**VTU EXAM –June 2018**

**CBCS SCHEME- SYSTEM VERILOG MTECH VLSI**

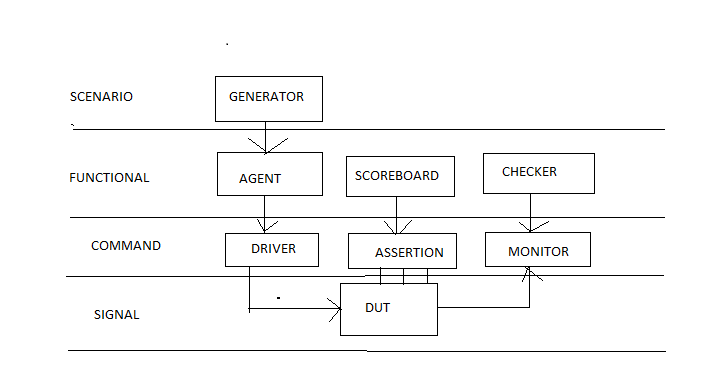
**SUB CODE 16EVE21**

**Module wise SOLUTIONS TO QUESTION PAPER**

**MODULE-1**

**1a. What are all the different levels and testing?Explain.**

At the bottom is the signal layer that contains the design under test and the signals that connect it to the testbench. The next level up is the command layer. The DUT’s inputs are driven by the driver that runs single commands such as bus read or write. The DUT’s output drives the monitor that takes signal transitions and groups them together into commands. Assertions also cross the command/signal layer, as they look at individual signals but look for changes across an entire command.



The functional layer

The functional layer feeds the command layer. The agent block (called the transactor in the VMM) receives higher-level transactions such as DMA read or write and breaks them into individual commands. These commands are also sent to the scoreboard that predicts the results of the transaction. The

checker compares the commands from the monitor with those in the scoreboard.

The scenario layer

The functional layer is driven by the generator in the scenario layer. What is a scenario? Remember that your job as a verification engineer is to make sure that this device accomplishes its intended task. An example device is an MP3 player that can concurrently play music from its storage, download new music from a host, and respond to input from the user, such as volume and track controls. Each of these operations is a scenario. Downloading a music file takes several steps, such as control register reads and writes to set up the operation, multiple DMA writes to transfer the song, and then another group

of reads and writes. The scenario layer of your testbench orchestrates all these steps with constrained-random values for parameters such as track size and memory location.

The blocks in the testbench environment (inside the dashed line) are written at the start of development. During the project they may evolve and you may add functionality, but these blocks should not change for individual tests. This is done by leaving “hooks” in the code so that a test can change the behavior of these blocks without having to rewrite them. You create these hooks with callbacks (section 8.7) and factory patterns.

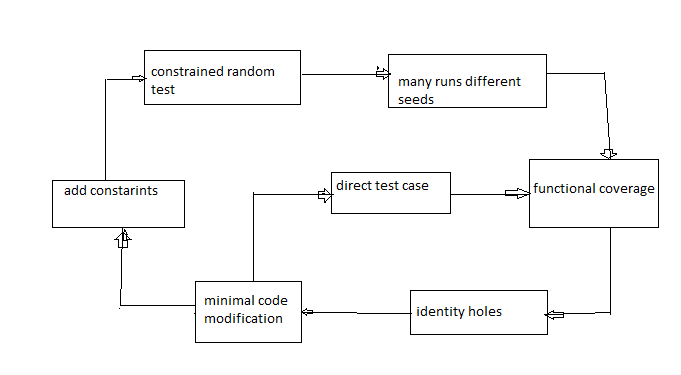
The test layer and functional coverage

You are now at the top of the testbench, the test layer, as Design bugs that occur between DUT blocks are harder to find as they involve multiple people reading and interpreting multiple specifications.

This top-level test is a conductor who does not play any musical instrument, but instead guides the efforts of others. The test contains the constraints to create the stimulus. Functional coverage measures the progress of all tests in fullfilling the requirements in the verification plan. The functional coverage code changes through the project as the various criteria complete. Because it is constantly being modified, it is not part of the environment.

