Verilog Tutorial

By

Deepak Kumar Tala

http://www.asic−world.com

Nov−22−2005

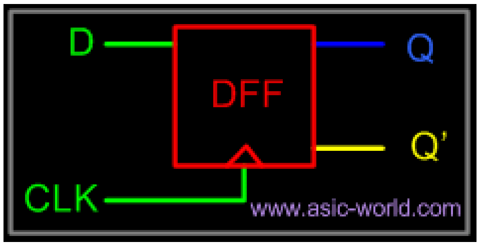
DISCLAIMER

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.




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

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

endmodule
```

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 than 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.

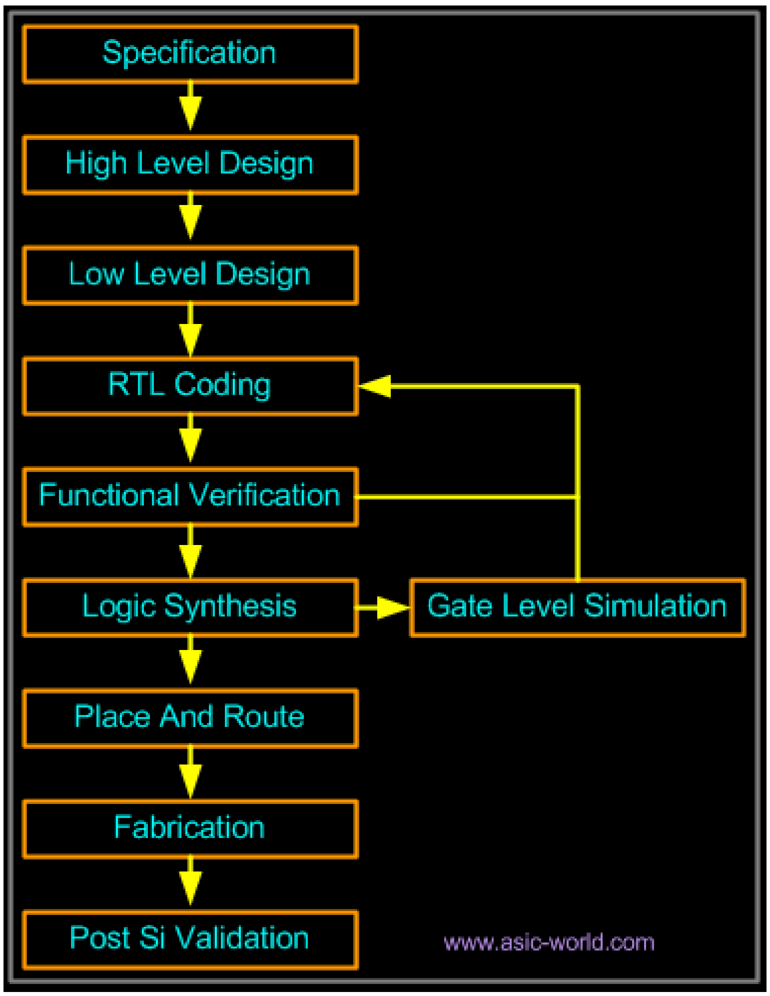
Bottom−Up Design

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

--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--

VERILOG IN ONE DAY

CHAPTER 2

Introduction

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

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

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

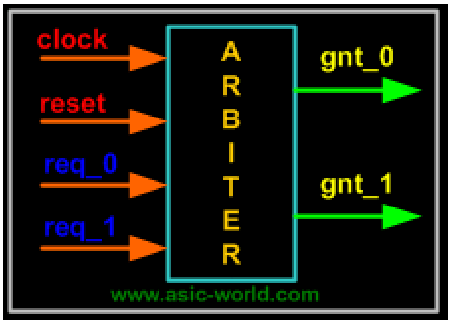
- Specifications (specs)
- High level design
- Low level (micro) design
- RTL coding
- Verification
- Synthesis.

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

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

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

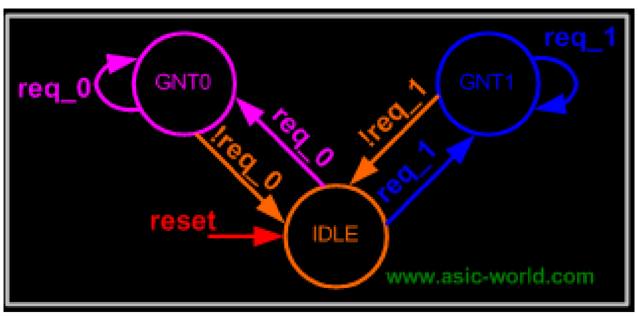
Block diagram of arbiter



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

Low level design

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



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

Modules

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

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

Code of module "arbiter"

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

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

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

Bi−Directional Ports Example −

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

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

Vector Signals Example −

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

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

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

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

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

(Drivers? What are those?)

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

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

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

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

Examples :

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

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

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

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. You'll see them someday.

Operators

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

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	–	Subtract
	%	Modulus
	+	Unary plus
	–	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Reduction	&	Bitwise negation
	~&	nand
	|	or
	~|	nor
	^	xor
	^~ ~^	xnor
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, They're (almost) just like C.

Control Statements

Wait, what's this? **if, else, repeat, while, for, case** − it's Verilog that looks exactly like C (and probably whatever other language you're used to programming in)! Even though the functionality appears to be the same as C, Verilog is an HDL, so the descriptions should translate to hardware. This means you've got to be careful when using control statements (otherwise your designs might not be implementable in hardware).

if−else

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

```
// begin and end act like curly braces in C/C++.
if (enable == 1'b1) begin
  data = 10; // Decimal assigned
  address = 16'hDEAD; // Hexadecimal
  wr_enable = 1'b1; // Binary
end else begin
  data = 32'b0;
  wr_enable = 1'b0;
  address = address + 1;
end
```

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

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

```
case(address)
  0 : $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.

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

While

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

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

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
```

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

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

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

for loop

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

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

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

repeat

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

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

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

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.
- repeat is the same as for−loop but without the incrementing variable.

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.

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.

Go on to the next page for the discussion of assign and always statements.

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.

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

Function and task have same syntax, one difference is task can have delays, where as function can not have any delay. Which means function can be used for modeling combination logic.

Second difference is function can return a value, where as Task can not return value.

Test Benches

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

Well we need to test it to see if it works according to specs. Most of the time, its same as we use to do in digital labs in college days. Drive the inputs, match the outputs with expected values. Lets look at the arbiter testbench.

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

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

reg gnt_0, gnt_1;

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

endmodule
// Testbench Code Goes here
module arbiter_tb;

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

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

always begin
   #5 clock = !clock;
```

```
end

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

endmodule
```

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.

NOTES

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

HISTORY OF VERILOG

CHAPTER 3

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 1364−2001.

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

DESIGN AND TOOL FLOW

CHAPTER 4

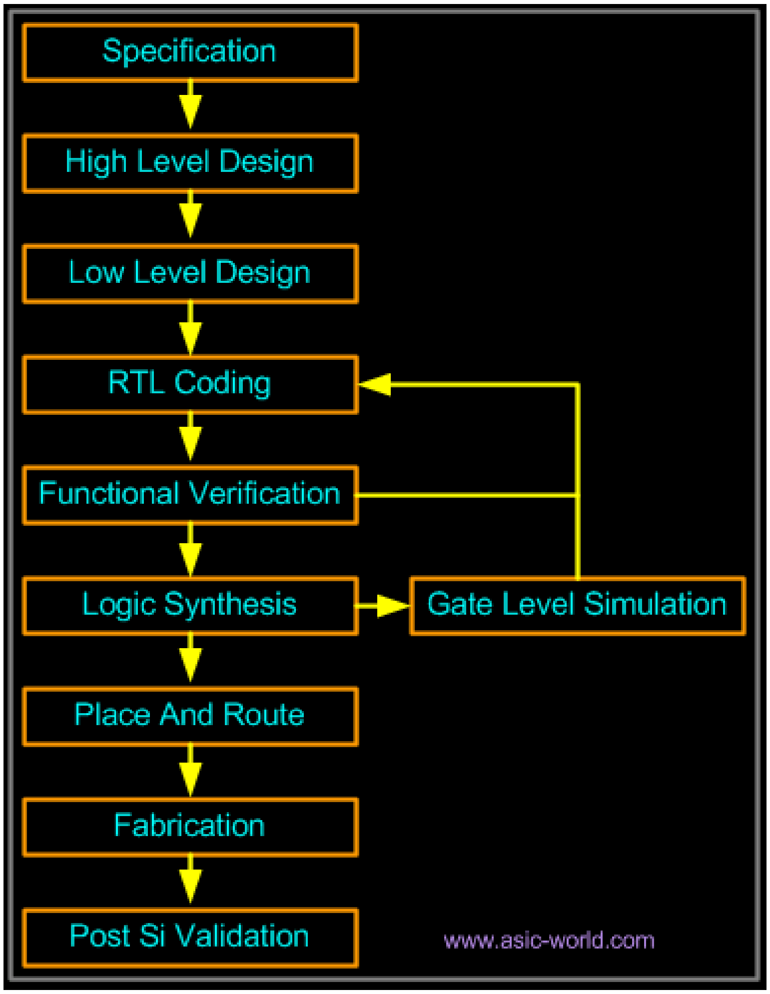
Introduction

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.

Various stages of ASIC/FPGA

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

Figure : Typical Design flow



Specification

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

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.

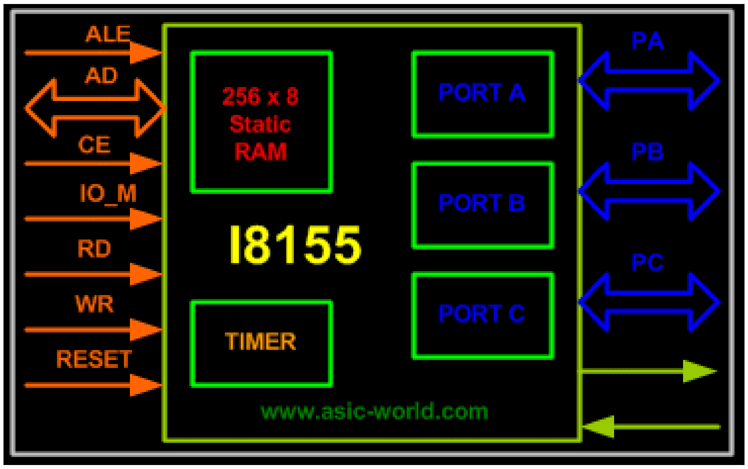


Figure : I8155 High Level Block Diagram

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. It is always a good idea if waveform is drawn at various interfaces. This is phase, where one spends lot of time.

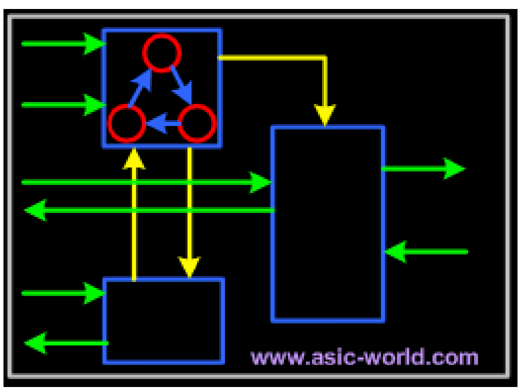


Figure : Sample Low level design

RTL Coding

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

```
module addbit (
a , // first input
b , // Second input
ci , // Carry input
sum , // sum output
co // carry output
);
//Input declaration
input a;
```

```
input b;
input ci;
//Ouput declaration
output sum;
output co;
//Port Data types
wire a;
wire b;
wire ci;
wire sum;
wire co;
//Code starts here
assign {co,sum} = a + b + ci;

endmodule // End of Module addbit
```

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 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 a design.

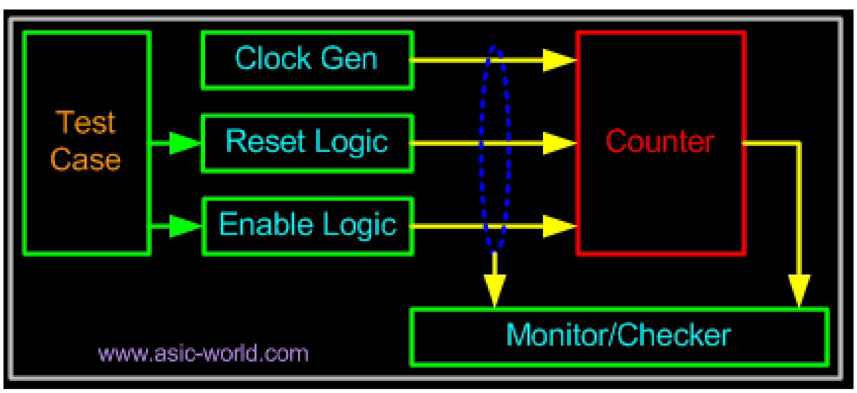


Figure : Sample Testbench Env

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.**

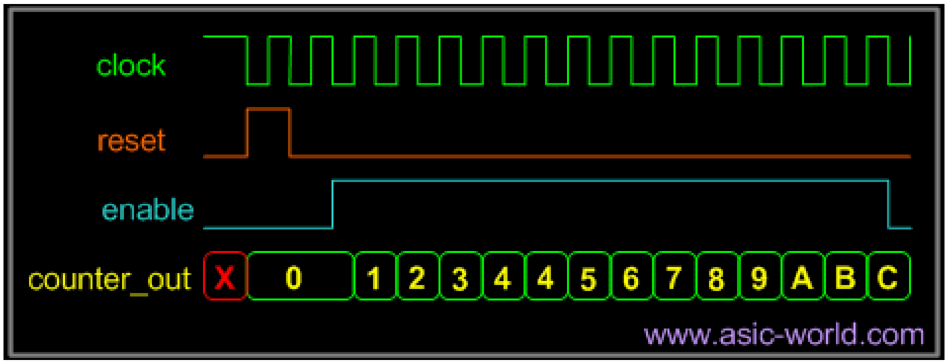


Figure : 4 bit Up Counter Waveform

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.

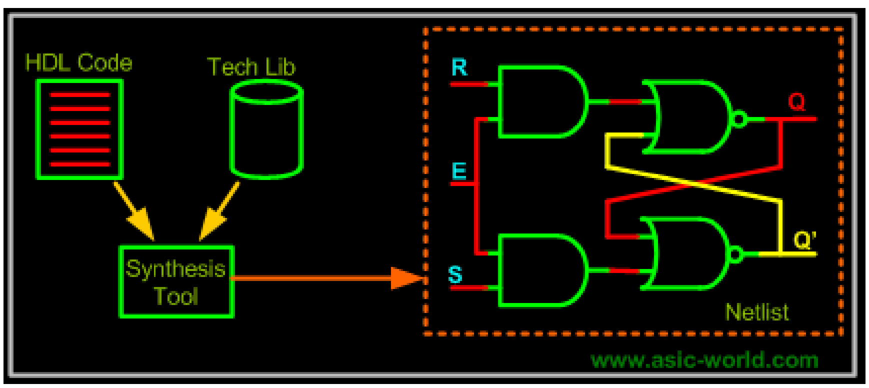


Figure : Synthesis Flow

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 placed, 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. Backend team normally dumps out SPEF (standard parasitic exchange format) /RSPF (reduced parasitic exchange format)/DSPF (detailed parasitic exchange format) from laylout tool like ASTRO to the frontend team, who then use read_parasitic command in tool like prime time to write out SDF (standard delay format) for gate level simulation purpose.

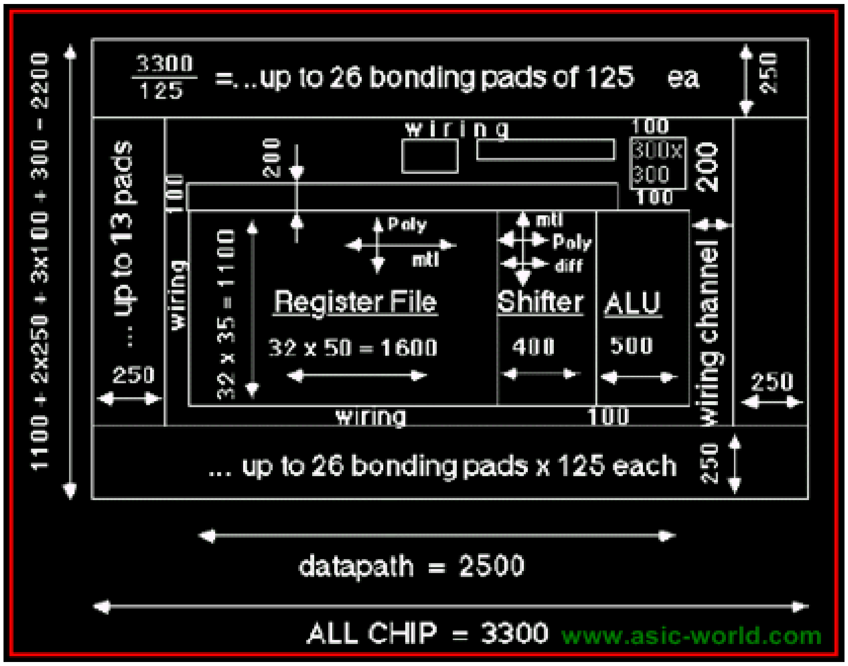


Figure : Sample micro–processor placement

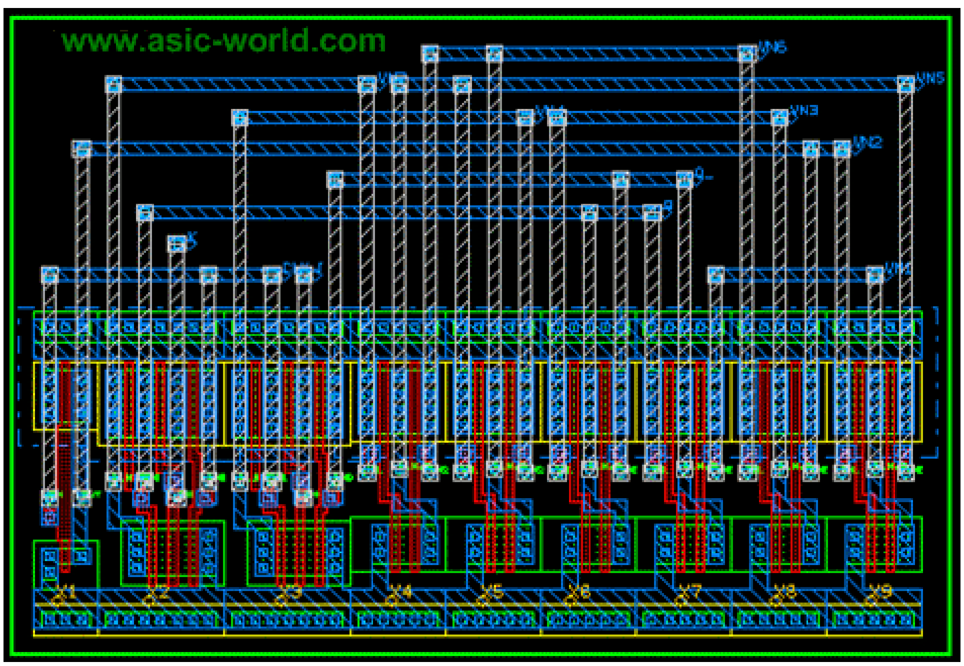


Figure : J–K Flip–Flop

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 5

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
```

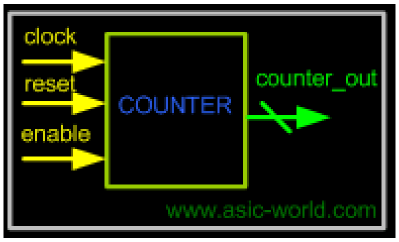
Words in green are comments, blue are reserved words, Any program in Verilog starts with reserved word module , In the above example line 8 contains module hello_world. (Note: We can have compiler pre-processor statements like `include, `define statements before module declaration)

Line 10 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 10 and end at line 13. In Verilog if you have multiple lines within a block, you need to use begin and end. Module ends with endmodule reserve mode, in this case line 15.

Hello World Program Output

```
Hello World by Deepak
```

Counter Design Block



Counter Design Specs

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

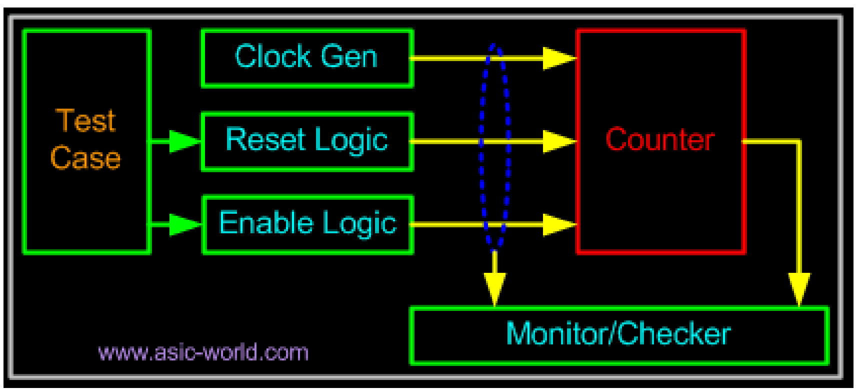
Counter Design

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

//––––––––––––Code Starts Here–––––––––––––––––––––––––
// Since this counter is a positive edge trigged one,
// We trigger the below block with respect to positive
// edge of the clock.
always @ (posedge clock)
begin : COUNTER // Block Name
   // At every rising edge of clock we check if reset is active
```

```
  // If active, we load the counter output with 4'b0000
  if (reset == 1'b1) begin
    counter_out <= #1 4'b0000;
  end
  // If enable is active, then we increment the counter
  else if (enable == 1'b1) begin
    counter_out <= #1 counter_out + 1;
  end
end // End of Block COUNTER

endmodule // End of Module counter
```

Counter Test Bench

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



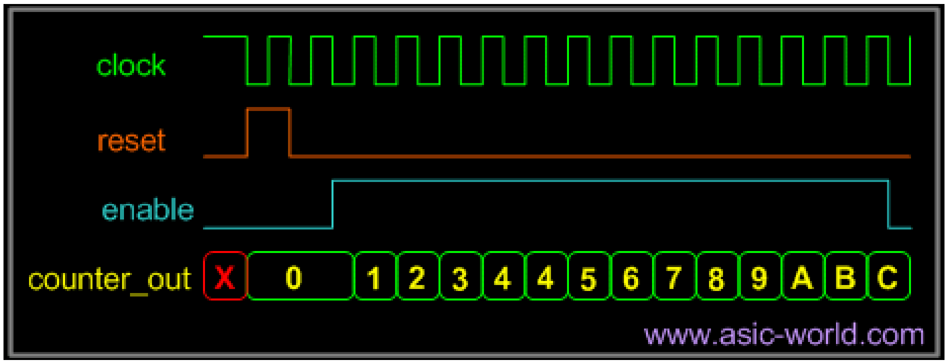
Counter testbench consists of clock generator, reset control, enable control and monitor/checker logic. Below is the simple code of testbench without the monitor/checker logic.

