Verilog Tutorial

By

Deepak Kumar Tala

http://www.asic-world.com

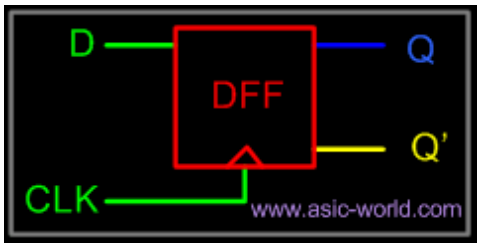
DISCLAIMER

I don't makes any claims, promises or guarantees about the accuracy, completeness, or adequacy of the contents of this tutorial and expressly disclaims liability for errors and omissions in the contents of this tutorial. No warranty of any kind, implied, expressed or statutory, including but not limited to the warranties of non−infringement of third party rights, title, merchantability, fitness for a particular purpose and freedom from computer virus, is given with respect to the contents of this tutorial or its hyperlinks to other Internet resources. Reference in this tutorial to any specific commercial products, processes, or services, or the use of any trade, firm or corporation name is for the information, and does not constitute endorsement, recommendation, or favoring by me. All the source code and Tutorials are to be used on your own risk. All the ideas and views in this tutorial are my own and are not by any means related to my employer.

INTRODUCTION
CHAPTER 1

Introduction

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




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

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

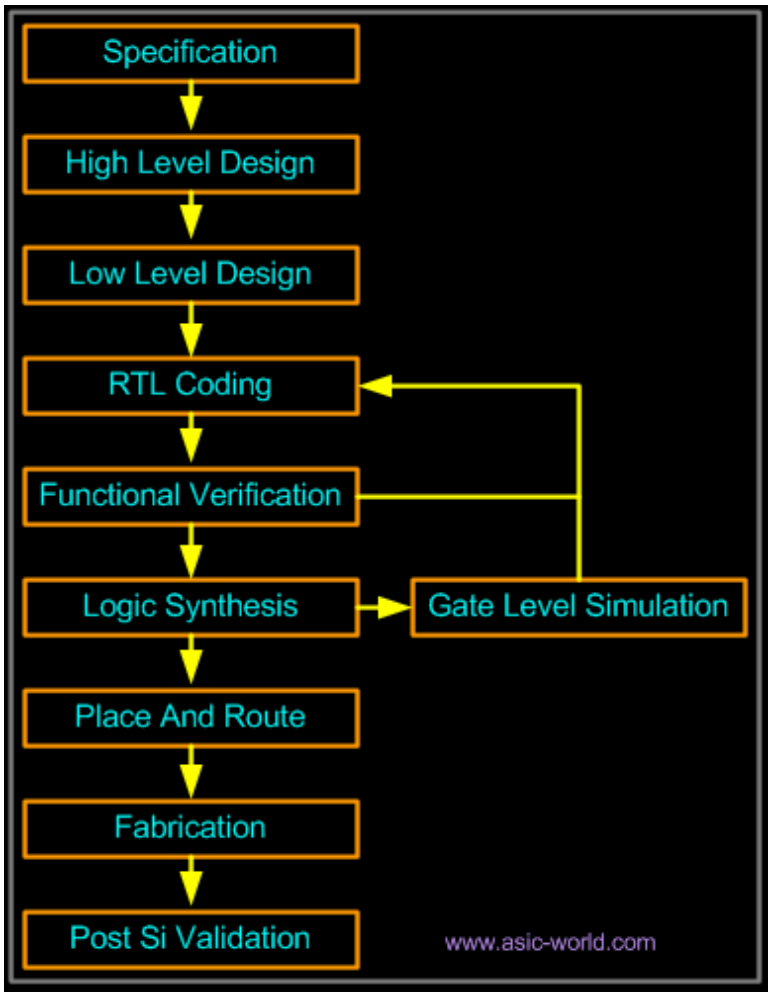
The traditional method of electronic design is bottom−up. Each design is performed at the gate−level using the standard gates (Refer to the Digital Section for more details) With increasing

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.*

NOTES

HISTORY OF VERILOG

CHAPTER 2

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.

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.

NOTES

DESIGN AND TOOL FLOW

CHAPTER 3

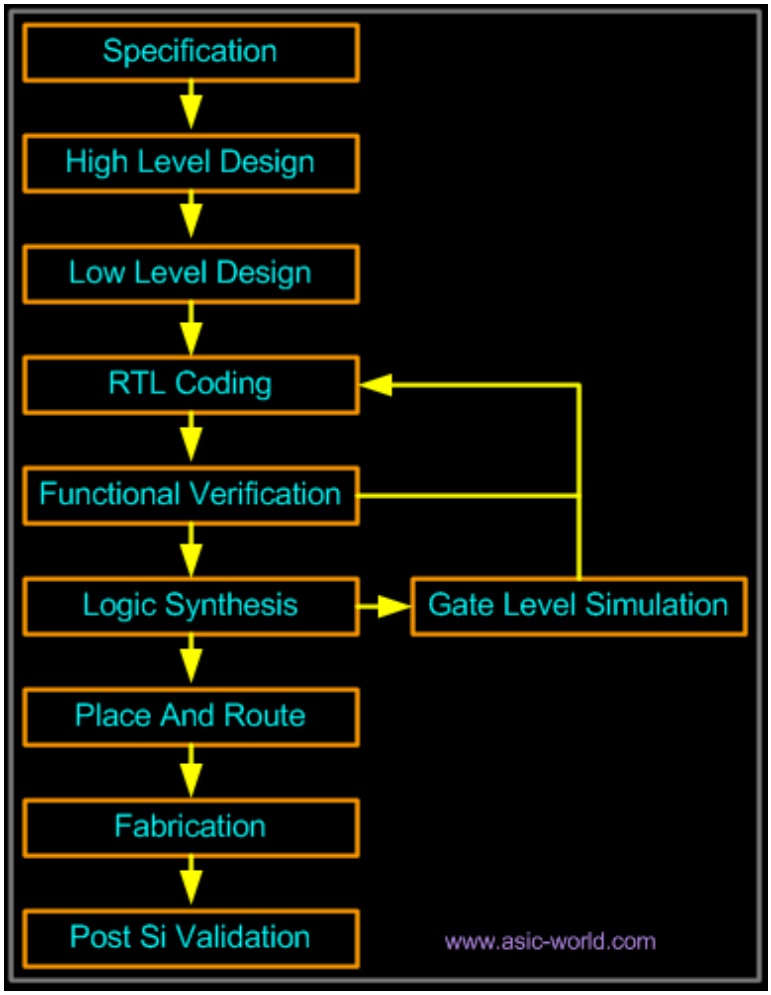
Introduction

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, Open Office.
- **High Level Design :** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.
- **Micro Design/Low level design:** Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word. For FSM StateCAD or some similar tool, Open Office.
- **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.
- **Post Si Validation :** For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place.

Figure : Typical Design flow



Specification

This is the stage at which we define what are the important parameters of the system/design that you are planning to design. 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, 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.

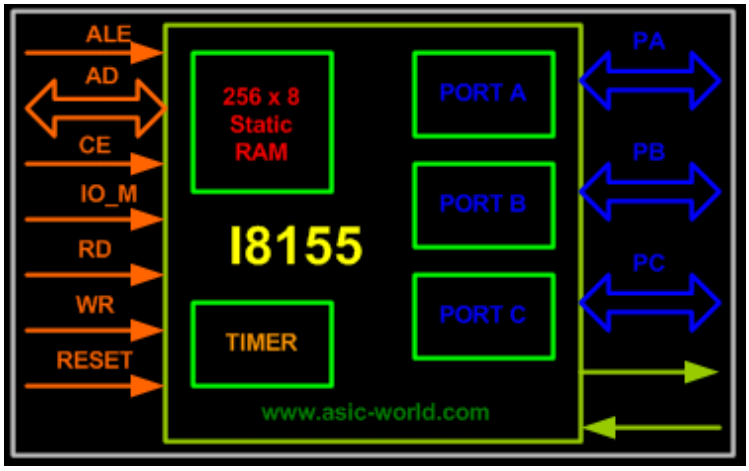


Figure : I8155 High Level Block Diagram

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. This is phase, where one spends lot of time.

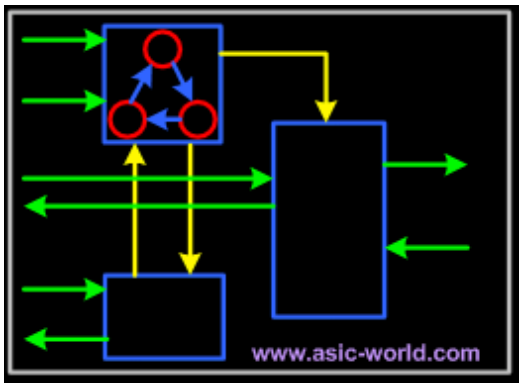


Figure : Sample Low level design

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.

```
1module addbit (
2a , // first input
3b , // Second input
4ci , // Carry input
5sum , // sum output
6co // carry output
7);
8//Input declaration
```

```
 9 input a;
10 input b;
11 input ci;
12 //Ouput declaration
13 output sum;
14 output co;
15 //Port Data types
16 wire a;
17 wire b;
18 wire ci;
19 wire sum;
20 wire co;
21 //Code starts here
22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit
```

Figure : Sample RTL code

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. Normally we spend 60–70% of time in verification of design.

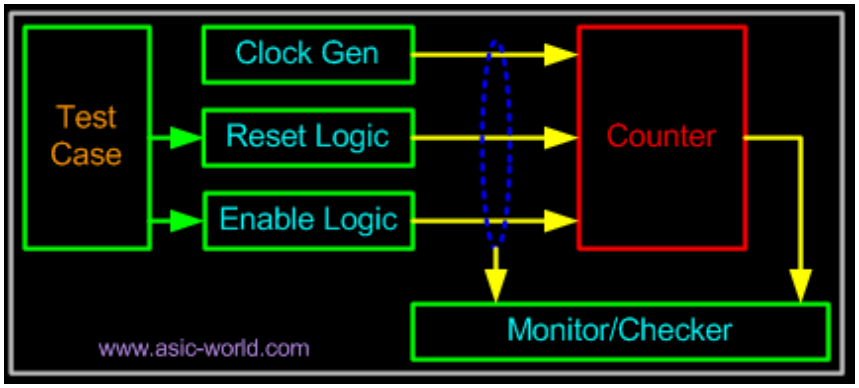


Figure : Sample Testbench Env

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 level simulation**.

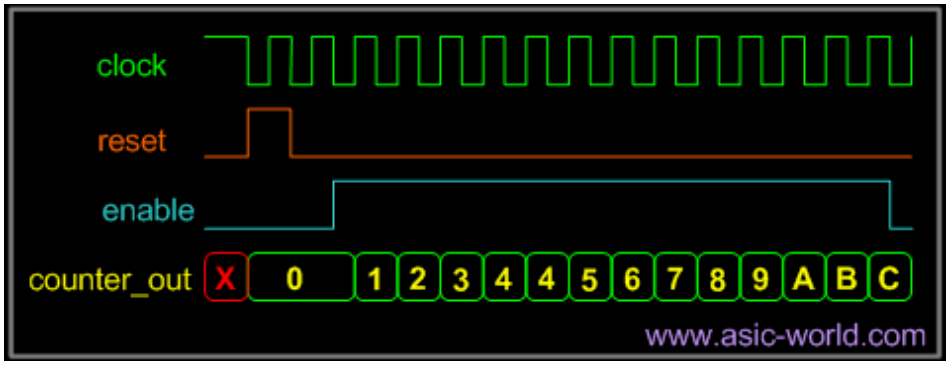


Figure : 4 bit Up Counter Waveform

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, they know only gate delays). After the synthesis there are couple of things that are normally done before passing the netlist to backend (Place and Route)

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

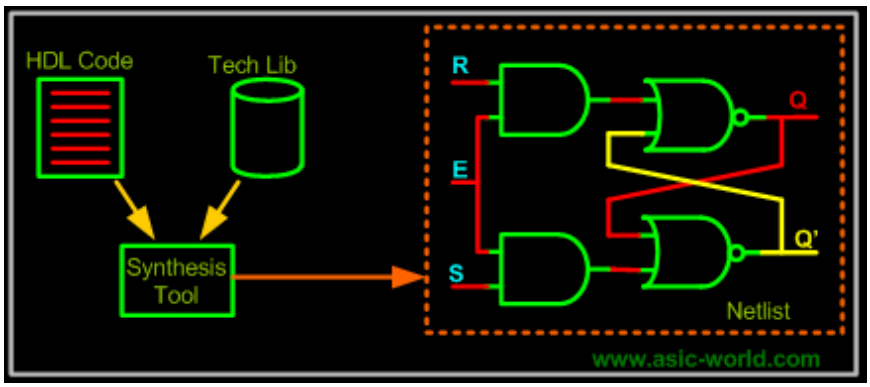


Figure : Synthesis Flow

Place & Route

Gatelevel netlist from the synthesis tool is taken and imported into place and route tool in Verilog netlist format. All the gates and flip–flops are places, Clock tree synthesis and reset is routed. After this each block is routed. Output of the P&R tool is GDS file, this files is used by foundry for fabricating the ASIC. Normally the P&R tool are used to output the SDF file, which is back annotated along with the gatelevel netlist from P&R into static analysis tool like Prime Time to do timing analysis.

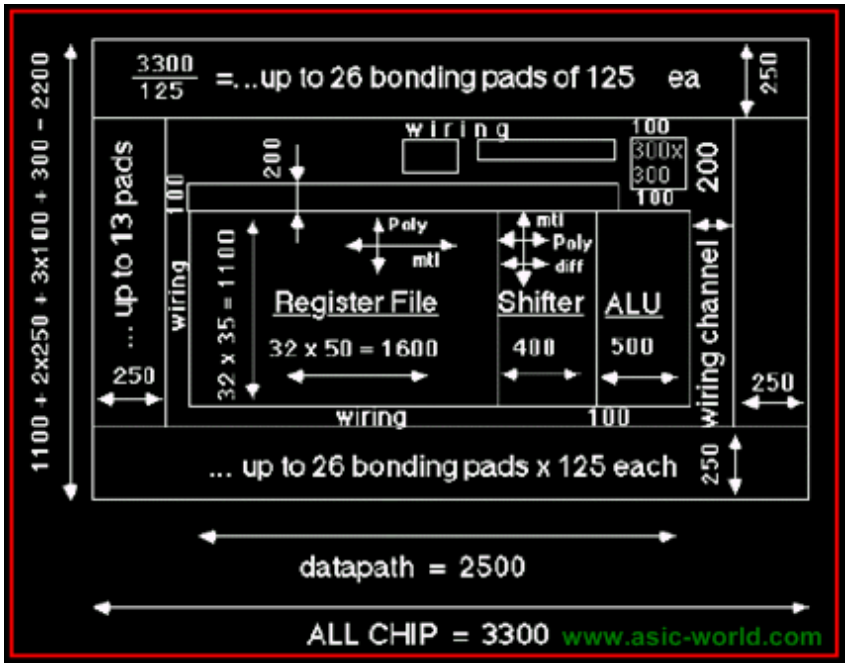


Figure : Sample micro–processor placement

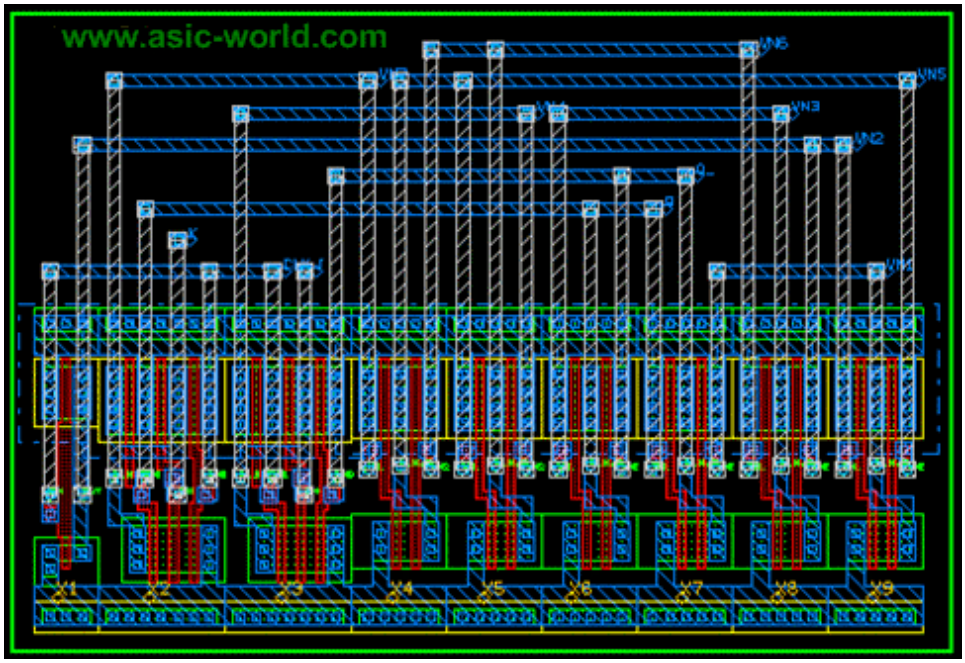


Figure : J–K Flip–Flop

Post Silicon Validation

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

NOTES

MY FIRST PROGRAM IN VERILOG

CHAPTER 4

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

```
1 //------------------------------------------------------
2 // This is my first Verilog Program
3 // Design Name : hello_world
4 // File Name : hello_world.v
5 // Function : This program will print 'hello world'
6 // Coder : Deepak
7 //------------------------------------------------------
8 module hello_world ;
9
10 initial begin
11    $display ( "Hello World by Deepak" );
12    #10 $finish;
13 end
14
15 endmodule // End of Module hello_world
```

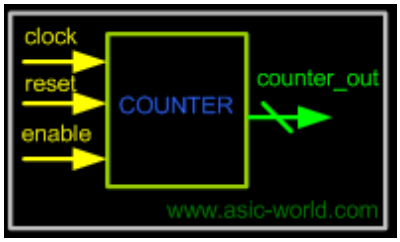
Words in green are comments, blue are reserved words, Any program in Verilog starts with reserved word module , In the above example line 7 contains module hello_world. (Note: We can have compiler pre-processor statements like `include, `define statements before module declaration)

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 within 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

```
Hello World by Deepak
```

Counter Design Block



Counter Design Specs

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

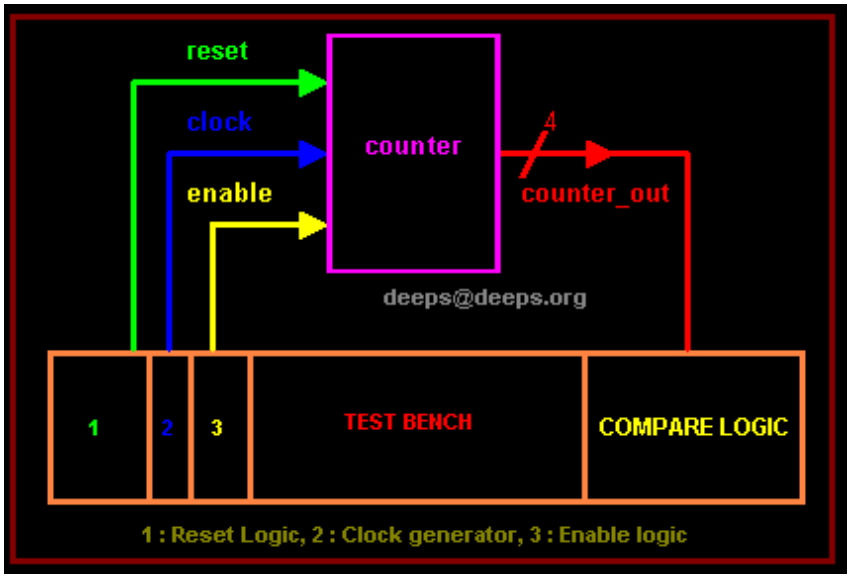
Counter Design

```
1 //------------------------------------------------------
2 // This is my second Verilog Design
3 // Design Name : first_counter
4 // File Name : first_counter.v
5 // Function : This is a 4 bit up-counter with
6 // Synchronous active high reset and
7 // with active high enable signal
8 //------------------------------------------------------
9 module first_counter (
10 clock , // Clock input ot the design
11 reset , // active high, synchronous Reset input
12 enable , // Active high enabel signal for counter
13 counter_out // 4 bit vector output of the counter
14 ); // End of port list
15 //-------------Input Ports-----------------------------
16 input clock ;
17 input reset ;
18 input enable ;
19 //-------------Output Ports----------------------------
20 output [3:0] counter_out ;
21 //-------------Input ports Data Type-------------------
22 // By rule all the input ports should be wires
23 wire clock ;
24 wire reset ;
25 wire enable ;
26 //-------------Output Ports Data Type------------------
27 // Output port can be a storage element (reg) or a wire
28 reg [3:0] counter_out ;
29
30 //-------------Code Starts Here------------------------
31 // Since this counter is a positive edge trigged one,
32 // We trigger the below block with respect to positive
33 // edge of the clock.
34 always @ (posedge clock)
35 begin : COUNTER // Block Name
36   // At every rising edge of clock we check if reset is active
```

```
37    // If active, we load the counter output with 4'b0000
38    if (reset == 1'b1) begin
39       counter_out <= #1 4'b0000;
40    end
41    // If enable is active, then we increment the counter
42    else if (enable == 1'b1) begin
43       counter_out <= #1 counter_out + 1;
44    end
45 end // End of Block COUNTER
46
47 endmodule // End of Module counter
```

Counter Test Bench

Any digital circuit, not matter how complex it is needs to be tested. For the counter logic, we need to provide clock, reset logic. Once counter is out of reset we toggle the enable input to counter, and check with waveform to see if counter is counting correctly. We do the same in Verilog.



