The testbench needs to drive the design and sample signals from the design, primarily through interfaces with clocking blocks.

Asynchronous signals such as rst pass through the interface with no delays.

When use modports with clocking blocks, a synchronous interface signal must be prefixed with both the interface name and the clocking block name. For example, arbif.cb.grant is legal, but arbif.grant is not legal.

We should always drive interface signals in a clocking block with a synchronous drive using the <= operator.

In System Verilog, synchronous bidirectional signals in interfaces are easier to use as the continuous assignment is added for you. When you write to the net from a program, System Verilog actually writes to a temporary variable that drives the net. Your program reads directly from the wire, seeing the value that is resolved from all the drivers. Design code in a module still uses the classic register plus continuous assignment statement.

Avoid race conditions by always putting the clock generator in a module.

You use $root to make unambiguous cross module references by specifying the absolute path.

2013-11-20

System Verilog的随机化机制

a class with random variables

constraints created by class builder

random-solver to solve the random values

Sample 6.1 Simple random class

**class Packet;**

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

**randc bit [7:0] kind;**

**constraint c {src > 10; src < 15}**

**endclass**

**Packet p;**

**initial begin**

**p = new();**

**assert (p.randomize())**

**else $fatal(0, “Packet::randomize failed”);**

**transmit(p);**

**end**

rand: think of rolling dice, each roll could be a new value or repeat the current one.

randc: which means random cyclic, 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.

The constraint expression is grouped using curly braces: {}, not begin … end. Because this code is declarative, not procedural, which uses begin … end.

The randomize() function returns 0 if a problem is found with the constraints.

You should not randomize an object in the class constructor. Your test may need to turn constraints on or off, change weights, or even add new constraints before randomization. The constructor is for initializing the object’s variables, and if you called randomize() at this early stage, you might end up throwing away the results.

All variables in your class should be random and public. In some cases, you may want to make some variable not be random, you can turn off the random variable.

randomize() function assigns random value to any variable labeled as rand or randc obeying the constraints. However, if there are conflicting constraints, the randomize() function will fail and return a value of 0. Therefore, once we call randomize(), we should check that the function is successfully returned with 1.

The constraint solver is a System Verilog simulator function. It solves the constraint expressions and chooses values that satisfy the constraints.

Sample 6.3 Constrained-random class

**class Stim;**

**const bit [31:0] CONGEST\_ADDR = 42;**

**typedef enum {READ, WRITE, CONTROL} stim\_e;**

**randc stim\_e kind;**

**rand bit [31:0] len, src, dst;**

**bit congestion\_test;**

**constraint c\_stim {**

**len < 1000;**

**len > 0;**

**if (congestion\_test) {**

**dst inside {[CONGEST\_ADDR-100:CONGEST\_ADDR+100]};**

**src == CONGEST\_ADDR;**

**}**

**else**

**src inside {0, [2:10], [100:107]};**

**}**

**endclass**

There can be only one relation operator which is one of “<=, <, ==, >, >=” in one constraint expression.

For example, the following constraint expression is wrong!

**0 < len < 1000; Wrong!**

inside operator specifies a set of number which is solved with equal possibility by the System Verilog constraint solver.

Sample 6.10 “inside” operator examples

**rand bit [6:0] b;**

**rand bit [5:0] e;**

**constraint c\_range{**

**b inside {[$:4], [20:$]};**

**e inside {[$:60]};**

**}**

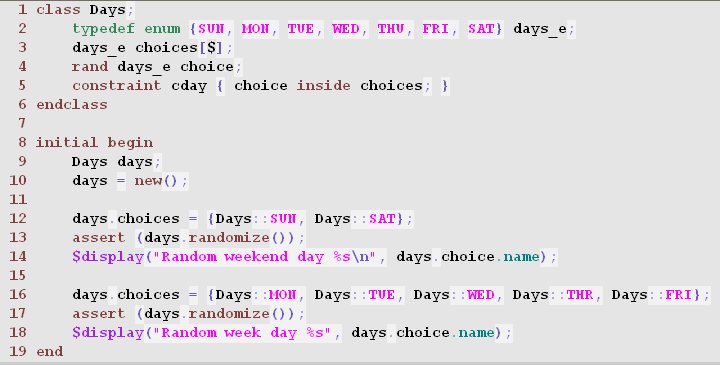
if you want to constrain the random variable not inside a range, you can use “!” plus “inside” operator, such as:

