stop** Interrupt **$finish** Terminate **$save**(“fn”) Save current simulation **$incsave**(“fn”) Delta-save since last save **$restart**(“fn”) Restart with saved simulation **$input**(“fn”) Read commands from file **$log[**(“fn”)] Enable output logging to file **$nolog** Disable output logging **$key**[(“fn”)] Enable input logging to file **$nokey** Disable input logging **$scope**(hiername) Set scope to hierarchy **$showscopes** Scopes at current scope **$showscopes**(1) All scopes at & below scope **$showvars** Info on all variables in scope **$showvars**(ID) Info on specified variable **$countdrivers**(net)>1 driver predicate **$list[**(ID)] List source of [named] block **$monitoron** Enable $monitor task $**monitoroff** Disable $monitor task **$dumpon** Enable val change dumping **$dumpoff** Disable val change dumping **$dumpfile**(“fn”) Name of dump file **$dumplimit(**size) Max size of dump file **$dumpflush** Flush dump file buffer **$dumpvars**(levels [{, MODID \| VARID}]) Variables to dump **$dumpall** Force a dump now **$reset[(0)]** Reset simulation to time 0 **$reset(1)** Reset and run again **$rese**t(0\|1, expr) Reset with reset_value*$reset_value Reset_value of last $reset **$reset**_count # of times $reset was used
MISCELLANEOUS	**$random[(ID)]** **$getpattern**(mem) Assign mem content **$rtoi(expr)** Convert real to integer **$itor(expr)** Convert integer to real **$realtobits(expr)** Convert real to 64-bit vector **$bitstoreal(expr)** Convert 64-bit vector to real

ESCAPE SEQUENCES IN FORMAT STRINGS	**\n, \t, \\, \”** newline, TAB, ‘\’, ‘“ **\xxx** character as octal value %% character ‘%’ %[w.d]e, %[w.d]E display real in scientific form %[w.d]f, %[w.d]F display real in decimal form %[w.d]g, %[w.d]G display real in shortest form %[0]h, %[0]H display in hexadecimal %[0]d, %[0]D display in decimal %[0]o, %[0]O display in octal %[0]b, %[0]B display in binary %[0]c, %[0]C display as ASCII character %[0]v, %[0]V display net signal strength %[0]s, %[0]S display as string %[0]t, %[0]T display in current time format %[0]m, %[0]M display hierarchical name
LEXICAL ELEMENTS	hierarchical identifier ::= {INSTID .} identifier identifier ::= letter | _ { alphanumeric | $ | _} escaped identifer ::= \ {nonwhite} decimal literal ::= [+|-]integer [. integer] [E|e[+|-] integer] based literal ::= integer ‘ base {hexdigit | x | z} base ::= b | o | d | h comment ::= // comment newline comment block ::= /* comment */

VERILOG IN ONE DAY

Introduction

I wish I could learn Verilog in one day, well that's every new learners dream. In next few pages I have made an attempt to make this dream a real one for those new learners. There will be some theory, some examples followed by some exercise. Only requirement for this "Verilog in One Day" is that you should be aware of at least one programming language. One thing that makes Verilog and software programming languages different is that, in Verilog execution of different blocks of code is concurrent, where as in software programming language it is sequential. Of course this tutorial is useful for those who have some background in Digital design back ground.

Life before Verilog was life of Schematics, where any design, let it be of any complexity use to designed thought schematics. This method of schematics was difficult to verify and was error prone, thus resulting in lot of design and verify cycles.

Whole of this tutorial is based around a arbiter design and verification. We will follow the typical design flow found here.

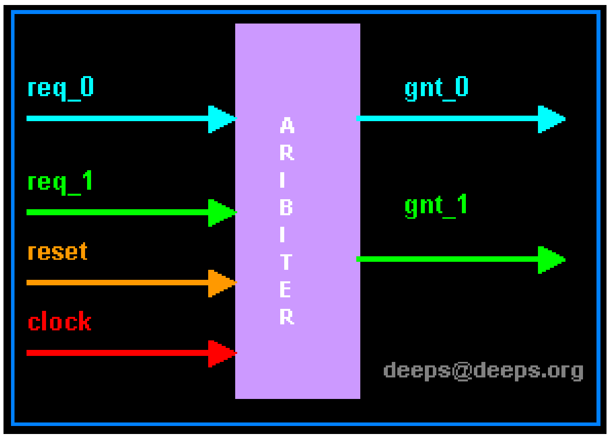
? Specs
? High level design
? Low level design or micro design
? RTL coding
? Verification
? Synthesis.

For anything to be designed, we need to have the specs. So lets define specs.

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

Once we have the specs, we can draw the block diagram. Since the example that we have taken is a simple one, For the record purpose we can have a block diagram as shown below.