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

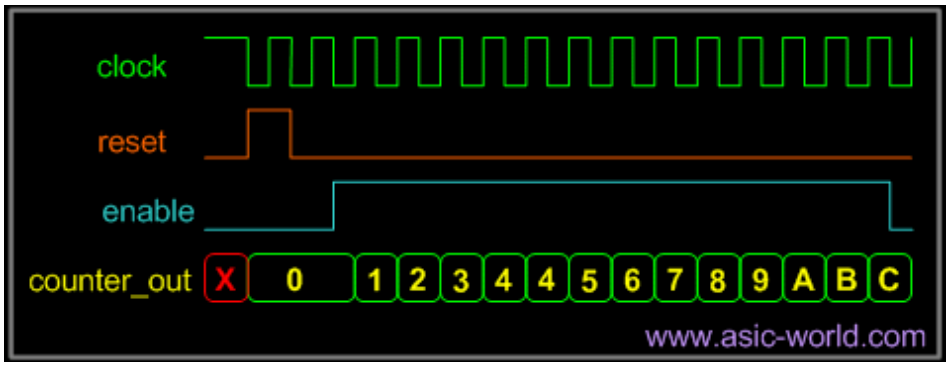
```
1 `include "first_counter.v"
2 module first_counter_tb();
3 // Declare inputs as regs and outputs as wires
4 reg clock, reset, enable;
5 wire [3:0] counter_out;
6
7 // Initialize all variables
8 initial begin
9    $display ( "time\t clk reset enable counter" );
10   $monitor ( "%g\t %b %b %b %b" ,
11   $time, clock, reset, enable, counter_out);
12   clock = 1; // initial value of clock
```

```
13    reset = 0; // initial value of reset
14    enable = 0; // initial value of enable
15    #5 reset = 1; // Assert the reset
16    #10 reset = 0; // De-assert the reset
17    #5 enable = 1; // Assert enable
18    #100 enable = 0; // De-assert enable
19    #10 $finish; // Terminate simulation
20end
21
22// Clock generator
23always begin
24    #5 clock = ~clock; // Toggle clock every 5 ticks
25end
26
27// Connect DUT to test bench
28first_counter U_counter (
29clock,
30reset,
31enable,
32counter_out
33);
34
35endmodule
```

```
time    clk reset enable counter
0       1   0     0      xxxx
5       0   1     0      xxxx
10      1   1     0      xxxx
11      1   1     0      0000
15      0   0     0      0000
20      1   0     1      0000
21      1   0     1      0001
25      0   0     1      0001
30      1   0     1      0001
31      1   0     1      0010
35      0   0     1      0010
40      1   0     1      0010
41      1   0     1      0011
45      0   0     1      0011
50      1   0     1      0011
51      1   0     1      0100
55      0   0     1      0100
60      1   0     1      0100
61      1   0     1      0101
65      0   0     1      0101
70      1   0     1      0101
71      1   0     1      0110
75      0   0     1      0110
80      1   0     1      0110
81      1   0     1      0111
85      0   0     1      0111
90      1   0     1      0111
91      1   0     1      1000
95      0   0     1      1000
100     1   0     1      1000
101     1   0     1      1001
105     0   0     1      1001
110     1   0     1      1001
111     1   0     1      1010
```

```
115   0  0   1    1010
120   1  0   0    1010
125   0  0   0    1010
```

Counter Waveform



NOTES

VERILOG HDL SYNTAX AND SEMANTICS

CHAPTER 5

Lexical Conventions

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

White Space

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

White space characters are :

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

Examples of White Spaces

Functional Equivalent Code

Bad Code : Never write code like this.

```
1module addbit(a,b,ci,sum,co);
2input a,b,ci;output sum co;
3wire a,b,ci,sum,co;endmodule
```

Good Code : Nice way to write code.

```
1module addbit (
2a,
3b,
4ci,
5sum,
6co);
7input a;
8input b;
9input ci;
10output sum;
11output co;
12wire a;
13wire b;
14wire ci;
15wire sum;
16wire co;
17
18endmodule
```

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 */

Some how I like single line comments.

Examples of Comments

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

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

endmodule
```

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 function or a module, a name so that it can be referenced from other places in a description.

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

Examples of legal identifiers

data_input mu

clk_input my$clk

i386 A

Escaped Identifiers

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

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

Examples of escape identifiers

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

```
1// There must be white space after the
2// string which uses escape character
3module \1dff (
4q, // Q output
5\q~ , // Q_out output
6d, // D input
7cl$k, // CLOCK input
8\reset* // Reset input
9);
```

```
10
11input d, cl$k, \reset* ;
12output q, \q~ ;
13
14endmodule
```

Numbers in Verilog

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

Integer Numbers

Verilog HDL allows integer numbers to be specified as

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

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

Example of Integer Numbers

Integer	Stored as
1	00000000000000000000000000000001
8'hAA	10101010
6'b10_0011	100011
'hF	00000000000000000000000000001111

Verilog expands to be fill the specified by working from right−to−left

- When is smaller than , then left−most bits of are truncated
- When is larger than , then left−most bits are filled, based on the value of the left−most bit in .
  - Left most '0' or '1' are filled with '0'
  - Left most 'Z' are filled with 'Z'
  - Left most 'X' are filled with 'X'

Example of Integer Numbers

Integer	Stored as
6'hCA	001010
6'hA	001010
16'bZ	ZZZZZZZZZZZZZZZZ
8'bx	xxxxxxxx

Real Numbers

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

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

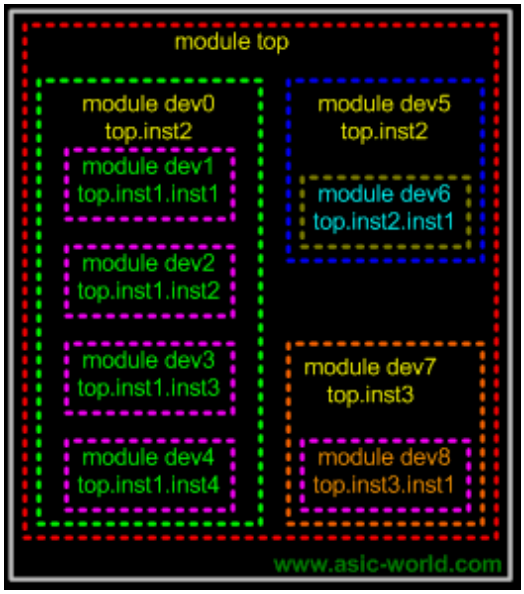
Below example file show how Verilog treats signed and unsigned numbers.

```
1  //****************************************
2  // Signed Number Example
3  //
4  // Written by Deepak Kumar Tala
5  //****************************************
6  module signed_number;
7
8  reg [31:0] a;
9
10 initial begin
11   a = 14'h1234;
12   $display ( "Current Value of a = %h" , a);
13   a = -14'h1234;
14   $display ( "Current Value of a = %h" , a);
15   a = 32'hDEAD_BEEF;
16   $display ( "Current Value of a = %h" , a);
17   a = -32'hDEAD_BEEF;
18   $display ( "Current Value of a = %h" , a);
19   #10 $finish;
20 end
21
22 endmodule
```

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

Modules

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



Ports

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

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

input [range_val:range_var] list_of_identifiers;

output [range_val:range_var] list_of_identifiers;

inout [range_val:range_var] list_of_identifiers;

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

Examples : Port Declaration

```
1input clk ; // clock input
2input [15:0] data_in ; // 16 bit data input bus
3output [7:0] count ; // 8 bit counter output
4inout 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 (locate the port which is causing compiler compile error), when any new port is added or deleted.

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

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

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

// Port Wires
```

```
27 wire [3:0] r1 ;
28 wire [3:0] r2 ;
29 wire ci ;
30 wire [3:0] result ;
31 wire carry ;
32
33 // Internal variables
34 wire c1 ;
35 wire c2 ;
36 wire c3 ;
37
38 // Code Starts Here
39 addbit u0 (
40 r1[0] ,
41 r2[0] ,
42 ci ,
43 result[0] ,
44 c1
45 );
46
47 addbit u1 (
48 r1[1] ,
49 r2[1] ,
50 c1 ,
51 result[1] ,
52 c2
53 );
54
55 addbit u2 (
56 r1[2] ,
57 r2[2] ,
58 c2 ,
59 result[2] ,
60 c3
61 );
62
63 addbit u3 (
64 r1[3] ,
65 r2[3] ,
66 c3 ,
67 result[3] ,
68 carry
69 );
70
71 endmodule // End Of Module adder
```

Modules connect by name

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

```
1 //-----------------------------------------------------
2 // This is simple adder Program
3 // Design Name : adder_implicit
4 // File Name : adder_implicit.v
5 // Function : This program shows how explicit
6 // port connection are done
7 // Coder : Deepak
8 //-----------------------------------------------------
9 module adder_explicit (
10 result , // Output of the adder
```

```
11 carry , // Carry output of adder
12 r1 , // first input
13 r2 , // second input
14 ci // carry input
15 );
16
17 // Input Port Declarations
18 input [3:0] r1 ;
19 input [3:0] r2 ;
20 input ci ;
21
22 // Output Port Declarations
23 output [3:0] result ;
24 output carry ;
25
26 // Port Wires
27 wire [3:0] r1 ;
28 wire [3:0] r2 ;
29 wire ci ;
30 wire [3:0] result ;
31 wire carry ;
32
33 // Internal variables
34 wire c1 ;
35 wire c2 ;
36 wire c3 ;
37
38 // Code Starts Here
39
40 // Code Starts Here
41
42 addbit u0 (
43 .a (r1[0]) ,
44 .b (r2[0]) ,
45 .ci (ci) ,
46 .sum (result[0]) ,
47 .co (c1)
48 );
49
50 addbit u1 (
51 .a (r1[1]) ,
52 .b (r2[1]) ,
53 .ci (c1) ,
54 .sum (result[1]) ,
55 .co (c2)
56 );
57
58 addbit u2 (
59 .a (r1[2]) ,
60 .b (r2[2]) ,
61 .ci (c2) ,
62 .sum (result[2]) ,
63 .co (c3)
64 );
65
66 addbit u3 (
67 .a (r1[3]) ,
68 .b (r2[3]) ,
69 .ci (c3) ,
70 .sum (result[3]) ,
71 .co (carry)
```

```
);

endmodule // End Of Module adder
```

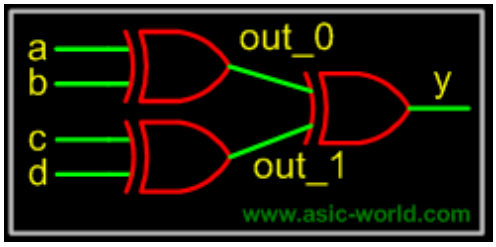
Instantiating a module

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

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

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

xor u1 (
out_1 ,
c ,
d
);

xor u2 (
y ,
out_0 ,
out_1
);
```

```
53
54endmodule // End Of Module parity
```

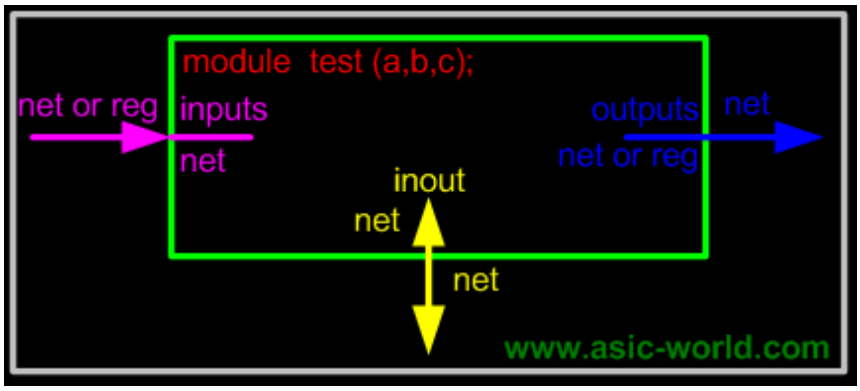
Question : What is difference between u0 in module adder and u0 in module parity?

Schematic



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
```

Hierarchical Identifiers

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

This is basically useful, while we want to see the signal inside a lower module or want to force a value on to internal module. Below example shows hows to monitor the value of internal module signal.

Example

```
1 //-------------------------------------------------------
2 // This is simple adder Program
3 // Design Name : adder_hier
4 // File Name : adder_hier.v
5 // Function : This program shows verilog hier path works
6 // Coder : Deepak
7 //-------------------------------------------------------
8 `include "addbit.v"
9 module adder_hier (
10 result , // Output of the adder
11 carry , // Carry output of adder
12 r1 , // first input
13 r2 , // second input
14 ci // carry input
15 );
16
17 // Input Port Declarations
18 input [3:0] r1 ;
19 input [3:0] r2 ;
20 input ci ;
21
22 // Output Port Declarations
23 output [3:0] result ;
24 output carry ;
```

```
25
26// Port Wires
27wire [3:0] r1 ;
28wire [3:0] r2 ;
29wire ci ;
30wire [3:0] result ;
31wire carry ;
32
33// Internal variables
34wire c1 ;
35wire c2 ;
36wire c3 ;
37
38// Code Starts Here
39addbit u0 (r1[0],r2[0],ci,result[0],c1);
40addbit u1 (r1[1],r2[1],c1,result[1],c2);
41addbit u2 (r1[2],r2[2],c2,result[2],c3);
42addbit u3 (r1[3],r2[3],c3,result[3],carry);
43
44endmodule // End Of Module adder
45
46module tb();
47
48reg [3:0] r1,r2;
49reg ci;
50wire [3:0] result;
51wire carry;
52
53// Drive the inputs
54initial begin
55    r1 = 0;
56    r2 = 0;
57    ci = 0;
58    #10 r1 = 10;
59    #10 r2 = 2;
60    #10 ci = 1;
61    #10 $display( "+-----------------------------------------------------------------+" );
62    $finish;
63end
64
65// Connect the lower module
66adder_hier U (result,carry,r1,r2,ci);
67
68// Hier demo here
69initial begin
70    $display( "+-----------------------------------------------------------------+" );
71    $display( "| r1 | r2 | ci | u0.sum | u1.sum | u2.sum | u3.sum |" );
72    $display( "+-----------------------------------------------------------------+" );
73    $monitor( "| %h | %h | %h | %h | %h | %h | %h |" ,
74    r1,r2,ci, tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum, tb.U.u3.sum);
75end
76
77endmodule
```

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

```
| a | 0 | 0 |  0  | 1 | 0 | 1 |
| a | 2 | 0 |  0  | 0 | 1 | 1 |
| a | 2 | 1 |  1  | 0 | 1 | 1 |
+-------------------------------------------------------+
```

Data Types

Verilog Language has two primary data types

- **Nets** – 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 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,suppy1	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

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

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

Example

```
1  //------------------------------------------------------
2  // Design Name : strings
3  // File Name : strings.v
4  // Function : This program shows how string
5  // can be stored in reg
6  // Coder  : Deepak Kumar Tala
7  //------------------------------------------------------
8  module strings();
9  // Declare a register variable that is 21 bytes
10 reg [8*21:0] string ;
11
12 initial begin
13    string = "This is sample string" ;
14    $display ( "%s \n" , string);
15 end
16
17 endmodule
```

```
This is sample string
```

NOTES

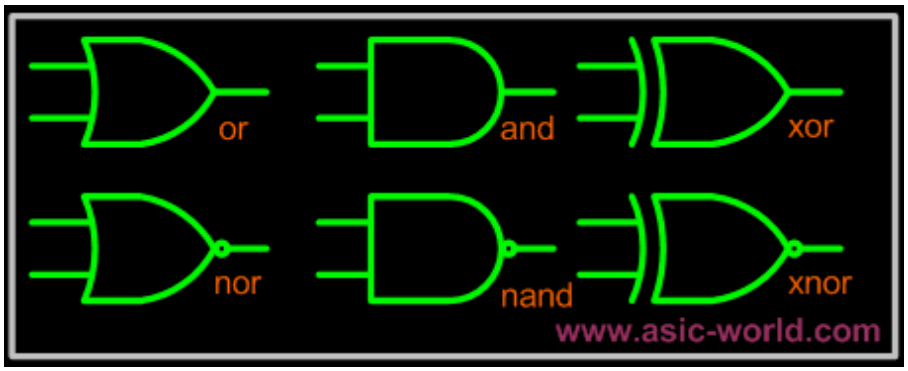
GATE LEVEL MODELING

CHAPTER 6

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. Also the output netlist formate from the synthesis tool which is imported into place and route tool is also in Verilog gate level primitives.

Gate Primitives



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

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

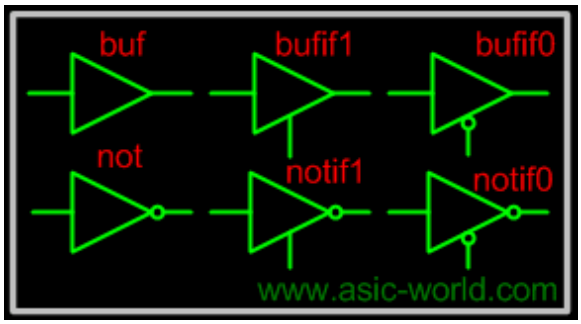
Examples

```
 1 module gates();
 2
 3 wire out0;
 4 wire out1;
 5 wire out2;
 6 reg in1,in2,in3,in4;
 7
 8 not U1(out0,in1);
 9 and U2(out1,in1,in2,in3,in4);
10 xor U3(out2,in1,in2,in3);
11
12 initial begin
```

```
13    $monitor( "in1 = %b in2 = %b in3 = %b in4 = %b out0 = %b out1 = %b out2 = %b"
      ,in1,in2,in3,in4,out0,out1,out2);
14    in1 = 0;
15    in2 = 0;
16    in3 = 0;
17    in4 = 0;
18    #1 in1 = 1;
19    #1 in2 = 1;
20    #1 in3 = 1;
21    #1 in4 = 1;
22    #1 $finish;
23  end
24
25  endmodule
```

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

Transmission Gate Primitives



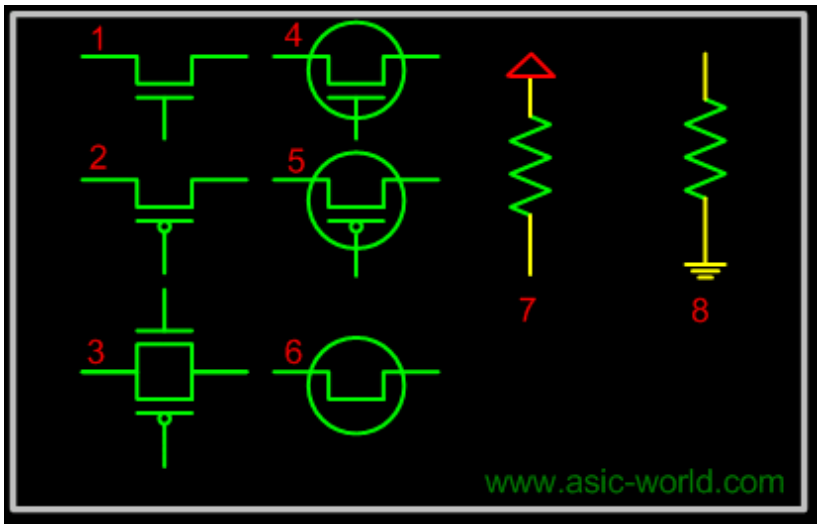
Gate	Description
not	N–output inverter
buf	N–output buffer.
bufif0	Tri–state buffer, Active low en.
bufif1	Tri–state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Examples