**1b. With neat diagram explain coverage and test bench components.**

While you want the simulator to generate the stimulus, you don’t want totally random values. You use the SystemVerilog language to describe the format of the stimulus and the simulator picks values that meet the constraints. Constraining the random values to become relevant stimuli is covered in Chapter 6. These values are sent into the design, and also into a high-level model that predicts what the result should be. The design’s actual output is compared with the predicted output. First, notice that a random test often covers a wider space than a directed one. This extra coverage may overlap other tests, or may explore new areas that you did not anticipate.



If these new areas find a bug, you are in luck! If the new area is not legal, you need to write more constraints to keep away. Lastly, you may still have to write a few directed tests to find cases not covered by any other constrained-random tests. Start at the upper left with basic constrained-random tests. Run them with many different seeds. When you look at the functional coverage reports, find the holes, where there are gaps. Now you make minimal code changes, perhaps with new constraints, or injecting errors or delays into the DUT. Spend most of your time in this outer loop, only writing directed tests for the few features that are very unlikely to be reached by random tests.

**2a. What do you mean by packed arrays ?For the below code write the packed array layout.**

Bit [3:0][7:0] bytes;

Bytes =32’h café –dada;

$display (bytes ,,

Bytes[3];

Bytes[3][7];;

For some data types, you may want both to access the entire value and also

divide it into smaller elements. For example, you may have a 32-bit register

that sometimes you want to treat as four 8-bit values and at other times as a

single, unsigned value. A SystemVerilog packed array is treated as both an

array and a single value. It is stored as a contiguous set of bits with no unused

space, unlike an unpacked array.

bit [3:0] [7:0] bytes;

bytes = 32’ café –dada;

$displayh(bytes,,

bytes[3],

bytes[3][7]);

bytes[3] bytes[1][6]

bytes 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

**2b. With an example explain different array method.**

PACKED ARRAY

bit [3:0] [7:0] bytes; // 4 bytes packed into 32-bits

bytes = 32’hdead\_beef;

$displayh(bytes,, // Show all 32-bits

bytes[3], // most significant byte "de"

bytes[3][7]);

Using dynamic arrays

intdyn[], d2[]; // Empty dynamic arrays

initial begin

dyn = new[5]; // Allocate 5 elements

foreach (dyn[j])

dyn[j] = j; // Initialize the elements

d2 = dyn; // Copy a dynamic array

d2[0] = 5; // Modify the copy

$display(dyn[0],d2[0]); // See both values (0 & 5)

dyn = new[20](dyn); // Expand and copy

dyn = new[100]; // Allocate 100 new integers

// Old values are lost

dyn.delete; // Delete all elements

end

Unpacked array declarations

bit [7:0] b\_unpacked[3]; // Unpacked

fixed-size arrays

intlo\_hi[0:15]; // 16 ints [0]..[15]

intc\_style[16]; // 16 ints [0]..

**MODULE – 2**

**3.a. Enumerate routine arguments with an example.**

C-style Routine Arguments

SystemVerilog and Verilog-2001 allow you to declare task and function arguments more cleanly and with less repetition. The following Verilog task

a routine definition or call with no arguments does not need the empty parentheses (). This book leaves them out except as needed for clarity.

SystemVerilog for Verification58

requires you to declare some arguments twice, once for the direction, and once for the type.

Example 3-6

task mytask2;

output [31:0] x;

reg [31:0] x;

input y;

...

endtask

With SystemVerilog, you can use the less verbose C-style. Note that you should use the universal input type of logic.

Example 3-7 C-style routine arguments

task mytask1

(output logic [31:0] x, input logic y);

...

endtask

Argument Direction

You can take even more shortcuts with declaring routine arguments. The direction and type default to “input logic” and are sticky, so you don’t have to repeat these for similar arguments. Here is a routine header written using the Verilog-1995 style.

Example 3-8 Verbose Verilog-style routine arguments

task T3;

input a, b;

logic a, b;

output [15:0] u, v;

bit [15:0] u, v;

...

endtask

You could rewrite this as follows.

Example 3-9 Routine arguments with sticky types

task T3(a, b, output bit [15:0] u, v);

The arguments a and b are input logic, 1 bit wide. The arguments u and v are 16-bit output bit types.

Advanced Argument Types

Verilog had a simple way to handle arguments: an input or inout was copied to a local variable at the start of the routine, while an output or inout was copied when the routine exited. No memories could be passed into a Verilog routine, only scalars. In SystemVerilog, you can specify that an argument is passed by reference, rather than copying its value. This argument type, ref, has several benefits over input, output, and inout. First, you can now pass an array into a routine.