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

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

```
   #5 enable = 1; // Assert enable
   #100 enable = 0; // De−assert enable
   #10 $finish; // Terminate simulation
end

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

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

endmodule
```

```
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 6

Lexical Conventions

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

White Space

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

White space characters are :

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

Examples of White Spaces

Functional Equivalent Code

Bad Code : Never write code like this.

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

Good Code : Nice way to write code.

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

endmodule
```

Comments

There are two forms to introduce comments.

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

Examples of Comments

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

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

endmodule
```

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 want to use a identifier to start with a numeric value then use a escape character as shown below.

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

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

```

endmodule
```

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

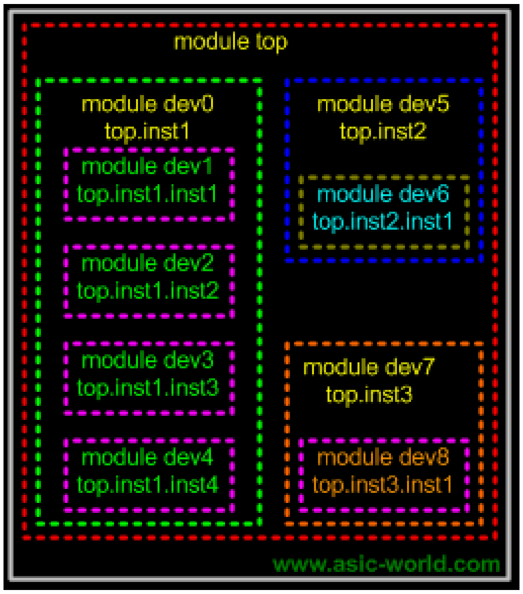
Below example file show how Verilog treats signed and unsigned numbers.

```
//*****************************************
// Signed Number Example
//
// Written by Deepak Kumar Tala
//*****************************************
module signed_number;

reg [31:0] a;

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

endmodule
```

```
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

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

Examples : A complete Example in Verilog

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

endmodule // End of Module addbit
```

Modules connected by port order (implicit)

Here order should match correctly. Normally it not a good idea to connect ports implicit. Could cause problem in debug (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 Kumar Tala
//-----------------------------------------------------
module adder_implicit (
result , // Output of the adder
carry , // Carry output of adder
r1 , // first input
r2 , // second input
ci // carry input
);

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

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

// Port Wires
```

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

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

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

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

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

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

endmodule // End Of Module adder
```

Modules connect by name

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

```
//-----------------------------------------------------
// This is simple adder Program
// Design Name : adder_explicit
// File Name : adder_explicit.v
// Function : Here the name should match
// with the leaf module, the order is not important.
// Coder : Deepak Kumar Tala
//-----------------------------------------------------
module adder_explicit (
result , // Output of the adder
```

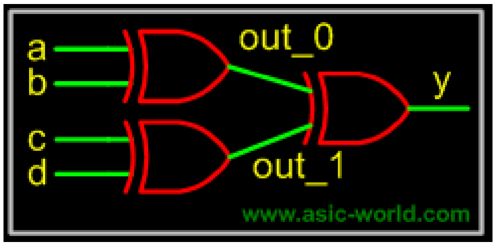
```
carry , // Carry output of adder
r1 , // first input
r2 , // second input
ci // carry input
);

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

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

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

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

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

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

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

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

endmodule // End Of Module adder
```

Instantiating a module

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

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

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

xor u1 (out_1,c,d);

xor u2 (y,out_0,out_1);

endmodule // End Of Module parity
```

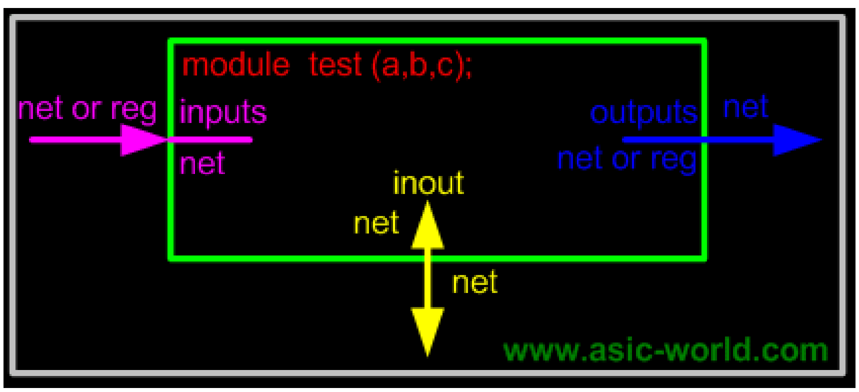
Question : What is difference between u0 in module adder and u0 in module parity?

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 Unconnected Port

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

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

endmodule

// D fli-flop
```

```
module dff (q, q_bar, clk, d, rst, pre);
input clk, d, rst, pre;
output q, q_bar;
reg q;

assign q_bar = ~q;

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

endmodule
```

Example – Explicit Unconnected Port

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

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

endmodule

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

assign q_bar = ~q;

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

endmodule
```

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

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

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

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

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

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

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

endmodule // End Of Module adder

```

```
module tb();

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

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

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

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

endmodule
```

```
+--------------------------------------------------------+
| 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,supply1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	Retains last value, when driven by z (tristate).

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

Example – wor

```
module test_wor();

wor a;
reg b, c;

assign a = b;
assign a = c;

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

endmodule
```

Simulator Output

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

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

Example – wand

```
module test_wand();

wand a;
reg b, c;

assign a = b;
assign a = c;

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

endmodule
```

Simulator Output

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

Example – tri

```
module test_tri();

tri a;
reg b, c;

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

initial begin
   $monitor( "%g a = %b b = %b c = %b" , $time, a, b, c);
   b  = 0;
```

```
    c  = 0;
    #1 b = 1;
    #1 b = 0;
    #1 c = 1;
    #1 b = 1;
    #1 b = 0;
    #1 $finish;
end

endmodule
```

Simulator Output

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

Example – trireg

```
module test_trireg();

trireg a;
reg b, c;

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

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

endmodule
```

Simulator Output

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

Register Data Types

* Registers store the last value assigned to them until another assignment statement changes their value.
* Registers represent data storage constructs.
* You can create 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

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

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

endmodule
```

```
This is sample string
```

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

GATE LEVEL MODELING

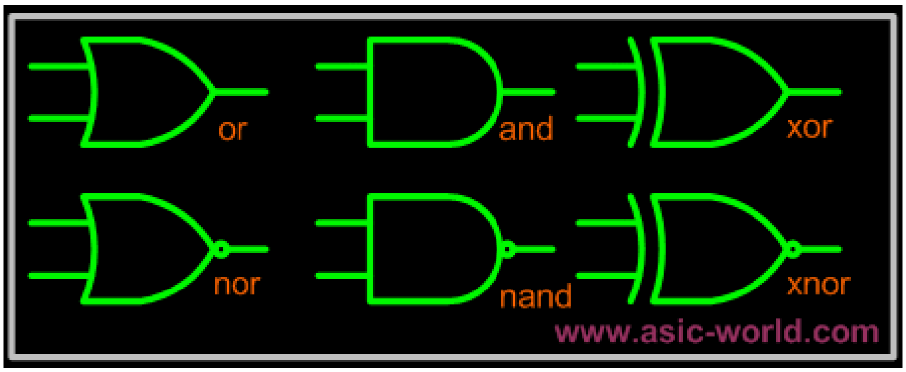
CHAPTER 7

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. These are rarely used in design (RTL Coding), but are used in post synthesis world for modeling the ASIC/FPGA cells, 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.

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

Gate Primitives



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

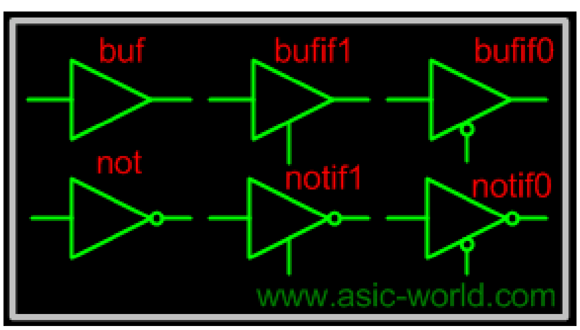
Gate	Description
and	N−input AND gate
nand	N−input NAND gate
or	N−input OR gate
nor	N−input NOR gate
xor	N−input XOR gate
xnor	N−input XNOR gate

Examples

```
module gates();

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

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

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

endmodule
```

```
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



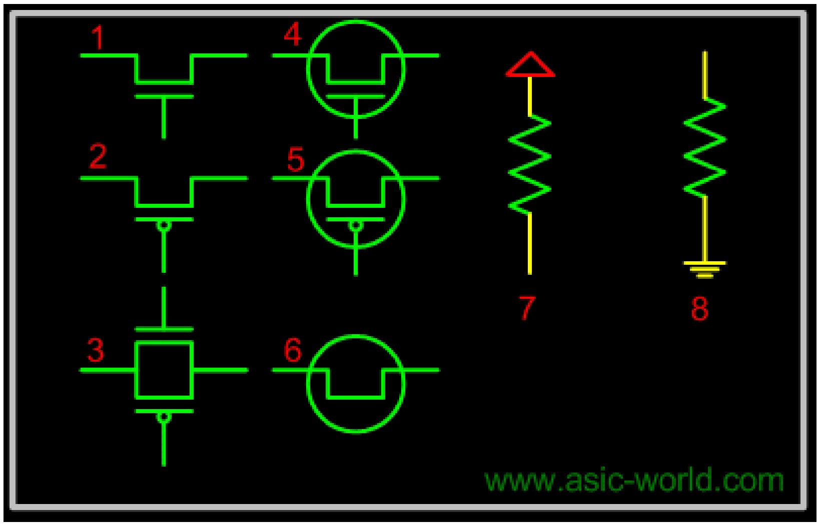
Gate	Description
not	N–output inverter
buf	N–output buffer.
bufif0	Tri–state buffer, Active low en.
bufif1	Tri–state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Examples

```
module transmission_gates();

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

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

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

always #2 in = ~in;

endmodule
```

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

Switch Primitives



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

```
module switch_primitives();

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

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

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

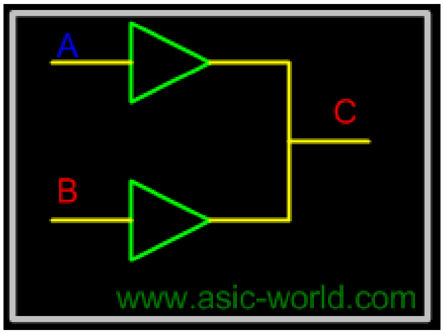
The Verilog HDL has got four logic values

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



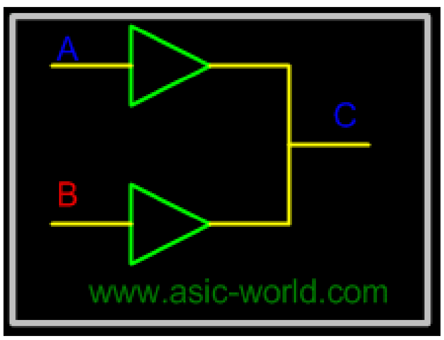
Two buffers that has output

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

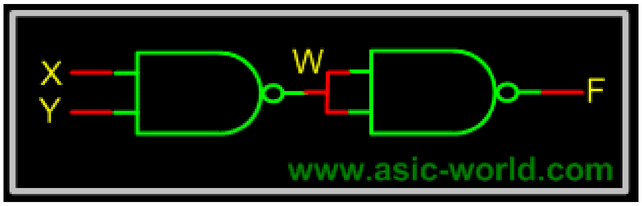
B : Large 1

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

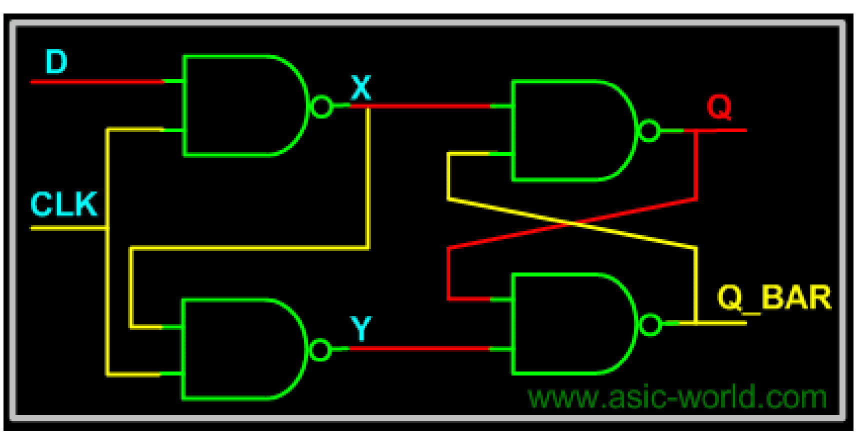


Code

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

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

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

endmodule
```

```
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

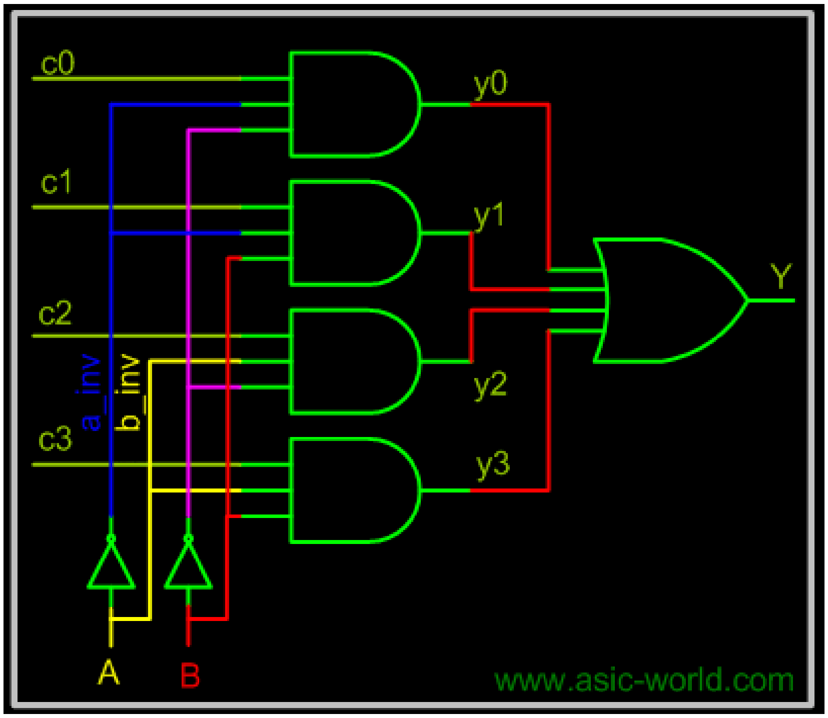


Verilog Code

```
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

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

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

```

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

endmodule
```

```
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.

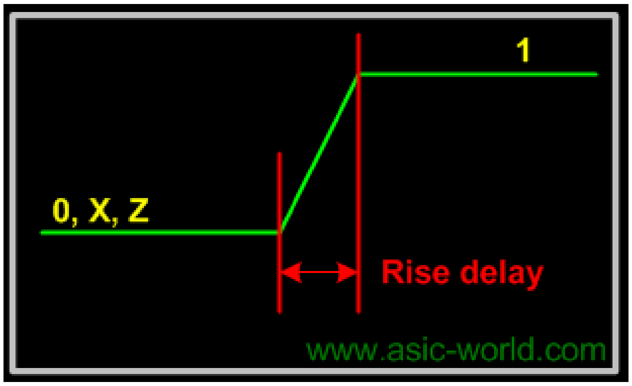
- Rise, Fall and Turn−off delays.
- Minimal, Typical, and Maximum delays.

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

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

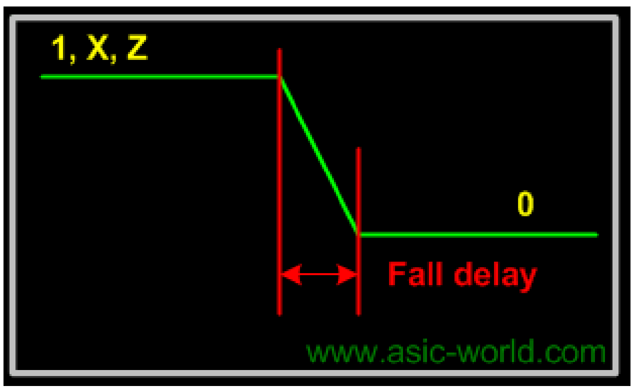
Rise Delay

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



Fall Delay

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



Turn–off Delay

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

Min Value

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

Typ Value

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

Max Value

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

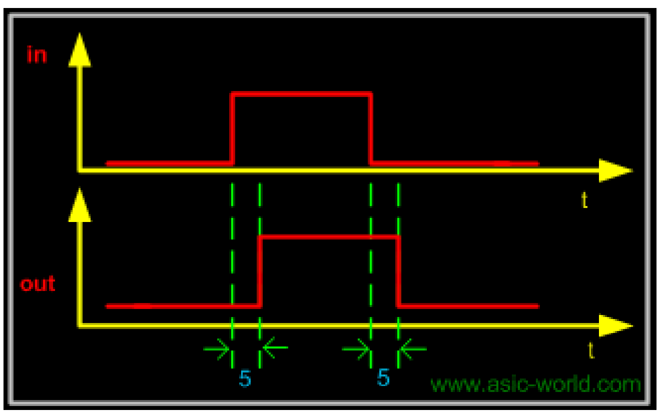
Example

Below are few examples to show the usage of delays.

Example – Single Delay

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

buf #(5) (out,in);

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

endmodule
```