```
1  module transmission_gates();
2
3  reg data_enable_low, in;
4  wire data_bus, out1, out2;
5
6  bufif0 U1(data_bus,in, data_enable_low);
7  buf U2(out1,in);
8  not U3(out2,in);
9
10 initial begin
11    $monitor( "in = %b data_enable_low = %b out1 = %b out2 = %b" ,in,data_enable_low, out1, out2);
12    data_enable_low = 0;
13    in = 0;
14    #4 data_enable_low = 1;
15    #8 $finish;
16 end
17
18 always #2 in = ~in;
19
20 endmodule
```

```
in = 0 data_enable_low = 0 out1 = 0 out2 = 1
in = 1 data_enable_low = 0 out1 = 1 out2 = 0
in = 0 data_enable_low = 1 out1 = 0 out2 = 1
in = 1 data_enable_low = 1 out1 = 1 out2 = 0
in = 0 data_enable_low = 1 out1 = 0 out2 = 1
in = 1 data_enable_low = 1 out1 = 1 out2 = 0
```

Switch Primitives



Gate	Description
1. pmos	Uni−directional PMOS switch
1. rpmos	Resistive PMOS switch
2. nmos	Uni−directional NMOS switch
2. rnmos	Resistive NMOS switch
3. cmos	Uni−directional CMOS switch
3. rcmos	Resistive CMOS switch
4. tranif1	Bi−directional transistor (High)
4. tranif0	Bi−directional transistor (Low)
5. rtranif1	Resistive Transistor (High)
5. rtranif0	Resistive Transistor (Low)
6. tran	Bi−directional pass transistor
6. rtran	Resistive pass transistor
7. pullup	Pull up resistor
8. pulldown	Pull down resistor

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

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

Examples

```
1 module switch_primitives();
2
3 wire net1, net2, net3;
4 wire net4, net5, net6;
5
6 tranif0 my_gate1 (net1, net2, net3);
7 rtranif1 my_gate2 (net4, net5, net6);
8
9 endmodule
```

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

Logic Values and signal Strengths

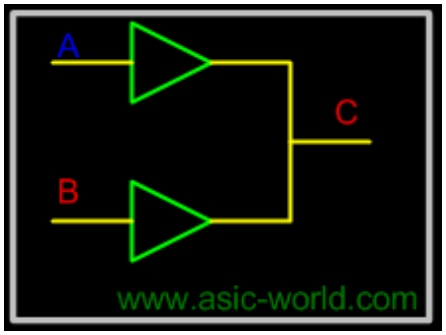
The Verilog HDL has got four logic values

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

Verilog Strength Levels

Strength Level	Specification Keyword
7 Supply Drive	supply0 supply1
6 Strong Pull	strong0 strong1
5 Pull Drive	pull0 pull1
4 Large Capacitance	large
3 Weak Drive	weak0 weak1
2 Medium Capacitance	medium
1 Small Capacitance	small
0 Hi Impedance	highz0 highz1

Example



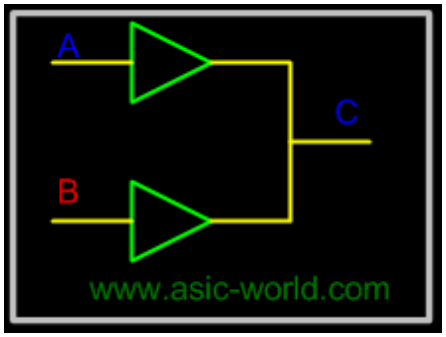
Two buffers that has output

A : Pull 1

B : Supply 0

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

Example



Two buffers that has output

A : Supply 1

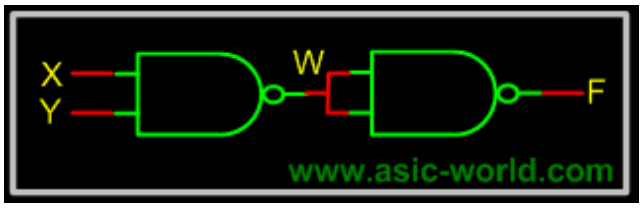
B : Large 1

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

Designing Using Primitives

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

AND Gate from NAND Gate



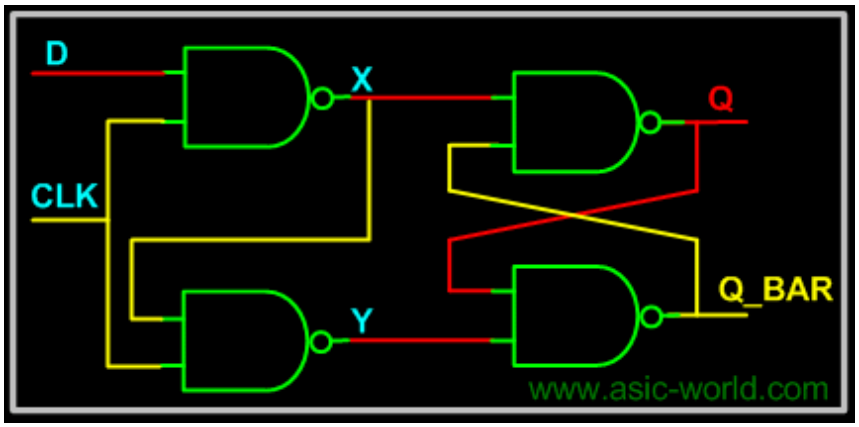
Code

```
1 // Structural model of AND gate from two NANDS
2 module and_from_nand();
3
4 reg X, Y;
5 wire F, W;
6 // Two instantiations of the module NAND
7 nand U1(W,X, Y);
8 nand U2(F, W, W);
9
10 // Testbench Code
11 initial begin
12    $monitor ( "X = %b Y = %b F = %b" , X, Y, F);
13    X = 0;
14    Y = 0;
15    #1 X = 1;
```

```
    #1 Y = 1;
    #1 X = 0;
    #1 $finish;
end

endmodule
```

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

D–Flip flop from NAND Gate

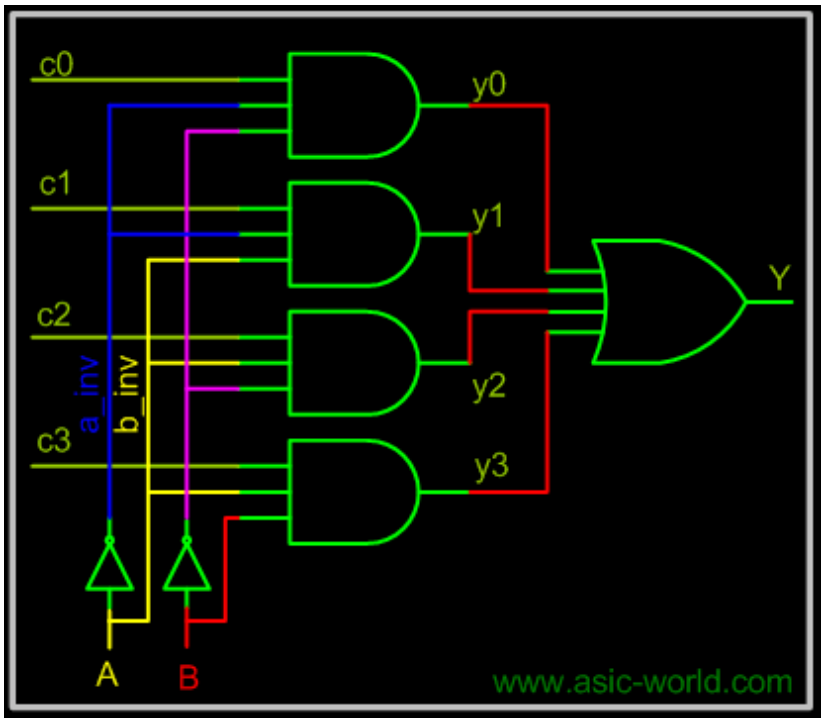


Verilog Code

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

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

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

always #2 CLK = ~CLK;

endmodule
```

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

Multiplexer from primitives



Verilog Code

```
1 module mux_from_gates ();
2 reg c0,c1,c2,c3,A,B;
3 wire Y;
4 //Invert the sel signals
5 not (a_inv, A);
6 not (b_inv, B);
7 // 3-input AND gate
8 and (y0,c0,a_inv,b_inv);
9 and (y1,c1,a_inv,B);
10 and (y2,c2,A,b_inv);
11 and (y3,c3,A,B);
12 // 4-input OR gate
13 or (Y, y0,y1,y2,y3);
14
15 // Testbench Code goes here
16 initial begin
17   $monitor ( "c0 = %b c1 = %b c2 = %b c3 = %b A = %b B = %b Y = %b" , c0, c1, c2, c3, A, B, Y);
18   c0 = 0;
```

```
19    c1 = 0;
20    c2 = 0;
21    c3 = 0;
22    A = 0;
23    B = 0;
24    #1 A  = 1;
25    #2 B  = 1;
26    #4 A  = 0;
27    #8 $finish;
28end
29
30always #1 c0 = ~c0;
31always #2 c1 = ~c1;
32always #3 c2 = ~c2;
33always #4 c3 = ~c3;
34
35endmodule
```

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

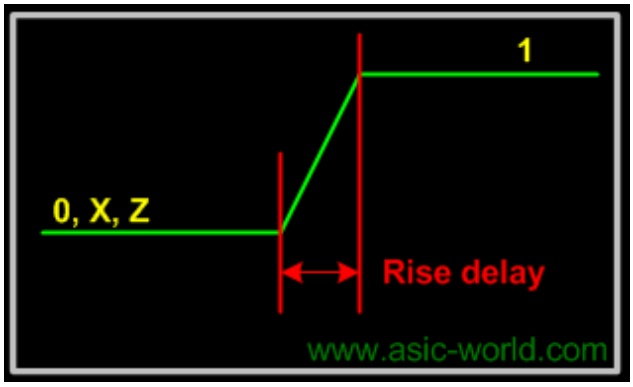
Gate and Switch delays

In real circuits , logic gates 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.

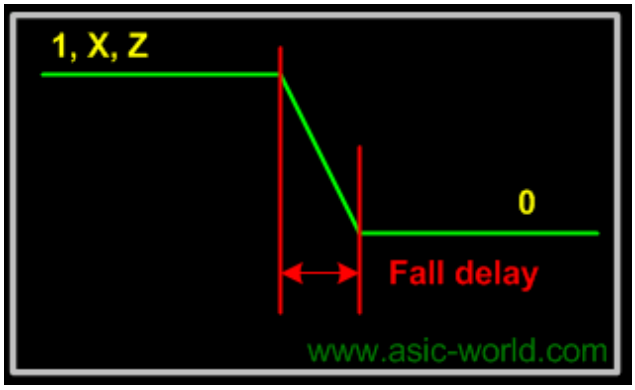
Rise Delay

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

Min Value

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

Typ Value

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

Max Value

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

Examples

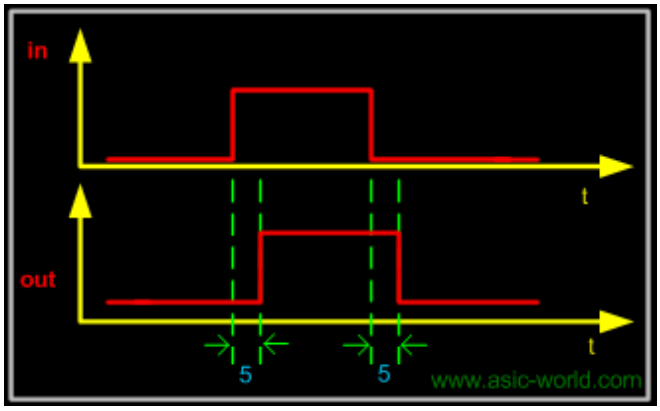
```
1  module delay_example();
2
3  wire out1,out2,out3,out4,out5,out6;
4  reg b,c;
5
6  // Delay for all transitions
7  or #5 u_or (out1,b,c);
8  // Rise and fall delay
9  and #(1,2) u_and (out2,b,c);
10 // Rise, fall and turn off delay
11 nor #(1,2,3) u_nor (out3,b,c);
12 //One Delay, min, typ and max
13 nand #(1:2:3) u_nand (out4,b,c);
14 //Two delays, min,typ and max
15 buf #(1:4:8,4:5:6) u_buf (out5,b);
16 //Three delays, min, typ, and max
17 notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (out6,b,c);
18
19 //Testbench code
20 initial begin
21    $monitor ( "Time = %g b = %b c=%b out1=%b out2=%b out3=%b out4=%b out5=%b out6=%b" , $time, b, c ,
      out1, out2, out3, out4, out5, out6);
22    b = 0;
23    c = 0;
24    #10 b = 1;
25    #10 c = 1;
26    #10 b = 0;
27    #10 $finish;
28 end
29
30 endmodule
```

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

Gate Delay Code Example

```
1  module buf_gate ();
2  reg in;
3  wire out;
4
5  buf #(5) (out,in);
6
7  initial begin
8    $monitor ( "Time = %g in = %b out=%b" , $time, in, out);
9    in = 0;
10   #10 in = 1;
11   #10 in = 0;
12   #10 $finish;
13 end
14
15 endmodule
```

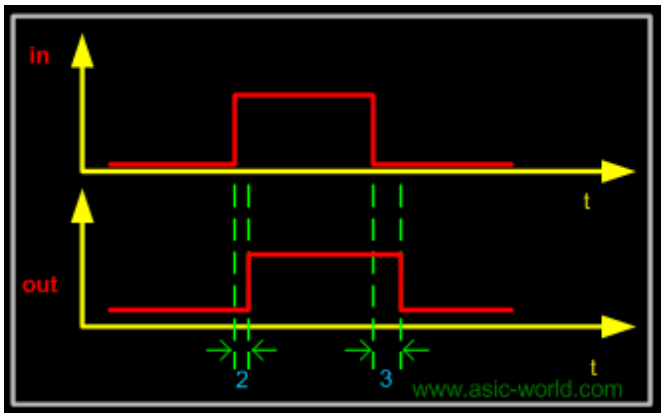
```
Time = 0 in = 0 out=x
Time = 5 in = 0 out=0
Time = 10 in = 1 out=0
Time = 15 in = 1 out=1
Time = 20 in = 0 out=1
Time = 25 in = 0 out=0
```



Gate Delay Code Example

```
1  module buf_gate1 ();
2  reg in;
3  wire out;
4
5  buf #(2,3) (out,in);
6
7  initial begin
8    $monitor ( "Time = %g in = %b out=%b" , $time, in, out);
9    in = 0;
10   #10 in = 1;
11   #10 in = 0;
12   #10 $finish;
13 end
14
15 endmodule
```

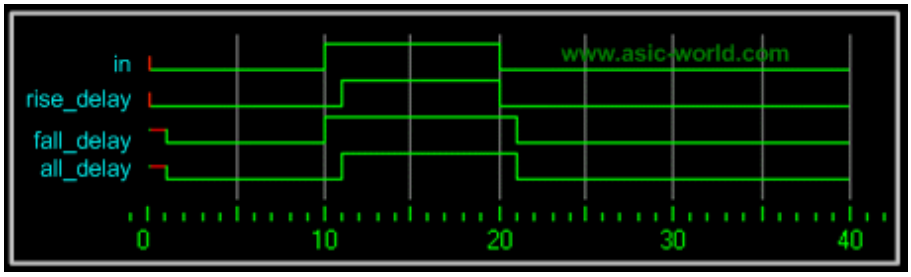
```
Time = 0 in = 0 out=x
Time = 3 in = 0 out=0
Time = 10 in = 1 out=0
Time = 12 in = 1 out=1
Time = 20 in = 0 out=1
Time = 23 in = 0 out=0
```



Gate Delay Code Example

```
1  module delay();
2  reg in;
3  wire rise_delay, fall_delay, all_delay;
4
5  initial begin
6      $monitor ( "Time = %g in = %b rise_delay = %b fall_delay = %b all_delay = %b" , $time, in, rise_delay, fall_delay,
       all_delay);
7      in = 0;
8      #10 in = 1;
9      #10 in = 0;
10     #20 $finish;
11 end
12
13 buf #(1,0)U_rise (rise_delay,in);
14 buf #(0,1)U_fall (fall_delay,in);
15 buf #1 U_all (all_delay,in);
16
17 endmodule
```

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



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

```
1 module n_in_primitive();
2
3 wire out1,out2,out3;
4 reg in1,in2,in3,in4;
5
6 // Two input AND gate
7 and u_and1 (out1, in1, in2);
8 // four input AND gate
9 and u_and2 (out2, in1, in2, in3, in4);
10 // three input XNOR gate
11 xnor u_xnor1 (out3, in1, in2, in3);
12
13 //Testbench Code
14 initial begin
15     $monitor ( "in1 = %b in2 = %b in3 = %b in4 = %b out1 = %b out2 = %b out3 = %b" , in1, in2, in3, in4, out1, out2, out3);
16     in1 = 0;
17     in2 = 0;
18     in3 = 0;
19     in4 = 0;
20     #1 in1 = 1;
21     #1 in2 = 1;
22     #1 in3 = 1;
23     #1 in4 = 1;
24     #1 $finish;
25 end
26
27 endmodule
```

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

N–Output Primitives

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

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

Examples

```
 1 module n_out_primitive();
 2
 3 wire out,out_0,out_1,out_2,out_3,out_a,out_b,out_c;
 4 wire in;
 5
 6 // one output Buffer gate
 7 buf u_buf0 (out,in);
 8 // four output Buffer gate
 9 buf u_buf1 (out_0, out_1, out_2, out_3, in);
10 // three output Invertor gate
11 not u_not0 (out_a, out_b, out_c, in);
12
13 endmodule
```

NOTES

USER DEFINED PRIMITIVES

CHAPTER 7

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. This is rather small number of primitives, if we need more complex primitives, then Verilog provides UDP, or simply User Defined Primitives. Using UDP we can model

- Combinational Logic
- Sequention Logic

We can include timing information along with this UDP to model complete ASIC library models.

Syntax

UDP begins with reserve word **primitive** and ends with **endprimitive**. This should follow by ports/terminals of primitive. This is kind of same as we do for module definition. UDP's should be defined outside **module** and **endmodule**

```
1 //This code shows how input/output ports
2 // and primitve is declared
3 primitive udp_syntax (
4 a, // Port a
5 b, // Port b
6 c, // Port c
7 d // Port d
8 );
9 output a;
10 input b,c,d;
11
12 // UDP function code here
13
14 endprimitive
```

In the above code, udp_syntax is the primitive name, it contains ports a, b,c,d.

The formal syntax of the UDP definition is as follows

```
<UDP>
  ::= primitive <name_of_UDP> ( <output_terminal_name>,
    <input_terminal_name> <,<input_terminal_name>>* ) ;
  <UDP_declaration>+
  <UDP_initial_statement>?
  <table_definition>
  endprimitive

<name_of_UDP>
  ::= <IDENTIFIER>

<UDP_declaration>
  ::= <UDP_output_declaration>
  ||= <reg_declaration>
  ||= <UDP_input_declaration>
```

```
<UDP_output_declaration>
  ::= output <output_terminal _name>;
<reg_declaration>
  ::=   reg <output_terminal_name> ;

<UDP_input_declaration>
  ::= input <input_terminal_name> <,<input_terminal_name>>* ;

<UDP_initial_statement>
  ::= initial <output_terminal_name> = <init_val> ;

<init_val>
  ::= 1'b0
  ||= 1'b1
  ||= 1'bx
  ||= 1
  ||= 0

<table_definition>
  ::= table
       <table_entries>
    endtable

<table_entries>
  ::= <combinational_entry>+
  ||= <sequential_entry>+

<combinational_entry>
  ::= <level_input_list> : <OUTPUT_SYMBOL> ;

<sequential_entry>
  ::= <input_list> : <state> : <next_state> ;

<input_list>
  ::= <level_input_list>
  ||= <edge_input_list>

<level_input_list>
  ::= <LEVEL_SYMBOL>+

<edge_input_list>
  ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*

<edge>
  ::= ( <LEVEL_SYMBOL> <LEVEL_SYMBOL> )
  ||= <EDGE_SYMBOL>

<state>
  ::= <LEVEL_SYMBOL>

<next_state>
  ::= <OUTPUT_SYMBOL>
  ||= –
```

UDP ports rules

- A UDP can contain only one output and up to 10 inputs max.
- Output Port should be the first port followed by one or more input ports.
- All UDP ports are scalar, i.e. Vector ports are not allowed.