Example 3-10 Passing arrays using ref and const function

void print\_sum (const ref inta[]);

int sum = 0;

for (inti=0; i<a.size; i++)

begin

sum += a[i];

$display("The sum of the arrays is ", sum);

endfunction

SystemVerilog allows you to pass array arguments without the ref direction, but the array is copied onto the stack, an expensive operation for all but the smallest arrays. Example 3-10 also shows the const modifier. As a result, the array a is initialized when print\_sum is called, but cannot be modified in the routine. Always use ref when passing arrays to a routine. If you don’t want the routine to change the array values, use the const ref type. With this, the compiler checks that your routine does not modify the array.

The second benefit of ref arguments is that a task can modify a variable and is instantly seen by the calling function.

.5.4 Default Argument Values As your testbench grows in sophistication, you may want to add additional controls to your code but not break existing code. For the function in Example 3-10, you might want to print a sum of just the middle values of the array. However, you don’t want to go back and rewrite every call to add extra arguments. In SystemVerilog you can specify a default value that is used if you leave out an argument in the call.

Example 3-12 Function with default argument values

function void print\_sum (ref inta[],

input int start = 0,

input int end = -1);

int sum = 0, last;

if (last == -1 || last >a.size)

last = a.size;

for (inti=start; i<last; i++) begin

sum += a[i];

$display("The sum of the arrays is ", sum);

endtask

You can call this task in the following ways. Note that the first call is compatible with both versions of the print\_sum routine.

Example 3-13 Using default argument values

print\_sum(a); // Sum a[0:size-1] – default

print\_sum(a, 2, 5); // Sum a[2:5]

print\_sum(a, 1); // Start at 1

print\_sum(a,, 3); // Sum a[0:3]

print\_sum(); // error: a has no default

**3.b. Specify how time values are constructed in system Verilog with an example.**

**Time units and precision**

When you rely on the ‘timescale compiler directive, you must compile the files in the proper order to be sure all the delays use the proper scale and precision. The timeunit and timeprecision declarations eliminate this ambiguity by precisely specifying the values for every module. Example 3-22 shows these declarations. Note that if you use these instead of ‘timescale, you must put them in every module that has a delay.

Time literals

SystemVerilog allows you to unambiguously specify a time value plus units. Your code can use delays such as 0.1ns or 20ps. Just remember to use timeunit and timeprecision or ‘timescale. You can make your code

even more time aware by using the classic Verilog $timeformat and $realtime routines.

Example 3-22 Time literals and $timeformat

module timing;

timeunit 1ns;

timeprecision 1ps;

initial begin

$timeformat(-9, 3, "ns", 8);

#1 $display("@%t", $realtime); // @1.000ns

#2ns $display("@%t", $realtime); // @3.000ns

#0.1ns $display("@%t", $realtime); // @3.100ns

#41ps $display("@%t, $realtime); // @3.141ns

end

endmodule

**4.a. List different tradeoffs of system Verilog.**

An interface cannot contain any instances such as modules or other interfaces.

There are trade-offs in using interfaces with modports as comparedwith traditional port connects with signals.

The advantages to using an interface are as follows.

 An interface is ideal for design reuse. When two blocks communicatewith a specified protocol using two or more signals, use an interface.If signals are repeated over and over, as in a networking switch, use avirtual interface as described in Chapter 10.

 The interface takes the jumble of signals that you declare over andover in every module or program and puts it in a central location,reducing the possibility of misconnecting signals.

 To add a new signal, you just have to declare it once in the interface,not in higher-level modules, once again reducing errors.

 Modports allow a module to easily tap a subset of signals from aninterface. You can specify signal direction for additional checking.The disadvantages of using an interface are as follows.

 For point-to-point connections, interfaces with modports are almost

as verbose as using ports with lists of signals. But all the declarations108 SystemVerilog for Verificationare still in one central location, reducing the chance for making anerror.

 You must now use the interface name in addition to the signal name,possibly making the modules more verbose.

