Module 4

Design of Embedded Processors

Lesson 23

Introduction to Hardware Description Languages-III

Instructional Objectives

At the end of the lesson the student should be able to

- Interface Verilog code to C & C++ using Programming Language Interface
- Synthesize a Verilog code and generate a netlist for layout
- Verify the generated code, and carry out optimization and debugging
- Classify various types of flows in Verification

3.1 Programming Language interface

3.1.1 Verilog

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

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

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

To achieve the 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 access routines***

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.

The block diagram representing above is as follows:

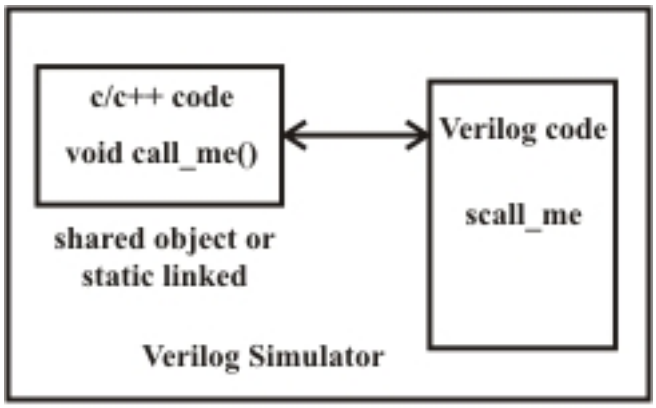


Fig. 23.1

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

Example - Hello World

Define a function ***hello ( ),*** which when called will print "Hello World". This example does not use any of the PLI standard functions (ACC, TF and VPI). For exact linking details, the simulator manuals must be referred. Each simulator implements its own strategy for linking with the C/C++ functions.

C Code

```
#include < stdio.h >
Void hello () {
printf ( "\nHello World\n" );
```

Verilog Code

```
module hello_pli ();
initial begin
$hello;
#10 $finish;
end
endmodule
```

3.1.2 Running a Simulation

Once linking is done, simulation is run as a normal simulation with slight modification to the command line options. These modifications tell the simulator that the PLI routines are being used (e.g. Modelsim needs to know which shared objects to load in command line).

Writing PLI Application **(**counter example**)**

Write the DUT reference model and Checker in C and link that to the Verilog Testbench.

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 the above requirements

3.1.3 PLI Application Specification

This can be well understood in context to the above counter logic. The objective is to design the PLI function ***$counter_monitor*** and check the response of the designed counter using it. This problem can be addressed to in the following steps:

- Implement the Counter logic in C.
- Implement the Checker logic in C.
- Terminate the simulation, whenever the checker fails.

This is represented in the block diagram in the figure 23.2.

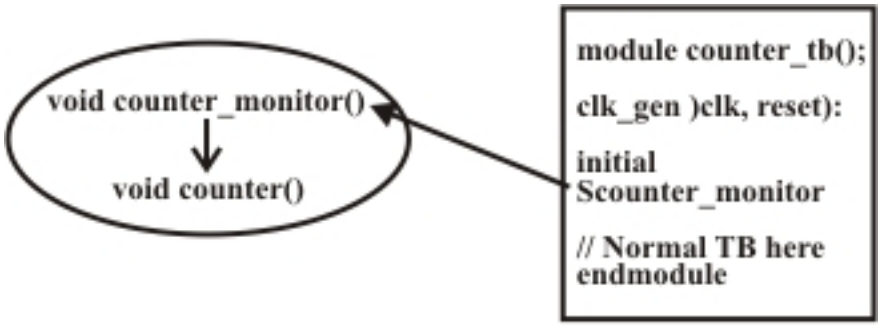


Fig. 23.2

Calling the C function

The change in clock signal is monitored and with its change the counter function is executed The ***acc_vcl_add*** routine is used. The syntax can be obtained in the Verilog PLI LRM.

acc_vcl_add routine basically monitors the list of signals and whenever any of the monitor signals change, it calls the user defined function (this function is called the ***Consumer C routine***). The ***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);

C Code – Basic

The desired C function is ***Counter_monitor*** , which is called from the Verilog Testbench. As like any other C code, header files specific to the application are included.Here the include e file comprises of the ***acc routines***.