**constraint c\_range {**

**!(c inside {[lo:hi]});**

**}**

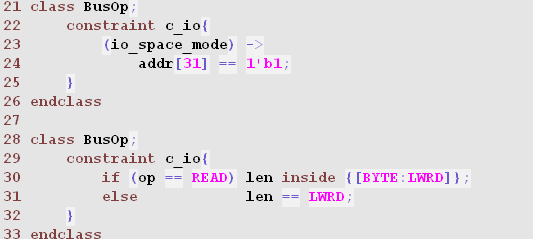
2013-11-21





The name() function returns a string with the name of an enumerated value.

Conditional constraints



You can turn on a constraint or turn off a constraint according to your need. Use the following syntax:

handle\_name.constraint\_name.constraint\_mode([input\_argument]);

input\_argument = 1, means to turn on the specified constraint;

input\_argument = 0, means to turn off the specified constraint;

You can use the following syntax to turn on or turn off all the constraints in an object:

handle\_name.constraint\_mode([input\_argument]);

function void pre\_randomize();

function void post\_randomize();

You can turn off the random property of a variable to make it not random any more.

The syntax is:

handle\_name.constraint\_name.random\_variable\_to\_be\_turned\_off.rand\_mode(input\_argument);

input\_argument = 0, means to turn off the specified random variable

input\_argument = 1, means to turn on the specified random variable

**the constraints to arrays**

constrain the array size

**class dyn\_size;**

**rand logic [31:0] d[];**

**constraint d\_size { d.size() inside {[1:10]}; }**

**endclass**

constrain the sum of elements

2013-11-22

# Disabling Multiple Threads

We can use disable statement to stop threads in a named block. System Verilog introduces **disable fork** statement so that we can *stop all child threads spawned by the current thread*.

Watch out, as you might unintentionally stop too many threads, such as those created from task calls. You should always surround the target code with a fork … join to limit the scope of a fork statement. The following example shows how to limit the scope of a disable fork.

Sample 7.17 Limiting the scope of a disable fork

initial begin

check\_trans(tr0); // Thread 0

// Create a thread to limit scope of disable

fork // Thread 1

begin

check\_trans(tr1); // Thread 2

fork // Thread 3

check\_trans(tr2); // Thread 4

join

// Stop threads 1-4, but leave 0 alone

#(TIME\_OUT/2) disable fork;

end

join

end

Note: a task call spawns a thread. The statement fork … join/join\_none/join\_any spawns a thread.

# Disable a task that was called multiple times

If you *disable a task from inside the task*, it is *like a return statement*, but *it also kills all threads started by the task*. *If the task has been called from multiple threads, disabling one will disable them all*.

In the following example, the wait\_for\_time\_out task is called three times, spawning three threads.

Sample 7.19 Using disable label to stop a task

task wait\_for\_time\_out(int id);

if (id == 0)

fork

begin

#2;

$display(“@%0t: disable wait\_for\_time\_out”, $time);

disable wait\_for\_time\_out;

end

join\_none

fork : just\_a\_little

begin

$display(“@%0t: %m: %0d entering thread”, $time, id);

#TIME\_OUT;

$display(“@%0t: %m: %0d done”, $time, id);

end

join\_none

endtask

initial begin

wait\_for\_time\_out(0); // Spawn thread 0

wait\_for\_time\_out(1); // Spawn thread 1

wait\_for\_time\_out(2); // Spawn thread 2

#(TIME\_OUT\*2) $display(“@%0t: All done”, $time);

end

# Interprocess Communication

All of this data exchange and control synchronization is called IPC, which is done in System Verilog with events, semaphores, and mailboxes.