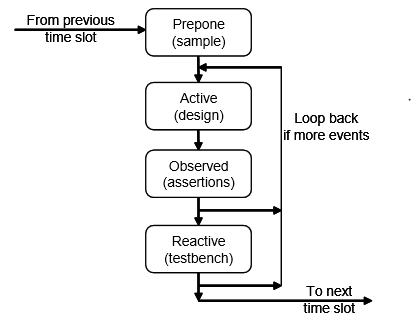
 If you are connecting two design blocks with a unique protocol thatwill not be reused, interfaces may be more work than just wiringtogether the ports.

 It is difficult to connect two different interfaces. A new interface(bus\_if) may contain all the signals of an existing one (arb\_if),plus new signals (address, data, etc.). But since interfaces cannot behierarchical, you have to break out the individual signals and drivethem appropriately.

**4.b. What are all the main time regions inside a system Verilog time step? Explain.**

The root of the problem is the mixing of design and testbench events during the same time slot (though even in pure RTL the same problem can happen). What if there were a way you could separate these events temporally, just as you separated the code? At 100ns, your testbench could sample the design outputs before the clock has had a chance to change and any design activity has occurred. By definition, these values would be the last possible ones from the previous time slot. Then, after all the design events are done, your testbench would start. How does SystemVerilog know to schedule the testbench events separately from the design events? In SystemVerilog, your testbench code is in a program block, which is similar to a module in that it can contain code and variables and be instantiated in other modules. However, a program cannot have any hierarchy such as instances of modules, interfaces, or other programs. This new division of the time slot was introduced in SystemVerilog. In Verilog, most events executed in the Active region. There are other regionfor nonblocking assignments, PLI execution, etc., but they can be ignored for the purposes of this book. See the IEEE 1800-2005 Language Reference Manual for more details.

Figure 5-4 Main regions inside a SystemVerilog time step



In SystemVerilog, several new regions were introduced. First to execute during a time slot is the Prepone region, which samples signals before any design activity. These samples are used by the testbench. Next is the Active region, where design events run. These include your RTL and gate code plus the clock generator. The third region is the Observed region, where assertions are evaluated. Following that is the Reactive region where the testbench executes. Note that time does not strictly flow forwards — events in the Observed and Reactive regions can trigger further design events in the Active region in the current cycle.

Table 5-1. Primary SystemVerilog scheduling regions

Name Activity

Prepone Sampling signals before design activity. For testbench input

Active Simulation of design code in modules

Observed Evaluation of SystemVerilog Assertions

Reactive Execution of testbench code in programs

**Module 3**

**5a. Write system Verilog code to describe simple class with random variable and constraints and test bench. Explain.**

Example 6-1 Simple random class

class Packet;

// The random variables

rand bit [31:0] src, dst, data[8];

randc bit [7:0] kind;

// Limit the values for src

constraint c {src> 10;

src< 15;}

endclass

Packet p;

initial begin

p = new; // Create a packet

assert (p.randomize());

transmit(p);

end

This class has four random variables. The first three use the rand modifier, so that every time you randomize the class, the variables are assigned a value. Think of rolling dice: each roll could be a new value or repeat the current one. The kind variable is randc, which means random cyclic, so that the random solver does not repeat a random value until every possible value has been assigned. Think of dealing cards from a deck: you deal out every card in the deck in random order, then shuffle the deck, and deal out the cards in a different order. Note that the constraint expression is grouped using curly braces: {}. This is because this code is declarative, not procedural, which uses begin...end. The randomize function returns 0 if a problem is found with the constraints. The procedural assertion is used to check the result, as shown in section 5.9. You need find the tool-specific switches to cause the assertion to terminate simulation. This book uses assert to test the result from randomize, but you may want to test the result and then call your special routine that prints any useful information and then gracefully shuts down then simulation. The constraint in Example 6-1 is an expression that limits the values for the src variable. In this case, SystemVerilog chooses between the values of 11, 12, 13, or 14. All variables in your classes should be random and public. This gives your test the maximum control over the DUT’s stimulus and control. You can always turn off a random variable. If you forget to make a variable random, you must edit the environment, which you want to avoid.