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

```
#include < stdio.h >
#include "acc_user.h"
typedef char * string;
handle clk ;
handle reset ;
handle enable ;
handle dut_count ;
int count ;
void counter_monitor()
{
acc_initialize();
clk = acc_handle_tfarg(1);
reset = acc_handle_tfarg(2);
enable = acc_handle_tfarg(3);
dut_count = acc_handle_tfarg(4);
acc_vcl_add(clk,counter,null,vcl_verilog_logic);
acc_close();
}
void counter ()
printf( "Clock changed state\n" );
```

Handles are used for accessing the Verilog objects. The handle is a predefined data type that is a pointer to a specific object in the design hierarchy. Each handle conveys information to access routines about a unique instance of an accessible object information about the object type and, also, how and where the data pertaining to it can be obtained. The information of specific object

to handle can be passed from the Verilog code as a parameter to the function ***$counter_monitor.*** This parameters can be accessed through the C-program with ***acc_handle_tfarg( )*** routine.

For instance ***clk = acc_handle_tfarg(1)*** basically makes that the ***clk*** is a handle to the first parameter passed. Similarly, all the other handles are assigned ***clk*** can now be added to the signal list that needs 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 the ***clk*** changes.

Verilog Code

Below is the code of a simple testbench for the counter example. If the object being passed is an ***instance***, then it should be passed inside double quotes. Since here all the objects are ***nets*** or ***wires***, there is no need to pass them inside the double quotes.

```
module counter_tb();
reg enable;;
reg reset;
reg clk_reg;
wire clk;
wire [3:0] count;
initial begin
clk = 0;
reset = 0;
$display( "Asserting reset" );
#10 reset = 1;
#10 reset = 0;
$display ( "Asserting Enable" );
#10 enable = 1;
#20 enable = 0;
$display ( "Terminating Simulator" );
#10 $finish;

End

Always
#5 clk_reg = !clk_reg;
assign clk = clk_reg;
initial begin
$counter_monitor(top.clk,top.reset,top.enable,top.count);
end

counter U(
clk (clk),
reset (reset),
enable (enable),
count (count)
);
endmodule
```

Access Routines

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

- Extract information pertaining to an object from the internal data representation.
- Write information pertaining to an object into the internal data representation.

Program Flow using access routines

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( )***

Utility Routines

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

3.2 Verilog and Synthesis

3.2.1 What is logic synthesis?

Logic synthesis is the process of converting a ***high-level description*** of design into an optimized ***gate-level netlist*** representation. Logic synthesis uses ***standard cell libraries*** which consist of ***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 form the ***technology library***. Normally, technology library is known by the minimum feature size ***(0.18u, 90nm).***
A circuit description is written in Hardware description language (HDL) such as Verilog Design constraints such as timing, area, testability, and power are considered during synthesis. Typical design flow with a large example is given in the last example of this lesson.