- UDP's can not have bidirectional ports.
- The output terminal of a sequential UDP requires an additional declaration as type reg.
- It is illegal to declare a reg for the output terminal of a combinational UDP

Body

Functionality of primitive (both combinational and sequential) is described inside a table, and it ends with reserve word endtable as shown in code below. For sequential UDP, we can use initial to assign initial value to output.

```
1  // This code shows how UDP body looks like
2  primitive udp_body (
3  a, // Port a
4  b, // Port b
5  c // Port c
6  );
7  output a;
8  input b,c;
9
10 // UDP function code here
11 // A = B | C;
12 table
13                                    // B C : A
14                                    ? 1  : 1;
15                                    1  ?  : 1;
16                                    0 0  : 0;
17 endtable
18
19 endprimitive
```

Note: A UDP cannot use 'z' in input table

TestBench to Check above UDP

```
1  `include "udp_body.v"
2  module udp_body_tb();
3
4  reg b,c;
5  wire a;
6
7  udp_body udp (a,b,c);
8
9  initial begin
10   $monitor( " B = %b C = %b A = %b" ,b,c,a);
11   b = 0;
12   c = 0;
13   #1 b = 1;
14   #1 b = 0;
15   #1 c = 1;
16   #1 b = 1'bx;
17   #1 c = 0;
18   #1 b = 1;
19   #1 c = 1'bx;
```

```
20    #1 b = 0;
21    #1 $finish;
22end
23
24endmodule
```

Simulator Output

```
B = 0 C = 0  A = 0
B = 1 C = 0  A = 1
B = 0 C = 0  A = 0
B = 0 C = 1  A = 1
B = x C = 1  A = 1
B = x C = 0  A = x
B = 1 C = 0  A = 1
B = 1 C = x  A = 1
B = 0 C = x  A = x
```

Table

Table is used for describing the function of UDP. Verilog reserve world **table** marks the start of table and reserve word **endtable** marks the end of table.

Each line inside a table is one condition, as and when a input changes, the input condition is matched and the output is evaluated to reflect the new change in input.

initial

initial statement is used for initialization of sequential UDP's. This statement begins with the keyword initial. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal reg.

```
1primitive udp_initial (a,b,c);
2output a;
3input b,c;
4reg a;
5// a has value of 1 at start of sim
6initial a = 1'b1;
7
8table
9                                   // udp_initial behaviour
10endtable
11
12endprimitive
```

Symbols

UDP uses special symbols to describe functions, like rising edge, don't care so on. Below table shows the symbols that are used in UDP's

Symbol	Interpretation	Explanation
?	0 or 1 or X	? means the variable can be 0 or 1 or x
b	0 or 1	Same as ?, but x is not included
f	(10)	Falling edge on an input
r	(01)	Rising edge on an input
p	(01) or (0x) or (x1) or (1z) or (z1)	Rising edge including x and z
n	(10) or (1x) or (x0) or (0z) or (z0)	Falling edge including x and z
*	(??)	All transitions
–	no change	No Change

We will see them in detail in next few pages.

Combinational UDPs

In combinational UDPs, the output is determined as a function of the current input. Whenever an input changes value, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row. This is kind of same as condition statements, each line in table is one condition.

Combinational UDPs have one field per input and one field for the output. Input fields and output fields are separated with colon. Each row of the table is terminated by a semicolon. For example, the following state table entry specifies that when the three inputs are all 0, the output is 0.

```
1 primitive udp_combo (.....);
2
3 table
4                                   0 0 0 : 0;
5                                   ...
6 endtable
7
8 endprimitive
```

The order of the inputs in the state table description must correspond to the order of the inputs in the port list in the UDP definition header. It is not related to the order of the input declarations.

Each row in the table defines the output for a particular combination of input states. If all inputs are specified as x, then the output must be specified as x. All combinations that are not explicitly specified result in a default output state of x.

Example

In below example entry, the ? represents a don't–care condition. This symbol indicates iterative substitution of 1, 0, and x. The table entry specifies that when the inputs are 0 and 1, the output is

1 no matter what the value of the current state is.

You do not have to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of x.

It is illegal to have the same combination of inputs, specified for different outputs.

```
1 // This code shows how UDP body looks like
2 primitive udp_body (
3 a, // Port a
4 b, // Port b
5 c // Port c
6 );
7 output a;
8 input b,c;
9
10 // UDP function code here
11 // A = B | C;
12 table
13                                          // B C : A
14                                          ? 1  : 1;
15                                          1  ?  : 1;
16                                          0 0  : 0;
17 endtable
18
19 endprimitive
```

TestBench to Check above UDP

```
1  `include "udp_body.v"
2  module udp_body_tb();
3
4  reg b,c;
5  wire a;
6
7  udp_body udp (a,b,c);
8
9  initial begin
10    $monitor( " B = %b C = %b A = %b" ,b,c,a);
11    b = 0;
12    c = 0;
13    #1 b = 1;
14    #1 b = 0;
15    #1 c = 1;
16    #1 b = 1'bx;
17    #1 c = 0;
18    #1 b = 1;
19    #1 c = 1'bx;
20    #1 b = 0;
21    #1 $finish;
22 end
```

```
23
24endmodule
```

Simulator Output

```
B = 0 C = 0  A = 0
B = 1 C = 0  A = 1
B = 0 C = 0  A = 0
B = 0 C = 1  A = 1
B = x C = 1  A = 1
B = x C = 0  A = x
B = 1 C = 0  A = 1
B = 1 C = x  A = 1
B = 0 C = x  A = x
```

Level Senstive Sequential UDP

Level−sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type reg, and there is an additional field in each table entry. This new field represents the current state of the UDP.

- The output is declared as reg to indicate that there is an internal state. The output value of the UDP is always the same as the internal state.
- A field for the current state has been added. This field is separated by colons from the inputs and the output.

Sequential UDPs have an additional field inserted between the input fields and the output field, compared to combinational UDP. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons.

```
1primitive udp_seq (.....);
2
3table
4                                                    0 0 0 : 0 : 0;
5                                                    ...
6endtable
7
8endprimitive
```

Example

```
primitive udp_latch(q, clk, d) ;
output q;
input clk, d;

reg q;

table
                                    //clk d q q+
                                    0 1 : ? : 1 ;
                                    0 0 : ? : 0 ;
                                    1  ? : ? : – ;
endtable

endprimitive
```

Edge–Sensitive UDPs

In level–sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge–sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs.

As in the combinational and the level–sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (–) in the output column indicates no value change.

All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 result in the output q going to x.

All transitions that should not affect the output must be explicitly specified. Otherwise, they will cause the value of the output to change to x. If the UDP is sensitive to edges of any input, the desired output state must be specified for all edges of all inputs.

Example

```
primitive udp_sequential(q, clk, d);
output q;
input clk, d;

reg q;

table
                                    // obtain output on rising edge of clk
                                    // clk d q q+
                                    (01) 0  : ?  : 0 ;
                                    (01) 1  : ?  : 1 ;
                                    (0?) 1  : 1  : 1 ;
                                    (0?) 0  : 0  : 0 ;
                                    // ignore negative edge of clk
                                    (?0)  ? : ? : – ;
                                    // ignore d changes on steady clk
                                    ? (??) : ? : – ;
endtable
```

```
19
20 endprimitive
```

Example UDP with initial

```
 1 primitive udp_sequential_initial(q, clk, d);
 2 output q;
 3 input clk, d;
 4
 5 reg q;
 6
 7 initial begin
 8    q = 0;
 9 end
10
11 table
12    // obtain output on rising edge of clk
13    // clk d q q+
14    (01) 0  :  ?  : 0 ;
15    (01) 1  :  ?  : 1 ;
16    (0?) 1  : 1  : 1 ;
17    (0?) 0  : 0  : 0 ;
18    // ignore negative edge of clk
19    (?0)  ?  :  ?  :  – ;
20    // ignore d changes on steady clk
21    ? (??)  :  ?  :  – ;
22 endtable
23
24 endprimitive
```

NOTES

VERILOG OPERATORS

CHAPTER 8

Arithmetic Operators

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

Example

```
 1 module arithmetic_operators();
 2
 3 initial begin
 4    $display ( " 5 + 10 = %d" , 5 + 10);
 5    $display ( " 5 – 10 = %d" , 5 – 10);
 6    $display ( " 10 – 5 = %d" , 10 – 5);
 7    $display ( " 10 * 5 = %d" , 10 * 5);
 8    $display ( " 10 / 5 = %d" , 10 / 5);
 9    $display ( " 10 / –5 = %d" , 10 / –5);
10    $display ( " 10 %s 3 = %d" ,
11    $display ( " +5 = %d" , +5);
12    $display ( " –5 = %d" , –5);
13    #10 $finish;
14 end
15
16 endmodule
```

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

Relational Operators

Operator	Description
a	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 (0)

Example

```
 1 module relational_operators();
 2
 3 initial begin
 4    $display ( " 5 <= 10 = %b" , (5 <= 10));
 5    $display ( " 5 >= 10 = %b" , (5 >= 10));
 6    $display ( " 1'bx <= 10 = %b" , (1'bx <= 10));
 7    $display ( " 1'bz <= 10 = %b" , (1'bz <= 10));
 8    #10 $finish;
 9 end
10
11 endmodule
```

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

Equality Operators

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

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

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

Note : The result is always 0 or 1.

Example

```
1 module equality_operators();
2
3 initial begin
4   // Case Equality
5   $display ( " 4'bx001 === 4'bx001 = %b" , (4'bx001 === 4'bx001));
6   $display ( " 4'bx0x1 === 4'bx001 = %b" , (4'bx0x1 === 4'bx001));
7   $display ( " 4'bz0x1 === 4'bz0x1 = %b" , (4'bz0x1 === 4'bz0x1));
8   $display ( " 4'bz0x1 === 4'bz001 = %b" , (4'bz0x1 === 4'bz001));
9   // Case Inequality
10  $display ( " 4'bx0x1 !== 4'bx001 = %b" , (4'bx0x1 !== 4'bx001));
11  $display ( " 4'bz0x1 !== 4'bz001 = %b" , (4'bz0x1 !== 4'bz001));
12  // Logical Equality
13  $display ( " 5 == 10 = %b" , (5 == 10));
14  $display ( " 5 == 5 = %b" , (5 == 5));
15  // Logical Inequality
16  $display ( " 5 != 5 = %b" , (5 != 5));
17  $display ( " 5 != 6 = %b" , (5 != 6));
18  #10 $finish;
19 end
20
21 endmodule
```

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

Logical Operators

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

- Expressions connected by && and || are evaluated from left to right
- Evaluation stops as soon as the result is known
- The result is a scalar value:

- 0 if the relation is false
- 1 if the relation is true
- x if any of the operands has unknown x bits

Example

```
1 module logical_operators();
2
3 initial begin
4    // Logical AND
5    $display ( "1'b1 && 1'b1 = %b" , (1'b1 && 1'b1));
6    $display ( "1'b1 && 1'b0 = %b" , (1'b1 && 1'b0));
7    $display ( "1'b1 && 1'bx = %b" , (1'b1 && 1'bx));
8    // Logical OR
9    $display ( "1'b1 || 1'b0 = %b" , (1'b1 || 1'b0));
10   $display ( "1'b0 || 1'b0 = %b" , (1'b0 || 1'b0));
11   $display ( "1'b0 || 1'bx = %b" , (1'b0 || 1'bx));
12   // Logical Negation
13   $display ( "! 1'b1 = %b" , (! 1'b1));
14   $display ( "! 1'b0 = %b" , (! 1'b0));
15   #10 $finish;
16 end
17
18 endmodule
```

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

Bit–wise Operators

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

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

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

Example

```
 1module bitwise_operators();
 2
 3initial begin
 4   // Bit Wise Negation
 5   $display ( " ~4'b0001 = %b" , (~4'b0001));
 6   $display ( " ~4'bx001 = %b" , (~4'bx001));
 7   $display ( " ~4'bz001 = %b" , (~4'bz001));
 8   // Bit Wise AND
 9   $display ( " 4'b0001 & 4'b1001 = %b" , (4'b0001 & 4'b1001));
10   $display ( " 4'b1001 & 4'bx001 = %b" , (4'b1001 & 4'bx001));
11   $display ( " 4'b1001 & 4'bz001 = %b" , (4'b1001 & 4'bz001));
12   // Bit Wise OR
13   $display ( " 4'b0001 | 4'b1001 = %b" , (4'b0001 | 4'b1001));
14   $display ( " 4'b0001 | 4'bx001 = %b" , (4'b0001 | 4'bx001));
15   $display ( " 4'b0001 | 4'bz001 = %b" , (4'b0001 | 4'bz001));
16   // Bit Wise XOR
17   $display ( " 4'b0001 ^ 4'b1001 = %b" , (4'b0001 ^ 4'b1001));
18   $display ( " 4'b0001 ^ 4'bx001 = %b" , (4'b0001 ^ 4'bx001));
19   $display ( " 4'b0001 ^ 4'bz001 = %b" , (4'b0001 ^ 4'bz001));
20   // Bit Wise XNOR
21   $display ( " 4'b0001 ~^ 4'b1001 = %b" , (4'b0001 ~^ 4'b1001));
22   $display ( " 4'b0001 ~^ 4'bx001 = %b" , (4'b0001 ~^ 4'bx001));
23   $display ( " 4'b0001 ~^ 4'bz001 = %b" , (4'b0001 ~^ 4'bz001));
24   #10 $finish;
25end
26
27endmodule
```

```
~4'b0001          = 1110
~4'bx001          = x110
~4'bz001          = x110
4'b0001 &  4'b1001 = 0001
4'b1001 &  4'bx001 = x001
4'b1001 &  4'bz001 = x001
4'b0001 |  4'b1001 = 1001
4'b0001 |  4'bx001 = x001
4'b0001 |  4'bz001 = x001
4'b0001 ^  4'b1001 = 1000
4'b0001 ^  4'bx001 = x000
```

```
4'b0001 ^  4'bz001 = z000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111
```

Reduction Operators

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

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

Example

```
1  module reduction_operators();
2
3  initial begin
4    // Bit Wise AND reduction
5    $display ( " & 4'b1001 = %b" , (& 4'b1001));
6    $display ( " & 4'bx111 = %b" , (& 4'bx111));
7    $display ( " & 4'bz111 = %b" , (& 4'bz111));
8    // Bit Wise NAND reduction
9    $display ( " ~& 4'b1001 = %b" , (~& 4'b1001));
10   $display ( " ~& 4'bx001 = %b" , (~& 4'bx001));
11   $display ( " ~& 4'bz001 = %b" , (~& 4'bz001));
12   // Bit Wise OR reduction
13   $display ( " | 4'b1001 = %b" , (| 4'b1001));
14   $display ( " | 4'bx000 = %b" , (| 4'bx000));
15   $display ( " | 4'bz000 = %b" , (| 4'bz000));
16   // Bit Wise OR reduction
17   $display ( " ~| 4'b1001 = %b" , (~| 4'b1001));
18   $display ( " ~| 4'bx001 = %b" , (~| 4'bx001));
19   $display ( " ~| 4'bz001 = %b" , (~| 4'bz001));
20   // Bit Wise XOR reduction
```

```
21    $display ( " ^ 4'b1001 = %b" , (^ 4'b1001));
22    $display ( " ^ 4'bx001 = %b" , (^ 4'bx001));
23    $display ( " ^ 4'bz001 = %b" , (^ 4'bz001));
24    // Bit Wise XNOR
25    $display ( " ~^ 4'b1001 = %b" , (~^ 4'b1001));
26    $display ( " ~^ 4'bx001 = %b" , (~^ 4'bx001));
27    $display ( " ~^ 4'bz001 = %b" , (~^ 4'bz001));
28    #10 $finish;
29  end
30
31  endmodule
```

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

Shift Operators

Operator	Description
<<	left shift
>>	right shift

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

Example

```
1 module shift_operators();
2
3 initial begin
4   // Left Shift
5   $display ( " 4'b1001 << 1 = %b" , (4'b1001 << 1));
6   $display ( " 4'b10x1 << 1 = %b" , (4'b10x1 << 1));
7   $display ( " 4'b10z1 << 1 = %b" , (4'b10z1 << 1));
8   // Right Shift
9   $display ( " 4'b1001 >> 1 = %b" , (4'b1001 >> 1));
10  $display ( " 4'b10x1 >> 1 = %b" , (4'b10x1 >> 1));
11  $display ( " 4'b10z1 >> 1 = %b" , (4'b10z1 >> 1));
12  #10 $finish;
13 end
14
15 endmodule
```

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

Concatenation Operators

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

Example

```
1 module concatenation_operator();
2
3 initial begin
4   // concatenation
5   $display ( " {4'b1001,4'b10x1} = %b" , {4'b1001,4'b10x1});
6   #10 $finish;
7 end
8
9 endmodule
```

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

Replication Operator Operators

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

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

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

Example

```
1  module replication_operator();
2
3  initial begin
4    // replication
5    $display ( " {4{4'b1001} = %b" , {4{4'b1001}});
6    // replication and concatenation
7    $display ( " {4{4'b1001,1'bz} = %b" , {4{4'b1001,1'bz}});
8    #10 $finish;
9  end
10
11 endmodule
```

```
{4{4'b1001}       = 1001100110011001
{4{4'b1001,1'bz}  = 1001z1001z1001z1001z
```

Conditional Operators

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

Example

```
1  module conditional_operator();
2
3  wire out;
4  reg enable,data;
5  // Tri state buffer
6  assign out = (enable) ? data : 1'bz;
7
8  initial begin
9    $display ( "time\t enable data out" );
10   $monitor ( "%g\t %b %b %b" ,$time,enable,data,out);
```

```
11    enable = 0;
12    data = 0;
13    #1 data = 1;
14    #1 data = 0;
15    #1 enable = 1;
16    #1 data = 1;
17    #1 data = 0;
18    #1 enable = 0;
19    #10 $finish;
20  end
21
22  endmodule
```

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

Operator Precedence

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

NOTES

VERILOG BEHAVIORAL MODELING

CHAPTER 9

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

```
 1 module initial_example();
 2 reg clk,reset,enable,data;
 3
 4 initial begin
 5     clk = 0;
 6     reset = 0;
 7     enable = 0;
 8     data = 0;
 9 end
10
11 endmodule
```

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

Example – always

```
1  module always_example();
2  reg clk,reset,enable,q_in,data;
3
4  always @ (posedge clk)
5  if (reset) begin
6     data <= 0;
7  end else if (enable) begin
8     data <= q_in;
9  end
10
11 endmodule
```

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

Procedural Assignment Statements

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

Example – Bad procedural assignment

```
1  module initial_bad();
2  reg clk,reset;
3  wire enable,data;
4
5  initial begin
6     clk = 0;
7     reset = 0;
8     enable = 0;
9     data = 0;
10 end
11
12 endmodule
```

Example – Good procedural assignment

```
1  module initial_good();
2  reg clk,reset,enable,data;
3
4  initial begin
5     clk = 0;
6     reset = 0;
7     enable = 0;
8     data = 0;
9  end
```

```
10
11endmodule
```

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"

```
1module initial_begin_end();
2reg clk,reset,enable,data;
3
4initial begin
5    #1 clk = 0;
6    #10 reset = 0;
7    #5 enable = 0;
8    #3 data = 0;
9end
10
11endmodule
```

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

Example – "fork–join"

```
1module initial_fork_join();
2reg clk,reset,enable,data;
3
4initial fork
5    #1 clk = 0;
6    #10 reset = 0;
7    #5 enable = 0;
8    #3 data = 0;
9join
10
11endmodule
```

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

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.

Example – sequential

```
 1 module sequential();
 2
 3 reg a;
 4
 5 initial begin
 6    #10 a = 0;
 7    #11 a = 1;
 8    #12 a = 0;
 9    #13 a = 1;
10    #14 $finish;
11 end
12
13 endmodule
```

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).

Example – Parallel

```
 1 module parallel();
 2
 3 reg a;
 4
 5 initial
 6 fork
 7    #10 a = 0;
 8    #11 a = 1;
 9    #12 a = 0;
10    #13 a = 1;
11    #14 $finish;
```

```
12 join
13
14 endmodule
```

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

```
 1 module fork_join();
 2
 3 reg clk,reset,enable,data;
 4
 5 initial begin
 6    $display ( "Starting simulation" );
 7    fork : FORK_VAL
 8       #1 clk = 0;
 9       #5 reset = 0;
10       #5 enable = 0;
11       #2 data = 0;
12    join
13    $display ( "Terminating simulation" );
14    #10 $finish;
15 end
16
17 endmodule
```

Blocking and Nonblocking assignment

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

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

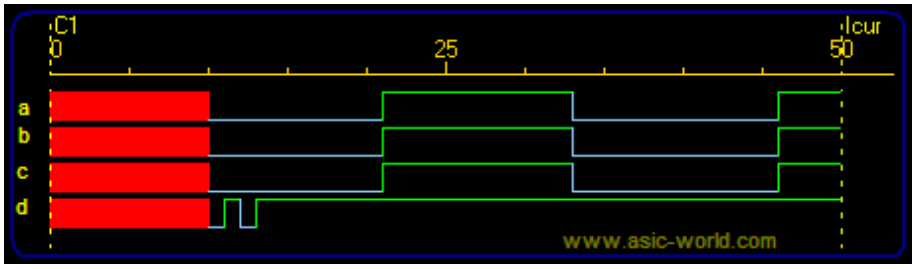
Note : Correct way to spell nonblocking is nonblocking and not non–blocking.