**5b. Write a system Verilog code for random strobe patterns**

parameter MAX\_TRANSFER\_LEN = 10;

class StrobePat;

rand bit strobe[MAX\_TRANSFER\_LEN];

constraint c\_set\_four{ strobe.sum == 3’h4; }

endclass

**6a. Write system Verilog code for to use rand case and rand range**.

You can use **randcase**to make a weighted choice between several actions, without having to create a class and instance. Example 6-54 chooses one of the three branches based on the weight. SystemVerilog adds up the weights (1+8+1 = 10), chooses a value in this range, and then picks the appropriate branch. The branches are not order dependent, the weights can be variables, and they do not have to add up to 100%.

**Example 6-54** Random control with randcaseand $urandom\_range

**initial begin**

**intlen;**

**randcase**

**1: len = $urandom\_range(0, 2); // 10%: 0, 1, or 2**

**8: len = $urandom\_range(3, 5); // 80%: 3, 4, or 5**

**1: len = $urandom\_range(6, 7); // 10%: 6 or 7**

**endcase**

**$display("len=%0d", len);**

**end**

The **$urandom\_range**function returns a random number in the specified

range.

**6b. List different random number functions in system Verilog.**

**The pre\_randomizeand post\_randomizeFunctions**

Sometimes you need to perform an action immediately before every **randomize** call or immediately afterwards. For example, you may want to set some nonrandom class variables (such as limits) before randomization starts, or you may need to calculate the error correction bits for random data. SystemVerilog lets you do this with two special void functions, **pre\_randomize**and **post\_randomize**. Section 3.3 showed that a void function does not return a value, but, because it is not a task, does not consume

time. If you want to call a debug routine from p**re\_randomize**or

**post\_randomize**, it must be a function.

**6.9.1 Building a bathtub distribution**

For some applications, you want a nonlinear random distribution. For instance, small and large packets are more likely to find a design bug such as buffer overflow than medium-sized packets. So you want a bathtub shaped distribution; high on both ends, and low in the middle. You could build an elaborate **dist**constraint, but it might require lots of tweaking to get the shape you want. Verilog has several functions for nonlinear distribution such as **$dist\_exponential**but none for a bathtub. However, you can build one by using the exponential function twice.

Across many randomizations, you will see the desired nonlinear distribution. You can use all the Verilog-1995 distribution functions this way, plus several that are new for SystemVerilog. Some of the useful = functions include the following.

􀂄**$dist\_exponential**— Exponential decay, as shown in Figure 6-1

􀂄**$dist\_normal**— Bell-shaped distribution

􀂄**$dist\_poisson**— Bell-shaped distribution

􀂄**$dist\_uniform**— Flat distribution

􀂄**$random** — Flat distribution, returning signed 32-bit random

􀂄**$urandom**— Flat distribution, returning unsigned 32-bit random

􀂄**$urandom\_range**— Flat distribution over a range

**6c. With an example enumerates diffferetdolution probabilities for randomization.**

Whenever you deal with random values, you need to understand the probability of the outcome. SystemVerilog does not guarantee the exact solution found by the random constraint solver, but you can influence the distribution. Any time you work with random numbers, you have to look at thousands or millions of values to average out the noise. Changing the tool version or random seed can cause different results. Some simulators, such as Synopsys VCS, have multiple solvers to allow you to trade memory usage vs. performance.

**6.5.1 Unconstrained**

Start with two variables with no constraints.