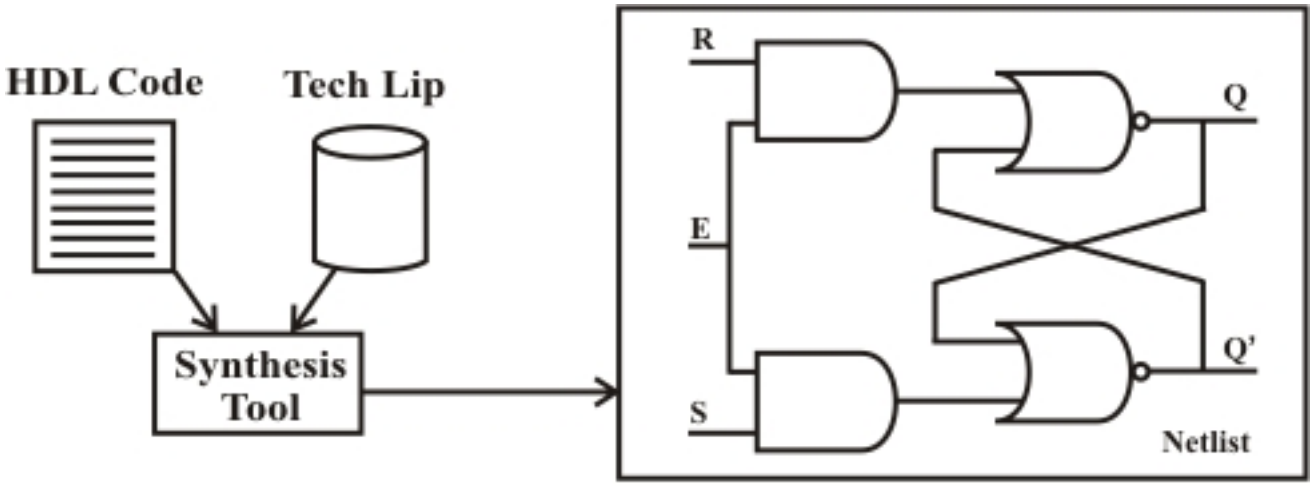


Fig. 23.3

3.2.2 Impact of automation on Logic synthesis

For large designs, manual conversions of the behavioral description to the gate-level representation are more prone to error. Prior to the development of modern sophisticated synthesis tools the earlier designers could never be sure that whether after fabrication the design constraints will be met. Moreover, a significant time of the design cycle was consumed in converting the high–level design into its gate level representation. On account of these, if the gate level design did not meet the requirements then the turnaround time for redesigning the blocks was also very high. Each designer implemented design blocks and there was very little consistency in design cycles, hence, although the individual blocks were optimized but the overall design still contained redundant logics. Moreover, timing, area and power dissipation was fabrication process specific and, hence, with the change of processes the entire process needed to be changed with the design methodology.

However, now automated logic synthesis has solved these problems. The high level design is less prone to human error because designs are described at higher levels of abstraction. High level design is done without much concentration on the constraints. The tool takes care of all the constraints and sees to it that the constraints are taken care of. The designer can go back, redesign and synthesize once again very easily if some aspect is found unaddressed. The turnaround time has also fallen down considerably. Automated logic synthesis tools synthesize the design as a whole and, thus, an overall design optimization is achieved. Logic synthesis allows a technology independent design. The tools convert the design into gates using cells from the standard cell library provided by the vendor.

Design reuse is possible for technology independent designs. If the technology changes the tool is capable of mapping accordingly.

Constructs Not Supported in Synthesis	
Construct Type	**Notes**
Initial	*Only in testbenches*
event	*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*

Example of a Non-Synthesizable Verilog construct

Codes containing one or more of the above constructs are not synthesizable. But even with synthesizable constructs, bad coding may cause serious synthesis concerns.

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 are also non-synthesizable e.g. 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, treating the above code as just a = b.

3.2.3 Constructs and Their Description

Construct Type	Keyword	Description
ports	***input, inout, output***	Use inout only at IO level.
parameters	***parameter***	This makes design more generic
module definition	***module***	
signals and variables	***wire, reg, tri***	Vectors are allowed
instantiation	***module instances primitive gate instances***	Eg- nand (out,a,b) bad idea to code RTL this way.
function and tasks	***function , task***	Timing constructs ignored

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

3.2.4 Operators and Their Description

Operator Type	Operator Symbol	DESCRIPTION
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

Constructs Supported In Synthesis

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

3.2.5 Overall Logic Circuit Modeling and Synthesis in brief

Combinational Circuit modeling using assign

RTL description – This comprises the high level description of the circuit incorporating the RTL constructs. Some functional verification is also done at this level to ensure the validity of the RTL description.

RTL for magnitude comparator

```
// module magnitude comparator
module magnitude_comparator(A_gt_B, A_lt_B, A_eq_B, A,_B);
//comparison output;
output A_gt_B, A_lt_B, A_eq_B ;
// 4- bit numbers input
input [3:0] A,B;
assign A_gt_B= (A>B) ; // A greater than B
assign A_lt_B= (A<B) ; // A greater than B
```

assign A_eq_B= (A==B) ; // A greater than B
endmodule

Translation

The RTL description is converted by the logic synthesis tool to an optimized, intermediate, internal representation. It understands the basic primitives and operators in the Verilog RTL description but overlooks any of the constraints.