Block diagram of arbiter

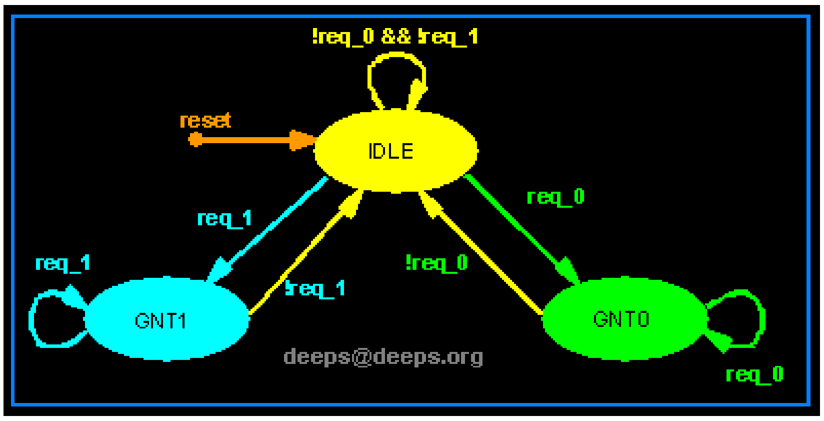


Normal digital design flow dictates that we draw a stated machine, from there we draw the truth table with next state transition for each flip-flop. And after that we draw kmaps and from kmaps we can get the optimized circuit. This method works just fine for small design, but with large design's this flow becomes complicated and error prone.

You may refer to the digital section to understand this flow (I think this flow tutorial in Digital section is still under construction).

Low level design

Here we can add the signals at the sub module level and also define the state machine if any in greater detail as shown in the figure below.



Modules

If you look at the arbiter block, we can see that it has got a name arbiter and input/output ports. Since Verilog is a HDL, it needs to support this, for this purpose we have reserve word "module".

module arbiter is same as block arbiter, Each module should follow with port list as shown in code below.

Code of module "arbiter"

If you look closely 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. (In Verilog 2001 we can define ports and port directions at one place), as shown in code below.

```
module aribiter (
    clock      , // clock
    reset      , // Active high, syn reset
    req_0      , // Request 0
    req_1      , // Request 1
    gnt_0      , // Grant 0
    gnt_1
    );
    /-------------Input Ports-----------------------------
    input       clock             ;
    input       reset              ;
    input       req_0             ;
    input       req_1             ;

    //-------------Output Ports----------------------------
    output      gnt_0              ;
    output      gnt_1              ;
```

As you can see, we have only two types of ports, input and output. But in real life we can have bi-directional ports also. Verilog allows us to define bi-directional ports as "inout"

Example :

```
inout  read_enable;
```

One make ask " How do I define vector signals", Well Verilog do provide simple means to declare this too.

Example :

```
inout [7:0] address;
```

where left most bit is 7 and rightmost bit is 0. This is little endian convesion.

Summary

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

Data Type

Oh god what this data type has to do with hardware ?. Well nothing special, it just that people wanted to write one more language that had data types (need to rephrase it!!!!). No hard feelings :-).

Actually there are two types of drivers in hardware...

What is this driver ?

Driver is the one which can drive a load. (guess, I knew it).

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

First one is called reg data type and second data type is called wire. You can refer to this page for getting more confused.

There are lot of other data types for making newbee life bit more harder. Lets not worry about them for now.

Examples :

```
wire  and_gate_output;

reg  d_flip_flop_output;

reg [7:0] address_bus;
```

Summary

- wire data type is used for connecting two points.
- reg data type is used for storing values.
- May god bless rest of the data types.

Operators

If you have seen the pre-request for this one day nightmare, you must have guessed now that Operators are same as the one found in any another programming language. So just to make life easies, all operators like in the list below are same as in C language.

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Divide
	+	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
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?:	Conditional

Example :

- a = b + c ; // That was very easy
- a = 1 << 5; // Hum let me think, ok shift '1' left by 5 position.
- a = !b   ; // Well does it invert b???
- a = ~b   ; // How many times do you want to assign to 'a', it could cause multiple-drivers.

Summary

- Lets attend C language training again.