## Events

In Verilog a thread waits for an event with the “ @ ” operator. This operator is edge-sensitive, and so it always blocks, waiting for the event to change. Another thread trigger the event with the “ -> ” operator, unblocking the first thread.

System Verilog enhances the Verilog event in several ways. An event is now a handle to a synchronization object that can be passed around to routines. This feature allows you to share events across objects without having to make the events global. The most common way is to pass the event into the constructor for an object.

## Mailbox

### Mailbox in a testbench

Sample 7.34-7.36 show a Generator and Driver exchanging transactions using a mailbox, and the top-level program. Note that the two classes need to be defined inside the program block so that they will see the definition of the bus interface.

Sample 7.34 Exchanging objects using a mailbox: the Generator class

**class Generator;**

**Transaction tr;**

**mailbox mbx;**

**function new(mailbox mbx);**

**this.mbx = mbx;**

**endfunction**

**task run(int count);**

**repeat (count) begin**

**tr = new();**

**assert(tr.randomize);**

**mbx.put(tr); // send out transaction**

**end**

**endtask**

**endclass**

Sample 7.35 Exchanging objects using a mailbox: the Driver class

**class Driver;**

**Transaction tr;**

**mailbox mbx;**

**function new(mailbox mbx);**

**this.mbx = mbx;**

**endfunction**

**task run(int count);**

**repeat (count) begin**

**mbx.get(tr); // fetch next transaction**

**@(posedge bus.cb.ack);**

**bus.cb.kind <= tr.kind;**

**...**

**end**

**endtask**

**endclass**

Sample 7.36 Exchanging objects using a mailbox: the program block

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

**`include "transaction.sv"**

**`include "generator.sv"**

**`include "driver.sv"**

**mailbox mbx;**

**Generator gen;**

**Driver drv;**

**int count;**

**initial begin**

**count = $urandom\_range(50);**

**mbx = new(); // construct the mailbox**

**gen = new(mbx); // construct the generator**

**drv = new(mbx); // construct the driver**

**fork**

**gen.run(count);**

**drv.run(count);**

**join // wait for both to finish**

**end**

**endprogram**

### Bounded Mailboxes

Sometimes we may want two threads work in lock steps so that the producer blocks until the consumer is done with the object. We can specify a maximum size for the mailbox while constructing it. The default size is 0, which means the size of the mailbox we create is infinite big, or unbounded. If we specify a number greater than 0, we construct a bounded mailbox. If the producer puts more transaction than the size, the producer blocks until the mailbox has a vacancy. The consumer gets the transaction and process the transaction, creating a vacancy.

### Synchronized testbenches

Why we want two threads operate in lock steps? Firstly, explain what the lock step is. If the producer blocks until the consumer gets the transaction and is done with the transaction, we call the two threads work in lock steps. If all the transactors of the hierarchical testbench communicates in this lock step way, the testbench is called synchronized. Secondly, we explain why we need the testbench to be synchronized. If the testbench is synchronized, the stimulus generation chain runs in lock step. The highest level (or the highest layer of a layered testbench) generator only completes when the last low level (or the layer lower than it but next to it) completes transmission. In this way, your testbench can tell precisely when all stimulus has been sent. If the threads operate unsynchronized, we need to add extra code to detect when the last transaction is applied to the DUT.

If we want two threads to run in lock step, we need a handshake in addition to the mailbox. We want the producer to be blocked after the consumer is done with the transaction. We can let the consumer not remove the transaction in the mailbox until the consumer is done with the transaction. Sample 7.41 shows the first attempt to synchronize two threads with a bounded mailbox. The Consumer uses the built-in mailbox method peek() to look at the data in the mailbox without removing. When the Consumer is done processing the data, it removes the data in the mailbox using get(). Then the Producer would detect the vacancy in the mailbox and it will put another transaction in the mailbox. If the Consumer loop uses the method get() at the beginning, the transaction would be removed immediately from the mailbox, so the Producer would detect a vacancy in the mailbox and put another transaction into the mailbox while the Consumer is still processing the last transaction.