Logic optimization

The logic is optimized to remove the ***redundant*** logic. It generates the optimized internal representation.

Technology library

The technology library contains standard library cells which are used during synthesis to replace the behavioral description by the actual circuit components. These are the basic building blocks. Physical layout of these, are done first and then area is estimated. Finally, modeling techniques are used to estimate the power and timing characteristics.
The library includes the following:

- Functionality of the cells
- Area of the different cell layout
- Timing information about the various cells
- Power information of various cells

The synthesis tools use these cells to implement the design.
// Library cells for abc_100 technology
VNAND// 2 – input nand gate
VAND// 2 – input and gate
VNOR // 2 – input nor gate
VOR// 2 – input or gate
VNOT// not gate
VBUF// buffer

Design constraints

Any circuit must satisfy at least three constraints viz. ***area, power*** *and* ***timing***. Optimization demands a compromise among each of these three constraints. Apart from these operating conditions-temperature etc. also contribute to synthesis complexity.

Logic synthesis

The logic synthesis tool takes in the RTL design, and generates an optimized gate level description with the help of technology library, keeping in pace with design constraints.

Verification of the gate –level netlist

An optimized gate level netlist must always be checked for its functionality and, in addition, the synthesis tool must always serve to meet the timing specifications. Timing verification is done in order to manipulate the synthesis parameters in such a way that different timing constraints like input delay, output delay etc. are suitably met.

Functional verification

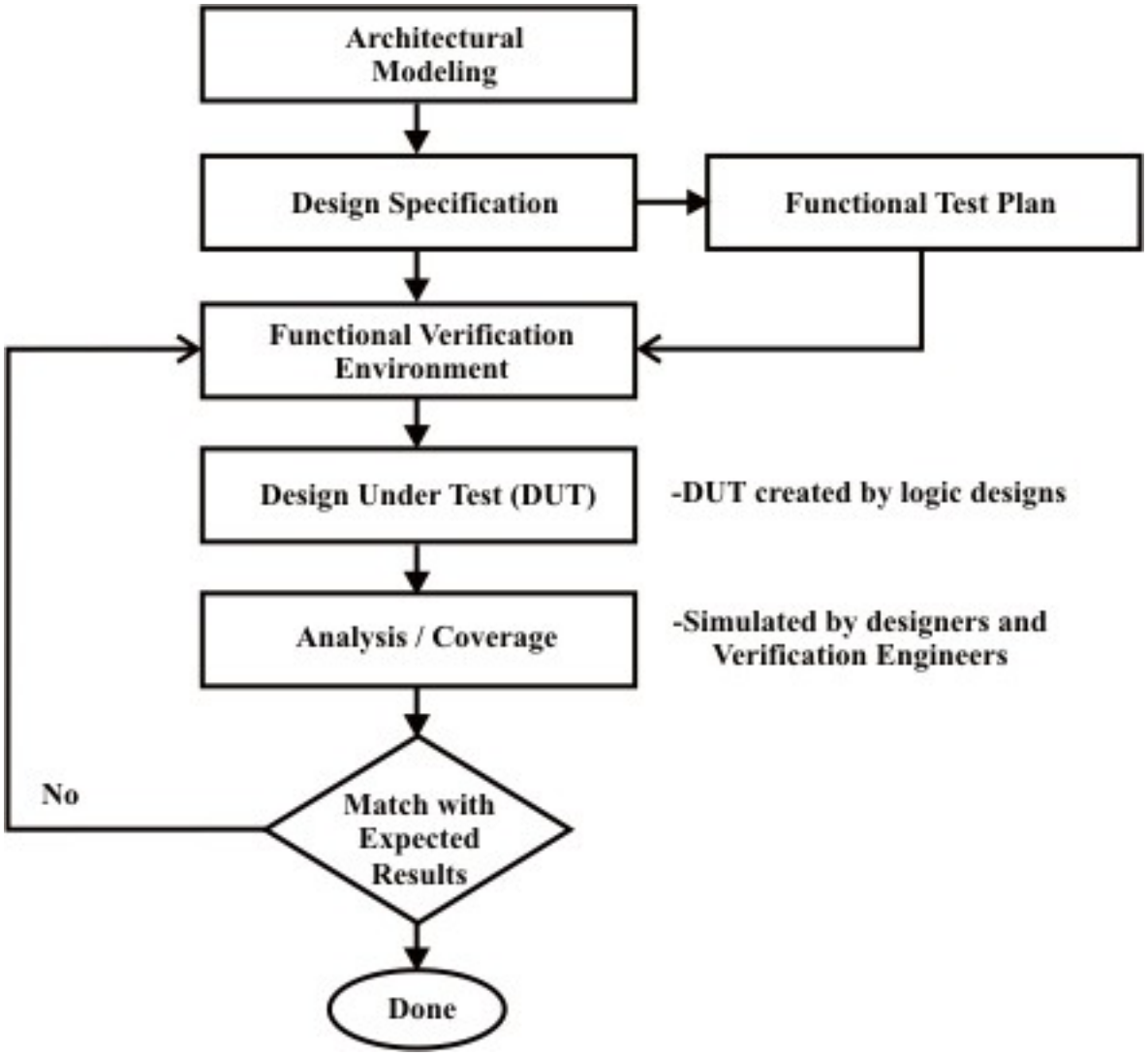
Identical stimulus is run with the original RTL and synthesized gate-level description of the design. The output is compared for matches.

```
module stimulus
reg [3:0] A, B;
wire A_GT_B, A_LT_B,  A_EQ_B;
// instantiate the magnitude comparator MC (A_GT_B, A_LT_B,  A_EQ_B,. A, B);
initial
$ monitor ($time, "A=%b, B=%b, A_GT_B=%b, A_LT_B=%b,  A_EQ_B=%b", A_GT_B,
A_LT_B,  A_EQ_B, A, B)
// stimulate the magnitude comparator

endmodule
```