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

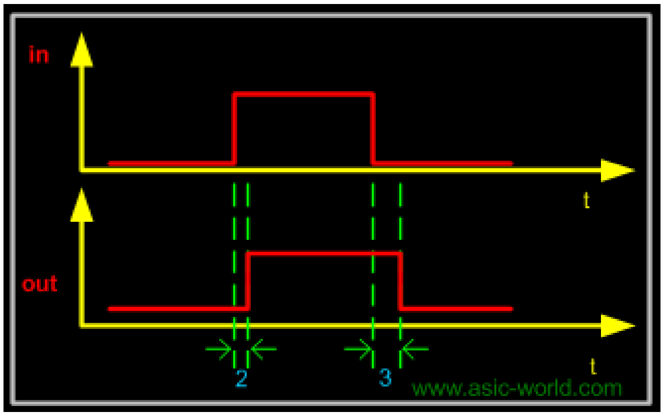


Example – Two Delays

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

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

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

```

```
endmodule
```

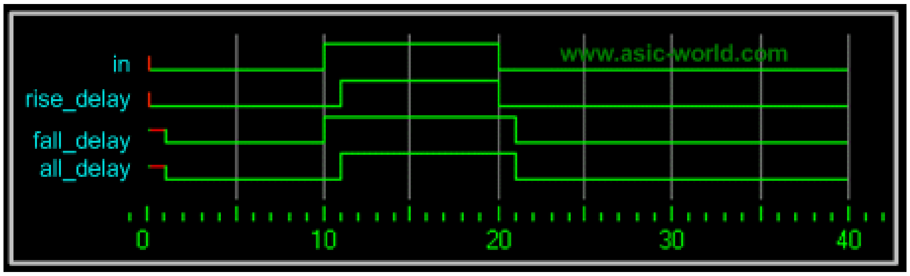
```
Time = 0 in = 0 out=x
Time = 3 in = 0 out=0
Time = 10 in = 1 out=0
Time = 12 in = 1 out=1
Time = 20 in = 0 out=1
Time = 23 in = 0 out=0
```



Example – All Delays

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

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

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

endmodule
```

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



Example – Complex Example

```
module delay_example();

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

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

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

endmodule
```

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

```
Time = 30 b = 0 c=1  out1=1 out2=1 out3=0 out4=0 out5=1 out6=0
Time = 32 b = 0 c=1  out1=1 out2=0 out3=0 out4=1 out5=1 out6=1
Time = 35 b = 0 c=1  out1=1 out2=0 out3=0 out4=1 out5=0 out6=1
```

N−Input Primitives

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

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

Examples

```
module n_in_primitive();

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

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

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

endmodule
```

```
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

```
module n_out_primitive();

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

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

endmodule
```

NOTES

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

−−

USER DEFINED PRIMITIVES

CHAPTER 8

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
- Sequential 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**

```
//This code shows how input/output ports
// and primitve is declared
primitive udp_syntax (
a, // Port a
b, // Port b
c, // Port c
d // Port d
);
output a;
input b,c,d;

// UDP function code here

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.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
b, // Port b
c // Port c
);
output a;
input b,c;

// UDP function code here
// A = B | C;
table
// B C : A
? 1  : 1;
1  ?  : 1;
0 0  : 0;
endtable

endprimitive
```

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

TestBench to Check above UDP

```
`include "udp_body.v"
module udp_body_tb();

reg b,c;
wire a;

udp_body udp (a,b,c);

initial begin
   $monitor( " B = %b C = %b A = %b" ,b,c,a);
   b = 0;
   c = 0;
   #1 b = 1;
   #1 b = 0;
   #1 c = 1;
   #1 b = 1'bx;
   #1 c = 0;
```

```
  #1 b = 1;
  #1 c = 1'bx;
  #1 b = 0;
  #1 $finish;
end

endmodule
```

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 an 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.

```
primitive udp_initial (a,b,c);
output a;
input b,c;
reg a;
// a has value of 1 at start of sim
initial a = 1'b1;

table
// udp_initial behaviour
endtable

endprimitive
```

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

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.

```
primitive udp_combo (.....);

table
0 0 0 : 0;
...
endtable

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.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
b, // Port b
c // Port c
);
output a;
input b,c;

// UDP function code here
// A = B | C;
table
// B C : A
? 1  : 1;
1  ?  : 1;
0 0  : 0;
endtable

endprimitive
```

TestBench to Check above UDP

```
`include "udp_body.v"
module udp_body_tb();

reg b,c;
wire a;

udp_body udp (a,b,c);

initial begin
   $monitor( " B = %b C = %b A = %b" ,b,c,a);
   b = 0;
   c = 0;
   #1 b = 1;
   #1 b = 0;
   #1 c = 1;
   #1 b = 1'bx;
   #1 c = 0;
   #1 b = 1;
   #1 c = 1'bx;
   #1 b = 0;
   #1 $finish;
end

endmodule
```

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 Sensitive 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.

```
primitive udp_seq (.....);

table
0 0 0 : 0 : 0;
...
endtable

endprimitive
```

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

endprimitive
```

Example UDP with initial

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

reg q;

initial begin
   q = 0;
end

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

endprimitive
```

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

VERILOG OPERATORS

CHAPTER 9

Arithmetic Operators

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

Example

```
module arithmetic_operators();

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

endmodule
```

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

Relational Operators

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

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

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

Example

```
module relational_operators();

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

endmodule
```

```
 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

```
module equality_operators();

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

endmodule
```

```
 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

```
module logical_operators();

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

endmodule
```

```
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

```
module bitwise_operators();

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

endmodule
```

```
~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

```
module reduction_operators();

initial begin
  // Bit Wise AND reduction
  $display ( " & 4'b1001 = %b" , (& 4'b1001));
  $display ( " & 4'bx111 = %b" , (& 4'bx111));
  $display ( " & 4'bz111 = %b" , (& 4'bz111));
  // Bit Wise NAND reduction
  $display ( " ~& 4'b1001 = %b" , (~& 4'b1001));
  $display ( " ~& 4'bx001 = %b" , (~& 4'bx001));
  $display ( " ~& 4'bz001 = %b" , (~& 4'bz001));
  // Bit Wise OR reduction
  $display ( " | 4'b1001 = %b" , (| 4'b1001));
  $display ( " | 4'bx000 = %b" , (| 4'bx000));
  $display ( " | 4'bz000 = %b" , (| 4'bz000));
  // Bit Wise OR reduction
  $display ( " ~| 4'b1001 = %b" , (~| 4'b1001));
  $display ( " ~| 4'bx001 = %b" , (~| 4'bx001));
  $display ( " ~| 4'bz001 = %b" , (~| 4'bz001));
  // Bit Wise XOR reduction
  $display ( " ^ 4'b1001 = %b" , (^ 4'b1001));
```

```
  $display ( " ^ 4'bx001 = %b" , (^ 4'bx001));
  $display ( " ^ 4'bz001 = %b" , (^ 4'bz001));
  // Bit Wise XNOR
  $display ( " ~^ 4'b1001 = %b" , (~^ 4'b1001));
  $display ( " ~^ 4'bx001 = %b" , (~^ 4'bx001));
  $display ( " ~^ 4'bz001 = %b" , (~^ 4'bz001));
  #10 $finish;
end

endmodule
```

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

Shift Operators

Operator	Description
<<	left shift
>>	right shift

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

Example

```
module shift_operators();

initial begin
  // Left Shift
  $display ( " 4'b1001 << 1 = %b" , (4'b1001 << 1));
  $display ( " 4'b10x1 << 1 = %b" , (4'b10x1 << 1));
  $display ( " 4'b10z1 << 1 = %b" , (4'b10z1 << 1));
  // Right Shift
```

```
   $display ( " 4'b1001 >> 1 = %b" , (4'b1001 >> 1));
   $display ( " 4'b10x1 >> 1 = %b" , (4'b10x1 >> 1));
   $display ( " 4'b10z1 >> 1 = %b" , (4'b10z1 >> 1));
   #10 $finish;
end

endmodule
```

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

Concatenation Operator

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

Example

```
module concatenation_operator();

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

endmodule
```

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

Replication Operator

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

```
module replication_operator();

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

endmodule
```

```
{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

```
module conditional_operator();

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

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

```
   #10 $finish;
end

endmodule
```

```
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 10

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

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

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

endmodule
```

In the above example, the initial block execution and always block execution starts at time 0. Always blocks waits for 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

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

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

endmodule
```

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

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

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

endmodule
```

Example − Good procedural assignment

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

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

```
endmodule
```

Procedural Assignment Groups

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

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

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

Example – "begin–end"

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

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

endmodule
```

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.

Simulator Output

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

Example – "fork–join"

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

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

endmodule
```

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.

Simulator Output

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

Sequential Statement Groups

The **begin − end** keywords:

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

```
module sequential();

reg a;

initial begin
  $monitor ( "%g a = %b" , $time, a);
  #10 a = 0;
  #11 a = 1;
```

```
  #12 a = 0;
  #13 a = 1;
  #14 $finish;
end

endmodule
```

Simulator Output

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

Parallel Statement Groups

The fork – join keywords:

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

Example – Parallel

```
module parallel();

reg a;

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

endmodule
```

Simulator Output

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

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

```
module fork_join();

reg clk,reset,enable,data;

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

endmodule
```

Simulator Output

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

Blocking and Nonblocking assignment

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

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

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

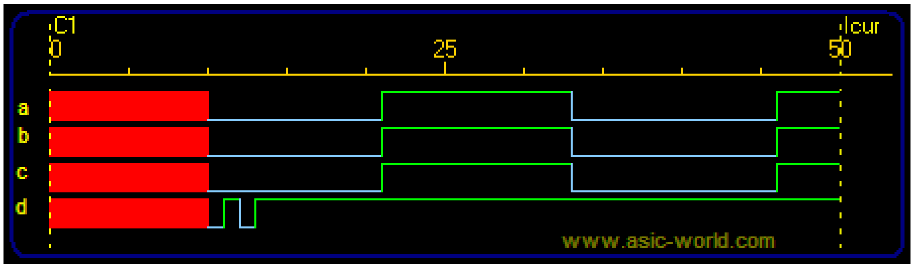
Example – blocking and nonblocking

```
module blocking_nonblocking();

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

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

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

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

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

endmodule
```

Simulator Output

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

Waveform



assign and deassign

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

Example – assign and deassign

```
module assign_deassign ();

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

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

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

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

```
endmodule

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

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

endmodule
```

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
@1  clk 1 rst 0 preset 0 d 0 q 0
@2  clk 0 rst 0 preset 0 d 0 q 0
@3  clk 1 rst 0 preset 0 d 0 q 0
@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 1
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```

force and release

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

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

Example − force and release

```
module force_release ();

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

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

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

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

endmodule

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

```
wire q;
reg q_reg;

assign q = q_reg;

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

endmodule
```

Simulator Output

```
@0  clk 0 rst 0 preset 0 d 0 q x
@1  clk 1 rst 0 preset 0 d 0 q 0
@2  clk 0 rst 0 preset 0 d 0 q 0
@3  clk 1 rst 0 preset 0 d 0 q 0
@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 0
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
```

The Conditional Statement if-else

The if – else statement controls the execution of other statements, In programming language like c, if – else controls the flow of program. When more than 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

```
module simple_if();

reg latch;
wire enable,din;

always @ (enable or din)
if (enable) begin
   latch <= din;
end

endmodule
```

Example- if-else

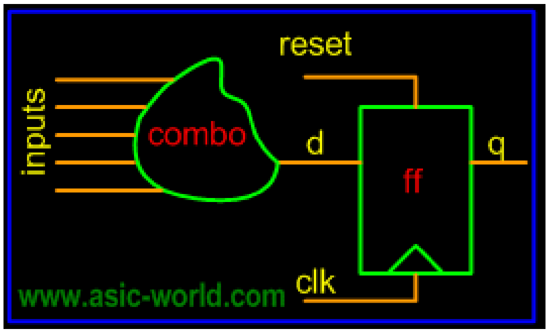
```
module if_else();

reg dff;
wire clk,din,reset;

always @ (posedge clk)
if (reset) begin
  dff <= 0;
end else begin
  dff <= din;
end

endmodule
```

Example– nested–if–else–if

```
module nested_if();

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

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

endmodule
```

Parallel if–else

In the above example, the (enable == 1'b1 && up_en == 1'b1) is given highest priority 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 nested if−else statements. 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.

```
module parallel_if();

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

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

endmodule
```

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.

Syntax of a case statement look as shown below.

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

Normal Case

Example– case

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

reg y;

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

endmodule
```

Example– case without default

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

reg y;

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

endmodule
```

Above example shows how to specify 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

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

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

endmodule
```

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

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

always @ (opcode or a or b or c)
casez(opcode)
  // Bit 0 is matched with "x"
  4'b1zzx : out = a; // Don't care about lower 3:1 bits
  4'b01?? : out = b; // The ? is same as z in a number
  4'b001? : out = c;
  default : $display ( "Error xxxx does matches 0000" );
endcase

endmodule
```

Example– casex

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

always @ (opcode or a or b or c)
casex(opcode)
  4'b1zzx : out = a; // Don't care 3:0 bits
  4'b01?? : out = b; // The ? is same as z in a number
  4'b001? : out = c;
  default : $display ( "Error xxxx does matches 0000" );
endcase

endmodule
```

Example– Comparing case, casex, casez

```
module case_compare;

reg sel;

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

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

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

always @ (sel)
casez (sel)
  1'b0 : $display( "CASEZ : Logic 0 on sel" );
  1'b1 : $display( "CASEZ : Logic 1 on sel" );
```

```
  1'bx : $display( "CASEZ : Logic x on sel" );
  1'bz : $display( "CASEZ : Logic z on sel" );
endcase

endmodule
```

Simulation Output

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

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

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

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

Looping Statements

Looping statements appear inside 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

```
module forever_example ();

reg clk;

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

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

endmodule
```

The repeat statement

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

syntax : repeat (< number >) < statement >

Example– repeat

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

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

```
end

endmodule
```

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;
  #1 $finish;
end

endmodule
```

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

```
module for_example();

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

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

endmodule
```

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

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

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

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

```
  b = 0;
  #10 a = 1;
  #10 b = 1;
  #10 a = 0;
  #10 b = 0;
  #10 $finish;
end

endmodule
```

Example – Tri–state buffer

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

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

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

endmodule
```

Propagation Delay

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

Example – Tri–state buffer

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

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

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

```
end

endmodule
```

Procedural Block Control

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

```
module dlatch_using_always();
reg q;

reg d, enable;

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

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

endmodule
```

An event sensitive delay at the beginning 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 modeling 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 initialize all the variables of that combo block to zero as soon as it enters.

Example – One bit Adder

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

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

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

endmodule
```

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

Example – 4–bit Adder

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

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

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

endmodule
```

Example – Ways to avoid Latches – Cover all conditions

```
module avoid_latch_else ();

reg q;
reg enable, d;

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

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

endmodule
```

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

```
module avoid_latch_init ();

reg q;
reg enable, d;

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

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

```
   #1 $finish;
end

endmodule
```

Sequential Logic using Procedural Coding

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

Sometimes it 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

```
module wrong_seq();

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

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

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

always
#1 clk1 = ~clk1;

always
#1.9 clk2 = ~clk2;

endmodule
```

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

```
module dff_async_reset_async_preset();

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

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

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

always
#1 clk = ~clk;

endmodule
```

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 drives.

```
module trigger_itself();

reg clk;

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

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

endmodule
```

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

```
module race_condition();
reg b;

initial begin
  b = 0;
end

initial begin
  b = 1;
end

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

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

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

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

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

endmodule
```

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 11

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

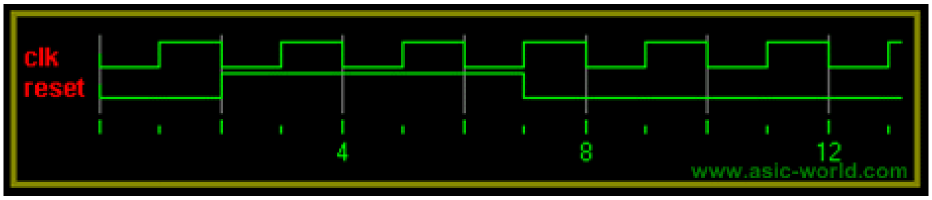
```
module clk_gen ();

reg clk, reset;

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

always
#1 clk = !clk;

endmodule
```

Simulation Output

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

```
TIME = 16 RESET = 0 CLOCK = 0
```

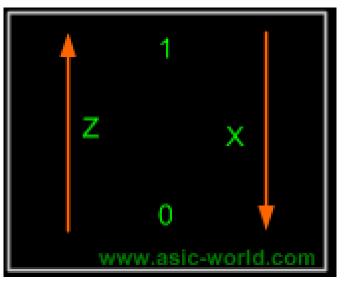
Waveform



Edge sensitive Event Controls

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

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



Example – Edge Wait

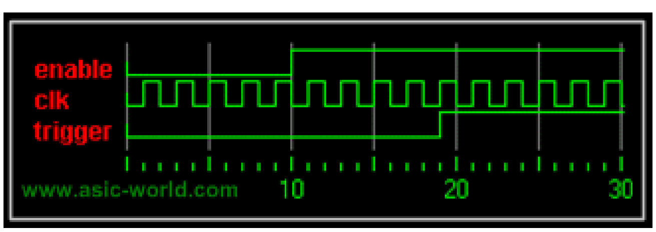
```
module edge_wait_example();

reg enable, clk, trigger;

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

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

```
   #1 enable = 0;
   #10 $finish;
end

always
#1 clk = ~clk;

endmodule
```

Simulator Output

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

Waveform



Level–Sensitive Even Controls (Wait statements)

Delays execution of the next statement until the evaluates as true

syntax : wait () ;

Example – Level Wait

```
module wait_example();

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

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

// Testbench Code here
initial begin
  $monitor ( "TIME = %g READ = %b READY = %b DATA = %b" , $time, mem_read, data_ready, data);
  data_bus = 0;
  mem_read = 0;
  data_ready = 0;
  #10 data_bus = 8'hDE;
  #10 mem_read = 1;
  #20 data_ready = 1;
  #1 mem_read = 1;
  #1 data_ready = 0;
  #10 data_bus = 8'hAD;
  #10 mem_read = 1;
  #20 data_ready = 1;
  #1 mem_read = 1;
  #1 data_ready = 0;
  #10 $finish;
end

endmodule
```

Simulator Output

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

Intra–Assignment Timing Controls

Intra–assignment controls 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.

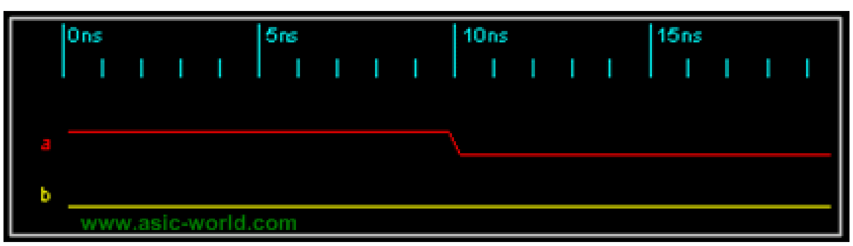
Example − Intra−Assignment

```
module intra_assign();

reg a, b;

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

endmodule
```

Simulation Output

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

Waveform



Modeling Combo Logic with Continuous Assignments

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

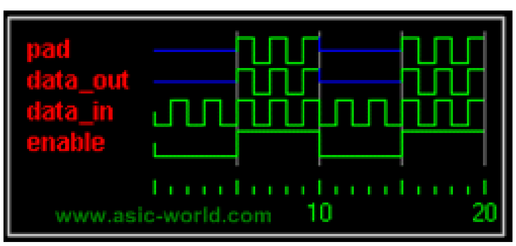
Example − Tri−state Buffer

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

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

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

endmodule
```

Simulation Output

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

Waveform



Example – Mux

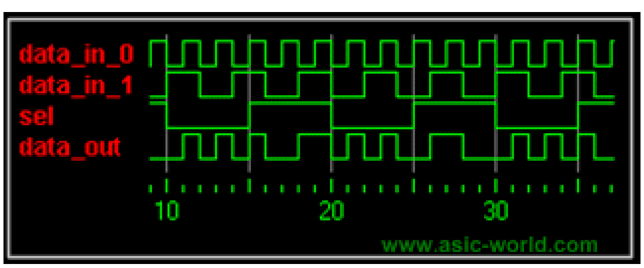
```
module mux_using_assign();
reg data_in_0, data_in_1;
wire data_out;
reg sel;

assign data_out = (sel) ? data_in_1 : data_in_0;

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

```
  sel = 0;
  #10 sel = 1;
  #10 $finish;
end

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

// Toggel data_in_1 at %1.5
always
#1.3 data_in_1 = ~data_in_1;

endmodule
```

Simulation Output

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

Waveform



NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

TASK AND FUNCTIONS

CHAPTER 12

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

```
module simple_task();

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

endmodule
```

Example – Task using Global Variables

```
module task_global();

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

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

endmodule
```

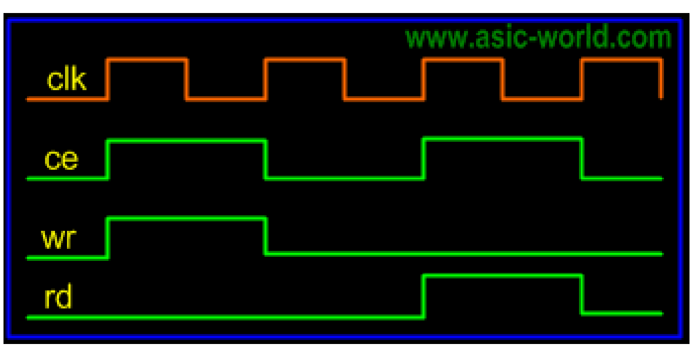
Calling a Task

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

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

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

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

endmodule
```

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.