Example – blocking and nonblocking

```
 1 module blocking_nonblocking();
 2
 3 reg a,b,c,d;
 4 // Blocking Assignment
 5 initial begin
 6    #10 a = 0;
 7    #11 a = 1;
 8    #12 a = 0;
 9    #13 a = 1;
10 end
11
12 initial begin
```

```
13    #10 b <= 0;
14    #11 b <= 1;
15    #12 b <= 0;
16    #13 b <= 1;
17  end
18
19  initial begin
20    c = #10 0;
21    c = #11 1;
22    c = #12 0;
23    c = #13 1;
24  end
25
26  initial begin
27    d <= #10 0;
28    d <= #11 1;
29    d <= #12 0;
30    d <= #13 1;
31  end
32
33  initial begin
34    $monitor( " TIME = %t A = %b B = %b C = %b D = %b" ,$time,a,b,c,d);
35    #50 $finish(1);
36  end
37
38  endmodule
```

Waveform



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. When more then one statement needs to be executed for a if conditions, then we need to use begin and end as seen in earlier examples.

Syntax : if

if (condition)

statements;

Syntax : if–else

if (condition)
statements;
else
statements;

Syntax : nested if–else–if

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

Example– simple if

```
1 module simple_if();
2
3 reg latch;
4 wire enable,din;
5
6 always @ (enable or din)
7 if (enable) begin
8     latch <= din;
9 end
10
11 endmodule
```

Example– if–else

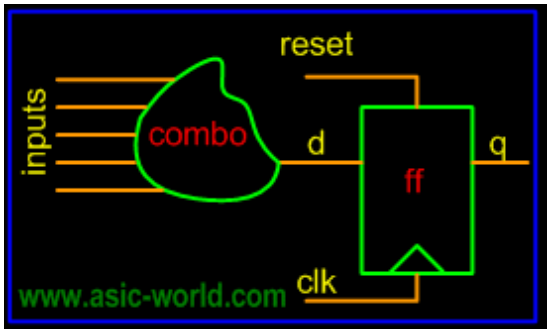
```
1 module if_else();
2
3 reg dff;
4 wire clk,din,reset;
5
6 always @ (posedge clk)
7 if (reset) begin
8     dff <= 0;
9 end else begin
10     dff <= din;
11 end
12
13 endmodule
```

Example– nested–if–else–if

```
1 module nested_if();
2
3 reg [3:0] counter;
4 wire clk,reset,enable, up_en, down_en;
5
6 always @ (posedge clk)
7 // If reset is asserted
8 if (reset == 1'b0) begin
9    counter <= 4'b0000;
10   // If counter is enable and up count is mode
11 end else if (enable == 1'b1 && up_en == 1'b1) begin
12   counter <= counter + 1'b1;
13   // If counter is enable and down count is mode
14 end else if (enable == 1'b1 && down_en == 1'b1) begin
15   counter <= counter - 1'b0;
16   // If counting is disabled
17 end else begin
18   counter <= counter; // Redundant code
19 end
20
21 endmodule
```

Parallel if–else

In the above example, the (enable == 1'b1 && up_en == 1'b1) is given highest pritority and condition (enable == 1'b1 && down_en == 1'b1) is given lowest priority. We normally don't include reset checking in priority as this does not falls in the combo logic input to the flip–flop as shown in figure below.



So when we need priority logic, we use nexted if–else statments. On other end if we don't want to implement priority logic, knowing that only one input is active at a time i.e. all inputs are mutually exclusive, then we can write the code as shown below.

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

```
 1 module parallel_if();
 2
 3 reg [3:0] counter;
 4 wire clk,reset,enable, up_en, down_en;
 5
 6 always @ (posedge clk)
 7 // If reset is asserted
 8 if (reset == 1'b0) begin
 9    counter <= 4'b0000;
10
11 end else begin
12    // If counter is enable and up count is mode
13    if (enable == 1'b1 && up_en == 1'b1) begin
14       counter <= counter + 1'b1;
15    end
16    // If counter is enable and down count is mode
17    if (enable == 1'b1 && down_en == 1'b1) begin
18       counter <= counter - 1'b0;
19    end
20 end
21
22 endmodule
```

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.

Synax of a case statement look as shown below.

case ()

< case1 > : < statement >

< case2 > : < statement >

.....

default : < statement >

endcase

Normal Case

Example– case

```
1  module mux (a,b,c,d,sel,y);
2  input a, b, c, d;
3  input [1:0] sel;
4  output y;
5
6  reg y;
7
8  always @ (a or b or c or d or sel)
9  case (sel)
10    0 : y = a;
11    1 : y = b;
12    2 : y = c;
13    3 : y = d;
14    default : $display( "Error in SEL" );
15 endcase
16
17 endmodule
```

Example– case without default

```
1  module mux_without_default (a,b,c,d,sel,y);
2  input a, b, c, d;
3  input [1:0] sel;
4  output y;
5
6  reg y;
7
8  always @ (a or b or c or d or sel)
9  case (sel)
10    0 : y = a;
11    1 : y = b;
12    2 : y = c;
13    3 : y = d;
14    2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
15    2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display( "Error in SEL" );
16 endcase
17
18 endmodule
```

Above example shows how to specifiy multiple case items as single case item.

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

Example– case with x and z

```
1 module case_xz(enable);
2 input enable;
3
4 always @ (enable)
5 case(enable)
6    1'bz : $display ( "enable is floating" );
7    1'bx : $display ( "enable is unknown" );
8    default : $display ( "enable is %b" ,enable);
9 endcase
10
11 endmodule
```

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 : Treates z as the don't care.
- casex : Treates x and z as don't care.

Example– casez

```
1  module casez_example(opcode,a,b,c,out);
2  input [3:0] opcode;
3  input [1:0] a,b,c;
4  output[1:0] out;
5  reg [1:0] out;
6
7  always @ (opcode or a or b or c)
8  casez(opcode)
9     // Bit 0 is matched with "x"
10    4'b1zzx : out = a; // Don't care about lower 3:1 bits
11    4'b01?? : out = b; // The ? is same as z in a number
12    4'b001? : out = c;
13    default : $display ( "Error xxxx does matches 0000" );
14 endcase
15
16 endmodule
```

Example– casex

```
1  module casex_example(opcode,a,b,c,out);
2  input [3:0] opcode;
3  input [1:0] a,b,c;
4  output[1:0] out;
5  reg [1:0] out;
6
7  always @ (opcode or a or b or c)
8  casex(opcode)
9     4'b1zzx : out = a; // Don't care 3:0 bits
10    4'b01?? : out = b; // The ? is same as z in a number
11    4'b001? : out = c;
```

```
12    default : $display ( "Error xxxx does matches 0000" );
13endcase
14
15endmodule
```

Example– Comparing case, casex, casez

```
 1module case_compare(sel);
 2
 3input sel;
 4
 5always @ (sel)
 6case (sel)
 7    1'b0 : $display( "Normal : Logic 0 on sel" );
 8    1'b1 : $display( "Normal : Logic 1 on sel" );
 9    1'bx : $display( "Normal : Logic x on sel" );
10    1'bz : $display( "Normal : Logic z on sel" );
11endcase
12
13always @ (sel)
14casex (sel)
15    1'b0 : $display( "CASEX : Logic 0 on sel" );
16    1'b1 : $display( "CASEX : Logic 1 on sel" );
17    1'bx : $display( "CASEX : Logic x on sel" );
18    1'bz : $display( "CASEX : Logic z on sel" );
19endcase
20
21always @ (sel)
22casez (sel)
23    1'b0 : $display( "CASEZ : Logic 0 on sel" );
24    1'b1 : $display( "CASEZ : Logic 1 on sel" );
25    1'bx : $display( "CASEZ : Logic x on sel" );
26    1'bz : $display( "CASEZ : Logic z on sel" );
27endcase
28
29endmodule
```

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. Normally we use forever statement in initial blocks.

syntax : forever < statement >

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

Example – Free running clock generator

```
1 module forever_example ();
2
3 reg clk;
4
5 initial begin
6    #1 clk = 0;
7    forever begin
8       #5 clk = !clk;
9    end
10 end
11
12 initial begin
13    $monitor ( "Time = %d clk = %b" ,$time, clk);
14    #100 $finish;
15 end
16
17 endmodule
```

The repeat statement

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

syntax : repeat (< number >) < statement >

Example– repeat

```
1  module repeat_example();
2  reg [3:0] opcode;
3  reg [15:0] data;
4  reg temp;
5
6  always @ (opcode or data)
7  begin
8     if (opcode == 10) begin
9        // Perform rotate
10       repeat (8) begin
11          #1 temp = data[15];
12          data = data << 1;
13          data[0] = temp;
```

```
    end
  end
end
// Simple test code
initial begin
  $display ( " TEMP DATA" );
  $monitor ( " %b %b " ,temp, data);
  #1 data = 18'hF0;
  #1 opcode = 10;
  #10 opcode = 0;
  #1 $finish;
end

endmodule
```

The while loop statement

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

syntax : while ()

Example– while

```
module while_example();

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

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

initial begin
  #1 data = 8'b11;
  #1 data = 8'b100;
  #1 data = 8'b1000;
  #1 data = 8'b1000_0000;
  #1 data = 8'b0;
```

```
27    #1 $finish;
28end
29
30endmodule
```

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 at the end of each pass through the loop.

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

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

Example– while

```
1module for_example();
2
3integer i;
4reg [7:0] ram [0:255];
5
6initial begin
7    for (i=0;i<=63;i=i+1) begin
8        #1 $display( " Address = %d Data = %h" ,i,ram[i]);
9        ram[i] <= 0; // Initialize the RAM with 0
10       #1 $display( " Address = %d Data = %h" ,i,ram[i]);
11   end
12   #1 $finish;
13end
14
15endmodule
```

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 any procedural assignments.
- The left–hand side of a continuous assignment must be net data type.

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

Example – One bit Adder

```
1 module adder_using_assign ();
2 reg a, b;
3 wire sum, carry;
4
5 assign #5 {carry,sum} = a+b;
6
7 initial begin
8    $monitor ( " A = %b B = %b CARRY = %b SUM = %b" ,a,b,carry,sum);
9    #10 a = 0;
10   b = 0;
11   #10 a = 1;
12   #10 b = 1;
13   #10 a = 0;
14   #10 b = 0;
15   #10 $finish;
16 end
17
18 endmodule
```

Example – Tri–state buffer

```
1 module tri_buf_using_assign();
2 reg data_in, enable;
3 wire pad;
4
5 assign pad = (enable) ? data_in : 1'bz;
6
7 initial begin
8    $monitor ( "ENABLE = %b DATA : %b PAD %b" ,enable, data_in,pad);
9    #1 enable = 0;
10   #1 data_in = 1;
11   #1 enable = 1;
12   #1 data_in = 0;
13   #1 enable = 0;
14   #1 $finish;
15 end
16
17 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

```
 1 module tri_buf_using_assign_delays();
 2 reg data_in, enable;
 3 wire pad;
 4
 5 assign #(1:2:3) pad = (enable) ? data_in : 1'bz;
 6
 7 initial begin
 8    $monitor ( "ENABLE = %b DATA : %b PAD %b" ,enable, data_in,pad);
 9    #10 enable = 0;
10    #10 data_in = 1;
11    #10 enable = 1;
12    #10 data_in = 0;
13    #10 enable = 0;
14    #10 $finish;
15 end
16
17 endmodule
```

Procedural Block Control

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

```
 1 module dlatch_using_always();
 2 reg q;
 3
 4 reg d, enable;
 5
 6 always @ (d or enable)
 7 if (enable) begin
 8    q = d;
 9 end
10
11 initial begin
12    $monitor ( " ENABLE = %b D = %b Q = %b" ,enable,d,q);
13    #1 enable = 0;
14    #1 d = 1;
15    #1 enable = 1;
16    #1 d = 0;
17    #1 d = 1;
18    #1 d = 0;
19    #1 enable = 0;
20    #10 $finish;
21 end
22
23 endmodule
```

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. There is one important rule that needs to be followed while modelling combinational logic. If you use conditional checking using "if", then you need to mention the "else" part. Missing the else part results in latch. If you don't like typing the else part, then you must initilize all the variables of that combo block to zero as soon as it enters.

Example – One bit Adder

```
1 module adder_using_always ();
2 reg a, b;
3 reg sum, carry;
4
5 always @ (a or b)
6 begin
7    {carry,sum} = a + b;
8 end
9
10 initial begin
11    $monitor ( " A = %b B = %b CARRY = %b SUM = %b" ,a,b,carry,sum);
12    #10 a = 0;
13    b = 0;
14    #10 a = 1;
15    #10 b = 1;
16    #10 a = 0;
17    #10 b = 0;
18    #10 $finish;
19 end
20
21 endmodule
```

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

Example – 4–bit Adder

```
1 module adder_4_bit_using_always ();
2 reg[3:0] a, b;
3 reg [3:0] sum;
4 reg carry;
5
6 always @ (a or b)
7 begin
8    {carry,sum} = a + b;
9 end
10
11 initial begin
12    $monitor ( " A = %b B = %b CARRY = %b SUM = %b" ,a,b,carry,sum);
13    #10 a = 8;
14    b = 7;
15    #10 a = 10;
16    #10 b = 15;
```

```
17    #10 a = 0;
18    #10 b = 0;
19    #10 $finish;
20end
21
22endmodule
```

Example – Ways to avoid Latches – Cover all conditions

```
 1module avoid_latch_else ();
 2
 3reg q;
 4reg enable, d;
 5
 6always @ (enable or d)
 7if (enable) begin
 8    q = d;
 9end else begin
10    q = 0;
11end
12
13initial begin
14    $monitor ( " ENABLE = %b D = %b Q = %b" ,enable,d,q);
15    #1 enable = 0;
16    #1 d = 0;
17    #1 enable = 1;
18    #1 d = 1;
19    #1 d = 0;
20    #1 d = 1;
21    #1 d = 0;
22    #1 d = 1;
23    #1 enable = 0;
24    #1 $finish;
25end
26
27endmodule
```

Example – Ways to avoid Latches – Init the variables to zero

```
 1module avoid_latch_init ();
 2
 3reg q;
 4reg enable, d;
 5
 6always @ (enable or d)
 7begin
 8    q = 0;
 9    if (enable) begin
10        q = d;
11    end
12end
13
14initial begin
```

```
15    $monitor ( " ENABLE = %b D = %b Q = %b" ,enable,d,q);
16    #1 enable = 0;
17    #1 d = 0;
18    #1 enable = 1;
19    #1 d = 1;
20    #1 d = 0;
21    #1 d = 1;
22    #1 d = 0;
23    #1 d = 1;
24    #1 enable = 0;
25    #1 $finish;
26  end
27
28  endmodule
```

Sequential Logic using Procedural Coding

To model sequential logic, a procedure block must be sensitive to positive edge or negative edge of clock. To model asychronous reset, procedure block must be sensitive to both clock and reset. All the assigments to sequential logic should be made throught nonblocking assignement.

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

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

Example – Bad coding – Using two clocks

```
1  module wrong_seq();
2
3  reg q;
4  reg clk1, clk2, d1, d2;
5
6  always @ (posedge clk1 or posedge clk2)
7  if (clk1) begin
8    q <= d1;
9  end else if (clk2) begin
10   q <= d2;
11 end
12
13 initial begin
14   $monitor ( "CLK1 = %b CLK2 = %b D1 = %b D2 %b Q = %b" , clk1, clk2, d1, d2, q);
15   clk1 = 0;
16   clk2 = 0;
17   d1 = 0;
18   d2 = 1;
19   #10 $finish;
20 end
21
```

```
22always
23#1 clk1 = ~clk1;
24
25always
26#1.9 clk2 = ~clk2;
27
28endmodule
```

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

```
1module dff_async_reset_async_preset();
2
3reg clk,reset,preset,d;
4reg q;
5
6always @ (posedge clk or posedge reset or posedge preset)
7if (reset) begin
8   q <= 0;
9end else if (preset) begin
10   q <= 1;
11end else begin
12   q <= d;
13end
14
15// Testbench code here
16initial begin
17   $monitor( "CLK = %b RESET = %b PRESET = %b D = %b Q = %b" ,clk,reset,preset,d,q);
18   clk  = 0;
19   #1 reset  = 0;
20   preset = 0;
21   d  = 0;
22   #1 reset = 1;
23   #2 reset = 0;
24   #2 preset = 1;
25   #2 preset = 0;
26   repeat (4) begin
27      #2 d  = ~d;
28   end
29   #2 $finish;
30end
31
32always
33#1 clk = ~clk;
34
35endmodule
```

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

```
module dff_sync_reset_sync_preset();

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

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

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

always
#1 clk = ~clk;

endmodule
```

A procedure can't trigger itself

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

```
module trigger_itself();

reg clk;

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

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

```

```
endmodule
```

Procedural Block Concurrency

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

```
module multiple_blocks ();
reg a,b;
reg c,d;
reg clk,reset;
// Combo Logic
always @ ( c)
begin
   a = c;
end
// Seq Logic
always @ (posedge clk)
if (reset) begin
   b <= 0;
end else begin
   b <= a & d;
end

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

endmodule
```

Race condition

```
1 module race_condition();
2 reg b;
3
4 initial begin
5    b = 0;
6 end
7
8 initial begin
9    b = 1;
10 end
11
12 endmodule
```

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. named block can be disabled using disable statement.

Example – Named Blocks

```
1  // This code find the lowest bit set
2  module named_block_disable();
3
4  reg [31:0] bit_detect;
5  reg [5:0] bit_position;
6  integer i;
7
8  always @ (bit_detect)
9  begin : BIT_DETECT
10    for (i = 0; i < 32 ; i = i + 1) begin
11       // If bit is set, latch the bit position
12       // Disable the execution of the block
13       if (bit_detect[i] == 1) begin
14          bit_position = i;
15          disable BIT_DETECT;
16       end else begin
17          bit_position = 32;
18       end
19    end
20 end
21
22 // Testbench code here
23 initial begin
24    $monitor( " INPUT = %b MIN_POSITION = %d" , bit_detect, bit_position);
25    #1 bit_detect = 32'h1000_1000;
26    #1 bit_detect = 32'h1100_0000;
27    #1 bit_detect = 32'h1000_1010;
28    #10 $finish;
29 end
```

```
30
31endmodule
```

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

NOTES

PROCEDURAL TIMING CONTROL

CHAPTER 10

Procedural blocks and timing controls.

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

Delay Controls

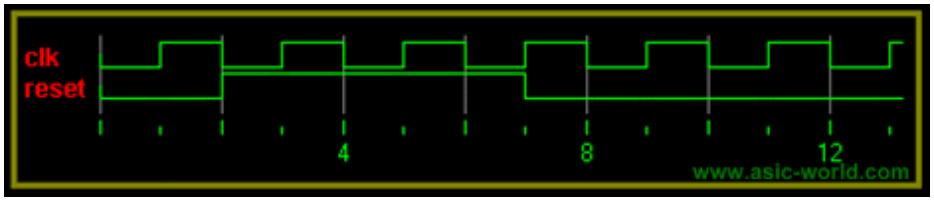
Delays the execution of a procedural statement by specific simulation time.

#< time > < statement >;

Example – clk_gen

```
1 module clk_gen ();
2
3 reg clk, reset;
4
5 initial begin
6    $monitor ( " RESET = %b CLOCK = %b" ,reset,clk);
7    clk = 0;
8    reset = 0;
9    #2 reset = 1;
10   #5 reset = 0;
11   #10 $finish;
12 end
13
14 always
15 #1 clk = !clk;
16
17 endmodule
```

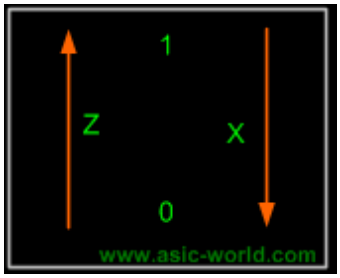
Waveform



Edge sensitive Event Controls

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

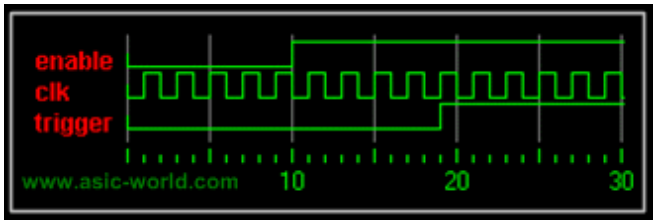
syntax : @ (< posedge >|< negedge > signal) < statement >;



Example – Edge Wait