**program automatic synch\_peek;**

**mailbox mbx;**

**class Producer;**

**task run();**

**for (int i=1; i<4; i++) begin**

**$display("Producer: before put(%0d)", i);**

**mbx.put(i);**

**end**

**endtask**

**endclass**

**class Consumer;**

**task run();**

**int i;**

**repeat (3) begin**

**mbx.peek(i); // peek integer from mbx**

**$display("Consumer: after get(%0d)", i);**

**mbx.get(i); // remove from mbx**

**end**

**endtask**

**endclass**

**Producer p;**

**Consumer c;**

**initial begin**

**// construct mailbox, producer, consumer**

**mbx = new(1); // bounded mailbox, limit: 1!**

**p = new();**

**c = new();**

**// run the producer and consumer in parallel**

**fork**

**p.run();**

**c.run();**

**join**

**end**

**endprogram**

From the example above, we can see that the Producer is still one transaction ahead of the Consumer. Why is the Producer still one transaction ahead of the consumer? This is because that a bounded mailbox with size=1 only blocks when you try to do a put of the second transaction. In the other words, the Producer is always blocked on putting the next transaction into the mailbox. However, what we want is the Producer would not try to put the transaction into the mailbox until the Consumer is blocked, which means the Consumer is idle. How to implement it? We can use handshake to handle it.

The consumer already blocks, waiting for the Producer using a mailbox. The producer needs to block, waiting for the Consumer to finish the transaction. This is done by adding a blocking statement to the Producer such as an event, a semaphore, or a second mailbox. Sample 7.43 uses an event to block the Producer after it puts data in the mailbox. The Consumer triggers the event after it consumes the data.

Sample 7.43 Producer-Consumer synchronized with an event

**program automatic mbx\_evt;**

**event handshake;**

**class Producer;**

**task run;**

**for (int i=1; i<4; i++) begin**

**$display("Producer: before put(%0d)", i);**

**mbx.put(i);**

**@handshake;**

**$display("Producer: after put(%0d)", i);**

**end**

**endtask**

**endclass**

**class Consumer;**

**task run;**

**int i;**

**repeat (3) begin**

**mbx.get(i);**

**$display("Consumer: after get(%0d)", i);**

**->handshake;**

**end**

**endtask**

**endclass**

**Producer p;**

**Consumer c;**

**initial begin**

**p = new();**

**c = new();**

**fork**

**p.run();**

**c.run();**

**join**

**end**

**endprogram**

From Sample 7.43, we can find that the Producer blocks at an event which is triggered by Consumer after the Consumer is done with the transaction. Unlike the example that indicates the one transaction ahead of the Consumer, in this way, the Producer is not blocked at trying to put the next transaction into the mailbox but blocked by waiting the Consumer to waiting the present transaction.

Another way to synchronize the Producer and Consumer threads is using a second mailbox, which sends a completion message from the Consumer back to the Producer, as shown in Sample 7.46.

Sample 7.46 Producer-Consumer synchronized with a mailbox

**program automatic mbx\_mbx2;**

**mailbox mbx, rtn;**

**class Producer;**

**task run();**

**int k;**

**for (int i=1; i<4; i++l) begin**

**$display("Producer: before put(%0d)", i);**

**mbx.put(i);**

**rtn.get(k);**

**$display("Producer: after get(%0d)", k);**

**end**

**endtask**

**endclass**

**class Consumer;**

**task run();**

**int i;**

**repeat (3) begin**

**$display("Consumer: before get");**

**mbx.get(i);**

**$display("Consumer: after get(%0d)", i);**

**rtn.put(-i);**

**end**

**endtask**

**endclass**

**Producer p;**

**Consumer c;**

**initial begin**

**p = new();**

**c = new();**

**fork**

**p.run();**

**c.run();**

**join**

**end**

**endprogram**

## Building a testbench with threads and IPC



We know the layered testbench is good for the verification of a design. The transactors such as Generator, Agent, Driver and so on are all classes. The upper layer transactor transmits transaction to the transactor in lower layer and next to it.

