SystemVerilog Tutorials

The following tutorials will help you to understand some of the new most important features in SystemVerilog. They also provide a number of code samples and examples, so that you can get a better "feel" for the language.

These tutorials assume that you already know some Verilog. If not, you might like to look at the KnowHow Verilog Designer's Guide first.

SystemVerilog Data Types

This tutorial describes the new data types that Systemverilog introduces. Most of these are synthesisable, and should make RTL descriptions easier to write and understand.

Integer and Real Types

SystemVerilog introduces several new data types. Many of these will be familiar to C programmers. The idea is that algorithms modelled in C can more easiliy be converted to SystemVerilog if the two languages have the same data types.

Verilog's variable types are *four-state*: each bit is 0,1,X or Z. SystemVerilog introduces new *two-state* data types, where each bit is 0 or 1 only. You would use these when you do not need X and Z values, for example in test benches and as for-loop variables. Using two-state variables in RTL models may enable simulators to be more efficient. Used appropriately, they should not affect the synthesis results.

TYPE	Description	Example
bit	user-defined size	bit [3:0] a_nibble;
byte	8 bits, signed	byte a, b;
shortint	16 bits, signed	shortint c, d;
int	32 bits, signed	int i,j;
longint	64 bits, signed	longint lword;

Two-state integer types

Note that, unlike in C, SystemVerilog specifies the number of bits for the fixed-width types.

TYPE	Description	Example
reg	user-defined size	reg [7:0] a_byte;
logic	identical to reg in every way	logic [7:0] a_byte;
integer	32 bits, signed	integer i, j, k;

Four-state integer types

logic is a better name than *reg*, so is preferred. As we shall see, you can use logic where in the past you have may have used reg or where you may have used wire.

TYPE	Description	Example
time	64-bit unsigned	time now;
shortreal	like float in C	shortreal f;
real	like double in C	double g;
realtime	identical to real	realtime now;

Non-integer types

Arrays

In Verilog-1995, you could define scalar and vector nets and variables. You could also define *memory arrays*, which are one-dimensional arrays of a variable type. Verilog-2001 allowed multi-dimensioned arrays of both nets and variables, and removed some of the restrictions on memory array usage.

SystemVerilog takes this a stage further and refines the concept of arrays and permits more operations on arrays.

In SystemVerilog, arrays may have either *packed* or *unpacked* dimensions, or both. Consider this example:

```
reg [3:0][7:0] register [0:9];
```

The packed dimensions are [3:0] and [7:0]. The unpacked dimension is [0:9]. (You can have as many packed and unpacked dimensions as you like.)

Packed dimensions:

- are guaranteed to be laid out contiguously in memory
- can be copied on to any other packed object
- can be sliced ("part-selects")
- are restricted to the "bit" types (bit, logic, int etc.), some of which (e.g. int) have a fixed size.

By contrast, unpacked dimensions can be arranged in memory in any way that the simulator chooses. You can reliably copy an array on to another array of the same type. For arrays with different types, you must use a cast, and there are rules for how an unpacked type is cast to a packed type. Unpacked arrays can be any type, such as arrays of reals.

SystemVerilog permits a number of operations on complete unpacked arrays and slices of unpacked arrays. For these, the arrays or slices involved must have the same type and the same shape – i.e. exactly the same number and lengths of unpacked dimensions. The packed dimensions may be different, as long as the array or slice elements have the same number of bits.

The permitted operations are:

- Reading and writing the whole array
- Reading and writing array slices
- Reading and writing array elements
- Equality relations on arrays, slices and elements

SystemVerilog also includes dynamic arrays (the number of elements may change during simulation) and associative arrays (which have a non-contiguous range).

To support all these array types, SystemVerilog includes a number of array querying functions and methods. For example, you could use $dimensions to find the number dimensions of an array variable.

Typedef

SystemVerilog's data type system allows you to define quite complex types. To make this kind of code clear, the *typedef* facility was introduced. Typedef allows users to create their own names for type definitions that they will use frequently in their code. Typedefs can be very convenient when building up complicated array definitions.

```
typedef reg [7:0] octet;
octet b;
```

is the same as

```
reg [7:0] b;
```

and

```
typedef octet [3:0]
quadOctet;
quadOctet qBytes [1:10];
```

is the same as

```
reg [3:0][7:0] qBytes [1:10];
```

Enum

SystemVerilog also introduces enumerated types, for example

```
enum { circle, ellipse, freeform } c;
```

Enumerations allow you to define a data type whose values have names. Such data types are appropriate and useful for representing state values, opcodes and other such non-numeric or symbolic data.