```
1 module edge_wait_example();
2
3 reg enable, clk, trigger;
4
5 always @ (posedge enable)
6 begin
7    trigger = 0;
8    // Wait for 5 clock cycles
9    repeat (5) begin
10      @ (posedge clk) ;
11   end
12   trigger = 1;
13 end
14
15 //Testbench code here
16 initial begin
17   $monitor ( "TIME : %d CLK : %b ENABLE : %b TRIGGER : %b" ,$time, clk,enable,trigger);
18   clk = 0;
19   enable = 0;
20   #5 enable = 1;
21   #1 enable = 0;
22   #10 enable = 1;
23   #1 enable = 0;
24   #10 $finish;
25 end
26
27 always
28 #1 clk = ~clk;
29
30 endmodule
```

Waveform



Level–Sensitive Even Controls (Wait statements)

Delays execution of the next statement until the evaluates as true

syntax : wait () ;

Example – Level Wait

```
1  module wait_example();
2
3  reg mem_read, data_ready;
4  reg [7:0] data_bus, data;
5
6  always @ (mem_read or data_bus or data_ready)
7  begin
8     data = 0;
9     while (mem_read == 1'b1) begin
10       // #1 is very important to avoid infinite loop
11       wait (data_ready == 1) #1 data = data_bus;
12    end
13 end
14
15 // Testbench Code here
16 initial begin
17    $monitor ( "%d READ = %b READY = %b DATA = %b" , $time, mem_read, data_ready, data);
18    data_bus = 0;
19    mem_read = 0;
20    data_ready = 0;
21    #10 data_bus = 8'hDE;
22    #10 mem_read = 1;
23    #20 data_ready = 1;
24    #1 mem_read = 1;
25    #1 data_ready = 0;
26    #10 data_bus = 8'hAD;
27    #10 mem_read = 1;
28    #20 data_ready = 1;
29    #1 mem_read = 1;
30    #1 data_ready = 0;
31    #10 $finish;
32 end
33
34 endmodule
```

Intra–Assignment Timing Controls

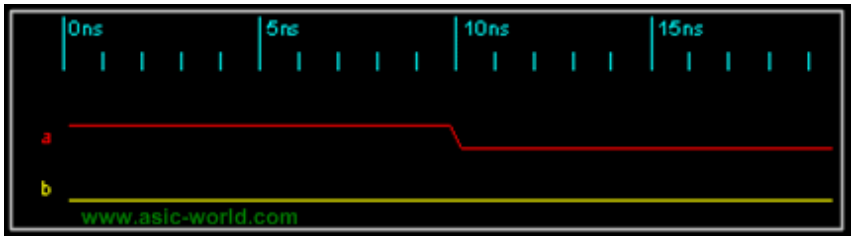
Intra–assignment controls evaluate the right side expression right always and assigns the result after the delay or event control.

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

Example – Intra–Assignment

```
1  module intra_assign();
2
3  reg a, b;
4
5  initial begin
6    $monitor( "TIME = %d A = %b B = %b" ,$time, a , b);
7    a = 1;
8    b = 0;
9    a = #10 0;
10   b = a;
11   #20 $display( "TIME = %d A = %b B = %b" ,$time, a , b);
12   $finish;
13 end
14
15 endmodule
```

Waveform



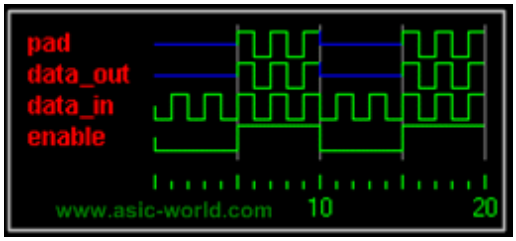
Modeling Combo Logic with Continuous Assignments

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

Example – Tri–state Buffer

```
1  module tri_buf_using_assign();
2  reg data_in, enable;
3  wire pad;
4
5  assign pad = (enable) ? data_in : 1'bz;
6
7  initial begin
8    $monitor ( "ENABLE = %b DATA : %b PAD %b" ,enable, data_in,pad);
9    #1 enable = 0;
10   #1 data_in = 1;
11   #1 enable = 1;
12   #1 data_in = 0;
13   #1 enable = 0;
14   #1 $finish;
15 end
16
17 endmodule
```

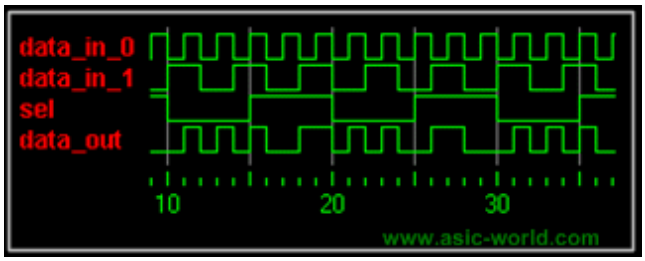
Waveform



Example – Mux

```
 1 module mux_using_assign();
 2 reg data_in_0, data_in_1;
 3 wire data_out;
 4 reg sel;
 5
 6 assign data_out = (sel) ? data_in_1 : data_in_0;
 7
 8 // Testbench code here
 9 initial begin
10    $monitor( "TIME = %d SEL = %b DATA0 = %b DATA1 = %b OUT = %b"
      ,$time,sel,data_in_0,data_in_1,data_out);
11    data_in_0 = 0;
12    data_in_1 = 0;
13    sel = 0;
14    #10 sel = 1;
15    #10 $finish;
16 end
17
18 // Toggel data_in_0 at #1
19 always
20 #1 data_in_0 = ~data_in_0;
21
22 // Toggel data_in_1 at %1.5
23 always
24 #1.3 data_in_1 = ~data_in_1;
25
26 endmodule
```

Waveform



NOTES

TASK AND FUNCTIONS

CHAPTER 11

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 and wait.
- 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

```
 1 module simple_task();
 2
 3 task convert;
 4 input [7:0] temp_in;
 5 output [7:0] temp_out;
 6 begin
 7    temp_out = (9/5) *( temp_in + 32)
 8 end
 9 endtask
10
11 endmodule
```

Example – Task using Global Variables

```
1  module task_global();
2
3  reg [7:0] temp_out;
4  reg [7:0] temp_in;
5
6  task convert;
7  begin
8      temp_out = (9/5) *( temp_in + 32);
9  end
10 endtask
11
12 endmodule
```

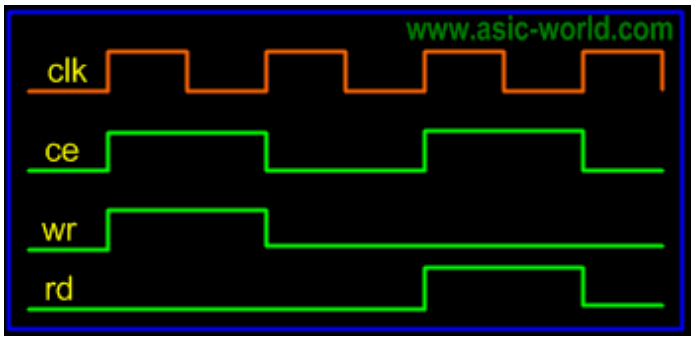
Calling a Task

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.

```
1  module task_calling (temp_a, temp_b, temp_c, temp_d);
2  input [7:0] temp_a, temp_c;
3  output [7:0] temp_b, temp_d;
4  reg [7:0] temp_b, temp_d;
5  `include "mytask.v"
6
7  always @ (temp_a)
8  begin
9      convert (temp_a, temp_b);
10 end
11
12 always @ (temp_c)
13 begin
14     convert (temp_c, temp_d);
15 end
16
17 endmodule
```

Example – CPU Write / Read Task

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



```
1  module bus_wr_rd_task();
2
3  reg clk,rd,wr,ce;
4  reg [7:0] addr,data_wr,data_rd;
5  reg [7:0] read_data;
6
7  initial begin
8     clk = 0;
9     read_data = 0;
10    rd = 0;
11    wr = 0;
12    ce = 0;
13    addr = 0;
14    data_wr = 0;
15    data_rd = 0;
16    // Call the write and read tasks here
17    #1 cpu_write(8'h11,8'hAA);
18    #1 cpu_read(8'h11,read_data);
19    #1 cpu_write(8'h12,8'hAB);
20    #1 cpu_read(8'h12,read_data);
21    #1 cpu_write(8'h13,8'h0A);
22    #1 cpu_read(8'h13,read_data);
23    #100 $finish;
24 end
25 // Clock Generator
26 always
27 #1 clk = ~clk;
28 // CPU Read Task
29 task cpu_read;
30 input [7:0] address;
31 output [7:0] data;
32 begin
33    $display ( "CPU Read task with address : %h" ,address);
34    $display ( " Driving CE, RD and ADDRESS on to bus" );
35    @ (posedge clk);
36    addr = address;
37    ce = 1;
38    rd = 1;
39    @ (negedge clk);
40    data = data_rd;
41    @ (posedge clk);
42    addr = 0;
43    ce = 0;
44    rd = 0;
45    $display ( " CPU Read data : %h" ,data);
46    $display ( "======================" );
47 end
48 endtask
49 // CU Write Task
50 task cpu_write;
51 input [7:0] address;
52 input [7:0] data;
53 begin
54    $display ( "CPU Write task with address : %h Data : %h" ,address,data);
55    $display ( " Driving CE, WR, WR data and ADDRESS on to bus" );
```

```
56    @ (posedge clk);
57    addr = address;
58    ce = 1;
59    wr = 1;
60    data_wr = data;
61    @ (posedge clk);
62    addr = 0;
63    ce = 0;
64    wr = 0;
65    $display ( "======================" );
66end
67endtask
68
69// Memory model for checking tasks
70reg [7:0] mem [0:255];
71
72always @ (addr or ce or rd or wr or data_wr)
73if (ce) begin
74    if (wr) begin
75        mem[addr] = data_wr;
76    end
77    if (rd) begin
78        data_rd = mem[addr];
79    end
80end
81
82endmodule
```

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.

Example – Simple Function

```
1  module simple_function();
2
3  function myfunction;
4  input a, b, c, d;
5  begin
6      myfunction = ((a+b) + (c–d));
7  end
8  endfunction
9
10 endmodule
```

Example – Calling a Function

```
1  module function_calling(a, b, c, d, e, f);
2
3  input a, b, c, d, e ;
4  output f;
5  wire f;
6  `include "myfunction.v"
7
8  assign f = (myfunction (a,b,c,d)) ? e :0;
9
10 endmodule
```

NOTES

SYSTEM TASK AND FUNCTION

CHAPTER 12

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.

$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.

NOTES

ART OF WRITING TESTBENCHES

CHAPTER 13

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 up 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

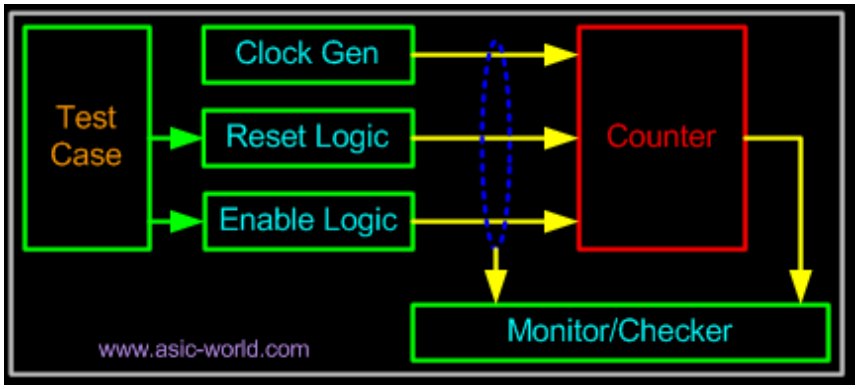
Lets assume that we have to verify a simple 4–bit up counter, which increments its count when ever enable is high and resets to zero, when reset is asserted high. Reset is synchronous to clock.

Code for Counter

```
1 //------------------------------------------------------
2 // Design Name : counter
3 // File Name : counter.v
4 // Function : 4 bit up counter
5 // Coder : Deepak
6 //------------------------------------------------------
7 module counter (clk, reset, enable, count);
8 input clk, reset, enable;
9 output [3:0] count;
10 reg [3:0] count;
11
12 always @ (posedge clk)
13 if (reset == 1'b1) begin
14    count <= 0;
15 end else if ( enable == 1'b1) begin
16    count <= count + 1;
17 end
18
19 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

```
1 module counter_tb;
2 reg clk, reset, enable;
3 wire [3:0] count;
4
5 counter U0 (
6 .clk (clk),
7 .reset (reset),
8 .enable (enable),
9 .count (count)
10 );
11
```

```
12 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

```
 1 module counter_tb;
 2 reg clk, reset, enable;
 3 wire [3:0] count;
 4
 5 counter U0 (
 6 .clk (clk),
 7 .reset (reset),
 8 .enable (enable),
 9 .count (count)
10 );
11
12 initial
13 begin
14     clk = 0;
15     reset = 0;
16     enable = 0;
17 end
18
19 always
20 #5 clk = !clk;
21
22 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 referring 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.asic–world.com\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...

```
 1 module counter_tb;
 2 reg clk, reset, enable;
 3 wire [3:0] count;
 4
 5 counter U0 (
 6 .clk (clk),
 7 .reset (reset),
 8 .enable (enable),
 9 .count (count)
10 );
11
12 initial begin
13     clk = 0;
14     reset = 0;
15     enable = 0;
16 end
17
18 always
19 #5 clk = !clk;
20
21 initial begin
22     $dumpfile ( "counter.vcd" );
23     $dumpvars;
24 end
25
26 initial begin
27     $display( "\t\ttime,\tclk,\treset,\tenable,\tcount" );
28     $monitor( "%d,\t%b,\t%b,\t%b,\t%d" ,$time, clk,reset,enable,count);
29 end
30
31 initial
32 #100 $finish;
33
34 //Rest of testbench code after this line
35
36 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.asic-world.com>veridos counter.v counter_tb.v
VeriWell for Win32 HDL Version 2.1.4 Fri Jan 17 21:33:25 2003

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

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


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

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

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

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

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

Adding Reset Logic

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

```
1  event reset_trigger;
2  event reset_done_trigger;
3
4  initial begin
5    forever begin
6      @ (reset_trigger);
7      @ (negedge clk);
8      reset = 1;
9      @ (negedge clk);
10     reset = 0;
11     -> reset_done_trigger;
12   end
13 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 – Assert/ Deassert 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 – Assert/Deassert enable and reset randomly.

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

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

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.

```
1 event terminate_sim;
2 initial begin
3     @ (terminate_sim);
4     #5 $finish;
5 end
```

and the modified test case #2 would like.

```
1  initial
2  begin: TEST_CASE
3      #10 -> reset_trigger;
4      @ (reset_done_trigger);
5      @ (negedge clk);
6      enable = 1;
7      repeat (10) begin
8          @ (negedge clk);
9      end
10     enable = 0;
11     #5 -> terminate_sim;
12 end
13
```

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.

```
1 reg [3:0] count_compare;
2
3 always @ (posedge clk)
4 if (reset == 1'b1) begin
5     count_compare <= 0;
6 end else if ( enable == 1'b1) begin
7     count_compare <= count_compare + 1;
8 end
```

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".

```
1always @ (posedge clk)
2if (count_compare != count) begin
3    $display ( "DUT Error at time %d" , $time);
4    $display ( " Expected value %d, Got Value %d" , count_compare, count);
5    #5 -> terminate_sim;
6end
```

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  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.

NOTES

MODELING MEMORIES AND FSM

CHAPTER 14

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

```
1 module memory();
2 reg [7:0] my_memory [0:255];
3
4 initial begin
5     $readmemh( "memory.list" , my_memory);
6 end
7 endmodule
```

Example – Memory.list file

```
1 //Comments are allowed
2 1100_1100 // This is first address i.e 8'h00
3 1010_1010 // This is second address i.e 8'h01
4 @ 55 // Jump to new address 8'h55
5 0101_1010 // This is address 8'h55
6 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.

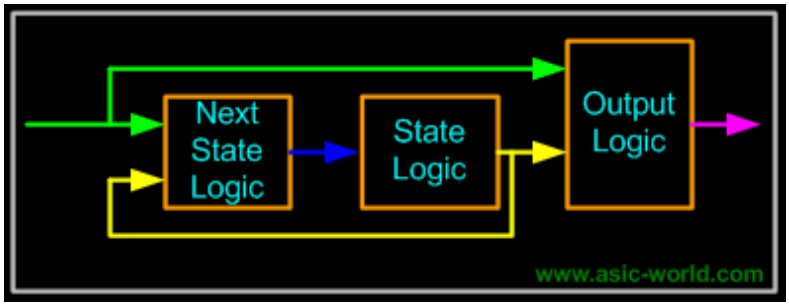
Introduction to FSM

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.

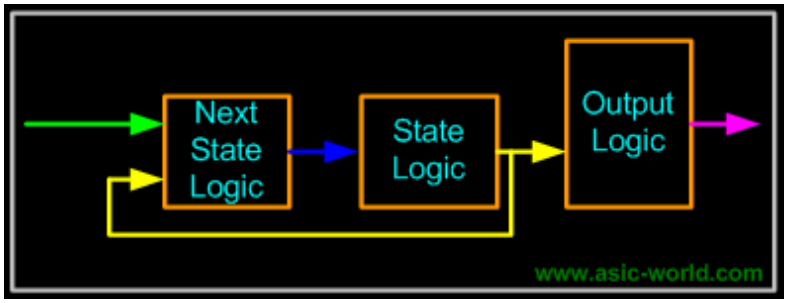
State machine Types

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.

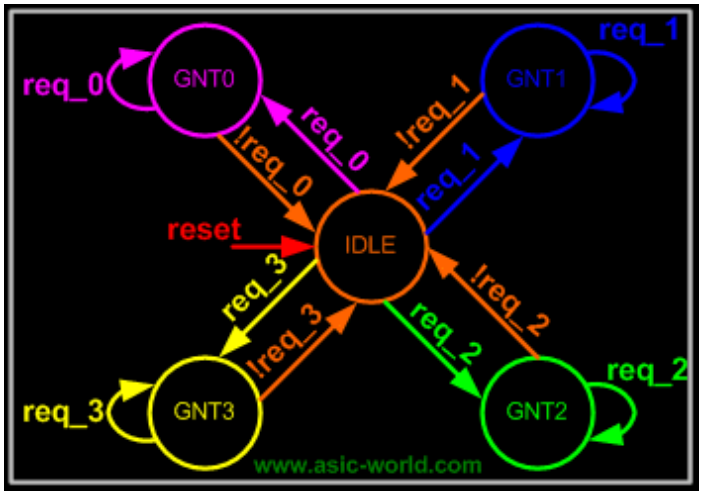
Modeling State machines.

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.

Example – Arbiter

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



Verilog Code

FSM code should have three sections,

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

Encoding Style

There are many encoding styles around, some of which are

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

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

One Hot Encoding

```
1parameter [4:0] IDLE  = 5'b0_0001;
2parameter [4:0] GNT0  = 5'b0_0010;
3parameter [4:0] GNT1  = 5'b0_0100;
4parameter [4:0] GNT2  = 5'b0_1000;
5parameter [4:0] GNT3  = 5'b1_0000;
```

Binary Encoding

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

Gray Encoding

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

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

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

```
30        end
31        GNT3 : if (req_3 == 1'b0) begin
32           next_state = IDLE;
33        end else begin
34           next_state = GNT3;
35        end
36        default : next_state = IDLE;
37     endcase
38end
```

Sequential Section

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

```
1always @ (posedge clock)
2begin : OUTPUT_LOGIC
3   if (reset == 1'b1) begin
4      gnt_0 <= #1 1'b0;
5      gnt_1 <= #1 1'b0;
6      gnt_2 <= #1 1'b0;
7      gnt_3 <= #1 1'b0;
8      state <= #1 IDLE;
9   end else begin
10     state <= #1 next_state;
11     case(state)
12        IDLE : begin
13           gnt_0 <= #1 1'b0;
14           gnt_1 <= #1 1'b0;
15           gnt_2 <= #1 1'b0;
16           gnt_3 <= #1 1'b0;
17        end
18        GNT0 : begin
19           gnt_0 <= #1 1'b1;
20        end
21        GNT1 : begin
22           gnt_1 <= #1 1'b1;
23        end
24        GNT2 : begin
25           gnt_2 <= #1 1'b1;
26        end
27        GNT3 : begin
28           gnt_3 <= #1 1'b1;
29        end
30        default : begin
31           state <= #1 IDLE;
32        end
33     endcase
34  end
35end
```

Full Code using binary encoding