### Transactor

A transactor receives transaction from its supervisor, processes it and transmits a transaction to the transactor in a lower layer and next to it. Sample 7.48 is a basic transactor.

Sample 7.48 Basic transactor

**class Agent;**

**mailbox gen2agt, agt2drv;**

**Transaction tr;**

**function new(mailbox gen2agt, agt2drv);**

**this.gen2agt = gen2agt;**

**this.agt2drv = agt2drv;**

**endfunction**

**task run();**

**forever begin**

**gen2agt.get(tr); // Get transaction from upstream block**

**... // Do some processing**

**agt2drv.put(tr); // Send it to downstream block**

**end**

**endtask**

**task wrap\_up; // Empty for now**

**endtask**

**endclass**

### Configuration class

The configuration class allows you to randomize the configuration of your system for every simulation. Sample 7.49 is a simple example and has just one variable and a basic constraint.

Sample 7.49 Configuration class

**class Config;**

**bit [31:0] run\_for\_n\_trans;**

**constraint reasonable{**

**run\_for\_n\_trans inside {[1:1000]};**

**}**

**endclass**

### Environment class

The Environment class holds the Generator, Agent, Driver, Monitor, Checker, Scoreboard, and Config objects, and the mailboxes between them. Sample 7.50 shows a basic Environment class.

Sample 7.50 Environment class

**class Environment;**

**Generator gen;**

**Agent agt;**

**Driver drv;**

**Monitor mon;**

**Checker chk;**

**Scoreboard scb;**

**Config cfg;**

**mailbox gen2agt;**

**mailbox agt2drv;**

**mailbox mon2chk;**

**extern function new();**

**extern function void gen\_cfg();**

**extern function void build();**

**extern task run();**

**extern task wrap\_up();**

**endclass**

**function void Environment::build();**

**// Initialize mailboxes**

**gen2agt = new();**

**agt2drv = new();**

**mon2chk = new();**

**// Initialize transactors**

**gen = new(gen2agt);**

**agt = new(gen2agt, agt2drv);**

**drv = new(agt2drv);**

**mon = new(mon2chk);**

**chk = new(mon2chk);**

**scb = new();**

**endfunction**

**task Environment::run();**

**fork**

**gen.run(cfg.run\_for\_n\_trans);**

**agt.run();**

**drv.run();**

**mon.run();**

**chk.run();**

**scb.run(cfg.run\_for\_n\_trans);**

**join**

**endtask**

**task Environment::wrap\_up();**

**fork**

**gen.wrap\_up();**

**agt.wrap\_up();**

**drv.wrap\_up();**

**mon.wrap\_up();**

**chk.wrap\_up();**

**scb.wrap\_up();**

**join**

**endtask**

### Test program

Sample 7.51 basic test program

**program automatic test;**

**Environment env;**

**iniital begin**

**env = new();**

**env.gen\_cfg();**

**env.build();**

**env.run();**

**env.wrap\_up();**

**end**

**endprogram**

# Advanced OOP techniques

## inheritance

Inheritance allows a new class to be derived from an existing one in order to share its variables and routines. The original class is known as the **base** or **super class**. The new class extended from the base class is called the **extended class**.

For example, assume the base transaction class has variables for the source and destination addresses, eight data words, and a CRC for error checking, plus routines for displaying the contents and calculating the CRC. The calc\_crc function is tagged as **virtual** so that it can be redefined, if needed.