```
module bus_wr_rd_task();

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

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

```
  @ (posedge clk);
  addr = address;
  ce = 1;
  wr = 1;
  data_wr = data;
  @ (posedge clk);
  addr = 0;
  ce = 0;
  wr = 0;
  $display ( "======================" );
end
endtask

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

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

endmodule
```

Simulation Output

```
1 CPU Write task with address : 11 Data : aa
1  -> Driving CE, WR, WR data and ADDRESS on to bus
======================
4 CPU Read  task with address : 11
4  -> Driving CE, RD and ADDRESS on to bus
7 CPU Read  data              : aa
======================
8 CPU Write task with address : 12 Data : ab
8  -> Driving CE, WR, WR data and ADDRESS on to bus
======================
12 CPU Read  task with address : 12
12  -> Driving CE, RD and ADDRESS on to bus
15 CPU Read  data              : ab
======================
16 CPU Write task with address : 13 Data : 0a
16  -> Driving CE, WR, WR data and ADDRESS on to bus
======================
20 CPU Read  task with address : 13
20  -> Driving CE, RD and ADDRESS on to bus
23 CPU Read  data              : 0a
======================
```

Function

A Verilog HDL function is same as task, with very little difference, like function cannot drive more than 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

```
module simple_function();

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

endmodule
```

Example – Calling a Function

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

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

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

endmodule
```

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

SYSTEM TASK AND FUNCTION

CHAPTER 13

Introduction

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

$display, $strobe, $monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like GTKWave. or Undertow or Debussy. $display and $strobe display once every time they are executed, where as $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.vcd")
- $dumpvar dumps all variables in the design.
- $dumpvar(1, top) dumps all the variables in module top and below, but not modules instantiated in top.
- $dumpvar(2, top) dumps all the variables in module top and 1 level below.
- $dumpvar(n, top) dumps all the variables in module top and n−1 levels below.
- $dumpvar(0, top) dumps all the variables in module top and all level below.
- $dumpon initiates the dump.
- $dumpoff stop dumping.

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

These commands write more selectively to files.

- $fopen opens an output file and gives the open file a handle for use by the other commands.
- $fclose closes the file and lets other programs access it.
- $fdisplay and $fwrite write formatted data to a file whenever they are executed. They are the same except $fdisplay inserts a new line after every execution and $write does not.
- $strobe also writes to a file when executed, but it waits until all other operations in the time step are complete before writing. Thus initial #1 a=1; b=0; $fstrobe(hand1, a,b); b=1; will write write 1 1 for a and b.
- $monitor writes to a file whenever any 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 14

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

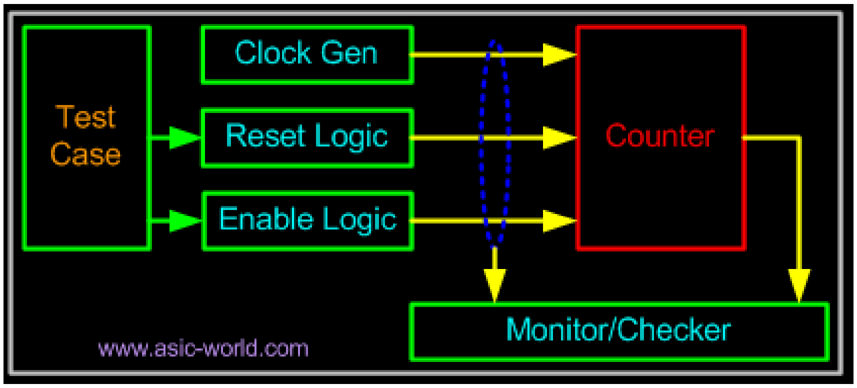
Lets assume that we have to verify a simple 4−bit up counter, which increments its count when ever enable is high and resets to zero, when reset is asserted high. Reset is synchronous to clock.

Code for Counter

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

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

endmodule
```

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

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

Writing TestBench

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

Test Bench

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

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

endmodule
```

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

Test Bench with Clock gen

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

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

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

always
#5 clk = !clk;

endmodule
```

Initial block in verilog is executed only once, thus simulator sets the value of clk, reset and enable to 0, which by looking at the counter code (of course you will be referring to 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...

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

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

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

always
#5 clk = !clk;

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

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

initial
#100 $finish;

//Rest of testbench code after this line

endmodule
```

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

$display is used for printing text or variables to stdout (screen), \t is for inserting tab. Syntax is same as printf. Second line $monitor is bit different, $monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). When ever anyone of them changes, it prints their value, in the respective radix specified.

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

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

```
C:\www.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

```
event reset_trigger;
event reset_done_trigger;

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

Adding test case logic

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

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

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

Test Case 1 − Asserting/ De−asserting reset

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

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

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

In this test case, we will trigger the reset logic and wait for the reset logic to complete its operation, before we start driving 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/De–assert enable and reset randomly.

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

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

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

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

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

and the modified test case #2 would like.

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

```

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

Adding compare Logic

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

```
reg [3:0] count_compare;

always @ (posedge clk)
if (reset == 1'b1) begin
  count_compare <= 0;
end else if ( enable == 1'b1) begin
  count_compare <= count_compare + 1;
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 prints out the expected and actual value, and also terminates the simulation by triggering the event "terminate_sim".

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

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 15

Memory Modeling

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

Syntax

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

Examples

Declaration

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

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

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

Storing Values

my_memory[address] = data_in;

Reading Values

data_out = my_memory[address];

Bit Read

Sometime there may be need to just read only one bit. Unfortunately Verilog does not allow to read only or write only one bit, the work around for such a problem is as shown below.

data_out = my_memory[address];

data_out_it_0 = data_out[0];

Initializing Memories

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

Syntax

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

Note : start_addr and stop_addr are optional.

Example − Simple memory

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

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

Example − Memory.list file

```
//Comments are allowed
1100_1100 // This is first address i.e 8'h00
1010_1010 // This is second address i.e 8'h01
@ 55 // Jump to new address 8'h55
0101_1010 // This is address 8'h55
0110_1001 // This is address 8'h56
```

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

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

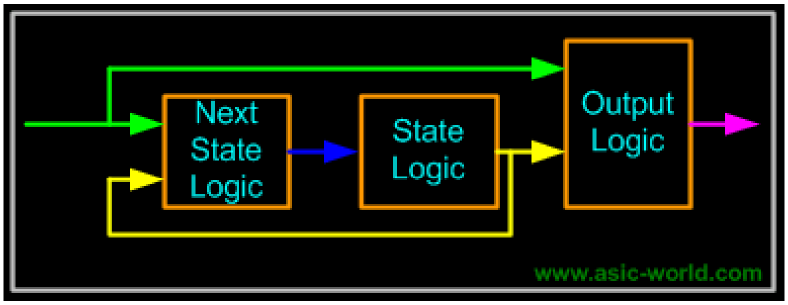
Introduction to FSM

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

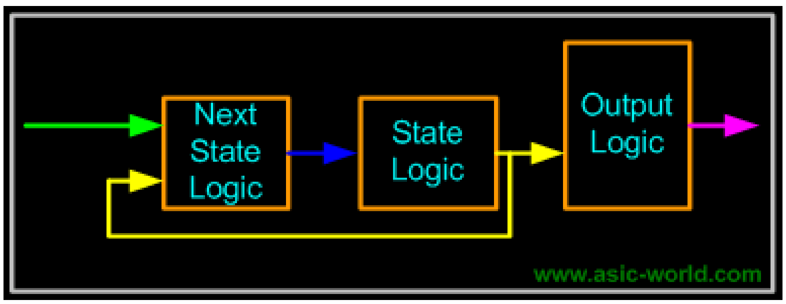
State machine Types

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

Mealy Model



Moore Model



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

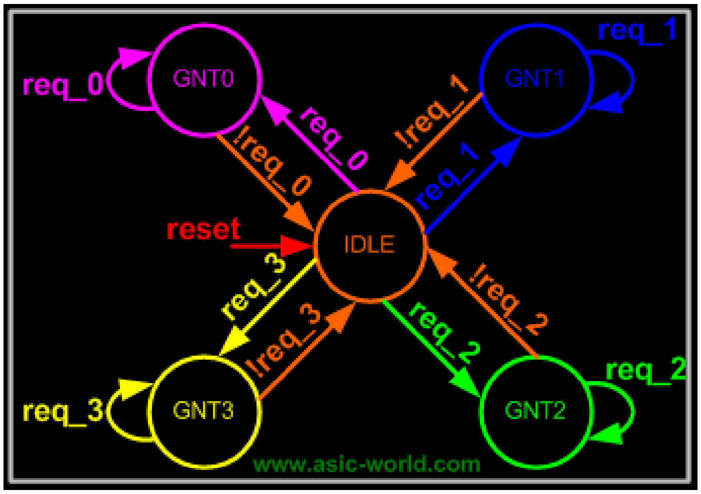
Modeling State machines.

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

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

Example − Arbiter

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



Verilog Code

FSM code should have three sections,

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

Encoding Style

There are many encoding styles around, some of which are

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

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

One Hot Encoding

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

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

```
            next_state = IDLE;
        end else begin
            next_state = GNT3;
        end
        default : next_state = IDLE;
    endcase
end
```

Sequential Section

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

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

Full Code using binary encoding

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

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

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

reg [2:0] state, next_state;

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

```
    end
    GNT0 : if (req_0 == 1'b0) begin
      next_state = IDLE;
    end else begin
      next_state = GNT0;
    end
    GNT1 : if (req_1 == 1'b0) begin
      next_state = IDLE;
    end else begin
      next_state = GNT1;
    end
    GNT2 : if (req_2 == 1'b0) begin
      next_state = IDLE;
    end else begin
      next_state = GNT2;
    end
    GNT3 : if (req_3 == 1'b0) begin
      next_state = IDLE;
    end else begin
      next_state = GNT3;
    end
    default : next_state = IDLE;
  endcase
end

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

```
        GNT3 : begin
          gnt_3 <= #1 1'b1;
        end
        default : begin
          state <= #1 IDLE;
        end
      endcase
    end
end

endmodule
```

Testbench

```
`include "fsm_full.v"

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

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

always
#2 clock = ~clock;


fsm_full U_fsm_full(
clock , // Clock
reset , // Active high reset
req_0 , // Active high request from agent 0
req_1 , // Active high request from agent 1
req_2 , // Active high request from agent 2
req_3 , // Active high request from agent 3
```

```
gnt_0 , // Active high grant to agent 0
gnt_1 , // Active high grant to agent 1
gnt_2 , // Active high grant to agent 2
gnt_3 // Active high grant to agent 3
);



endmodule
```

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 16

Introduction

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

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

We can override the default values with either using defparam or by passing new set of parameters during instantiating. We call this as parameter over riding.

Parameters

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

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

Parameter Override using defparam

```
module secret_number;
parameter my_secret = 0;

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

endmodule

module defparam_example();

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

secret_number U0();
secret_number U1();

endmodule
```

Parameter Override during instantiating.

```
module secret_number;
parameter my_secret = 0;

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

endmodule

module param_overide_instance_example();

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

endmodule
```

Passing more than one parameter

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

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

endmodule
```

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

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

// Controller Code

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

endmodule
```

Verilog 2001

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

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

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

endmodule
```

Was this copied from VHDL?

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

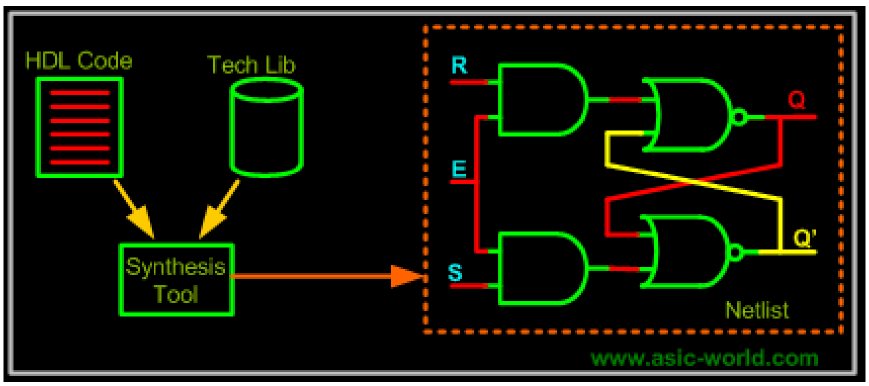
VERILOG SYNTHESIS TUTORIAL
CHAPTER 17

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 than one always blocks. Well it will surely cause synthesis error.

Example − Initial Statement

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

initial begin
   q <= 0;
end

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

```
end

endmodule
```

Delays

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

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

a = b;

Comparison to X and Z are always ignored

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

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

endmodule
```

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, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
named Blocks	disable	Disabling of named block supported.
loops	for, while, forever	While and forever loops must contain @(posedge clk) or @(negedge clk)

Operators and their Effect.

One common problem that seems to occur, 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

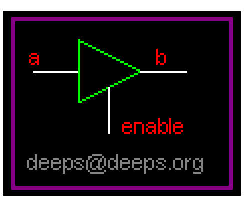
From what we have learn in digital design, we know that there could be only two types of digital circuits. One is combinational circuits and second is sequential circuits. There are very few rules that need to be followed to get good synthesis output and avoid surprises.

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

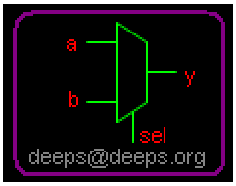
```
assign y = (a&b) | (c^d);
```

Tri-state buffer



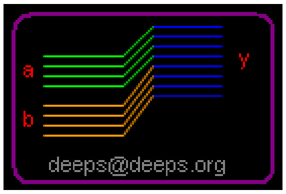

```
module tri_buf (a,b,enable);
input a;
output b;
input enable;
wire b;

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

endmodule
```

Mux




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

assign y = (sel) ? b : a;

endmodule
```

Simple Concatenation




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

assign y = {a,b};

endmodule
```

1 bit adder with carry

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

```
wire a;
wire b;
wire ci;
wire sum;
wire co;
//Code starts here
assign {co,sum} = a + b + ci;

endmodule // End of Module addbit
```

Multiply by 2

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

assign product  = a << 1;

endmodule
```

3 is to 8 decoder

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

endmodule
```

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

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

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

endmodule
```

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

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

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

endmodule
```

Verilog Coding Style

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

* Use meaningful names for signals and variables
* Don't mix level and edge sensitive in one always block
* Avoid mixing positive and negative edge−triggered flip−flops
* Use parentheses to optimize logic structure
* Use continuous assign statements for simple combo logic.
* Use 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 18

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.

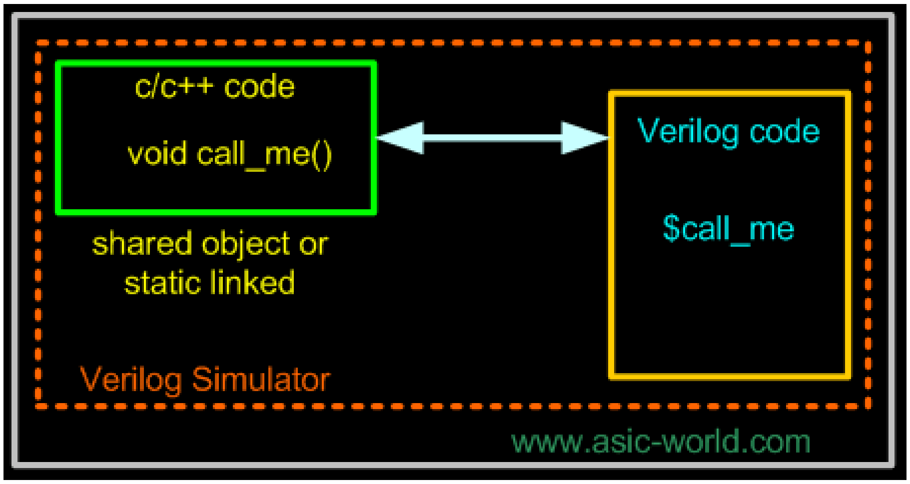
To achieve above few application of PLI, C code should have the access to the internal data structure of the Verilog simulator. To facilitate this Verilog PLI provides with something called acc routines or simply access routines.

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

```
#include <stdio.h>

void hello () {
  printf ( "\nHello Deepak\n" );
}
```

Verilog Code

```
module hello_pli ();

initial begin
  $hello;
  #10 $finish;
end

endmodule
```

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.

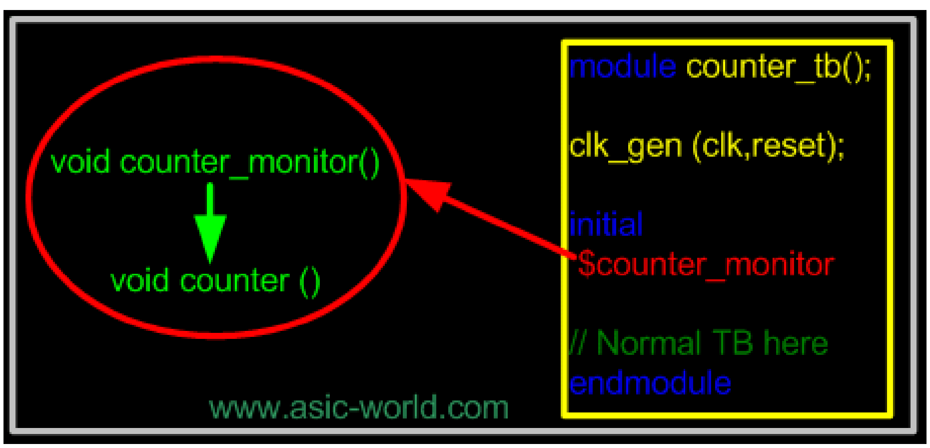
- Means of calling the C model, when ever there is change in input signals (Could be wire or reg or types).
- Means to get the value of the changes signals in Verilog code or any other signals in Verilog code from inside the C code.
- Means to drive the value on any signal inside the Verilog code from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

```
#include "acc_user.h"

handle clk ;
handle reset ;
handle enable ;
handle dut_count ;
void counter ();

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 () {
   io_printf( "Clock changed state\n" );
}
```

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 objects 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 explanation, 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.

```
module counter_tb();
reg enable;
reg reset;
reg clk_reg;
wire clk;
wire [3:0] count;

initial begin
  enable = 0;
  clk_reg = 0;
  reset = 0;
  $display( "%g , Asserting reset" , $time);
  #10 reset = 1;
  #10 reset = 0;
  $display ( "%g, Asserting Enable" , $time);
  #10 enable = 1;
  #55 enable = 0;
  $display ( "%g, Deasserting Enable" , $time);
  #1 $display ( "%g, Terminating Simulator" , $time);
  #1 $finish;
end

always begin
  #5 clk_reg = !clk_reg;
end

assign clk = clk_reg;

initial begin
  $counter_monitor(counter_tb.clk, counter_tb.reset, counter_tb.enable, counter_tb.count);
```

```
end

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

endmodule
```

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

```
0 , Asserting reset
Clock changed state
Clock changed state
Clock changed state
20, Asserting Enable
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
Clock changed state
85, Deasserting Enable
Clock changed state
86, Terminating Simulator
```

C Code – Full

So now that we see that our function gets called when 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,"format")

But the value returned is a string, so we need to convert that into integer if, multi−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.

```
#include "acc_user.h"

typedef char * string;
handle clk ;
handle reset ;
handle enable ;
handle dut_count ;
int count ;
int sim_time;
string high = "1" ;
void counter ();
int pli_conv (string in_string, int no_bits);

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 () {
  p_acc_value value;
  sim_time = tf_gettime();
  string i_reset = acc_fetch_value(reset, "%b" ,value);
  string i_enable = acc_fetch_value(enable, "%b" ,value);
  string i_count = acc_fetch_value(dut_count, "%b" ,value);
  string i_clk = acc_fetch_value(clk, "%b" ,value);
  int size_in_bits= acc_fetch_size (dut_count);
  int tb_count = 0;
  // Counter function goes here
  if (*i_reset == *high) {
    count = 0;
    io_printf( "%d, dut_info : Counter is reset\n" , sim_time);
  }
  else if ((*i_enable == *high) && (*i_clk == *high)) {
    if ( count == 15 ) {
      count = 0;
    } else {
      count = count + 1;
    }
  }
```

```
  // Counter Checker function goes checker logic goes here
  if ((*i_clk != *high) && (*i_reset != *high)) {
    tb_count = pli_conv(i_count,size_in_bits);
    if (tb_count != count) {
      io_printf( "%d, dut_error : Expect value %d, Got value %d\n" , sim_time, count, tb_count);
      tf_dofinish();
    } else {
      io_printf( "%d, dut_info : Expect value %d, Got value %d\n" , sim_time, count, tb_count);
    }
  }
}

// Multi-bit vector to integer conversion.
int pli_conv (string in_string, int no_bits) {
  int conv = 0;
  int i = 0;
  int j = 0;
  int bin = 0;
  for ( i = no_bits-1; i >= 0; i = i - 1) {
    if (*(in_string + i) == 49) {
      bin = 1;
    } else if (*(in_string + i) == 120) {
      io_printf ( "%d, Warning : X detected\n" , sim_time);
      bin = 0;
    } else if (*(in_string + i) == 122) {
      io_printf ( "%d, Warning : Z detected\n" , sim_time);
      bin = 0;
    } else {
      bin = 0;
    }
    conv = conv + (1 << j)*bin;
    j ++;
  }
  return conv;
}
```

You can compile and simulate the above code with Simulator you have as in next few paragraphs.