3.3 Verification

3.3.1 Traditional verification flow



Traditional verification follows the following steps in general.

1. To verify, first a design specification must be set. This requires analysis of architectural trade-offs and is usually done by simulating various architectural models of the design.
2. Based on this specification a functional test plan is created. This forms the framework for verification. Based on this plan various test vectors are applied to the DUT (design under test), written in verilog. Functional test environments are needed to apply these test vectors.
3. The DUT is then simulated using traditional software simulators.
4. The output is then analyzed and checked against the expected results. This can be done manually using waveform viewers and debugging tools or else can be done automatically by verification tools. If the output matches expected results then verification is complete.

5. Optionally, additional steps can be taken to decrease the risk of future design respin. These include Hardware Acceleration, Hardware Emulation and assertion based Verification.

Functional verification

When the specifications for a design are ready, a functional test plan is created based on them. This is the fundamental framework of the functional verification. Based on this test plan, test vectors are selected and given as input to the ***design_under_test(DUT).*** The DUT is simulated to compare its output with the desired results. If the observed results match the expected values, the verification part is over.

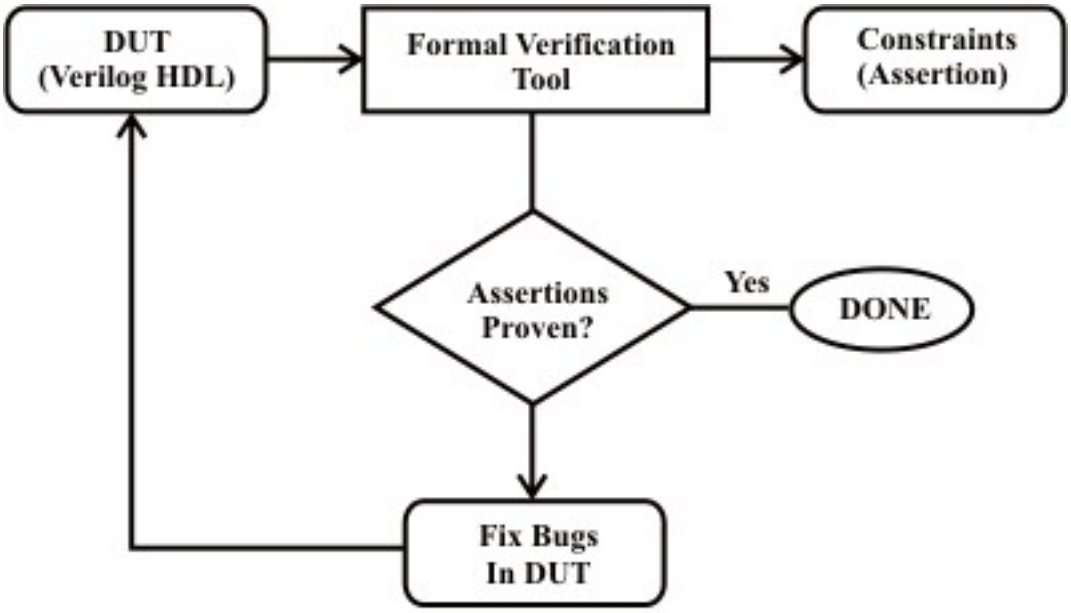
Functional verification Environment

The verification part can be divided into three substages :

- ***Block level verification:*** verification is done for ***blocks*** of code written in verilog using a number of test cases.
- ***Full chip verification:*** The goal of ***full chip verification***, i.e, all the feature of the full chip described in the test plan is complete.
- ***Extended verification:*** This stage depicts the ***corner state bugs.***

3.3.2 Formal Verification

A formal verification tool proves a design by manipulating it as much as possible. All input changes must, however, conform to the constraints for behaviour validation. Assertions on interfaces act as constraints to the formal tool. Assertions are made to prove the assertions in the RTL code false. However, if the constraints are too tight then the tool will not explore all possible behaviours and may wrongly report the design as faulty.
Both the formal and the semi-formal methodologies have come into precedence with the increasing complexity of design.

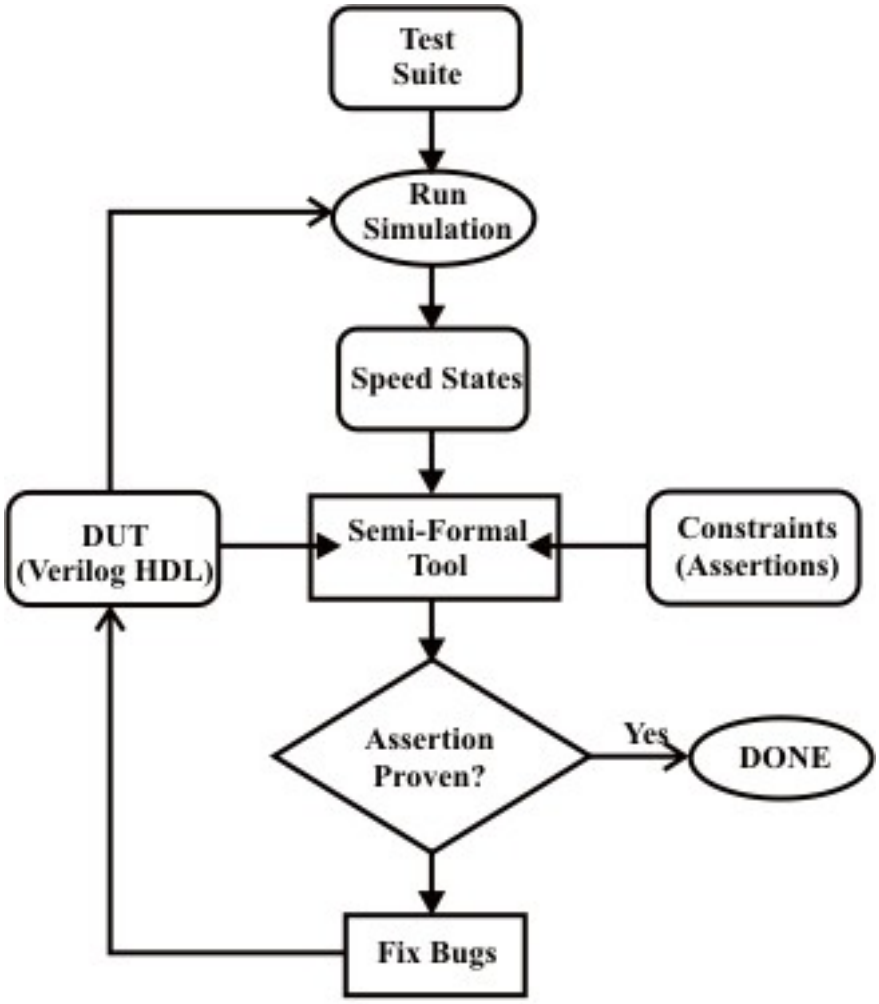


Formal verification flow

3.3.3 Semi- formal verification

Semi formal verification combines the traditional verification flow using test vectors with the power and thoroughness of formal verification.