Sample 8.1 Base transaction class

**class Transaction;**

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

**bit [31:0] crc;**

**virtual function void calc\_crc;**

**crc = src ^ dst ^ data.xor;**

**endfunction**

**function void display(input string prefix="");**

**$display("%sTr: src=%h, dst=%h, crc=%h", prefix, src, dst, crc);**

**endfunction**

**endclass**

### Extending the base class

In Sample 8.1, we defined a base transaction class. The transaction class’s CRC is good. If we want to inject CRC error to the design under test, we can extends implement it by extending the method calc\_crc. Sample 8.2 is an example of how to extend a class.

Sample 8.2 Extended transaction class

**class BadTr extends Transaction;**

**rand bit bad\_crc;**

**virtual function void calc\_crc;**

**super.calc\_crc(); // compute good CRC**

**if (bad\_crc) crc = ~crc; // corrupt the CRC to inject error**

**endfunction**

**virtual function void display(input string prefix="");**

**$write("%sBadTr: bad\_crc=%b, ", prefix, bad\_crc);**

**endfunction**

**endclass : BadTr**

Note that in Sample 8.2, the variable crc is used without a hierarchical identifier. The BadTr class can see all the variables from the original Transaction plus its own variables such as bad\_crc. The calc\_crc function in the extended class call calc\_crc in the base class using the **super** prefix.



Always declare routines inside a class as virtual so that they can be redefined in an extended class. This applies to all tasks and functions except the new() function. System Verilog always calls the new function based on the handle’s type.

### More OOP Terminology

**Property**: a variable in a class;

**Method**: a task or function in a class;

**Parent class** or **Super class**: the original class which would be extended;

**Derived class** or **subclass**: the extended class;

**Prototype for a routine**: the first line that shows the argument list and return type;

### Constructors in extended class

When you start extending classes, there is one rule about constructors (new() function) to keep in mind. If your base class constructor has any argument, the constructor in the extended class must have a constructor and must call the base’s constructor on its first line, as shown in Sample 8.3.

Sample 8.3 Constructor with argument in an extended class

**class Base1;**

**int var;**

**function new(input int var);**

**this.var = var;**

**endfunction**

**endclass**

**class Extended extends Base1;**

**function new(input int var);**

**super.new(var); // must be first line of new()**

**// other constructor actions**

**endfunction**

**endclass**

### Driver class

OOP rules say that if you have a handle of the base type (Transaction), it can also point to an object of an extended type (BadTr), which means you can assign a handle pointing to a subclass to the handle of a base class type. This is because the handle tr can only reference the variables src, dst, crc, and data, and the routine calc\_crc. So you can send BadTr objects into the Driver without changing it.

**Transaction tr;**

**BadTr bad\_tr;**

**initial begin**

**bad\_tr = new();**

**tr = bad\_tr;**

**tr.calc\_crc(); // Which routine to be called?**

**// bad\_tr.calc\_crc is called.**

**end**

When the diver calls tr.calc\_crc(), which one will be called, the one in Transaction or BadTr? Since calc\_crc() was declared as a virtual method, System Verilog chooses the proper method based on the type of objected stored in tr. If the object is of type Transaction, System Verilog calls the task Transaction::calc\_crc. If the object stored in tr is of type BadTr, System Verilog calls BadTr::calc\_crc.

Sample 8.4 an example of a driver class

**class Driver;**

**mailbox gen2drv;**

**function new(input mailbox gen2drv);**

**this.gen2drv = gen2drv;**

**endfunction**

**task main;**

**Transaction tr;**

**forever begin**

**gen2drv.get(tr); // Get transactoin from generator**

**tr.calc\_crc(); // Process the transaction**

**@ifc.cb.src = tr.src;// Send transaction**

**...**

**end**

**endtask**

**endclass**

### Simple Generator class

Sample 8.5 Generator class

**class Generator;**

**mailbox gen2drv;**

**Transaction tr;**

**function new(input mailbox gen2drv);**

**this.gen2drv = gen2drv;**

**endfunction**

**task run();**

**forever begin**

**tr = new(); // Construct transaction**

**assert(tr.randomize()); // Randomize it**

**gen2drv.put(tr); // Send to Driver**

**end**

**endtask**

**endclass**