Linking With Simulator

The counter example we saw, we will link with below simulators, If you want to see with any other simulator let me know.

- VCS
- Modelsim

I use Linux for my tutorials, so if you want to read how to link in windows or solaris, then refer to simulator manual for details.

VCS

With VCS simulator you need to create a tab file. For our example tab file looks like below.

$counter_monitor call=counter_monitor acc=rw:*

Here $counter_monitor is name of user defined function that will be used in Verilog code, call=counter_monitor is the C function which will be called when $counter_monitor is called in Verilog. acc=rw:* is telling that we are using access routines with read and write access to simulator internal data. :* means that this should be applicable to all the modules in design.

Command line options for compiling the code is as below.

vcs −R −P pli_counter.tab pli_counter_tb.v counter.v pli_full_example.c −CFLAGS "−g −I$VCS_HOME/`vcs −platform`/lib" +acc+3

Since we are using callbacks we have to use +acc+3, rest of the options are simple, you can refer to VCS user guide.

Modelsim

Like VCS, modelsim simulator has its own way for communicating with PLI. We need to create function listing all user defined function that will be referred in verilog and corresponding C function to call. Unlike VCS, we need to do it in C file as shown below.

```
#include "veriuser.h"
#include "pli_full_example.c"

s_tfcell veriusertfs[] = {
   {usertask, 0, 0, 0, counter_monitor, 0, "$counter_monitor" },
   {0} // last entry must be 0
   };
```

vlib work

vlog pli_counter_tb.v counter.v

gcc −c −g −I$MODEL/include pli_full_example_modelsim.c

ld −shared −E −o pli_full_example.sl pli_full_example_modelsim.o

vsim −c counter_tb −pli pli_full_example.sl

In vsim command line, type "run −all" to start the simulation.

Refer to modelsim user guide for details or for how to compile and link in Windows.

Counter Simulation Output

```
0 , Asserting reset
10, dut_info : Counter is reset
15, dut_info : Counter is reset
20, Asserting Enable
20, dut_info  : Expect value 0, Got value 0
30, dut_info  : Expect value 0, Got value 0
40, dut_info  : Expect value 1, Got value 1
50, dut_info  : Expect value 2, Got value 2
60, dut_info  : Expect value 3, Got value 3
70, dut_info  : Expect value 4, Got value 4
80, dut_info  : Expect value 5, Got value 5
85, Deasserting Enable
86, Terminating Simulator
```

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()	This function returns the handle to the scope of an object. The scope can be either a module, task, function, named parallel block, or named sequential block.
acc_handle_by_name()	This routine returns the handle to a Verilog−HDL object based on the specified name and scope.
acc_handle_parent()	This function returns handle for the parent primitive instance or module instance of an object
acc_handle_port()	This function returns handle for a module port
acc_handle_hiconn()	This function returns the hierarchically higher net connection to a scalar module port or a bit of a vector port
acc_handle_loconn()	This function returns the hierarchically lower net connection to a scalar module port or a bit of a vector

	port.
acc_handle_path()	This function returns a handle to an inter−module path that represents the connection from an output port to an input port
acc_handle_modpath()	This function returns handle to the path of a module
acc_handle_datapath()	This function returns a handle to a datapath for a module instance for the specified edge−sensitive module path
acc_handle_pathin()	This function returns handle for the first net connected to a module path source
acc_handle_pathout()	This function returns handle for the first net connected to a module path destination
acc_handle_condition()	This function returns a handle to the conditional expression for the specified path
acc_handle_tchk()	This function returns handle for the specified timing check of a module (or cell)
acc_handle_tchkarg1()	This function returns handle for the net connected to the first argument of a timing check
acc_handle_tchkarg2()	This function returns handle for the net connected to the second argument of a timing check
acc_handle_simulated_net()	This function returns the simulated net associated with the collapsed net passed as an argument
acc_handle_terminal()	This function returns handle for a primitive_terminal
acc_handle_conn()	This function returns handle to the net connected to a primitive terminal
acc_handle_tfarg()	This function returns handle for the specified argument of the system task or function associated (through the PLI mechanism) with your C−language routine`
acc_fetch_attribute()	This function returns the value of a parameter or specparam named as an attribute in your source description
acc_fetch_paramtype()	This function returns the data type of a parameter as one of three predefined integer constants.
acc_fetch_paramval()	This function returns the value of a parameter or specparam
acc_fetch_defname()	This function returns a pointer to the defining name of a module instance or primitive instance
acc_fetch_fullname()	This function returns a pointer to the full hierarchical name of any named object or module path
acc_fetch_name()	This function returns a pointer to the instance name of any named object or module path
acc_fetch_delays()	This function fetches different delay values for different objects
acc_fetch_size()	

	This function returns the bit size of a net, register, or port.
acc_fetch_range()	This function retrieves the most significant bit and least significant bit range values for a vector.
acc_fetch_tfarg()	This function returns value of the specified argument of the system task or function associated (through the PLI mechanism) with your C−language routine
acc_fetch_direction()	This function returns the direction of a port or terminal as one of three predefined integer constants.
acc_fetch_index()	This function returns a zero−based integer index for a port or terminal
acc_fetch_edge()	This function returns the edge specifier (type) of a path input or output terminal as one of these predefined integer constants.
acc_set_value()	This function returns a pointer to a character string indicating the logic or strength value of a net, register or variable.
acc_initialize()	This function initializes the environment for access routines
acc_close()	This function frees internal memory used by access routines; resets all configuration parameters to default values
acc_configure()	This function sets parameters that control the operation of various access routines
acc_product_version()	This function returns a pointer to a character string that indicates what version of a Verilog simulator is linked to the access routines
acc_version()	This function returns a pointer to a character string that indicates version number of your access routine software
acc_count()	This function returns an integer count of the number of objects related to a particular reference object
acc_collect()	This function returns a pointer to an array that contains handles for all objects related to a particular reference object
acc_free()	This function frees memory allocated by acc_collect
acc_compare_handles()	This function returns true if the two input handles refer to the same object
acc_object_in_typelist()	This function determines whether an object fits a type or fulltype or exhibits a property specified in an input array
acc_object_of_type()	This function determines whether an object fits a specified type or fulltype, or exhibits a specified property
acc_next_cell()	This function returns the next cell instance within the region that includes the entire hierarchy below a module

acc_next_child()	This function returns the next child of a module
acc_next_modpath()	This function returns the next path of a module
acc_next_net()	This function returns the next net of a module
acc_next_parameter()	This function returns the next parameter within a module
acc_next_port()	This function returns the next input, output or inout port of a module in the order specified by the port list
acc_next_portout()	This function returns the next output or inout port of a module in the order specified by the port list
acc_next_primitive()	This function returns the next gate, switch or user–defined primitive (UDP) within a module
acc_next_specparam()	This function returns the next specparam within a module
acc_next_tchk()	This function returns the next timing check within a module
acc_next_terminal()	This function returns the next terminal of a gate, switch or user–defined primitive (UDP)
acc_next()	This function within a scope, returns the next object of each type specified in object_type_array
acc_next_topmod()	This function returns the next top–level module
acc_next_cell_load()	This function returns the next load on a net inside a cell
acc_next_load()	This function returns the next primitive terminal driven by a net
acc_next_driver()	This function returns the next primitive terminal that drives a net
acc_next_hiconn()	This function returns the next hierarchically higher net connection to a port of a module
acc_next_loconn()	This function returns the next hierarchically lower net connection to a port of a module
acc_next_bit()	This function returns the handles of each bit in an expanded vector port or expanded vector net
acc_next_input()	This function returns a handle to the next input path terminal of the specified module path or datapath
acc_next_output()	This function returns a handle to the next output path terminal of the specified module path or datapath

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.

```
#include <acc_user.h>

void pli_func() {
  acc_initialize();
  // Main body: Insert the user application code here
  acc_close();
}
```

- 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 Utility Routines

Routine	Description
tf_getp()	Get integer value of a task or function argument (parameter) for the current instance.
tf_putp()	Passes a 32−bit integer value to an argument of the current task or function.
tf_getrealp()	Get real value of a task or function argument for the current instance.
tf_putrealp()	Passes a 64−bit floating point value to an argument of the current task or function.
tf_getlongp()	Get long argument value (64 bits) for the current instance.
tf_putlongp()	Passes a 64−bit integer argument value to the current instance of a task or function.
tf_strgetp()	Get formatted argument values.
tf_getcstringp()	Get a parameter value as a character string.
tf_strdelputp()	Write a value to an argument with the specified delay, where the write value is specified as a character string.
tf_strlongdelputp()	Write value to an argument with specified long (64−bit) delay, where the write value is specified as a character string.
tf_strrealdelputp()	Write value to an argument with specified long (64−bit) delay, where the write value is specified as a character string.
tf_copypvc_flag()	Copy parameter value change flag
tf_movepvc_flag()	Move argument value change flag.
tf_testpvc_flag()	Test argument value change flag for the current task/function instance.
tf_getpchange()	Get number of the next argument that changed value in the current task or function instance.
tf_gettime()	Get the current simulation time.
tf_getlongtime()	Gets current simulation time as a 64−bit integer value, in the timescale of the calling task or function.

tf_getrealtime()	Get the current simulation time scaled to the calling module's timescale.
tf_strgettime()	Get current simulation time as a character string.
tf_gettimeunit()	Get the timescale units for the module that contains the current task call.
tf_gettimeprecision()	Get the timescale precision of the module that contains the current task call.
tf_synchronize()	Synchronize processing of the current system task or function's arguments.
tf_rosynchronize()	Synchronize processing of the current system task or function's arguments to the end of the current time slot with event generation suppressed.
tf_getnextlongtime()	Get the time of the next scheduled simulation event.
tf_setdelay()	Reactivate a user task at a particular integer simulation time.
tf_setlongdelay()	Reactivate user task at a particular simulation time specified by a 64−bit integer.
tf_setrealdelay()	Activate user task at a particular simulation time.
tf_clearalldelays()	Clear all scheduled reactivation delays.
io_printf	Print a formatted message to the output channel of the software product that invoked the PLI application and to a log file.
io_mcdprintf()	Write a formatted message to one or more files.
tf_warning()	Report a warning.
tf_error()	Report error.
tf_text()	Stores error information for later display by tf_message.
tf_message()	Reports user−generated error message in standard error−message syntax.
tf_getinstance()	Get current instance pointer.
tf_mipname()	Get hierarchical path name of the current module instance.
tf_spname()	Get a scope's hierarchical path name as a string.
tf_setworkarea()	Store a work area pointer for the current user task/function instance in a cell.
tf_getworkarea()	Get the work area pointer stored by the routine tf_setworkarea().
tf_nump()	Get number of arguments in the argument list of the current task or function instance.
tf_typep()	Get data type of an argument of the current task or function.
tf_sizep()	Get size of an argument of the current task/function instance.
tf_dostop()	Enable interactive mode.
tf_dofinish()	Finish simulation.
mc_scan_plusargs()	Get command line plus (+) options.
tf_compare_long()	Compare two 64−bit integer values.

tf_add_long()	Add two 64–bit integer values.
tf_subtract_long()	Subtract one 64–bit integer value from another.
tf_multiply_long()	Multiply two 64 bit integer values.
tf_divide_long()	Divide two 64–bit integer values.
tf_long_to_real()	Convert a long integer to a real (double–precision floating point) number.
tf_longtime_tostr()	Convert 64–bit integer time value to string.
tf_real_tf_long()	Convert a real (double precision floating point) number to a long (64–bit) integer.
tf_write_save()	Append a block of data to the save file.
tf_read_restart()	Get a block of data from a previously written save file.

Other Usefull functions

Other then acc_* and tf_* routines, simulators provide other functions as listed below.

- veriusertfs
- endofcompile_routines
- err_intercept

veriusertfs

For all the simulator other then VCS, veriusertfs should be provided. The veriusertfs array is a table that the PLI uses to associate Verilog Simulator system tasks or functions with user–written applications. Below table shows the fields in the array that you fill in for each new system task or function you wish to create. The fields appear in the array in the order shown.

Field	Description
type	Keyword that indicates whether the routine is a system task or function, usertask defines a system task or userfunction defines a system function that can return a user–defined value or userrealfunction defines a system function that returns a real value
data	Data argument (0 means do not pass a data argument).
checktf	Pointer to optional user–supplied routine that checks parameters of the system task or function (0 means no checktf routine).
sizetf	Pointer to a user–supplied routine that returns the size in bits of the value returned by a system function (0 means no sizetf routine).
calltf	Pointer to user–supplied application––the main routine that is called when the Veritool executes the system task or function during simulation (0 means no calltf routine).
misctf	

	Pointer to optional miscellaneous routine (0 means no misctf routine).
$tfname	Quoted literal string defining the name of the system task or function; $ must be the first character, followed by any combination of letters, numbers and underscore (_) characters.

endofcompile_routines

This data structure, as the name suggests, can be used for declaring functions that will be invoked at the end of the simulation. The present example does not need it. Nonetheless there has to be a default definition on as shown below.

```
int (*endofcompile_routines[])() ={0};
```

err_intercept

This function, which returns either true or false (i.e. a boolean type), can be used to enhance the error detection mechanism of the code and can be ignored for small applications. A typical example of this routine would be as shown below.

```
bool err_intercept(level, facility,code)
int level; char * facility; char *code;
{ return (true); }
```

PLI Example

As in rest of the tutorial, lets verify counter example with test vector generation, monitor, checker everything build with C code. Most of the common pli functions have been used to show usage of these functions.

Testbench will have below components.

- Clock generation in C
- Clock generator HDL wrapper
- Test Generation in C
- Test Gen HDL Wrapper
- Monitor and Checker in C
- Monitor Checker HDL Wrapper
- DUT/Monitor/Clock/Test Generation instance in Verilog

It is good idea to write HDL wrappers for the C functions that will be calling them.

Clock Generator

Normally we don't want to have clock generators in PLI, it is always to put them in Verilog.

```c
#include "acc_user.h"
#include "veriuser.h"

// Define the ON and OFF time of clock
#define PERIOD 5

// Data structure
struct clkData {
  int clk;
  int clkCnt;
};

// Main routine which toggles the clock
void clkGen () {
  // Get the stored workarea
  struct clkData *data = ( struct clkData * )tf_igetworkarea(tf_getinstance());
  if (data->clkCnt == PERIOD) {
    data->clk = (data->clk == 0) ? 1 : 0;
    data->clkCnt = 0;
    //io_printf("%d Current clk = %d\n",tf_gettime(), data->clk);
  } else {
    data->clkCnt ++;
  }
  // Drive the clock signal in HDL
  tf_putp (1, data->clk);
}

// checktf() routine
// This function inits the objects and also stores the object in workarea
void clkInit() {
  struct clkData *data = ( struct clkData * )malloc( sizeof( struct clkData ) );
  data->clkCnt = 0;
  data->clk  = 0;
  tf_setworkarea(data);
}

// misctf() routine
// This routine is called after 1 tick
void clkReactive (int data, int reason, int paramvc) {
  // if callback reason is reactive, then call clkGen function
  if (reason == reason_reactivate) {
    clkGen();
  }
  // Set the callback delay to 1 tick
  tf_setdelay(1);
}
```

Clock Generator HDL Wrapper

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

initial $clkGen(clk);

endmodule
```

Counter Monitor

```
#include "acc_user.h"
#include "veriuser.h"
#include <malloc.h>
#include <string.h>

struct myCounter {
	handle count;
	handle enable;
	handle reset;
	handle clk;
	char *count_value;
	char *enable_value;
	char *reset_value;
	char *clk_value;
	char *clk_last_value;
	int checker_count;
	int count_width;
	int error;
	int error_time;
};

// Multi-bit vector to integer conversion.
int pliConv (char *in_string, int no_bits, int sim_time) {
	int conv = 0;
	int i = 0;
	int j = 0;
	int bin = 0;
	for ( i = no_bits-1; i >= 0; i = i - 1) {
		if (*(in_string + i) == 49) {
			bin = 1;
		} else if (*(in_string + i) == 120) {
			io_printf ( "%d counterMonitor : WARNING : X detected\n" ,
			sim_time);
			bin = 0;
		} else if (*(in_string + i) == 122) {
			io_printf ( "%d counterMonitor : WARNING : Z detected\n" ,
			sim_time);
			bin = 0;
		} else {
			bin = 0;
		}
		conv = conv + (1 << j)*bin;
```

```
                                        j ++;
                    }
                    return conv;
}

void counterModel (struct myCounter *counter) {
                    int current_value ;
                    int time = tf_gettime();
                    // Our model checks only at posedge
                    if ((strcmp(counter->clk_value, "1" ) == 0) && (strcmp(counter->clk_last_value, "0" ) == 0)) {
                    // Conver the current count value
                    current_value  = pliConv(counter->count_value,counter->count_width,time);
                    // Check input control signal to floating or UnKnown
                    if (strcmp(counter->reset_value, "x" ) == 0) {
                    io_printf( "%d counterMonitor : WARNING : reset is x\n" , time);
}
if (strcmp(counter->reset_value, "z" ) == 0) {
io_printf( "%d counterMonitor : WARNING : reset is z\n" , time);
}
if (strcmp(counter->enable_value, "x" ) == 0) {
io_printf( "%d counterMonitor : WARNING : enable is x\n" , time);
}
if (strcmp(counter->enable_value, "z" ) == 0) {
io_printf( "%d counterMonitor : WARNING : enable is z\n" , time);
}
// Increment monitor counter and compare only if enable is 1 and reset is not active
if (strcmp(counter->enable_value, "1" ) == 0 && strcmp(counter->reset_value, "0" ) == 0) {
if (counter->checker_count != current_value) {
                    io_printf( "%d counterMonitor : ERROR : Current value of monitor is %d dut is %d\n" ,
                    time, counter->checker_count, current_value);
                    counter->error ++;
                    if (counter->error == 1) counter->error_time = time;
} else {
                    io_printf( "%d counterMonitor : INFO : Current value of monitor is %d dut is %d\n" ,
                    time, counter->checker_count, current_value);
}
counter->checker_count = (counter->checker_count == 15) ? 0 : counter->checker_count + 1;
// Reset monitor counter if reset is active
} else if (strcmp(counter->reset_value, "1" ) == 0) {
io_printf( "%d counterMonitor : INFO : Reset is asserted\n" , time);
counter->checker_count = 0;
}
}
// Update the clock state
strcpy(counter->clk_last_value,counter->clk_value);
}

// misctf
void counterMonitor(int data, int reason, int paramvc) {
                    struct myCounter *counter = (struct myCounter *) tf_igetworkarea(tf_getinstance());
                    if ((reason == reason_paramvc) || (reason == reason_paramdrc)) {
                                        tf_synchronize( );
                    } else if (reason == reason_synch) {
                                        counter->clk  = acc_handle_tfarg(1);
```