**Example 6-17** Class Unconstrained

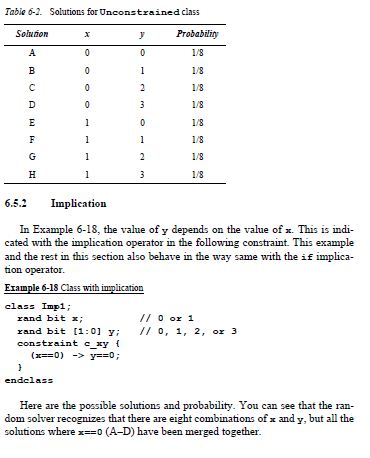
**class Unconstrained;**

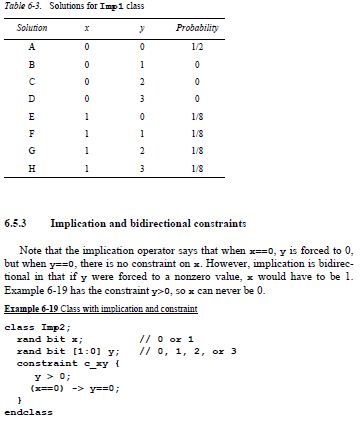
**rand bit x; // 0 or 1**

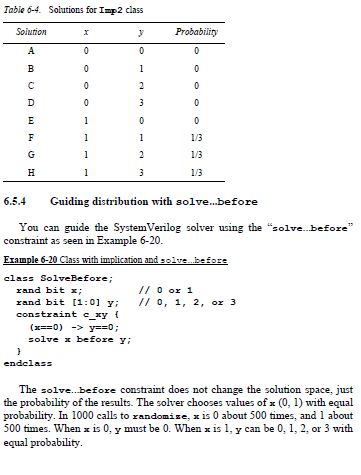
**rand bit [1:0] y; // 0, 1, 2, or 3**

**endclass**

There are eight possible solutions. Because there are no constraints, each has the same probability. You have to run thousands of randomizations to see the actual results approach the listed probabilities.



****

****

**7 a. Write a system Verilog sample code a describe of using fork….join and begin….end.**

Example: Interaction of begin...end and fork...join

initial begin

$display("@%0d: start fork...join example", $time);

#10 $display("@%0d: sequential after #10", $time);

fork

$display("@%0d: parallel start", $time);

#50 $display("@%0d: parallel after #50", $time);

#10 $display("@%0d: parallel after #10", $time);

begin

#30 $display("@%0d: sequential after #30", $time);

#10 $display("@%0d: sequential after #10", $time);

end

join

$display("@%0d: after join", $time);

#80 $display("@%0d: final after #80", $time);

End

Output from begin...end and fork...join

@0: start fork...join example

@10: sequential after #10

@10: parallel start

@20: parallel after #10

@40: sequential after #30

@50: sequential after #10

@60: parallel after #50

@60: after join

@140: final after #80

**7 b. What are dynamic threads? With an example code explain how dynamic threads are created.**

Verilog’s threads are very predictable. You can read the source code and count the initial, always, and fork...join blocks to know how many threads were in a module. System Verilog lets you create threads dynamically, and does not require you to wait for them to finish.

In Example below, the testbench generates random transactions and sends them to a DUT that stores them for some predetermined time, and then returns them. The testbench has to wait for the transaction to complete, but does not want to stop the generator.

Example: Dynamic thread creation

program automatic test(busif.TB bus);

task wait\_for\_tr(Transaction tr);

fork

begin

wait (bus.cb.addr != tr.addr);

$display("@%0d: Addr match %d", $time, tr.addr);

end

join\_none

endtask

Transaction tr;

initial

repeat (10)

begin

tr = new;

if (!tr.randomize)

$finish;

transmit(tr);

wait\_for\_tr(tr);

end

endprogram

When the wait\_for\_tr task is called, it spawns off a thread to watch the bus for the matching transaction address. During a normal simulation, many of these threads run concurrently.

**8a. Enumerate the techniques used to disable threads with an example.**

(i)Disabling a single thread

Just as you may need to create threads in the testbench, you may also need to stop them. The Verilog disable statement works on SystemVerilog threads.

Example: Disabling a thread parameter TIME\_OUT = 1000;

task wait\_for\_tr(Transaction tr);

fork

begin

fork :timeout\_block

wait (bus.cb.addr != tr.addr);

#TIME\_OUT $display("@%0d: Error: timeout", $time);

join\_any

disable timeout\_block;

$display("@%0d: Addr match %d", $time, tr.addr);

end

join\_none

endtask

This version has two threads inside a fork...join\_any such that the simple wait is done in parallel with a delayed display. If the correct bus address comes back quickly enough, the wait construct completes, the join\_any executes, and then the disable kills off the remaining thread. However, if the bus address does not get the right value before the TIME\_OUT delay completes, the error message is printed, the join\_any executes, and the disable kills the thread with the wait.

(ii) Disabling multiple threads

System Verilog introduces the disable fork statement so you can stop all child threads that have been spawned from the current thread. Example below has an additional begin...end block inside the fork...join to make the statements sequential. The following sections show how you can asynchronously disable multiple threads.