Control Statements

Did we come across "if else"," repeat", "while", "for" "case". Man this is getting boring, Looks like Verilog was picked from C language. Functionality of Verilog Control statement is same as C language. Since Verilog is a HDL (Hardware Description Language), this control statements should translate to Hardware, so better be careful when you use control statements. We will see this in detail in synthesis sub-section.

if-else

if-else statement is used for checking a condition to execute a portion of code. If condition does not satisfy, then execute code in other portion of code as shown in code below.

```
if (enable == 1'b1) begin
   data = 10; // Decimal assigned
   address = 16'hDEAD; // Hexa decimal
   wr_enable = 1'b1; // Binary
end else begin
   data  = 32'b0;
   wr_enable = 1'b0;
   address = address + 1;
end
```

One could use any operators in the condition checking as in the case of C language. If needed we can have nested if else statements, statements without else is also ok, but then it has its own problem when modeling combinational logic, if statement without else results in a Latch (this is not always true).

case

Case statement is used where we have one variable, which needs to be checked for multiple values. Like a address decoder, where input is address and it needs to checked for all the values that it can take. In Verilog we have casex and casez, This are good for reading, but for implementation purpose just avoid them. You can read about them in regular Verilog text.

Any case statement should begin with case reserved word, and end with encase reserved word. It is always better to have default statement, as this always takes care of un-covered case. Like in FSM, if all cases are not covered and FSM enters into a un-covered statement, this could result in FSM hanging. If we default statement with return to idle state, could bring FSM to safe state.

```
  case(address)
    0 : $display ("It is 11:40PM");
    1 : $display ("I am feeling sleepy");
    2 : $display  ("Let me skip this tutorial");
    default : $display  ("Need to complete");
  endcase
```

Looks like address value was 3 and so I am still writing this tutorial. 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 combination statement, the synthesis tool will infer Latch.

While

While statement checks if a condition results in Boolean true and executed the code within the begin and end statements. Normally while loop is not used for real life modeling, but used in Test benches

```
  while(free_time) begin
    $display ("Continue with webpage
    development");
  end
```

As long as free_time variable is set, code within the begin and end will be executed. i.e print "Continue with web development". Lets looks at a more strange example, which uses most of the constructs of Verilog. Well you heard it right. Verilog has very few reserve words then VHDL, and in this few, we use even lesser few for actual coding. So good of Verilog....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
```

We will visit this code later, you can find the RTL and Test bench for above here.

for loop

"for-loop" statement in Verilog is very close to C language "for-loop" statement, only difference is that ++ and -- operators is not supported in Verilog. So we end up using var = var + 1, as shown below.

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

Above code prints the value of i from 0 to 15. Using of for loop for RTL, should be done only after careful analysis.

repeat

"repeat" statement in Verilog is same as to C language "repeat" statement, Below code is simple example of a repeat statement.

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

Above example output will be same as the for-loop output. One question that comes to mind, why the hell someone would like to use repeat for implementing hardware.

Summary

- while, repeat statements are same as C language.
- if-else and case statements requires all the cases to covered for combinational logic.
- for-loop same as C, but no ++ and -- operators.

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.

- Combination elements can be modeled using assign and always statements.

? Sequential elements can be modeled using only always statement.
? There is third type, which is used in test benches only, it is called initial statement.

Before we discuss about this modeling, lets go back to the second example of while statement. In that example we had used lot of features of Verilog. Verilog allows user to give name to block of code, block of code is something that starts with reserve word "begin" and ends with reserve word "end". Like in the example we have "COUNT" as name of the block. This concept is called named block.

We can disable a block of code, by using reserve word "disable <block name>". In the above example, after the each incremented of counter, COUNT block of code is disabled.

Initial Blocks

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

Example

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

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

Always Blocks

As name suggest, always block executes always. Unlike initial block, which executes only once, at the beginning of simulation. Second difference is always block should have sensitive list or delay associated with it.

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

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

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

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

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

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

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

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

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

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

Assign Statement

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

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

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

```
assign out = data;
```

Above example is a simple buffer.

Task and Function

Just repeating same old thing again and again, Like any other programming language, Verilog provides means to address repeated used code, this are called Task and Functions. I wish I had something similar for the webpage, just call it to print this programming language stuff again and again.

Below code 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
```

function and task have same syntax, few difference is task can have delays, where function can not have any delay. Which means function can be used for modeling combination logic. You can find the example code here.

Test Benches

Ok, now 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, its same as we use to do in digital labs in college days. Drive the inputs, match the outputs with expected values. Lets look at the arbiter testbench.

```
module arbiter_tb;

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

initial begin
   $monitor ("req0=%b, req1=%b,
   gnt0=%b,gnt1=%b", req0,req0,gnt0,gnt1);
   clock = 0;
   reset = 0;
   req0 = 0;
   req1 = 0;
   #5 reset = 1;
   #15 reset = 0;
   #10 req0 = 1;S
   #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; // Generate clock
end

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

```
   );

   endmodule
```

Its 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 the needed variables, we initialize all the inputs to know 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 always block.

After we have 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
```

I have used Icarus Verilog simulator to generate the above output. You can get the code of the arbiter with testbench here.