```
        counter->reset  = acc_handle_tfarg(2);
        counter->enable = acc_handle_tfarg(3);
        counter->count  = acc_handle_tfarg(4);
        // Get the values
        counter->clk_value  = acc_fetch_value(counter->clk, "%b" , 0);
        counter->reset_value  = acc_fetch_value(counter->reset, "%b" , 0);
        counter->enable_value = acc_fetch_value(counter->enable, "%b" , 0);
        counter->count_value  = acc_fetch_value(counter->count, "%b" , 0);
        counter->count_width  = acc_fetch_size (counter->count);
        // Call the counter model
        counterModel (counter);
    }
    // Print simulation stats when $finish is called
    if (reason == reason_finish) {
        io_printf( "============================================\n" );
        if (counter->error != 0) {
            io_printf ( " Simulation : FAILED\n" );
            io_printf ( " Mismatched %d\n" ,counter->error);
            io_printf ( " First Mismatch at time %d\n" , counter->error_time);
        } else {
            io_printf ( " Simulation : PASSED\n" );
        }
        io_printf( "============================================\n" );
    }
}

// calltf()
void initCounter(int data, int reason) {
    struct myCounter *counter;
    // Allocate memory for all variables necessary to manage a
    // single instance of the model.
    counter = (struct myCounter *) malloc (sizeof(struct myCounter));
    // Initialize this instance of the model.
    counter->clk  = acc_handle_tfarg(1);
    counter->reset  = acc_handle_tfarg(2);
    counter->enable = acc_handle_tfarg(3);
    counter->count  = acc_handle_tfarg(4);
    // Save a copy of the present clk value.
    counter->clk_last_value  = acc_fetch_value(counter->clk, "%b" , 0);
    // Enable callback of `counter_monitor` whenever
    // any argument to `$counter_monitor` changes.
    tf_asynchon();
    // Set initial counter value to 0.
    counter->checker_count = 0;
    counter->error = 0;
    counter-> error_time = 0;
    // Save the model data with this instance of `$counterMonitor`.
    tf_setworkarea((char *)counter);
}
```

Counter Monitor HDL Wrapper

```verilog
module counterMonitor (clk, reset, enable, count);
input clk, reset, enable;
input [3:0] count;

wire clk, reset, enable;
wire [3:0] count;

initial $counterMonitor(clk,reset,enable,count);

endmodule
```

Counter TestGen

Syntax for test file

delay : Command = Value

Where
Delay : Delay in clock ticks
Command : reset or enable
Value : 0 or 1

```c
#include "acc_user.h"
#include "veriuser.h"
#include "string.h"
#include "stdio.h"

#define IDLE 0
#define INCR 1
#define WAIT 2
#define DRIVE 3
#define DONE 4

struct testGenObject {
        char* testFile;
        int debug;
        char cmdArray[100] [100];
        int cmdSize;
        int CmdPointer;
        char* command;
        int wait;
        int value;
        int clkCnt;
        int state;
        handle count;
        handle enable;
        handle reset;
        handle clk;
        char* clk_value;
```

```
        char *clk_last_value;
};

static struct testGenObject *object;

// Increment counter
void waitTicks () {
        object->clkCnt = object->clkCnt + 1;
}

// This function loads the content of test file into
// object command array
void loadTest() {
        FILE *testFile;
        char currentLine [100];
        object->cmdSize = 0;
        if((testFile = fopen(object->testFile, "r" )) == NULL) {
        printf( "Error Opening File.\n" );
}
while (fgets(currentLine, sizeof(currentLine), testFile) != NULL ) {
        // Store the line cmdArray
        strcpy(object->cmdArray[object->cmdSize], currentLine);
        // print the line number and data
        if (object->debug) printf( "Line %d: %s\n" , object->cmdSize, object->cmdArray[object->cmdSize]);
        // Get each line from the test file
        object->cmdSize ++;
}
// Close the test file
fclose(testFile);
}

// This function process command line options
void processCmdOptions () {
        // Get debug option
        if (mc_scan_plusargs( "plidebug" ) != NULL) {
        object->debug = 1;
} else {
        object->debug = 0;
}
// Get test file name
if (mc_scan_plusargs( "test=" ) == NULL) {
printf( "ERROR : No test file option passed, use +test=testfile\n" );
} else {
object->testFile = mc_scan_plusargs( "test=" );
if (object->debug) printf( "Test file name %s\n" ,object->testFile);
}
}

void doTest() {
        char* ptoks;
        char* tcmd;
        s_setval_delay delay_s;
        s_setval_value value_s;
        // Get current clock value
        object->clk_value = acc_fetch_value(object->clk, "%b" , 0);
        // BFM drives only at rising edge of clock
```

```
  if (!strcmp(object->clk_last_value, "1" ) && !strcmp(object->clk_value, "0" )) {
  switch (object->state) {
    case IDLE  : if (object->debug) printf( "%d Current state is IDLE\n" , tf_gettime());
    if (object->CmdPointer < object->cmdSize) {
      tcmd = object->cmdArray[object->CmdPointer];
      if (object->debug) printf ( "Test line %d current command-%s" ,object->CmdPointer, tcmd);
      ptoks = strtok(tcmd, ":" );
      int lcnt = 0;
      while(ptoks != NULL) {
        if (*ptoks != '=') {
          if (lcnt == 0) {
            object->wait = atoi(ptoks);
            if (object->debug) printf( "Wait : %d\n" , object->wait);
          } else if (lcnt == 1) {
            object->command = ptoks;
            if (object->debug) printf( "Command : %s\n" , ptoks);
          } else {
            object->value = atoi(ptoks);
            if (object->debug) printf( "Value : %d\n" , object->value);
          }
          lcnt ++;
        }
        ptoks = strtok(NULL, " " );
      }
      object->CmdPointer ++ ;
      if (object->wait == 0) {
        if (object->debug) printf( "%d Next State DRIVE\n" , tf_gettime());
        object->state = DRIVE;
        doTest();
      } else {
        if (object->debug) printf( "%d Next State WAIT\n" , tf_gettime());
        object->state = WAIT;
      }
    } else {
      if (object->debug) printf( "%d Next State DONE\n" , tf_gettime());
      object->state = DONE;
    }
    break;
    case WAIT  : if (object->debug) printf( "%d Current state is WAIT : %d\n" , tf_gettime(), object->clkCnt);
    if ((object->clkCnt + 1) >= object->wait) {
      object->wait = 0;
      object->clkCnt = 0;
      if (object->debug) printf( "%d Next State DRIVE\n" , tf_gettime());
      object->state = DRIVE;
      doTest();
    } else {
      waitTicks();
    }
    break;
```

```
          case DRIVE : if (object->debug) printf( "%d Current state is DRIVE\n" , tf_gettime());
          value_s.format  = accIntVal;
          delay_s.model  = accNoDelay;
          delay_s.time.type = accTime;
          delay_s.time.low  = 0;
          delay_s.time.high = 0;
          if (!strcmp(object->command, "reset" )) {
          value_s.value.integer = object->value;
          acc_set_value(object->reset,&value_s,&delay_s);
     } else if (!strcmp(object->command, "enable" )) {
     value_s.value.integer = object->value;
     acc_set_value(object->enable,&value_s,&delay_s);
} else {
     if (object->debug) printf( "ERROR : What command do you want\n" );
}
if (object->debug) printf( "%d Next State IDLE\n" , tf_gettime());
object->state = IDLE;
break;
case DONE  : if (object->debug) printf( "%d Current state is DONE\n" , tf_gettime());
tf_dofinish();
break;
default  : object->state = IDLE;
break;
}
}
object->clk_last_value = acc_fetch_value(object->clk, "%b" , 0);
}

void initCounterTestGen () {
     //acc_initialize( );
     //acc_configure( accDisplayErrors, "false" );
     object = (struct testGenObject *) malloc (sizeof(struct testGenObject));
     // Load the initial and defaule values
     object->testFile = "simple.tst" ;
     object->cmdSize = 0;
     object->CmdPointer = 0;
     object->clkCnt = 0;
     object->state = IDLE;
     // Initialize this instance of the model.
     object->clk  = acc_handle_tfarg(1);
     object->reset  = acc_handle_tfarg(2);
     object->enable = acc_handle_tfarg(3);
     object->count  = acc_handle_tfarg(4);
     // Drive inactive signals on all inputs
     tf_putp (2, 0);
     tf_putp (3, 0);
     // Save a copy of the present clk value.
     object->clk_last_value = acc_fetch_value(object->clk, "%b" , 0);
     // Get the command line testfile name and debug option
     processCmdOptions();
     // Open the testfile and make array of command
     loadTest(object);
     // Add callback when ever clock changes
```

```
        acc_vcl_add( object->clk, doTest, object->clk_value, vcl_verilog_logic );
        // All acc routines should have this
        acc_close();
}
```

Counter TestGen HDL Wrapper

```
module counterTestGen (clk, reset, enable, count);
input clk;
output reset, enable;
input [3:0] count;

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

initial $counterTestGen(clk,reset,enable,count);

endmodule
```

HDL TestBench Top

```
module top();
wire clk;
wire [3:0] count;
wire enable;
wire reset;

// Connect the clk generator
clkGen clkGen(.clk (clk));

// Connect the DUT
counter dut(
.clk (clk),
.reset (reset),
.enable (enable),
.count (count)
);

// Connect the Monitor/Checker
counterMonitor monitor(
.clk (clk),
.reset (reset),
.enable (enable),
.count (count)
);

// Connect the Test Generator
counterTestGen test(
.clk (clk),
.reset (reset),
.enable (enable),
.count (count)
);

endmodule
```

Sample : Test File

```
1 : reset = 1
10 : reset = 0
5 : enable = 1
20 : enable = 0
```

Sample Ouput

```
5 counterMonitor : WARNING : X detected
5 counterMonitor : WARNING : X detected
5 counterMonitor : WARNING : X detected
5 counterMonitor : WARNING : X detected
17 counterMonitor : WARNING : X detected
17 counterMonitor : WARNING : X detected
17 counterMonitor : WARNING : X detected
17 counterMonitor : WARNING : X detected
29 counterMonitor : WARNING : X detected
29 counterMonitor : WARNING : X detected
29 counterMonitor : WARNING : X detected
29 counterMonitor : WARNING : X detected
29 counterMonitor : INFO  : Reset is asserted
41 counterMonitor : INFO  : Reset is asserted
53 counterMonitor : INFO  : Reset is asserted
65 counterMonitor : INFO  : Reset is asserted
77 counterMonitor : INFO  : Reset is asserted
89 counterMonitor : INFO  : Reset is asserted
101 counterMonitor : INFO  : Reset is asserted
113 counterMonitor : INFO  : Reset is asserted
125 counterMonitor : INFO  : Reset is asserted
137 counterMonitor : INFO  : Reset is asserted
149 counterMonitor : INFO  : Reset is asserted
233 counterMonitor : INFO  : Current value of monitor is 0 dut is 0
245 counterMonitor : INFO  : Current value of monitor is 1 dut is 1
257 counterMonitor : INFO  : Current value of monitor is 2 dut is 2
269 counterMonitor : INFO  : Current value of monitor is 3 dut is 3
281 counterMonitor : INFO  : Current value of monitor is 4 dut is 4
293 counterMonitor : INFO  : Current value of monitor is 5 dut is 5
305 counterMonitor : INFO  : Current value of monitor is 6 dut is 6
317 counterMonitor : INFO  : Current value of monitor is 7 dut is 7
329 counterMonitor : INFO  : Current value of monitor is 8 dut is 8
341 counterMonitor : INFO  : Current value of monitor is 9 dut is 9
353 counterMonitor : INFO  : Current value of monitor is 10 dut is 10
365 counterMonitor : INFO  : Current value of monitor is 11 dut is 11
377 counterMonitor : INFO  : Current value of monitor is 12 dut is 12
389 counterMonitor : INFO  : Current value of monitor is 13 dut is 13
401 counterMonitor : INFO  : Current value of monitor is 14 dut is 14
413 counterMonitor : INFO  : Current value of monitor is 15 dut is 15
425 counterMonitor : INFO  : Current value of monitor is 0 dut is 0
437 counterMonitor : INFO  : Current value of monitor is 1 dut is 1
449 counterMonitor : INFO  : Current value of monitor is 2 dut is 2
461 counterMonitor : INFO  : Current value of monitor is 3 dut is 3
473 counterMonitor : INFO  : Current value of monitor is 4 dut is 4
=========================================
 Simulation : PASSED
=========================================
```

Verilog Procedural Interface (VPI)

The Verilog Procedural Interface (VPI) is a C−programming interface for the Verilog Hardware Description Language (HDL). VPI is the third−generation procedural interface to Verilog HDL. VPI provides consistent, object−oriented access to the complete Verilog HDL.

VPI consists of a set of access and utility routines that you call from standard C programming language functions. These routines interact with the instantiated simulation objects contained in your Verilog HDL design.

VPI is same as verilog PLI 1.0 and PLI 2.0, So if something is working in PLI 2.0, it can be easily ported over to VPI. VPI is cleaner then earlier PLI 2.0. All the functions and tasks are start with vpi_*. Example vpi_printf().

Steps : Writing Application Using VPI

Following steps needs to be performed to able to write a C application and interface with Verilog simulator.

- Write C function
- Associating C Functions with a New System Task
- Registering New System Tasks
- Invoking System Tasks

Write C function

Writing C function/routine is same as in PLI 2.0, only difference is we need to include vpi_user.h instead of acc_user.h and veriuser.h. Also we use vpi_* functions to access and modify objects in Verilog simulator.

Below is example code of a simple C function

```
#include "vpi_user.h"

void hello() {
   vpi_printf( "\n\nHello Deepak\n\n" );
}
```

Associating C Functions with a New System Task

To associate your C function with a system task, create a data structure of type s_vpi_systf_data and a pointer to that structure. The vpi_systf_data data type is defined in the vpi_user.h include file. Below is is data structure of s_vpi_systf_data.

```
typedef struct t_vpi_systf_data {
  PLI_INT32 type; // vpiSysTask, vpiSysFunc
  PLI_INT32 sysfunctype; // vpiSysTask, vpi[Int,Real,Time,Sized, SizedSigned]Func
  PLI_BYTE8 *tfname; // First character must be `$'
  PLI_INT32 (*calltf)(PLI_BYTE8 *);
  PLI_INT32 (*compiletf)(PLI_BYTE8 *);
  PLI_INT32 (*sizetf)(PLI_BYTE8 *); // For sized function callbacks only
  PLI_BYTE8 *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Field Name	Field Description
type	task– That does not return a value. or function– That can return a value.
sysfunctype	If type is function, sysfunctype indicates the type of the value that the calltf function returns.
tfname	This quoted literal string defines the name of the system task or function. The first character must be $.
calltf	This field is a pointer to your application routine
compiletf	This field is a pointer to a routine that the simulator calls once each time it compiles an instance of the task or function. Enter NULL if you have no such routine.
sizetf	This field is a pointer to a routine that returns the size, in bits, of the value that the system task or function returns.
user_data	This field is a pointer to optional data. You can retrieve this data by calling the vpi_get_systf_info() routine.

Example for hello routine

```
#include "hello_vpi.c"

// Associate C Function with a New System Task
void registerHelloSystfs() {
  s_vpi_systf_data task_data_s;
  p_vpi_systf_data task_data_p = &task_data_s;
  task_data_p->type = vpiSysTask;
  task_data_p->tfname = "$hello" ;
  task_data_p->calltf = hello;
  task_data_p->compiletf = 0;

  vpi_register_systf(task_data_p);
}

// Register the new system task here
void (*vlog_startup_routines[ ] ) () = {
  registerHelloSystfs,
  0 // last entry must be 0
};
```

Registering New System Tasks

After you initialize the s_vpi_systf_data data structure, you must register your new system task so that the simulator can execute it. In the above example of hello routine, line number 14 to 17 does this.

Invoking System Tasks

You can call your new system tasks in initial block or in always block as shown in below code.

```
module hello_pli ();

initial begin
  $hello;
  #10 $finish;
end

endmodule
```

Linking with Simulator

Like PLI1.0 and PLI2.0, each simulator has its own way of linking VPI routines into simulator. We will see following simulators as examples

- VCS
- Modelsim
- NCSim

VCS

With VCS simulator you need to create a tab file. For our example tab file looks like below.

$hello call=hello

Here $hello is name of user defined function that will be used in Verilog code, call=hello is the C function which will be called when $hello is called in Verilog.

Command line options for compiling the code is as below.

vcs –R –P hello.tab hello_vpi.c hello_pli.v +vpi –CFLAGS "–I$VCS_HOME/`vcs –platform`/lib"

Only difference between VPI and PLI is option +vpi in command line.

Modelsim

Like VCS, modelsim simulator has its own way for communicating with PLI. We need to create function listing all userdefined function that will be referred in verilog and corresponding C function to call. Unlike VCS, we need to do it in C file as shown below. We discussed it already in

"**Associating C Functions with a New System Task**".

```
#include "hello_vpi.c"

// Associate C Function with a New System Task
void registerHelloSystfs() {
  s_vpi_systf_data task_data_s;
  p_vpi_systf_data task_data_p = &task_data_s;
  task_data_p->type = vpiSysTask;
  task_data_p->tfname = "$hello" ;
  task_data_p->calltf = hello;
  task_data_p->compiletf = 0;

  vpi_register_systf(task_data_p);
}

// Register the new system task here
void (*vlog_startup_routines[ ] ) () = {
  registerHelloSystfs,
  0 // last entry must be 0
};
```

```
vlib work
vlog hello_pli.v
gcc -c -g -I$MODEL/include hello_vpi_modelsim.c
ld -shared -E -o hello_vpi.sl hello_vpi_modelsim.o
vsim -c hello_pli -pli hello_vpi.sl
```

In vsim command line, type "run -all" to start the simulation.

Refer to modelsim user guide for details.

NCSim

Compiling, linking for ncsim is some what close to modelsim. Unlike modelsim ncsim supports many ways to link. Below are list of steps to perform.

```
gcc -c -g -I$CDS_INST_DIR/tools/include hello_vpi_modelsim.c
ld -shared -E -o libvpi.so hello_vpi_modelsim.o
ncverilog hello_pli.v
```

Refer to ncsim or ncverilog user manual for detail on linking vpi

VPI Routines

Routine	Description
vpi_chk_error()	Checks whether the previous VPI call has caused an error and retrieves information about that error.
vpi_compare_objects()	Compares two object handles to determine whether they reference the same object.
vpi_control()	Passes information from user code to the simulator.
vpi_flush()	Flushes the data from the stimulator output channel and log file output buffers.
vpi_free_object()	Frees the memory that is allocated for VPI objects.
vpi_get()	Retrieves the value of an integer or Boolean property of an object.
vpi_get_cb_info()	Retrieves information about a simulation−related callback.
vpi_get_data()	Gets data from an implementation's save/restart location.
vpi_get_delays()	Retrieves the delays or timing limits of an object.
vpi_get_str()	Retrieves the value of a string property of an object.
vpi_get_systf_info()	Retrieves information about a user−defined system task or function.
vpi_get_time()	Retrieves the current simulation time, using the timescale of the object.
vpi_get_userdata()	Retrieves the user data value from an implementation's system task/function storage location.
vpi_get_value()	Retrieves the simulation value of an object.
vpi_handle()	Returns a handle to an object that has a one−to−one relationship.
vpi_handle_by_index()	Returns a handle to an object, using the object's index number within a parent object.
vpi_handle_by_name()	Returns a handle to an object that has a specific name.
vpi_handle_multi()	Returns a handle to an object that has a many−to−one relationship.
vpi_iterate()	Returns an iterator handle that you use to access objects that are in a one−to−many relationship.
vpi_mcd_open()	Opens a file for writing.
vpi_mcd_close()	Close one or more files opened by the vpi_mcd_open() routine.
vpi_mcd_flush()	Use this function to flush the data from the given MCD output buffers.
vpi_mcd_name()	Returns the name of a file that is represented by a channel descriptor.
vpi_mcd_printf()	Writes to one or more files opened by vpi_mcd_open().
vpi_printf()	Writes to the output channel of the simulator that has invoked the VPI application and to the simulator log file.
vpi_put_data()	Puts data into an implementation's save/restart location.

vpi_put_delays()	Sets the delays or timing limits of an object.
vpi_put_userdata()	Puts the user data value into an implementation's system task/function storage location.
vpi_put_value()	Sets a value on an object.
vpi_register_cb()	Registers a simulation−related callback.
vpi_register_systf()	Registers a user−defined system task or function callback.
vpi_remove_cb()	Removes a simulation callback that has been registered with vpi_register_cb().
vpi_scan()	Scans the Verilog HDL hierarchy for objects that have a one−to−many relationship.
vpi_vprintf()	Writes to the output channel of the simulator that has invoked the VPI application and to the simulator log file using varargs that are already started.