Example: Limiting the scope of a disable fork

initial begin

wait\_for\_tr(tr0);

fork

begin

wait\_for\_tr(tr1);

fork

wait\_for\_tr(tr2);

join

#(TIME\_OUT/2) disable fork;

end

join

end

The code calls wait\_for\_tr that starts thread 0. Next a fork...join creates thread 1. Inside this thread, one is spawned by the wait\_for\_tr task and one by the innermost fork...join, which spawns thread 4 by calling the task. After a delay, a disable fork stops the child threads. Only threads 2, 3, and 4 are below thread 1, so they are the only ones stopped. Thread 0 is outside the fork...join block that has the disable, so it is unaffected.

Example below is the more robust version, with disable with a label that explicitly names the threads that you want to stop.

Example: Using disable label to stop threads

initial begin

wait\_for\_tr(tr0);

begin : threads\_1\_2

wait\_for\_tr(tr1);

wait\_for\_tr(tr2);

end

#(TIME\_OUT/2) disable threads\_1\_2;

join

end

8 b. Explain mailboxes. How to create a mailboxe in system Verilog explain with an example.

A mailbox is an object and thus has to be instantiated by calling the new function. This takes an optional size argument to limit the number of entries in the mailbox. If the size is 0 or not specified, the mailbox is unbounded and can hold an unlimited number of entries. You put data into a mailbox with the put task, and remove it with the get task. A put can block if the mailbox is full and a get blocks if it is empty. The peek task gets a copy of the data in the mailbox but does not remove it. The data can be a single value, such as an integer, or logic of any size.

Example: Exchanging objects using a mailbox: the Generator class

program mailbox\_example(bus\_if.TB bus, ...);

class Generator;

Transaction tr;

mailbox mbx;

function new(mailbox mbx);

this.mbx = mbx;

endfunction

task run;

repeat (10) begin

tr = new;

assert(tr.randomize);

mbx.put(tr);

end

endtask

endclass

class Driver;

Transaction tr;

mailbox mbx;

function new(mailbox mbx);

this.mbx = mbx;

endfunction

task run;

repeat (10) begin

mbx.get(tr);

@(posedgebusif.cb.ack);

bus.cb.kind<= tr.kind;

...

end

endtask

endclass

mailbox mbx;

Generator gen;

Driver drv;

initial begin

mbx = new;

gen = new(mbx);

drv = new(mbx);

fork

gen.run();

drv.run();

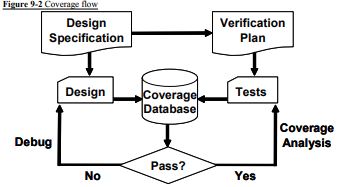
join

end

endprogram

**Module 5**

**9 a. Explain Coverage flow of functional coverage with an diagram.**

Gathering coverage data You can run the same random testbench over and over, simply by changing the random seed, to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage. Figure 9-2 Coverage flow You then analyze the coverage data to decide how to modify your tests. If the coverage levels are steadily growing, you may just need to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, you can add additional constraints to generate more “interesting” stimuli. When you reach a plateau, some parts of the design are not being exercised, so you need to create more tests. Lastly, when your functional coverage values near 100%, check the bug rate. If bugs are still being found, you may not be measuring true coverage for some areas of your design

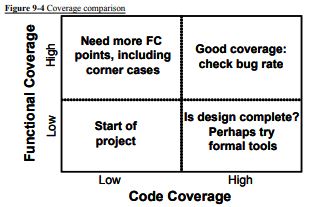
**9 b. Enumerate the action required for storing coverage data**

Each simulation vendor has its own format for storing coverage data and as well as its own analysis tools. You need to perform the following actions with those tools.

– Run a test with multiple seeds. For a given set of constraints (and coverage groups), compile the testbench and design into a single executeable. Now you need to run this constraint set over and over with different random seeds. You can use the Unix system clock as a seed, but be careful, as your batch system may start multiple jobs simultaneously. These jobs may run on different servers or may start on a single server with multiple processors. – Check for pass/fail. Functional coverage information is only valid for a successful simulation. When a simulation fails because there is a design bug, the coverage information must be discarded. The coverage data measures how many items in the verification plan are complete, and this plan is based on the design specification. If the design does not match the specification, the coverage data is useless. Some verification teams periodically measure all functional coverage from scratch so that it reflects the current state of the design. – Analyze coverage across multiple runs. You need to measure how successful each constraint set is, over time. If you are not yet getting 100% coverage for the areas that are targeted by the constraints, but the amount is still growing, run more seeds. If the coverage level has plateaued, with no recent progress, it is time to modify the constraints. Only if you think that reaching the last few test cases for one particular section may take too long for constrained-random simulation should you consider writing a directed test. Even then, continue to use random stimulus for the other sections of the design, in case this “background noise” finds a bug.