```
 1 module fsm_full(
 2 clock , // Clock
 3 reset , // Active high reset
 4 req_0 , // Active high request from agent 0
 5 req_1 , // Active high request from agent 1
 6 req_2 , // Active high request from agent 2
 7 req_3 , // Active high request from agent 3
 8 gnt_0 , // Active high grant to agent 0
 9 gnt_1 , // Active high grant to agent 1
10 gnt_2 , // Active high grant to agent 2
11 gnt_3 // Active high grant to agent 3
12 );
13 // Port declaration here
14 input clock ; // Clock
15 input reset ; // Active high reset
16 input req_0 ; // Active high request from agent 0
17 input req_1 ; // Active high request from agent 1
18 input req_2 ; // Active high request from agent 2
19 input req_3 ; // Active high request from agent 3
20 output gnt_0 ; // Active high grant to agent 0
21 output gnt_1 ; // Active high grant to agent 1
22 output gnt_2 ; // Active high grant to agent 2
23 output gnt_3 ; // Active high grant to agent
24
25 // Internal Variables
26 reg gnt_0 ; // Active high grant to agent 0
27 reg gnt_1 ; // Active high grant to agent 1
28 reg gnt_2 ; // Active high grant to agent 2
29 reg gnt_3 ; // Active high grant to agent
30
31 parameter [2:0] IDLE  = 3'b000;
32 parameter [2:0] GNT0  = 3'b001;
33 parameter [2:0] GNT1  = 3'b010;
34 parameter [2:0] GNT2  = 3'b011;
35 parameter [2:0] GNT3  = 3'b100;
36
37 reg [2:0] state, next_state;
38
39 always @ (state or req_0 or req_1 or req_2 or req_3)
40 begin
41    next_state = 0;
42    case(state)
43       IDLE : if (req_0 == 1'b1) begin
44          next_state = GNT0;
45       end else if (req_1 == 1'b1) begin
46          next_state= GNT1;
47       end else if (req_2 == 1'b1) begin
48          next_state= GNT2;
49       end else if (req_3 == 1'b1) begin
```

```
50          next_state= GNT3;
51        end else begin
52          next_state = IDLE;
53        end
54        GNT0 : if (req_0 == 1'b0) begin
55          next_state = IDLE;
56        end else begin
57          next_state = GNT0;
58        end
59        GNT1 : if (req_1 == 1'b0) begin
60          next_state = IDLE;
61        end else begin
62          next_state = GNT1;
63        end
64        GNT2 : if (req_2 == 1'b0) begin
65          next_state = IDLE;
66        end else begin
67          next_state = GNT2;
68        end
69        GNT3 : if (req_3 == 1'b0) begin
70          next_state = IDLE;
71        end else begin
72          next_state = GNT3;
73        end
74        default : next_state = IDLE;
75      endcase
76  end
77
78  always @ (posedge clock)
79  begin : OUTPUT_LOGIC
80    if (reset) begin
81      gnt_0 <= #1 1'b0;
82      gnt_1 <= #1 1'b0;
83      gnt_2 <= #1 1'b0;
84      gnt_3 <= #1 1'b0;
85      state <= #1 IDLE;
86    end else begin
87      state <= #1 next_state;
88      case(state)
89        IDLE : begin
90          gnt_0 <= #1 1'b0;
91          gnt_1 <= #1 1'b0;
92          gnt_2 <= #1 1'b0;
93          gnt_3 <= #1 1'b0;
94        end
95        GNT0 : begin
96          gnt_0 <= #1 1'b1;
97        end
98        GNT1 : begin
99          gnt_1 <= #1 1'b1;
100       end
```

```
101          GNT2 : begin
102             gnt_2 <= #1 1'b1;
103          end
104          GNT3 : begin
105             gnt_3 <= #1 1'b1;
106          end
107          default : begin
108             state <= #1 IDLE;
109          end
110       endcase
111    end
112end
113
114endmodule
```

Testbench

```
  1`include "fsm_full.v"
  2
  3module fsm_full_tb();
  4reg clock , reset ;
  5reg req_0 , req_1 , req_2 , req_3;
  6wire gnt_0 , gnt_1 , gnt_2 , gnt_3 ;
  7
  8initial begin
  9   $display( "Time\t R0 R1 R2 R3 G0 G1 G2 G3" );
 10   $monitor( "%g\t %b %b %b %b %b %b %b %b" , $time, req_0, req_1, req_2, req_3, gnt_0, gnt_1, gnt_2, gnt_3);
 11   clock = 0;
 12   reset = 0;
 13   req_0 = 0;
 14   req_1 = 0;
 15   req_2 = 0;
 16   req_3 = 0;
 17   #10 reset = 1;
 18   #10 reset = 0;
 19   #10 req_0 = 1;
 20   #20 req_0 = 0;
 21   #10 req_1 = 1;
 22   #20 req_1 = 0;
 23   #10 req_2 = 1;
 24   #20 req_2 = 0;
 25   #10 req_3 = 1;
 26   #20 req_3 = 0;
 27   #10 $finish;
 28end
 29
 30always
 31#2 clock = ~clock;
 32
 33
 34fsm_full U_fsm_full(
 35clock , // Clock
 36reset , // Active high reset
 37req_0 , // Active high request from agent 0
```

```
38 req_1 , // Active high request from agent 1
39 req_2 , // Active high request from agent 2
40 req_3 , // Active high request from agent 3
41 gnt_0 , // Active high grant to agent 0
42 gnt_1 , // Active high grant to agent 1
43 gnt_2 , // Active high grant to agent 2
44 gnt_3 // Active high grant to agent 3
45 );
46
47
48
49 endmodule
```

Simulator Output

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

NOTES

PARAMETERIZED MODULES

CHAPTER 15

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

```
 1 module secret_number;
 2 parameter my_secret = 0;
 3
 4 initial begin
 5    $display( "My secret number is %d" , my_secret);
 6 end
 7
 8 endmodule
 9
10 module defparam_example();
11
12 defparam U0.my_secret = 11;
13 defparam U1.my_secret = 22;
14
15 secret_number U0();
16 secret_number U1();
17
18 endmodule
```

Parameter Override during instantiating.

```
1  module secret_number;
2  parameter my_secret = 0;
3
4  initial begin
5      $display( "My secret number in module is %d" , my_secret);
6  end
7
8  endmodule
9
10 module param_overide_instance_example();
11
12 secret_number #(11) U0();
13 secret_number #(22) U1();
14
15 endmodule
```

Passing more then one parameter

```
1  module ram_sp_sr_sw (
2  clk , // Clock Input
3  address , // Address Input
4  data , // Data bi-directional
5  cs , // Chip Select
6  we , // Write Enable/Read Enable
7  oe // Output Enable
8  );
9
10 parameter DATA_WIDTH = 8 ;
11 parameter ADDR_WIDTH = 8 ;
12 parameter RAM_DEPTH = 1 << ADDR_WIDTH;
13 // Actual code of RAM here
14
15 endmodule
```

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

```
1  module ram_controller (); //Some ports
2
3  // Controller Code
4
5  ram_sp_sr_sw #(16,8,256) ram(clk,address,data,cs,we,oe);
6
7  endmodule
```

Verilog 2001

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

```
1module ram_controller (); //Some ports
2
3ram_sp_sr_sw #(
4.DATA_WIDTH(16),
5.ADDRE_WIDTH(8),
6.RAM_DEPTH(256)) ram(clk,address,data,cs,we,oe);
7
8endmodule
```

Was this copied from VHDL?

NOTES

VERILOG SYNTHESIS TUTORIAL

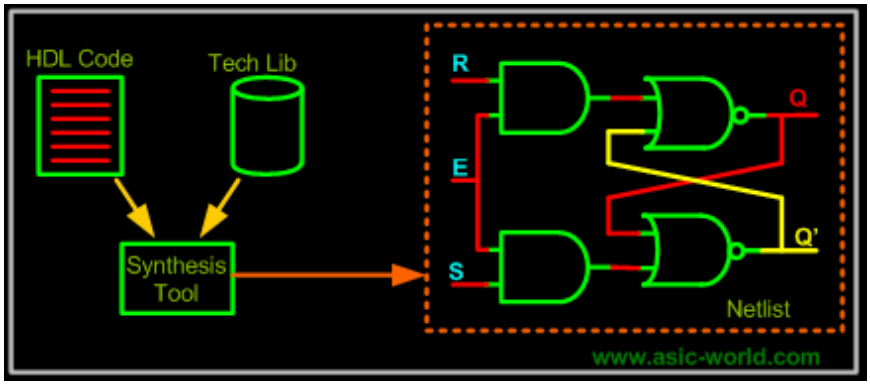
CHAPTER 16

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 experienced in college, every thing (all the digital circuits) is designed manually. Draw K−maps, optimize the logic, Draw the schematic. This is how engineers used to design digital logic circuits in early days. Well this works fine as long as the design is 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 while reading earlier chapters, how could this be represented in Hardware. Example would be "delays". There is no way we could synthesize delays, but of course 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 nonblocking assignments to get same effect.
primitives	Only gate level primitives are supported
table	UDP and tables are not supported.

Example of Non–Synthesizable Verilog construct.

Any code that contains 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.

Example – Initial Statement

```
1 module synthesis_initial(
2 clk,q,d);
3 input clk,d;
4 output q;
5 reg q;
6
7 initial begin
8     q <= 0;
9 end
10
11 always @ (posedge clk)
12 begin
```

```
13    q <= d;
14end
15
16endmodule
```

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

```
 1module synthesis_compare_xz (a,b);
 2output a;
 3input b;
 4reg a;
 5
 6always @ (b)
 7begin
 8    if ((b == 1'bz) || (b == 1'bx)) begin
 9        a = 1;
10    end else begin
11        a = 0;
12    end
13end
14
15endmodule
```

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
	/	Division
	+	Add
	–	Subtract
	%	Modulus
	+	Unary plus
	–	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality

Reduction	&	Bitwise negation
	~&	nand
	|	or
	~|	nor
	^	xor
	^~ ~^	xnor
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

Logic Circuit Modeling

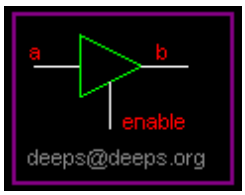
From what we have learn in digital design, we know that there could be only two types of digital circuits. One is combinational circuits and second is sequential circuits. There are very few rules that need to be followed to get good synthesis output and avoid surprises.

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

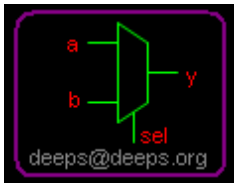
assign y = (a&b) | (c^d);

Tri–state buffer



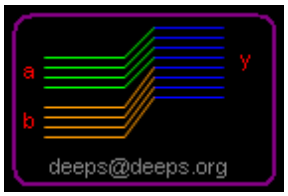

```
1 module tri_buf (a,b,enable);
2 input a;
3 output b;
4 input enable;
5 wire b;
6
7 assign b = (enable) ? a : 1'bz;
8
9 endmodule
```

Mux




```
1module mux_21 (a,b,sel,y);
2input a, b;
3output y;
4input sel;
5wire y;
6
7assign y = (sel) ? b : a;
8
9endmodule
```

Simple Concatenation




```
1module bus_con (a,b);
2input [3:0] a, b;
3output [7:0] y;
4wire [7:0] y;
5
6assign y = {a,b};
7
8endmodule
```

1 bit adder with carry

```
1module addbit (
2a , // first input
3b , // Second input
4ci , // Carry input
5sum , // sum output
6co // carry output
7);
8//Input declaration
9input a;
10input b;
11input ci;
12//Ouput declaration
13output sum;
14output co;
```

```
15 //Port Data types
16 wire a;
17 wire b;
18 wire ci;
19 wire sum;
20 wire co;
21 //Code starts here
22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit
```

Multiply by 2

```
1  module muliply (a,product);
2  input [3:0] a;
3  output [4:0] product;
4  wire [4:0] product;
5
6  assign product  = a << 1;
7
8  endmodule
```

3 is to 8 decoder

```
1  module decoder (in,out);
2  input [2:0] in;
3  output [7:0] out;
4  wire [4:0] out;
5  assign out  = (in == 3'b000 ) ? 8'b0000_0001 :
6  (in == 3'b001 ) ? 8'b0000_0010 :
7  (in == 3'b010 ) ? 8'b0000_0100 :
8  (in == 3'b011 ) ? 8'b0000_1000 :
9  (in == 3'b100 ) ? 8'b0001_0000 :
10 (in == 3'b101 ) ? 8'b0010_0000 :
11 (in == 3'b110 ) ? 8'b0100_0000 :
12 (in == 3'b111 ) ? 8'b1000_0000 : 8'h00;
13
14 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 using always

```
1  module decoder_always (in,out);
2  input [2:0] in;
3  output [7:0] out;
4  reg [4:0] out;
5
6  always @ (in)
7  begin
8     out = 0;
9     case (in)
10       3'b001 : out = 8'b0000_0001;
11       3'b010 : out = 8'b0000_0010;
12       3'b011 : out = 8'b0000_0100;
13       3'b100 : out = 8'b0000_1000;
14       3'b101 : out = 8'b0001_0000;
15       3'b110 : out = 8'b0100_0000;
16       3'b111 : out = 8'b1000_0000;
17    endcase
18 end
19
20 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 nonblocking assignments for sequential circuits.

Simple Flip–Flop

```
1  module flif_flop (clk,reset, q, d);
2  input clk, reset, d;
3  output q;
4  reg q;
5
6  always @ (posedge clk )
7  begin
8     if (reset == 1) begin
9        q <= 0;
10    end else begin
11       q <= d;
12    end
13 end
14
15 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 nonblocking for sequential and blocking for combo logic
- Don't mix blocking and nonblocking 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.

NOTES

VERILOG PLI TUTORIAL

CHAPTER 17

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.

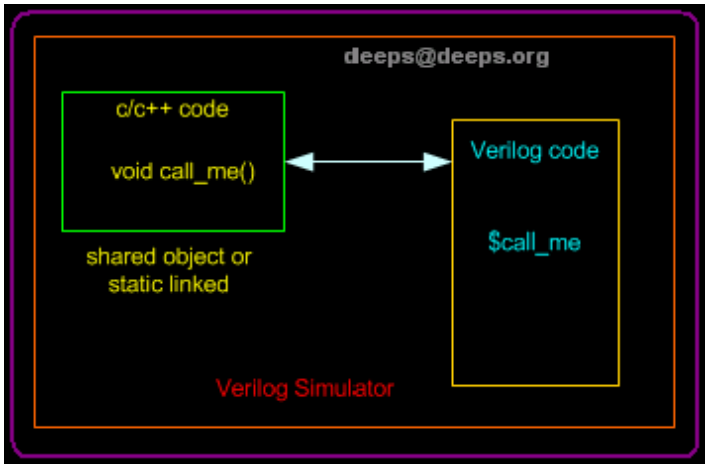
To achieve above few application of PLI, C code should have the access to the internal data structure of the Verilog simulator. To facilitate this Verilog PLI provides with something called acc routines or simply access routines.

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.

You can get Verilog 2001 LRM or PLI 1.0 IEEE document for details of each and every functions provided. Verilog IEEE LRM's are written in such a way that anyone with hardware background can understand. If you are unable to get hold of above IEEE docs, then you can buy PLI books listed in books section.

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 (This is called linking, and you need to refer to simulator user guide to understand how this is done).
- Once linked just run the simulator like any other Verilog simulation.



During execution of the Verilog code by the simulator, when ever the simulator encounters the user defines system tasks (the one which starts with $), the execution control is passed to PLI routine (C/C++ function).

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

```
1#include < stdio.h >
2void hello () {
3    printf ( "\nHello Deepak\n" );
4}
```

Verilog Code

```
1module hello_pli ();
2
3initial begin
4    $hello;
5    #10 $finish;
6end
7
8endmodule
```

Running the Simulation

Once linking is done, simulation is run as a normal simulation as we had seen earlier with slight modification to the command line options. Like we need to tell the simulator that we are using PLI (Modelsim needs to know which shared objects to load in command line).

Writing PLI Application

Example that we saw was too basic and is no good for any practical purpose. Lets consider our infamous counter example and write the DUT reference model and Checker in C and link that to Verilog Testbench. First lets list out the requirements for writing a C model using PLI.

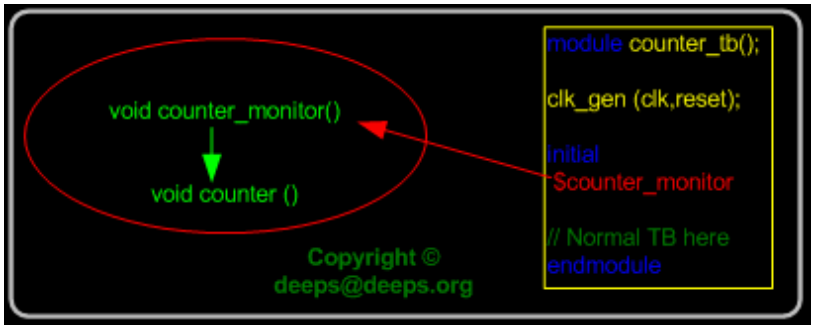
- Means of calling the C model, when ever there is change in input signals (Could be wire or reg or types).
- Means to get the value of the changes signals in Verilog code or any other signals in Verilog code from inside the C code.
- Means to drive the value on any signal inside the Verilog code from C code.

There are set of routines (functions), that Verilog PLI provides which satisfy above requirements.

PLI Application Specification.

Lets define the requirements for our infamous counter testbench requirements using PLI. We will call out PLI function as $counter_monitor.

- Implements a Counter logic in C.
- Implements Checker logic in C.
- Terminates the simulation, when ever checker fails.



Calling the C function.

Writing counter in C is so cool, but when do we increment the counter value. Well we need to monitor the change in clock signal. (Note : By the way, it normally good idea to drive reset and clock from Verilog code.) When ever the clock changes, counter function needs to be executed. This can be achieved by using below routine.

- Use acc_vcl_add routine. The syntax of which can be found in Verilog PLI LRM.

acc_vcl_add routines basically allows us to monitor list of signals, and when ever any of the monitor signals change, it calls the user defined function (i.e this function is called Consumer C

routine). VCL routine has four arguments

- Handle to the monitored object
- Consumer C routine to call when the object value changes
- String to be passed to consumer C routine
- Predefined VCL flags: vcl_verilog_logic for logic monitoring vcl_verilog_strength for strength monitoring

acc_vcl_add(net, display_net, netname, vcl_verilog_logic);

Lets look at the code below, before we go into details.

C Code – Basic

Counter_monitor is our C function, which will be called from the Verilog Testbench. As like any another C code, we need to include the header files, specific to application that we are developing. In our case we need to include acc routines include file.

The access routine acc_initialize initializes the environment for access routines and must be called from your C–language application program before the program invokes any other access routines. and before exiting a C–language application program that calls access routines, it is necessary to also exit the access routine environment by calling acc_close at the end of the program.

```
1#include < stdio.h >
2#include "acc_user.h"
3
4typedef char * string;
5handle clk ;
6handle reset ;
7handle enable ;
8handle dut_count ;
9int count ;
10
11void counter_monitor()
12{
13
14   acc_initialize();
15   clk = acc_handle_tfarg(1);
16   reset = acc_handle_tfarg(2);
17   enable = acc_handle_tfarg(3);
18   dut_count = acc_handle_tfarg(4);
19   acc_vcl_add(clk,counter,null,vcl_verilog_logic);
20   acc_close();
21}
22
23void counter ()
24{
25   printf( "Clock changed state\n" );
26}
```

For accessing the Verilog objects, we use handle, A handle is a predefined data type that is a pointer to a specific object in the design hierarchy. Each handle conveys information to access routines about a unique instance of an accessible object information about the object¿s type, plus how and where to find data about the object. But how do we pass the information of specific object to handle. Well we can do this by number of ways, but for now, we will pass it from Verilog as parameters to $counter_monitor , this parameters can be accessed inside the C−program with acc_handle_tfarg() routine. Where the argument is numbers as in the code.

So clk = acc_handle_tfarg(1) basically makes the clk as the handle to first parameter passed. Similarly we assign all the handle's. Now we can add clk to the signal list that need to be monitored using the routine acc_vcl_add(clk,counter,null,vcl_verilog_logic). Here clk is the handle, counter is the user function to execute, when clk changes.

The function counter() does not require any explation, it is simple Hello world type code.

Verilog Code

Below is the code of the simple testbench for the counter example. We call the C−function using the syntax shown in code below. If object thats been passed is a instant, then it should be passed inside double quotes. Since all our objects are nets or wires, there is no need to pass them inside double quote.

```
1 module counter_tb();
2 reg enable;;
3 reg reset;
4 reg clk_reg;
5 wire clk;
6 wire [3:0] count;
7
8 initial begin
9    enable = 0;
10   clk = 0;
11   reset = 0;
12   $display( "Asserting reset" );
13   #10 reset = 1;
14   #10 reset = 0;
15   $display ( "Asserting Enable" );
16   #10 enable = 1;
17   #20 enable = 0;
18   $display ( "Terminating Simulator" );
19   #10 $finish;
20 end
21
22 always
23 #5 clk_reg = !clk_reg;
24
25 assign clk = clk_reg;
26
27 initial begin
28   $counter_monitor(top.clk,top.reset,top.enable,top.count);
29 end
```

```
30
31counter U(
32.clk (clk),
33.reset (reset),
34.enable (enable),
35.count (count)
36);
37
38endmodule
```

Depending on the simulator in use, the compile and running various. When you run the code above with the C code seen earlier we get following output

```
Asserting reset
Clock changed state
Clock changed state
Clock changed state
Asserting Enable
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Terminating Simulator
Clock changed state
Clock changed state
$finish at simulation time 60
```

C Code – Full

So now that we see that our function gets called whenever there is change in clock, we can write the counter code. But wait, there is a problem, every time counter function makes a exit, the local variables will loose its value. There are couple of ways we can preserve state of the variables.