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

WHAT'S NEW IN VERILOG 2001

CHAPTER 19

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 may need simulator with Verilog 2001 support for testing examples listed below

Comma used in sensitive list

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

```
module comma_example();

reg a, b, c, d, e;
reg [2:0] sum, sum95;

// Verilog 2k example for usage of comma
always @ (a, b, c, d, e)
begin : SUM_V2K
  sum = a + b + c + d + e;
end

// Verilog 95 example for above code
always @ (a or b or c or d or e)
begin : SUM_V95
  sum95 = a + b + c + d + e;
end

initial begin
  $monitor ( "%g a=%b b=%b c=%b d=%b e=%b sum=%b sum95=%b" ,
  $time, a, b, c, d, e, sum, sum95);
  #1 a = 1;
  #1 b = 1;
  #1 c = 1;
  #1 d = 1;
  #1 e = 1;
  #1 $finish;
end

endmodule
```

We can use same for edge senstive code also as shown below code

```
module comma_edge_example();

reg clk, reset, d;
reg q, q95;

// Verilog 2k example for usage of comma
always @ (posedge clk, posedge reset)
begin : V2K
  if (reset) q <= 0;
```

```
  else q <= d;
end

// Verilog 95 example for above code
always @ (posedge clk or posedge reset)
begin : V95
  if (reset) q95 <= 0;
  else q95 <= d;
end

initial begin
  $monitor ( "%g clk=%b reset=%b d=%b q=%b q95=%b" ,
  $time, clk, reset, d, q, q95);
  clk = 0;
  reset = 0;
  d = 0;
  #4 reset = 1;
  #4 reset = 0;
  #1 d = 1;
  #10 $finish;
end

initial #1 forever clk = #1 ~clk;

endmodule
```

Combinational logic sensitive list

Verilog 2001 allows us to use star in sensitive list instead of listing all the variables in RHS. This kind of removed the typo mistakes and thus avoids simulation and synthesis mismatches.

```
module star_example();

reg a, b, c, d, e;
reg [2:0] sum, sum95;

// Verilog 2k example for usage of star for combo logic
always @ (*)
begin : SUM_V2K
  sum = a + b + c + d + e;
end

// Verilog 95 example for above code
always @ (a or b or c or d or e)
begin : SUM_V95
  sum95 = a + b + c + d + e;
end

initial begin
  $monitor ( "%g a=%b b=%b c=%b d=%b e=%b sum=%b sum95=%b" ,
  $time, a, b, c, d, e, sum, sum95);
  #1 a = 1;
  #1 b = 1;
  #1 c = 1;
  #1 d = 1;
```

```
  #1 e = 1;
  #1 $finish;
end

endmodule
```

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.

```
module v2k_reg();

// v2k allows to init variables
reg a = 0;
// Here only last variable is set to 0, i.e d = 0
// Rest b, c are set to x
reg b, c, d = 0;
// reg data type can be signed in v2k
// We can assign with signed constants
reg signed [7:0] data = 8'shF0;

// Function can return signed values
// Its ports can contain signed ports
function signed [7:0] adder;
input a_in;
input b_in;
input c_in;
input signed [7:0] data_in;
begin
  adder = a_in + b_in + c_in + data_in;
end
endfunction

endmodule
```

New operators

Verilog 2001 introduced two new operator that are of interest to designers. They are

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

Signed shift operator

```
module signed_shift();

reg [7:0] unsigned_shift;
reg [7:0] signed_shift;
reg signed [7:0] data = 8'hAA;

initial begin
  unsigned_shift = data >> 4;
  $display ( "unsigned shift = %b" , unsigned_shift);
  signed_shift = data >>> 4;
  $display ( "signed shift = %b" , signed_shift);
  unsigned_shift = data << 1;
  $display ( "unsigned shift = %b" , unsigned_shift);
  signed_shift = data <<< 1;
  $display ( "signed shift = %b" , signed_shift);
  $finish;
end

endmodule
```

Power operator

```
module power_operator();

reg [3:0] base = 2;
reg [5:0] exponent = 1;
reg [31:0] result = 0;

initial begin
  $monitor ( "base = %d exponent = %d result = %d" , base, exponent, result);
  repeat (10) begin
    #1 exponent = exponent + 1;
  end
  #1 $finish;
end

always @ (*)
begin
  result = base ** exponent;
end

endmodule
```

Port Declaration

Verilog 2001 allows to have port direction and data type in the port list of module as shown in below example.

```
module ansiport_example();

reg read,write = 0;
reg [7:0] data_in  = 0;
reg [3:0] address = 0;
wire [7:0] data_v95, data_notype, data_ansi;

initial begin
  $monitor ( "%g read=%b write=%b address=%b data_in=%h data_v95=%h data_notype=%h data_ansi=%h" ,
  $time, read, write, address, data_in, data_v95, data_notype, data_ansi);
  #1 read = 0; // why only for read
  #3 repeat (16) begin
    data_in = $random;
    write = 1;
    #1 address = address + 1;
  end
  write = 0;
  address = 0;
  #3 repeat (16) begin
    read = 1;
    #1 address = address + 1;
  end
  read = 0;
  #1 $finish;
end

memory_v95 U (read, write, data_in, address, data_v95);
memory_ansi_notype W (read, write, data_in, address, data_notype);
memory_ansi V (read, write, data_in, address, data_ansi);

endmodule
// Verilog 95 code
module memory_v95 ( read, write, data_in, address, data_out);
input read;
input write;
input [7:0] data_in;
input [3:0] address;
output [7:0] data_out;

reg [7:0] data_out;
reg [7:0] mem [0:15];

always @ (*)
begin
  if (write) mem[address] = data_in;
end

always @ (read, address)
begin
  if (read) data_out = mem[address];
end

endmodule

// Verilog 2k with notype in port list
module memory_ansi_notype (
input read,
input write,
```

```
input [7:0] data_in,
input [3:0] address,
output [7:0] data_out
);
reg [7:0] mem [0:15];

always @ (*)
begin
  if (write) mem[address] = data_in;
end

assign data_out = (read) ? mem[address] : 0;

endmodule

// Verilog 2k with width and data type listed
module memory_ansi (
input wire read,
input wire write,
input wire [7:0] data_in,
input wire [3:0] address,
output reg [7:0] data_out
);

reg [7:0] mem [0:15];

always @ (*)
begin
  if (write) mem[address] = data_in;
end

always @ (read, address)
begin
  if (read) data_out = mem[address];
end

endmodule
```

Generate Blocks

This feature has been taken from VHDL with some modification. It is possible to use for loop to mimic multiple instants. Below code is example of usage of generate statement of Verilog 2001.

```
module generate_example();

reg read,write = 0;
reg [31:0] data_in  = 0;
reg [3:0] address = 0;
wire [31:0] data_out;

initial begin
  $monitor ( "%g read=%b write=%b address=%b data_in=%h data_out=%h" ,
  $time, read, write, address, data_in, data_out);
  #1 read = 0; // why only for read
  #3 repeat (16) begin
    data_in = $random;
```

```
      write = 1;
      #1 address = address + 1;
   end
   write = 0;
   address = 0;
   #3 repeat (16) begin
      read = 1;
      #1 address = address + 1;
   end
   read = 0;
   #1 $finish;
end

genvar i;

generate
for (i=0; i < 4; i=i+1) begin : MEM
   memory U (read, write, data_in[(i*8)+7:(i*8)], address,data_out[(i*8)+7:(i*8)]);
end
endgenerate

endmodule

// Lower module that will be connected muliple times
module memory (
input wire read,
input wire write,
input wire [7:0] data_in,
input wire [3:0] address,
output reg [7:0] data_out
);

reg [7:0] mem [0:15];

always @ (*)
begin
   if (write) mem[address] = data_in;
end

always @ (read, address)
begin
   if (read) data_out = mem[address];
end

endmodule
```

Multi Dimension Array

Verilog 1995 allows one−dimensional arrays of variables. Verilog 2001 allows more than two dimension with support for arrays of variable and net data types. While array assignments, Verilog 2001 allows part selection and variables to used in part selection as shown in below example.

```
module multi_array();

reg read_v95, read_multi, read_bit;

// Verilog 1995 and Verilog 2001 allow
// 1 dimensional arrays
reg [7:0] address;
reg [7:0] memory [0:255];
wire [7:0] data_out;

assign data_out = (read_v95) ? memory[address] : 0;

// Verilog 2001 allows multi dimention array
reg [7:0] address1, address2;
reg [15:0] array [0:255][0:255];
wire [7:0] data_array = (read_multi) ? array[address1][address2] : 0;

// Verilog 2001 allows bit and part select within arrays
wire [7:0] data_bit = (read_bit) ? array[1][200][12:5] : 8'b0;

// Verilog 2001 allows indexed vector part selects
reg [31:0] double_word;
reg [2:0] byte_no;
wire [7:0] pos_offset = double_word[byte_no*8 +:8];
wire [7:0] neg_offset = double_word[byte_no*8 -:8];

initial begin
   // Check bit and part select within array
   #1 array[1][200] = 16'h1234;
   #1 read_bit  = 1;
   #1 array[1][200] = 16'hAAAA;
   // Check indexed vector part selects
   double_word = 32'hDEAD_BEEF;
   #1 byte_no = 2;
   #1 byte_no = 1;
   #1 $finish;
end

always @ (*)
$display ( "double_word : %h byte_no : %d pos_offset : %h neg_offset : %h" , double_word, byte_no, pos_offset, neg_offset);

always @ (*)
$display ( "array[1][200] : %h read_bit : %b data_bit : %h" , array[1][200],read_bit, data_bit);

endmodule
```

Re–entrant tasks and recursive functions

Verilog 2001 adds a new keyword, automatic. This keyword when added to task, makes the task re–entrant. All task declarations with automatic are allocated dynamically for each concurrent task entry.

A function with added automatic keyword allows the function to be called recursively.

Example : Task

```
module re_entrant_task();

task automatic print_value;
input [7:0] value;
input [7:0] delay;
begin
  #(delay) $display( "%g Passed Value %d Delay %d" , $time, value, delay);
end
endtask

initial begin
  fork
    #1 print_value (10,7);
    #1 print_value (8,5);
    #1 print_value (4,2);
  join
  #1 $finish;
end

endmodule
```

Example : Function

```
module recursuve_function();

function automatic [31:0] calFactorial;
input [7:0] number;
begin
  if (number == 1) begin
    calFactorial = 1;
  end else begin
    calFactorial = number * (calFactorial(number - 1));
  end
end
endfunction

initial begin
  $display ( "Factorial of 1 : %d" , calFactorial(1));
  $display ( "Factorial of 4 : %d" , calFactorial(4));
  $display ( "Factorial of 8 : %d" , calFactorial(8));
  $display ( "Factorial of 16 : %d" , calFactorial(16));
  $display ( "Factorial of 32 : %d" , calFactorial(32));
  #1 $finish;
end

endmodule
```

In-line parameter passing by name

Verilog 1995 allows 2 ways to override parameters declared in a module.

- defparam
- By using #

Verilog 2001 adds way to fix the problems that are introduced by # usage. This is done as same as port connection by name. i.e parameter are over ridden by name unlike by position.

```
module parameter_v2k();
parameter D_WIDTH = 4;
parameter A_WIDTH = 9;

reg [A_WIDTH-1:0] address = 0;
reg [D_WIDTH-1:0] data_in = 0;
wire [D_WIDTH-1:0] data_out;
reg rd,wr;

initial begin
   $monitor ( "%g addr %d din %h dout %d read %b write %b" ,
   $time, address, data_in, data_out, rd, wr);
   rd = 0;
   wr = 0;
   #1 repeat (10) begin
      wr = 1;
      data_in = $random;
      #1 address = address + 1;
   end
   wr = 0;
   address  = 0;
   data_in = 0;
   #1 repeat (10) begin
      rd = 1;
      #1 address = address + 1;
   end
   rd = 0;
   #1 $finish;
end

memory #(.AWIDTH(A_WIDTH),.DWIDTH(D_WIDTH)) U (address, data_in, data_out, rd, wr);

endmodule

// Memory model
module memory (address, data_in, data_out, rd, wr);
parameter DWIDTH = 8;
parameter AWIDTH = 8;
parameter DEPTH  = 1 << AWIDTH;

input [AWIDTH-1:0] address;
input [DWIDTH-1:0] data_in;
output [DWIDTH-1:0] data_out;
input rd,wr;

reg [DWIDTH-1:0] mem [0:DEPTH-1];

always @ (*)
if (wr) mem[address] = data_in;
```

```

assign data_out = (rd) ? mem[address] : 0;

endmodule
```

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.

NOTES

--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--
--

ASSERTIONS IN VERILOG
CHAPTER 20

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 writing 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.

Normally assertions are implemented by the designers to safe guard their design, so they code the assertions into their RTL. Simple example of a assertion would be, writing into FIFO, when it is full. Traditionally verification engineers have been using assertions in their verification environments without knowing that they are assertion. For verification simple application of assertion would be checking protocols. Example, expecting the grant of a arbiter to be asserted after one clock cycle and before two cycles after the assertion of request.

New few pages we will see simple examples on usage of assertions using Open Verification Library and PSL assertions.

What You Need?

For using Open Verification Library examples you need Open Verification Library from Accellera. For running PSL examples you need simulator that can support PSL.

Then you need bit of patience to go through the manuals to learn in details Assertions and try out more examples.

Verification Of FIFO

Our first example is verification of synchronous FIFO. Here we will build simple testbench around the FIFO model and use simple assertions to show how they can be used to check simple protocol. If you have any better suggestion, please let me know.

FIFO Model

Below code is original code found in examples directory.

```
//=============================================
// Function : Synchronous (single clock) FIFO
// Coder : Deepak Kumar Tala
// Date : 31−October−2002
//=============================================
module syn_fifo (
clk , // Clock input
rst , // Active high reset
wr_cs , // Write chip select
rd_cs , // Read chipe select
data_in , // Data input
rd_en , // Read enable
wr_en , // Write Enable
data_out , // Data Output
```

```
empty , // FIFO empty
full // FIFO full
);

// FIFO constants
parameter DATA_WIDTH = 8;
parameter ADDR_WIDTH = 8;
parameter RAM_DEPTH = (1 << ADDR_WIDTH);
// Port Declarations
input clk ;
input rst ;
input wr_cs ;
input rd_cs ;
input rd_en ;
input wr_en ;
input [DATA_WIDTH-1:0] data_in ;
output full ;
output empty ;
output [DATA_WIDTH-1:0] data_out ;

//-----------Internal variables-------------------
reg [ADDR_WIDTH-1:0] wr_pointer;
reg [ADDR_WIDTH-1:0] rd_pointer;
reg [ADDR_WIDTH :0] status_cnt;
reg [DATA_WIDTH-1:0] data_out ;
wire [DATA_WIDTH-1:0] data_ram ;

//-----------Variable assignments---------------
assign full = (status_cnt == (RAM_DEPTH-1));
assign empty = (status_cnt == 0);

//-----------Code Start---------------------------
always @ (posedge clk or posedge rst)
begin : WRITE_POINTER
  if (rst) begin
    wr_pointer <= 0;
  end else if (wr_cs && wr_en ) begin
    wr_pointer <= wr_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_POINTER
  if (rst) begin
    rd_pointer <= 0;
  end else if (rd_cs && rd_en ) begin
    rd_pointer <= rd_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_DATA
  if (rst) begin
    data_out <= 0;
  end else if (rd_cs && rd_en ) begin
    data_out <= data_ram;
  end
end
```

```

always @ (posedge clk or posedge rst)
begin : STATUS_COUNTER
  if (rst) begin
    status_cnt <= 0;
    // Read but no write.
  end else if ((rd_cs && rd_en) && !(wr_cs && wr_en)
  && (status_cnt != 0)) begin
    status_cnt <= status_cnt - 1;
    // Write but no read.
  end else if ((wr_cs && wr_en) && !(rd_cs && rd_en)
  && (status_cnt != RAM_DEPTH)) begin
    status_cnt <= status_cnt + 1;
  end
end

ram_dp_ar_aw #(DATA_WIDTH,ADDR_WIDTH)DP_RAM (
.address_0 (wr_pointer) , // address_0 input
.data_0 (data_in) , // data_0 bi-directional
.cs_0 (wr_cs) , // chip select
.we_0 (wr_en) , // write enable
.oe_0 (1'b0) , // output enable
.address_1 (rd_pointer) , // address_q input
.data_1 (data_ram) , // data_1 bi-directional
.cs_1 (rd_cs) , // chip select
.we_1 (1'b0) , // Read enable
.oe_1 (rd_en) // output enable
);

endmodule
```

Ram Model

```
//=============================================
// Function : Asynchronous read write RAM
// Coder : Deepak Kumar Tala
// Date : 18-April-2002
//=============================================
module ram_dp_ar_aw (
address_0 , // address_0 Input
data_0 , // data_0 bi-directional
cs_0 , // Chip Select
we_0 , // Write Enable/Read Enable
oe_0 , // Output Enable
address_1 , // address_1 Input
data_1 , // data_1 bi-directional
cs_1 , // Chip Select
we_1 , // Write Enable/Read Enable
oe_1 // Output Enable
);

parameter DATA_WIDTH = 8 ;
```

```
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;

//--------------Input Ports-----------------------
input [ADDR_WIDTH-1:0] address_0 ;
input cs_0 ;
input we_0 ;
input oe_0 ;
input [ADDR_WIDTH-1:0] address_1 ;
input cs_1 ;
input we_1 ;
input oe_1 ;

//--------------Inout Ports-----------------------
inout [DATA_WIDTH-1:0] data_0 ;
inout [DATA_WIDTH-1:0] data_1 ;

//--------------Internal variables----------------
reg [DATA_WIDTH-1:0] data_0_out ;
reg [DATA_WIDTH-1:0] data_1_out ;
reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];

//--------------Code Starts Here------------------
// Memory Write Block
// Write Operation : When we_0 = 1, cs_0 = 1
always @ (address_0 or cs_0 or we_0 or data_0
or address_1 or cs_1 or we_1 or data_1)
begin : MEM_WRITE
  if ( cs_0 && we_0 ) begin
    mem[address_0] <= data_0;
  end else if (cs_1 && we_1) begin
    mem[address_1] <= data_1;
  end
end

// Tri-State Buffer control
// output : When we_0 = 0, oe_0 = 1, cs_0 = 1
assign data_0 = (cs_0 && oe_0 && !we_0) ? data_0_out : 8'bz;

// Memory Read Block
// Read Operation : When we_0 = 0, oe_0 = 1, cs_0 = 1
always @ (address_0 or cs_0 or we_1 or oe_0)
begin : MEM_READ_0
  if (cs_0 && !we_0 && oe_0) begin
    data_0_out <= mem[address_0];
  end else begin
    data_0_out <= 0;
  end
end

//Second Port of RAM
// Tri-State Buffer control
// output : When we_0 = 0, oe_0 = 1, cs_0 = 1
assign data_1 = (cs_1 && oe_1 && !we_1) ? data_1_out : 8'bz;
// Memory Read Block 1
// Read Operation : When we_1 = 0, oe_1 = 1, cs_1 = 1
always @ (address_1 or cs_1 or we_1 or oe_1)
begin : MEM_READ_1
  if (cs_1 && !we_1 && oe_1) begin
```

```
      data_1_out <= mem[address_1];
    end else begin
      data_1_out <= 0;
    end
end

endmodule // End of Module ram_dp_ar_aw
```

Testbench Code

In below testbench code below, we are causing over flow and under flow condition. What I mean to say is, FIFO depth is 8, so we can do 8 writes, without reading from FIFO. If we do 9 writes then 9th data over writes the content of FIFO.