**9 c. List and explain different functional coverage strategies.**

Before you write the first line of test code, you need to anticipate what are the key design features, corner cases, and possible failure modes. This is how you write your verification plan. Don’t think in terms of data values only; instead, think about what information is encoded in the design. The plan should spell out the significant design states. 9.3.1 Gather information, not data A classic example is a FIFO. How can you be sure you have thoroughly tested a 1K FIFO memory? You could measure the values in the read and write indices, but there are over a million possible combinations. Even if you were able to simulate that many cycles, you would not want to read the coverage report. At a more abstract level, a FIFO can hold from 0 to N-1 possible values. So what if you just compare the read and write indices to measure how full orempty the FIFO is? You would still have 1K coverage values. If your testbench pushed 100 entries into the FIFO, then pushed in 100 more, do you really need to know if the FIFO ever had 150 values? Not as long as you can successfully read out all values. The corner cases for a FIFO are Full and Empty. If you can make the FIFO go from Empty (the state after reset) through Full and back down to Empty, you have covered all the levels in between. Other interesting states involve the indices as they pass between all 1’s and all 0’s. A coverage report for these cases is easy to understand. You may have noticed that the interesting states are independent of the FIFO size. Once again, look at the information, not the data values. Design signals with a large range (more than a few dozen possible values) should be broken down into smaller ranges, plus corner cases. For example, your DUT may have a 32-bit address bus, but you certainly don’t need to collect 4 billion samples. Check for natural divisions such as memory and IO space. For a counter, pick a few interesting values, and always try to rollover counter values from all 1’s back to 0. 9.3.2 Only measure what you are going to use Gathering functional coverage data can be expensive, so only measure what you will analyze and use to improve your tests. Your simulations may run slower as the simulator monitors signals for functional coverage, but this approach has lower overhead than gathering waveform traces and measuring code coverage. Once a simulation completes, the database is saved to disk. With multiple testcases and multiple seeds, you can fill disk drives with functional coverage information. But if you never look at the final coverage reports, don’t perform the initial measurements. There are several ways to control cover data: at compilation, instantiation, or triggering. You could use switches provided by the simulation vendor, conditional compilation, or suppression of the gathering of coverage data. The last of these is less desirable because the post-processing report is filled with sections with 0% coverage, making it harder to find the few enabled ones. 9.3.3 Measuring completeness Like your kids in the backseat on a family vacation, your manager constantly asks you, “Are we there yet?” How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination. At the start of a project, both code and functional coverage are low. As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage. Create additional constraints and tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing.



What if the functional coverage is high but the code coverage is low? Your tests are not exercising the full design, and you need to revise your verification plan and add more functional coverage points to locate untested functionality. A more difficult situation is high code coverage but low functional coverage. Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If the functionality is there, but your tests can’t reach it, you might need a formal verification tool that can extract the design’s states and create appropriate stimulus. The goal is both high code and functional coverage. But don’t plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up? Worse yet, are they being found deliberately, or did your testbench happen to stumble across a particular combination of states that no one had anticipated? On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches. Try different approaches such as new combinations of design blocks and error generators.

10 a Write a system Verilog code to test using functional coverage callback.

program automatic test;

Environment env;

initial begin

Driver\_cbs\_coverage dcc;

env = new;

env.gen\_cfg;

env.build;

// Create and register the coverage callback

dcc = new;

env.drv.cbs.push\_back(dcc); // Put into driver’s Q

env.run;

env.wrapup.

end

endprogram

10 b. Write system Verilog code for basic cross coverage and also mention coverage summary report for basic cross coverage.

Example 9-28 Basic cross coverage

class Transaction;

rand bit [3:0] kind;

rand bit [2:0] port;

endclass

Transaction tr;

covergroupCovPort;

kind: coverpointtr.kind; // Create cover point kind

port: coverpointtr.port // Create cover point port

cross kind, port; // Cross kind and port

endgroup