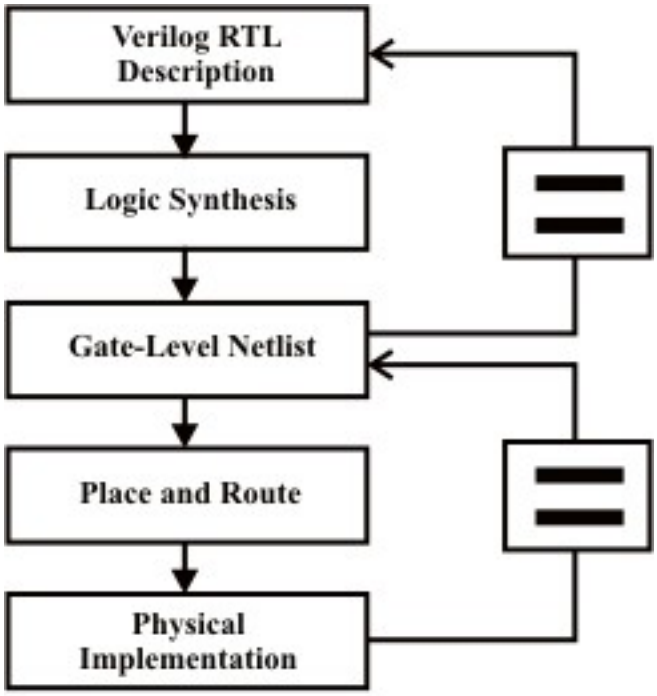
- Semi-formal methods supplement simulation with test vectors
- Embedded assertion checks define the properties targeted by formal methods
- Embedded assertion checks defines the input constraints
- Semi-formal methods explore limited space exhaustibility from the states reached by simulation, thus, maximizing the effect of simulation.The exploration is limited to a certain point around the state reached by simulation.



Semi-formal verification flow

3.3.4 Equivalence checking

After ***logic synthesis*** and ***place and route*** tools create a gate level netlist and physical implementations of the RTL design, respectively, it is necessary to check whether these functionalities match the original RTL design. Here comes equivalence checking. It is an application of formal verification. It ensures that the gate level or physical netlist has the same functionality as the Verilog RTL that was simulated. A logical model of both the RTL and gate level representations is constructed. It is mathematically proved that their functionality are same.



Equivalence checking

3.4 Some Exercises

3.4.1 PLI

i) Write a user defined system task, $count_and_gates, which counts the number of and gate primitive in a module instance. Hierarchical module instance name is the input to the task. Use this task to count the number of and gates in a 4-to-1 multiplexer.

3.4.2 Verilog and Synthesis

i) A 1-bit full subtractor has three inputs x, y, z(previous borrow) and two outputs D(difference) and B(borrow). The logic equations for D & B are as follows

D=x'y'z+ x'yz'+ xy'z' + xyz

B= x'y + x'z+ yz

Write the verilog RTL description for the full subtractor. Synthesize the full using any technology library available. Apply identical stimulus to the RTL and gate level netlist and compare the outputs.

ii) Design a 3-8 decoder, using a Verilog RTL description. A 3-bit input a[2:0] is provided to the decoder. The output of the decoder is out[7:0]. The output bit indexed by a[2:0] gets the value 1, the other bits are 0. Synthesize the decoder, using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and gate level netlist and compare the outputs.

iii) Write the verilog RTL description for a 4-bit binary counter with synchronous reset that is active high.(hint: use always loop with the @ (posedge clock)statement.) synthesize the counter using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and gate level netlist and compare the outputs.