Similarly if we read from FIFO, when FIFO is empty it causes underflow. This kind of things happen when the code interface block is buggy. We can code assertion either in RTL or in the testbench. For assertions like our example, it is better that the RTL designer codes it along with his code.

```
module fifo_tb ();
parameter DATA_WIDTH = 8;
// Limit depth to 8
parameter ADDR_WIDTH = 3;

reg clk, rst, rd_en, wr_en;
reg [DATA_WIDTH-1:0] data_in ;
wire [DATA_WIDTH-1:0] data_out ;
wire empty, full;
integer i;

initial begin
  $monitor ( "%g wr:%h wr_data:%h rd:%h rd_data:%h" , $time, wr_en, data_in, rd_en, data_out);
  clk = 0;
  rst = 0;
  rd_en = 0;
  wr_en = 0;
  data_in = 0;
  #5 rst = 1;
  #5 rst = 0;
  @ (negedge clk);
  wr_en = 1;
  // We are causing over flow
  for (i = 0 ; i < 10; i = i + 1) begin
    data_in  = i;
    @ (negedge clk);
  end
  wr_en  = 0;
  @ (negedge clk);
  rd_en = 1;
  // We are causing under flow
  for (i = 0 ; i < 10; i = i + 1) begin
    @ (negedge clk);
```

```
   end
   rd_en = 0;
   #100 $finish;
end

always #1 clk = !clk;

syn_fifo #(DATA_WIDTH,ADDR_WIDTH) fifo(
.clk (clk) , // Clock input
.rst (rst) , // Active high reset
.wr_cs (1'b1) , // Write chip select
.rd_cs (1'b1) , // Read chipe select
.data_in (data_in) , // Data input
.rd_en (rd_en) , // Read enable
.wr_en (wr_en) , // Write Enable
.data_out (data_out), // Data Output
.empty (empty) , // FIFO empty
.full (full) // FIFO full
);

endmodule
```

Assertion with OVL

Now that we have seen the code of FIFO and the testbench, lets see the example of using OVL to build assertions for the FIFO. To use OVL, we need to first install the OVL package. Then we need include the assertion file that we need to use. In our example we are using assert_fifo_index.vlib, we use synopsys translate_off to make the synthesis tools not to read the code within **synopsys translate_off** and **synopsys translate_on**. We want to do this, as this is simulation code and not meant for synthesis. Next we need to enable assertions by `define OVL_ASSERT_ON. There are many other defines that we can use to control the OVL assertion, details of each of the options can be found in the OVL manual.

Assertion in RTL

In the below code, we use following assertions to get feel of OVL assertions

- **assert_fifo_index :** Prints error when ever there is over flow or under flow error.
- **assert_always :** Prints error when ever write happens when fifo full flag is set
- **assert_never :** Prints error when ever read happens when fifo empty flag is set
- **assert_increment :** Prints error when ever write pointer increments by value > 1

We can set the various parameters for each of the assertions to get desired function, for assert_fifo_index, the parameters that need to be set are. For other refer OVL documentation.

- severity_level : `OVL_ERROR, This can be set to fatal, or warning, so on.
- depth of FIFO : This is set to depth of FIFO.

• msg : Message that we want to print.

Details on usage of this assertion could be found from OVL manual. Now to compile we need to pass the +incdir path to VLIB install directory.

Example : Lets assume vlib is installed in home directory (/home/deepak) with vlib as name and we are using verilog XL to compile.

verilog +incdir+/home/deepak/vlib syn_fifo_assert.v fifo_tb.v ram_dp_ar_aw.v

```
//=============================================
// Function : Synchronous (single clock) FIFO
// With Assertion
// Coder : Deepak Kumar Tala
// Date : 1-Nov-2005
//=============================================
// synopsys translate_off
`define OVL_ASSERT_ON
`define OVL_INIT_MSG
`include "assert_fifo_index.vlib"
`include "assert_always.vlib"
`include "assert_never.vlib"
`include "assert_increment.vlib"
// synopsys translate_on
module syn_fifo (
clk , // Clock input
rst , // Active high reset
wr_cs , // Write chip select
rd_cs , // Read chipe select
data_in , // Data input
rd_en , // Read enable
wr_en , // Write Enable
data_out , // Data Output
empty , // FIFO empty
full // FIFO full
);

// FIFO constants
parameter DATA_WIDTH = 8;
parameter ADDR_WIDTH = 8;
parameter RAM_DEPTH = (1 << ADDR_WIDTH);
// Port Declarations
input clk ;
input rst ;
input wr_cs ;
input rd_cs ;
input rd_en ;
input wr_en ;
input [DATA_WIDTH-1:0] data_in ;
output full ;
output empty ;
output [DATA_WIDTH-1:0] data_out ;

//-----------Internal variables-------------------
```

```
reg [ADDR_WIDTH-1:0] wr_pointer;
reg [ADDR_WIDTH-1:0] rd_pointer;
reg [ADDR_WIDTH :0] status_cnt;
reg [DATA_WIDTH-1:0] data_out ;
wire [DATA_WIDTH-1:0] data_ram ;

//-----------Variable assignments---------------
assign full = (status_cnt == (RAM_DEPTH-1));
assign empty = (status_cnt == 0);

//-----------Code Start---------------------------
always @ (posedge clk or posedge rst)
begin : WRITE_POINTER
  if (rst) begin
    wr_pointer <= 0;
  end else if (wr_cs && wr_en ) begin
    wr_pointer <= wr_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_POINTER
  if (rst) begin
    rd_pointer <= 0;
  end else if (rd_cs && rd_en ) begin
    rd_pointer <= rd_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_DATA
  if (rst) begin
    data_out <= 0;
  end else if (rd_cs && rd_en ) begin
    data_out <= data_ram;
  end
end

always @ (posedge clk or posedge rst)
begin : STATUS_COUNTER
  if (rst) begin
    status_cnt <= 0;
    // Read but no write.
  end else if ((rd_cs && rd_en) && !(wr_cs && wr_en)
  && (status_cnt != 0)) begin
    status_cnt <= status_cnt - 1;
    // Write but no read.
  end else if ((wr_cs && wr_en) && !(rd_cs && rd_en)
  && (status_cnt != RAM_DEPTH)) begin
    status_cnt <= status_cnt + 1;
  end
end

ram_dp_ar_aw #(DATA_WIDTH,ADDR_WIDTH) DP_RAM (
.address_0 (wr_pointer) , // address_0 input
.data_0 (data_in) , // data_0 bi-directional
```

```
.cs_0 (wr_cs) , // chip select
.we_0 (wr_en) , // write enable
.oe_0 (1'b0) , // output enable
.address_1 (rd_pointer) , // address_q input
.data_1 (data_ram) , // data_1 bi-directional
.cs_1 (rd_cs) , // chip select
.we_1 (1'b0) , // Read enable
.oe_1 (rd_en) // output enable
);

// Add assertion here
// synopsys translate_off
// Assertion to check overflow and underflow
assert_fifo_index #(
`OVL_ERROR , // severity_level
(RAM_DEPTH-1) , // depth
1 , // push width
1 , // pop width
`OVL_ASSERT , // property type
"my_module_err"  , // msg
`OVL_COVER_NONE , //coverage_level
1) no_over_under_flow (
.clk (clk), // Clock
.reset_n (~rst), // Active low reset
.pop (rd_cs & rd_en), // FIFO Write
.push (wr_cs & wr_en) // FIFO Read
);

// Assertion to check full and write
assert_always #(
`OVL_ERROR , // severity_level
`OVL_ASSERT , // property_type
"fifo_full_write" , // msg
`OVL_COVER_NONE // coverage_level
) no_full_write (
.clk (clk),
.reset_n (~rst),
.test_expr (!(full && wr_cs && wr_en))
);

// Assertion to check empty and read
assert_never #(
`OVL_ERROR , // severity_level
`OVL_ASSERT , // property_type
"fifo_empty_read" , // msg
`OVL_COVER_NONE // coverage_level
) no_empty_read (
.clk (clk),
.reset_n (~rst),
.test_expr ((empty && rd_cs && rd_en))
);

// Assertion to check if write pointer increments by just one
assert_increment #(
`OVL_ERROR , // severity_level
```

```
ADDR_WIDTH , // width
1 , // value
`OVL_ASSERT , // property_typ
"Write_Pointer_Error"  , // msg
`OVL_COVER_NONE // coverage_level
) write_count (
.clk (clk),
.reset_n (~rst),
.test_expr (wr_pointer)
);
// synopsys translate_on
endmodule
```

Simulator Output

First OVL message is init message, used to check if the OVL lib was included and ready. my_module_err is message user defined message, that OVL uses in print statements.

Rest of the messages are OVERFLOW and UNDERFLOW messages.

```
OVL_NOTE: `OVL_VERSION: ASSERT_FIFO_INDEX initialized
@ fifo_tb.fifo.no_over_under_flow.ovl_init_msg_t Severity: 1, Message: my_module_err
OVL_NOTE: `OVL_VERSION: ASSERT_ALWAYS initialized
@ fifo_tb.fifo.no_full_write.ovl_init_msg_t Severity: 1, Message: fifo_full_write
OVL_NOTE: `OVL_VERSION: ASSERT_NEVER initialized
@ fifo_tb.fifo.no_empty_read.ovl_init_msg_t Severity: 1, Message: fifo_empty_read
0 wr:0 wr_data:00 rd:0 rd_data:xx
5 wr:0 wr_data:00 rd:0 rd_data:00
10 wr:1 wr_data:00 rd:0 rd_data:00
12 wr:1 wr_data:01 rd:0 rd_data:00
14 wr:1 wr_data:02 rd:0 rd_data:00
16 wr:1 wr_data:03 rd:0 rd_data:00
18 wr:1 wr_data:04 rd:0 rd_data:00
20 wr:1 wr_data:05 rd:0 rd_data:00
22 wr:1 wr_data:06 rd:0 rd_data:00
24 wr:1 wr_data:07 rd:0 rd_data:00
    OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : OVERFLOW : severity 1 :
    time 25 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
    OVL_ERROR : ASSERT_ALWAYS : fifo_full_write :  : severity 1 : time 25 :
    fifo_tb.fifo.no_full_write.ovl_error_t
26 wr:1 wr_data:08 rd:0 rd_data:00
    OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : OVERFLOW : severity 1 :
    time 27 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
28 wr:1 wr_data:09 rd:0 rd_data:00
    OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : OVERFLOW : severity 1 :
    time 29 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
30 wr:0 wr_data:09 rd:0 rd_data:00
32 wr:0 wr_data:09 rd:1 rd_data:00
33 wr:0 wr_data:09 rd:1 rd_data:08
35 wr:0 wr_data:09 rd:1 rd_data:09
39 wr:0 wr_data:09 rd:1 rd_data:03
41 wr:0 wr_data:09 rd:1 rd_data:04
43 wr:0 wr_data:09 rd:1 rd_data:05
45 wr:0 wr_data:09 rd:1 rd_data:06
    OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : UNDERFLOW : severity 1 :
    time 47 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
47 wr:0 wr_data:09 rd:1 rd_data:07
```

```
      OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : UNDERFLOW : severity 1 :
      time 49 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
      OVL_ERROR : ASSERT_NEVER : fifo_empty_read :  : severity 1 : time 49 :
      fifo_tb.fifo.no_empty_read.ovl_error_t
49 wr:0 wr_data:09 rd:1 rd_data:08
      OVL_ERROR : ASSERT_FIFO_INDEX : my_module_err : UNDERFLOW : severity 1 :
      time 51 : fifo_tb.fifo.no_over_under_flow.ovl_error_t
      OVL_ERROR : ASSERT_NEVER : fifo_empty_read :  : severity 1 : time 51 :
      fifo_tb.fifo.no_empty_read.ovl_error_t
51 wr:0 wr_data:09 rd:1 rd_data:09
52 wr:0 wr_data:09 rd:0 rd_data:09
```

OVL Assertion List

Below is list of OVL assertions that are common in usage, you can always refer to the document that comes with the OVL library to get more information and complete list of assertions available.

Assertion	Description
assert_always	The assert_always assertion checker checks the single−bit expression test_expr at each rising edge of clk to verify the expression evaluates to TRUE.
assert_always_on_edge	The assert_always_on_edge assertion checker checks the single−bit expression sampling_event for a particular type of transition.
assert_change	The assert_change assertion checker checks the expression start_event at each rising edge of clk to determine if it should check for a change in the value of test_expr. If start_event is sampled TRUE, the checker evaluates test_expr and re−evaluates test_expr at each of the subsequent num_cks rising edges of clk. If the value of test_expr has not been sampled changed from its start value by the last of the num_cks cycles, the assertion fails.
assert_cycle_sequence	The assert_cycle_sequence assertion checker checks the expression event_sequenceat the rising edges of clk to identify whether or not the bits in event_sequence assert sequentially on successive rising edges of clk.
assert_decrement	The assert_decrement assertion checker checks the expression test_expr at each rising edge of clk to determine if its value has changed from its value at the previous rising edge of clk. If so, the checker verifies that the new value equals the previous value decremented by value. The checker allows the value of test_expr to wrap, if the total change equals the decrement value.
assert_even_parity	The assert_even_parity assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a value that has even parity. A value has even parity if it is 0 or if the number of bits set to 1 is even.
assert_fifo_index	The assert_fifo_index assertion checker tracks the numbers of pushes (writes) and pops (reads) that occur for a FIFO or queue memory structure. This checker does permit simultaneous pushes/ pops on the queue within the same clock cycle. It ensures the FIFO never overflows (i.e., too many pushes occur without enough pops) and

	never underflows (i.e., too many pops occur without enough pushes).
assert_increment	The assert_increment assertion checker checks the expression test_expr at each rising edge of clk to determine if its value has changed from its value at the previous rising edge of clk. If so, the checker verifies that the new value equals the previous value incremented by value. The checker allows the value of test_expr to wrap, if the total change equals the increment value.
assert_never	The assert_never assertion checker checks the single−bit expression test_expr at each rising edge of clk to verify the expression does not evaluate to TRUE.
assert_one_hot	The assert_one_hot assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a one−hot value. A one−hot value has exactly one bit set to 1.
assert_range	The assert_range assertion checker checks the expression test_expr at each rising edge of clk to verify the expression falls in the range from minto max, inclusive. The assertion fails if test_expr< min or max < test_expr.
assert_one_cold	The assert_one_cold assertion checker checks the expression test_expr at each rising edge of clk to verify the expression evaluates to a one−cold or inactive state value. A one−cold value has exactly one bit set to 0.

Assertion with PSL

Now that we have seen the example of FIFO assertion using OVL, lets see the example of using PSL to build assertions for the FIFO. PSL assertions can be coding two ways.

- inline Coding : In this method, psl assertions are coded into verilog code as comment.
- External File : In this method, psl assertions are coded into separate file with vunit as name.

inline Coding

- All assertions appear within a consecutive series of comments appropriate for the context
- The first assertion statement line must begin with the psl keyword.
- Both the psl keyword and the keyword that follows it must be on the same line.
- Specify a label followed by a colon.
- Assert the behavior described in the property by using the assert or assume keyword
- Describe the behavior of the design.
- Assertions cannot be embedded in Verilog tasks, functions, or UDPs
- Example : // psl label: assert behavior;

External File

- To add assertions to an existing design without modifying the source text, as with legacy IP.
- To experiment with assertions before embedding them in the source file.
- When you are working in teams where the assertions are not created by the HDL author.

```
vunit verification_unit_name (module_name) {
  default clock = clock_edge;
  property_name: assert behavior;
  property_name: cover {behavior};
}
```

For more details refer to PSL usage guide that comes with simulator.

Assertion in RTL

In the below code, we use psl assertion to check if no write is done when FIFO is full and also check if no read is done when FIFO is empty.

We can code psl assertion inline with code with // or /* */. Before we write any assertion, we need to declare the clock as in the example.

ncverilog +assert verilog_file1.v verilog_file2.v

```
//=============================================
// Function : Synchronous (single clock) FIFO
// With Assertion
// Coder : Deepak Kumar Tala
// Date : 31-October-2002
//=============================================
module syn_fifo (
clk , // Clock input
rst , // Active high reset
wr_cs , // Write chip select
rd_cs , // Read chipe select
data_in , // Data input
rd_en , // Read enable
wr_en , // Write Enable
data_out , // Data Output
empty , // FIFO empty
full // FIFO full
);

// FIFO constants
parameter DATA_WIDTH = 8;
parameter ADDR_WIDTH = 8;
parameter RAM_DEPTH = (1 << ADDR_WIDTH);
// Port Declarations
input clk ;
input rst ;
```

```
input wr_cs ;
input rd_cs ;
input rd_en ;
input wr_en ;
input [DATA_WIDTH-1:0] data_in ;
output full ;
output empty ;
output [DATA_WIDTH-1:0] data_out ;

//-----------Internal variables-------------------
reg [ADDR_WIDTH-1:0] wr_pointer;
reg [ADDR_WIDTH-1:0] rd_pointer;
reg [ADDR_WIDTH :0] status_cnt;
reg [DATA_WIDTH-1:0] data_out ;
wire [DATA_WIDTH-1:0] data_ram ;

//-----------Variable assignments---------------
assign full = (status_cnt == (RAM_DEPTH-1));
assign empty = (status_cnt == 0);

//-----------Code Start---------------------------
always @ (posedge clk or posedge rst)
begin : WRITE_POINTER
  if (rst) begin
    wr_pointer <= 0;
  end else if (wr_cs && wr_en ) begin
    wr_pointer <= wr_pointer + 1;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_POINTER
  if (rst) begin
    rd_pointer <= 0;
    data_out <= 0;
  end else if (rd_cs && rd_en ) begin
    rd_pointer <= rd_pointer + 1;
    data_out <= data_ram;
  end
end

always @ (posedge clk or posedge rst)
begin : READ_DATA
  if (rst) begin
    data_out <= 0;
  end else if (rd_cs && rd_en ) begin
    data_out <= data_ram;
  end
end

always @ (posedge clk or posedge rst)
begin : STATUS_COUNTER
  if (rst) begin
    status_cnt <= 0;
    // Read but no write.
  end else if ((rd_cs && rd_en) && !(wr_cs && wr_en)
  && (status_cnt != 0)) begin
```

```
84        status_cnt <= status_cnt - 1;
85        // Write but no read.
86    end else if ((wr_cs && wr_en) && !(rd_cs && rd_en)
87    && (status_cnt != RAM_DEPTH)) begin
88        status_cnt <= status_cnt + 1;
89    end
90 end
91
92 ram_dp_ar_aw #(DATA_WIDTH,ADDR_WIDTH) DP_RAM (
93 .address_0 (wr_pointer) , // address_0 input
94 .data_0 (data_in) , // data_0 bi-directional
95 .cs_0 (wr_cs) , // chip select
96 .we_0 (wr_en) , // write enable
97 .oe_0 (1'b0) , // output enable
98 .address_1 (rd_pointer) , // address_q input
99 .data_1 (data_ram) , // data_1 bi-directional
100 .cs_1 (rd_cs) , // chip select
101 .we_1 (1'b0) , // Read enable
102 .oe_1 (rd_en) // output enable
103 );
104
105 // Add assertion here
106 // psl default clock = (posedge clk);
107 // psl ERRORwritefull: assert never {full && wr_en && wr_cs};
108 // psl ERRORreadempty: assert never {empty && rd_en && rd_cs};
109
110 endmodule
```

Simulator Output

When ever there is violation, Assertion is printed.

```
0 wr:0 wr_data:00 rd:0 rd_data:xx
5 wr:0 wr_data:00 rd:0 rd_data:00
10 wr:1 wr_data:00 rd:0 rd_data:00
12 wr:1 wr_data:01 rd:0 rd_data:00
14 wr:1 wr_data:02 rd:0 rd_data:00
16 wr:1 wr_data:03 rd:0 rd_data:00
18 wr:1 wr_data:04 rd:0 rd_data:00
20 wr:1 wr_data:05 rd:0 rd_data:00
22 wr:1 wr_data:06 rd:0 rd_data:00
24 wr:1 wr_data:07 rd:0 rd_data:00
ncsim: *E,ASRTST (syn_fifo_psl.v,107): (time 25 NS)
Assertion fifo_tb.fifo.ERRORwritefull has failed
26 wr:1 wr_data:08 rd:0 rd_data:00
28 wr:1 wr_data:09 rd:0 rd_data:00
30 wr:0 wr_data:09 rd:0 rd_data:00
32 wr:0 wr_data:09 rd:1 rd_data:00
33 wr:0 wr_data:09 rd:1 rd_data:08
35 wr:0 wr_data:09 rd:1 rd_data:09
39 wr:0 wr_data:09 rd:1 rd_data:03
41 wr:0 wr_data:09 rd:1 rd_data:04
43 wr:0 wr_data:09 rd:1 rd_data:05
45 wr:0 wr_data:09 rd:1 rd_data:06
47 wr:0 wr_data:09 rd:1 rd_data:07
ncsim: *E,ASRTST (syn_fifo_psl.v,108): (time 49 NS)
Assertion fifo_tb.fifo.ERRORreadempty has failed
49 wr:0 wr_data:09 rd:1 rd_data:08
```

```
ncsim: *E,ASRTST (syn_fifo_psl.v,108): (time 51 NS)
Assertion fifo_tb.fifo.ERRORreadempty has failed
51 wr:0 wr_data:09 rd:1 rd_data:09
52 wr:0 wr_data:09 rd:0 rd_data:09
Simulation complete via $finish(1) at time 152 NS + 0
fifo_tb.v:36   #100 $finish;
```

Post Processing

Like with any other simulation, we need to have post processing scripts to process the messages that are printed by simulator to declare if the simulation as passed or failed.

NOTES

−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−
−−

COMPILER DIRECTIVES

CHAPTER 21

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 language, 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 simulation 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 22

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 */