- Declare the counter variable as global
- Use tf_setworkarea() and tf_getworkarea() routine to store and restore the values of the local variables.

Since we have only one variable, we can use the first solution. i.e. declare count as global variable.

To write equivalent model for the counter, clock, reset, enable signal input to DUT is required and to code checker, out of the DUT count is required. To read the values from the Verilog code, we have PLI routine.

acc_fetch_value(handle,"formate")

but the value returned is a string, so we need to convert that into integer if, muli–bit vector signal is read using this routine. pli_conv is a function which does this conversion. Routine tf_dofinish() is

used for terminating simulation, when DUT and TB count value does not match or in other words, when simulation mismatch occurs.

Rest of the code is self explanatory. (Now time is 11:45PM, time to bed)

```
#include < stdio.h >
#include "acc_user.h"

typedef char * string;
handle clk ;
handle reset ;
handle enable ;
handle dut_count ;
int count ;

void counter_monitor()
{

  acc_initialize();
  clk = acc_handle_tfarg(1);
  reset = acc_handle_tfarg(2);
  enable = acc_handle_tfarg(3);
  dut_count = acc_handle_tfarg(4);
  acc_vcl_add(clk,counter,null,vcl_verilog_logic);
  acc_close();
}

void counter ()
{

  string i_reset = acc_fetch_value(reset, "%b" );
  string i_enable = acc_fetch_value(enable, "%b" );
  string i_count = acc_fetch_value(dut_count, "%b" );
  string i_clk = acc_fetch_value(clk, "%b" );
  string high = "1" ;
  int size_in_bits= acc_fetch_size (dut_count);
  int tb_count = 0;
  // Counter function goes here
  if (*i_reset == *high) {
     count = 0;
  }
  else if ((*i_enable == *high) && (*i_clk == *high)) {
     if ( count == 15 ) {
        count = 0;
     } else {
        count = count + 1;
     }
     // Counter Checker function goes herecker logic goes here
     if ((*i_clk != *high) && (*i_reset != *high)) {
        tb_count = pli_conv(i_count,size_in_bits);
        if (tb_count != count) {
           printf( "dut_error : Expect value %d, Got value %d\n" , count, tb_count);
```

```
48              tf_dofinish();
49          }
50      }
51  }
52  // Multi-bit vector to integer conversion.
53  int pli_conv (string in_string,int no_bits)
54  {
55      int conv = 0;
56      int i = 0;
57      int j = 0;
58      int bin = 0;
59      for ( i = no_bits-1; i >= 0; i = i - 1) {
60          if (*(in_string + i) == 49) {
61              bin = 1;
62          } else if (*(in_string + i) == 120) {
63              printf ( "Warning : X detected" );
64              bin = 0;
65          } else if (*(in_string + i) == 122) {
66              printf ( "Warning : Z detected" );
67              bin = 0;
68          } else {
69              bin = 0;
70          }
71          conv = conv + (1 << j)*bin;
72          j ++;
73      }
74      return conv;
75  }
```

You can compile and simulate the above code with Simulator you have.

Note : There could be mistakes in the way I have written the code or taken the approach in explaining PLI. Please mail me if you feel that it needs to be fixed or you have better way to show how PLI tutorial should be written.

PLI Routines.

PLI 1.0 provides two types of routines, they are

- access routine
- task and function routine.

PLI 2.0 combined access routines and task and function routines into VPI routines, and also clarified the confusion in PLI 1.0.

Access Routines

Access routines are C programming language routines that provide procedural access to information within Verilog–HDL. Access routines perform one of two operations

Read Operation : read data about particular objects in your circuit design directly from internal data structures. Access routines can read information about the following objects

- Module instances
- Module ports
- Module paths
- Inter–module paths
- Top–level modules
- Primitive instances
- Primitive terminals
- Nets
- Registers
- Parameters
- Specparams
- Timing checks
- Named events
- Integer, real and time variables

Write Operation : Write new information about objects in your circuit design into the internal data structures. Access routines can write to following objects.

- Inter–module paths.
- Module paths.
- Primitive instances.
- Timing checks.
- Register logic values.
- Sequential UDP logic values.

Based on the operation performed by access routines, they are classified into 6 categories as shown below.

- Fetch : This routines return a variety of information about different objects in the design hierarchy.
- Handle : This routines return handles to a variety of objects in the design hierarchy.
- Modify : This routines alter the values of a variety of objects in the design hierarchy.
- Next : When used inside a loop construct, next routines find each object of a given type that is related to a particular reference object in the design hierarchy.
- Utility : This routines perform a variety of operations, such as initializing and configuring the access routine environment.

- Vcl : The Value Change Link (VCL) allows a PLI application to monitor the value changes of selected objects.

Access Routines Reference

Routine	Description
acc_handle_scope()	
acc_handle_by_name()	
acc_handle_by_object()	
acc_handle_interactive_scope()	
acc_handle_calling_mod_m()	
acc_handle_parent()	
acc_handle_port()	
acc_handle_hiconn()	
acc_handle_loconn()	
acc_handle_path()	
acc_handle_modpath()	
acc_handle_datapath()	
acc_handle_pathin()	
acc_handle_pathout()	
acc_handle_condition()	
acc_handle_tchk()	
acc_handle_notifier()	
acc_handle_tchkarg1()	
acc_handle_tchkarg2()	
acc_handle_simulated_net()	
acc_handle_terminal()	
acc_handle_conn()	
acc_handle_tfinst()	
acc_handle_tfarg()	
acc_handle_itfarg()	
acc_fetch_argc()	
acc_fetch_argv()	
acc_fetch_attribute()	
acc_fetch_attribute_int()	
acc_fetch_attribute_str()	
acc_fetch_paramtype()	
acc_fetch_paramval()	

acc_fetch_defname()	
acc_fetch_fullname()	
acc_fetch_name()	
acc_fetch_delay_mode()	
acc_fetch_delay()	
acc_fetch_size()	
acc_fetch_range()	
acc_fetch_tfarg()	
acc_fetch_itfarg()	
acc_fetch_ifarg_int()	
acc_fetch_itfarg_int()	
acc_fetch_tfarg_str()	
acc_fetch_itfarg_str()	
acc_fetch_precision()	
acc_fetch_timescale_info()	
acc_fetch_direction()	
acc_fetch_index()	
acc_fetch_edge()	
acc_fetch_polarity()	
acc_fetch_pulsere()	
acc_set_value()	
acc_append_delays()	
acc_replace_delays()	
acc_initialize()	
acc_close()	
acc_configure()	
acc_reset_buffer()	
acc_product_type()	
acc_product_version()	
acc_version()	
acc_set_scope()	
acc_set_interactive_scope()	
acc_count()	
acc_collect()	
acc_free()	
acc_compare_handles()	
acc_object_in_typelist()	
acc_object_of_type()	

acc_next_cell()	
acc_next_child()	
acc_next_modpath()	
acc_next_net()	
acc_next_parameter()	
acc_next_port()	
acc_next_portout()	
acc_next_primitive()	
acc_next_specparam()	
acc_next_tchk()	
acc_next_terminal()	
acc_next_scope()	
acc_next()	
acc_next_topmod()	
acc_next_cell_load()	
acc_next_load()	
acc_next_driver()	
acc_next_hiconn()	
acc_next_loconn()	
acc_next_bit()	
acc_next_input()	
acc_next_output()	

Program Flow using access routines

As seen in the earlier example, there set of steps that need to be performed before we could write a user application. This can be shown as in the below program.

```
1#include < acc_user.h >
2
3void pli_func() {
4   acc_initialize();
5   // Main body: Insert the user application code here
6   acc_close();
7}
```

- acc_user.h : all data–structure related to access routines
- acc_initialize() : initialize variables and set up environment
- main body : User–defined application
- acc_close() : Undo the actions taken by the function acc_initialize()

Handle to Objects

Handle is a predefined data type, is similar to that of a pointer in C, can be used to point to an object in the design database, can be used to refer to any kind of object in the design database. Handle is backbone of access routine methodology and the most important new concept introduced in this part of PLI 1.0.

Declarations

- handle my_handle;
- handle clock;
- handle reset;

Value change link(VCL)

The Value Change Link (VCL) allows a PLI application to monitor the value changes of selected objects. The VCL can monitor value changes for the following objects.

- Events.
- Scalar and vector registers.
- Scalar nets.
- Bit−selects of expanded vector nets.
- Unexpanded vector nets.

The VCL cannot extract information about the following objects:

- Bit−selects of unexpanded vector nets or registers.
- Part−selects.
- Memories.
- Expressions.

Utility Routines

Interaction between the Verilog system and the user¿s routines is handled by a set of routines that are supplied with the Verilog system. Library functions defined in PLI1.0 Perform a wide variety of operations on the parameters passed to the system call is used to do a simulation synchronization or to implement conditional program breakpoint .

This routines are also called Utility routines. Most of these routines are in two forms: one dealing with the current call, or ¿instance,¿ and another dealing with an instance other than the current one and referenced by an instance pointer.

Classification of Utiliy Routines

Routine	Description
tf_getp()	
tf_putp()	
tf_getrealp()	
tf_igetrealp()	
tf_iputp()	
tf_putrealp()	
tf_iputrealp()	
tf_getlongp()	
tf_igetlongp()	
tf_putlongp()	
tf_iputlongp()	
tf_strgetp()	
tf_getcstringp()	
tf_strdelputp()	
tf_strlongdelputp()	
tf_strrealdelputp()	
tf_copypvc_flag()	
tf_icopypvc_flag()	
tf_movepvc_flag()	
tf_imovepvc_flag()	
tf_testpvc_flag()	
tf_itestpvc_flag()	
tf_getpchange()	
tf_igetpchange()	
tf_gettime()	
tf_getlongtime()	
tf_getrealtime()	
tf_strgettime()	
tf_gettimeunit()	
tf_gettimeprecision()	
tf_synchronize()	
tf_rosynchronize()	
tf_getnextlongtime()	
tf_setdelay()	
tf_setlongdelay()	

tf_setrealdelay()	
tf_clearalldelays()	
io_printf()	
io_mcdprintf()	
tf_warning()	
tf_error()	
tf_text()	
tf_message()	
tf_getinstance()	
tf_mipname()	
tf_spname()	
tf_setworkarea()	
tf_getworkarea()	
tf_nump()	
tf_typed()	
tf_sized()	
tf_dostop()	
tf_dofinish()	
mc_scan_plusargs()	
tf_compare_long()	
tf_add_long()	
tf_subtract_long()	
tf_multiply_long()	
tf_divide_long()	
tf_long_to_real()	
tf_longtime_tostr()	
tf_real_tf_long()	
tf_write_save()	
tf_read_restart()	

NOTES

WHAT'S NEW IN VERILOG 2001

CHAPTER 18

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)
```

H1START

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;
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 tut

NOTES

ASSERTIONS IN VERILOG

CHAPTER 19

Introduction

Verification with assertions refers to the use of an assertion language to specify expected behavior in a design, Tools that evaluate the assertions relative to the design under verification

Assertion–based verification is of most use to design and verification engineers who are responsible for the RTL design of digital blocks and systems. ABV lets design engineers capture verification information during design. It also enables internal state, datapath, and error precondition coverage analysis.

Simple example of assertion could be a FIFO, when ever ever FIFO is full and write happens, it is illegal. So designer of FIFO can write assertion which checks for this condition and asserts failure.

Assertions Languages

Currently there are multiple ways available for writting assertions as shown below.

- Open Verification Library (OVL).
- Formal Property Language Sugar
- SystemVerilog Assertions

Most assertions can be written in HDL, but HDL assertions can be lengthy and complicated. This defeats the purpose of assertions, which is to ensure the correctness of the design. Lengthy, complex HDL assertions can be hard to create and subject to bugs themselves.

In this tutorial we will be seeing verilog based assertion (OVL) and PSL (sugar).

Advantages of using assertions

- Testing internal points of the design, thus increasing observability of the design
- Simplifying the diagnosis and detection of bugs by constraining the occurrence of a bug to the assertion monitor being checked
- Allowing designers to use the same assertions for both simulation and formal verification.

Implementing assertion monitors

Assertion monitors address design verification concerns and can be used as follows to increase design confidence:

- Combine assertion monitors to increase the coverage of the design (for example, in interface circuits and corner cases).

- Include assertion monitors when a module has an external interface. In this case, assumptions on the correct input and output behavior should be guarded and verified.
- Include assertion monitors when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores), or may not completely understand the module. In these cases, guarding the module with assertion monitors may prevent incorrect use of the module.

Triggering assertion monitors

An assertion monitor is triggered when an error condition occurs usually, in the following cycle. However, when the test_expr is not synchronized with the assertion clock clk, either a non–deterministic delay or false assertion triggering may occur. To avoid this consequence, always line up test_expr with the assertion monitor sampling clock clk. Non–deterministic triggering delay refers to the delay between the time the error condition occurs and the time it is detected. False triggering can occur with more complex assertions if the test_expr and assertion clock clk are not synchronized.

To Be Completed

To Be Completed

NOTES

COMPILER DIRECTIVES

CHAPTER 20

Introduction

A compiler directive may be used to control the compilation of a Verilog description. The grave accent mark, `, denotes a compiler directive. A directive is effective from the point at which it is declared to the point at which another directive overrides it, even across file boundaries. Compiler directives may appear anywhere in the source description, but it is recommended that they appear outside a module declaration. This appendix presents those directives that are part of IEEE–1364.

As in any lanaguage, each compiler has its own way of handling command line options and supported compiler directives in code. Below we will see some standard and common compiler directives. For specific compiler directives, please refer to simulator manuals.

`include

The `include compiler directive lets you insert the entire contents of a source file into another file during Verilog compilation. The compilation proceeds as though the contents of the included source file appear in place of the `include command. You can use the `include compiler directive to include global or commonly–used definitions and tasks, without encapsulating repeated code within module boundaries.

`define

This compiler directive is used for defining text MACROS, this is normally defined in verilog file "name.vh". Where name cane be module that you are coding. Since `define is compiler directive, it can used across multiple files.

`undef

The `undef compiler directive lets you remove definitions of text macros created by the `define compiler directive and the +define+ command–line plus option. You can use the `undef compiler directive undefine a text macro that you use in more than one file.

`ifdef

Optionally includes lines of source code during compilation. The `ifdef directive checks that a macro has been defined, and if so, compiles the code that follows. If the macro has not been define, compiler compiles the code (if any) following the optional `else directive. You can control what code is compiled by choosing whether to define the text macro, either with `define or with +define+. The `endif directive marks the end of the conditional code.

`timescale

The `timescale compiler directive specifies the time unit and precision of the modules that follow the directive. The time unit is the unit of measurement for time values, such as the simulation time and delay values. The time precision specifies how simulator rounds time values. The rounded time values that simulator uses are accurate to within the unit of time that you specify as the time

precision. The smallest−specified time precision determines the accuracy at which simulator must run, and thus the precision affects simlation performance and memory consumption.

String	Unit
s	Seconds
ms	Miliseconds
us	Microseconds
ns	Nanoseconds
ps	Picoseconds
fs	femtoseconds

`resetall

The `resetall directive sets all compiler directives to their default values.

`defaultnettype

The `defaultnettype directive allows the user to override the ordinary default type (wire) of implicitly declared nets. It must be used outside a module. It specifies the default type of all nets that are declared in modules that are declared after the directive.

`nounconnected_drive and `unconnected_drive

The `unconnected_drive and `nounconnected_drive directives cause all unconnected input ports of modules between the directives to be pulled up or pulled down, depending on the argument of the `unconnected_drive directive. The allowed arguments are pull0 and pull1.

NOTES

VERILOG QUICK REFERENCE

CHAPTER 21

Verilog Quick Reference

This is still in very early stage, need time to add more on this.

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);
nand (out, in1, ..., inN);
or (out, in1, ..., inN);
nor (out, in1, ..., inN);
xor (out, in1, ..., inN);
xnor (out, in1, ..., inN);
buf (out1, ..., outN, in);
not (out1, ..., outN, in);
bufif1 (out, in, ctl);
bufif0 (out, in, ctl);
notif1 (out, in, ctl);
notif0 (out, in, ctl);
pullup (out);
pulldown (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);

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 [
lvalue [
–> 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 ===, !==, ==, !=
&&, ||, >, >=,
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
$reset(0|1, expr) Reset with reset_value*$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 */

NOTES

VERILOG IN ONE DAY

CHAPTER 22

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.

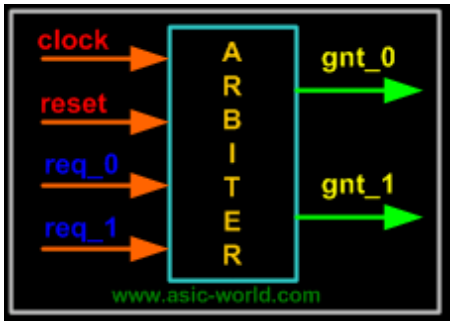
- Specs
- High level design
- Low level design or micro design
- RTL coding
- Verification
- Synthesis.

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

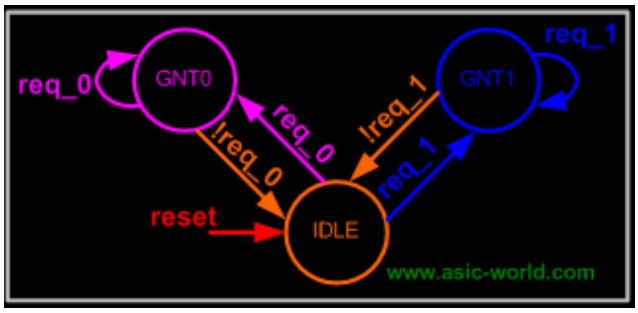


Normal digital design flow dictates that we draw a stated machine, from there we draw the truth table with next state transition for each flip–flop. And after that we draw kmaps and from kmaps we can get the optimized circuit. This method works just fine for small design, but with large designs this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

```
1module arbiter (
2clock , // clock
3reset , // Active high, syn reset
4req_0 , // Request 0
5req_1 , // Request 1
6gnt_0 , // Grant 0
7gnt_1
8);
9//--------------Input Ports-------------------------------
10input clock ;
11input reset ;
12input req_0 ;
13input req_1 ;
14//--------------Output Ports------------------------------
15output gnt_0 ;
16output 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 conversion.

Summary

- We learn how a block/module is defined in Verilog
- We learn 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 newbie 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
	/	Division
	+	Add
	–	Subtract
	%	Modulus
	+	Unary plus
	–	Unary minus

Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Reduction	&	Bitwise negation
	~&	nand
	|	or
	~|	nor
	^	xor
	^~ ~^	xnor
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

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 for loop seen earlier. Below code is simple example of a repeat statement.

```
1repeat (16) begin
2    $display ( "Current value of i is %d" , i);
3    i = i + 1;
4end
```

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, if−else, case(switch) 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 ". 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 nonblocking assignment. "=" executes code sequentially inside a begin and end, where as nonblocking "<=" 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–impedance. 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.

```
1 function parity;
2 input [31:0] data;
3 integer i;
4 begin
5    parity = 0;
6    for (i= 0; i < 32; i = i + 1) begin
7       parity = parity ^ data[i];
8    end
9 end
10 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.

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.

```
1  module arbiter_tb;
2
3  reg clock, reset, req0,req1;
4  wire gnt0,gnt1;
5
6  initial begin
7
8     $monitor ( "req0=%b, req1=%b, gnt0=%b,gnt1=%b" , req0,req0,gnt0,gnt1);
9     clock = 0;
10    reset = 0;
11    req0 = 0;
12    req1 = 0;
13    #5 reset = 1;
14    #15 reset = 0;
15    #10 req0 = 1;S
16    #10 req0 = 0;
17    #10 req1 = 1;
18    #10 req1 = 0;
19    #10 {req0,req1} = 2'b11;
20    #10 {req0,req1} = 2'b00;
21    #10 $finish;
22 end
23
```

```
24always begin
25
26    #5 clock = !clock; // Generate clock
27end
28
29arbiter U0 (
30.clock (clock),
31.reset (reset),
32.req_0 (req0),
33.req_1 (req1),
34.gnt_0 (gnt0),
35.gnt_1 (gnt1)
36);
37
38endmodule
```

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.