Typedef is commonly used together with enum, like this:

```
typedef enum { circle, ellipse, freeform } ClosedCurve;
ClosedCurve c;
```

The named values of an enumeration type act like constants. The default type is *int*. You can copy them to and from variables of the enumeration type, compare them with one another and so on. Enumerations are strongly typed. You can't copy a numeric value into a variable of enumeration type, unless you use a type-cast:

```
c = 2;              // ERROR
c = ClosedCurve'(2); // Casting - okay
```

However, when you use an enumeration in an expression, the value you are working with is the literal's integer equivalent; so, for example, it's okay to compare an enumeration variable with an integer; and it's okay to use an enumeration value in an integer expression.

Stuct and Union

Finally, SystemVerilog introduces *struct* and *union* data types, similar to those in C.

```
struct {
  int x, y;
} p;
```

Struct members are selected using the .name syntax:

```
p.x = 1;
```

Structure literals and expressions may be formed using braces.

```
p = {1,2};
```

It is often useful to declare a new structure type using typedef and then declare variables using the new type. Note also that structs may be packed.

```
typedef struct packed {
  int x, y;
} Point;
Point p;
```

Unions are useful where the same hardware resources (like a register) can store values of different types (e.g. integer, floating point, …)

SystemVerilog RTL Tutorial

This tutorial introduces some the new features in SystemVerilog that will make RTL design easier and more productive.

New Operators

SystemVerilog adds a number of new operators, mostly borrowed from C. These include increment (++) and decrement (--), and assignment operators (+=, -=, ...). The *wild equality* operators (=== and !==) act like the comparisons in a casex statement, with X and Z values meaning “don’t care”.

New loop statements

Also from C is the *do-while* loop statement and *break* and *continue*. The new *foreach* loop is used with array variables. The *for*loop has been enhanced, so that the following is permitted:

```
for (int i = 15, logic j = 0 ; i > 0 ; i--, j = ~j) ...
```

Labelling

In Verilog, you may label *begin* and *fork* statements:

```
begin : a_label
```

In SystemVerilog the label may be repeated at the end:

```
end : a_label
```

This is useful for documenting the code. The label at the end must be the same as the one at the beginning. Modules, tasks and functions may also have their names repeated at the end:

```
module MyModule ...
...
```

```
endmodule : MyModule
```

In SystemVerilog any procedural statement may be labelled:

```
loop : for (int i=0; ...
```

This is especially useful *for* loops, because they can then be disabled. Despite enhancing named blocks in this way, one reason for using them is removed: in SystemVerilog variables may be declared in unnamed blocks!

Relaxed Assignment Rules

Perhaps the hardest Verilog feature for beginners (and even experienced Verilog users are tripped up by it from time to time) is the difference between variables and nets. SystemVerilog consigns the confusion to history: variables may be assigned using procedural assignments, continuous assignments and be being connected to the outputs of module instances. Unfortunately, you still can''t connect variables to *inout* ports, although you can pass them using *ref* ports.

This means that, in SystemVerilog, you would tend to use the *logic* data type most of the time, where in Verilog you would sometimes use *reg* and sometimes *wire*. In fact *reg* and *logic* are completely interchangeable, but *logic* is a more appropriate name.

There are some restrictions, though. You are not allowed to assign the same variable from more than one continuous assignment or output port connection. This is because there is no resolution for variables like there is for nets in the case of multiple drivers. Also, if you assign a variable in one of these way, you may not assign the same variable using procedural assignments.

Port Connection Shorthand

Suppose you are using Verilog-2001 and are writing a testbench for a module which has the following declaration:

```
module Design (input Clock, Reset, input [7:0] Data, output [7:0] Q);
```

In the testbench, you might declare regs and wires:

```
reg Clock, Reset;
reg [7:0] Data;
wire [7:0] Q;
```

and you would instance the module like this:

```
Design DUT ( Clock, Reset, Data, Q );
```

or, better, like this:

```
Design DUT ( .Clock(Clock), .Reset(Reset), .Data(Data), .Q(Q) );
```

But this is a bit repetitive. SystemVerilog allows you to use the following shorthand notation:

```
Design DUT ( .Clock, .Reset, .Data, .Q );
```

where appropriately named nets and variables have previously been declared, perhaps like this:

```
logic Clock, Reset;
logic [7:0] Data;
logic [7:0] Q;
```

If even this is too verbose, you can also write this:

```
Design DUT ( .* );
```

which means “connect all ports to variables or nets with the same names as the ports”. You do not need to connect all the ports in this way. For example,

```
Design DUT ( .Clock(SysClock), .* );
```

means “connect the Clock port to SysClock, and all the other ports to variables or nets with the same names as the ports.”

Synthesis Idioms

Verilog is very widely used for RTL synthesis, even though it wasn’t designed as a synthesis language. It is very easy to write Verilog code that simulates correctly, and yet produces an incorrect design.For example, it is easy unintentionally to infer transparent latches. One of the ways in which SystemVerilog addresses this is through the introduction of new *always* keywords: *always_comb*, *always_latch* and *always_ff*.

always_comb is used to describe combinational logic. It implicitly creates a complete sensitivity list by looking at the variables and nets that are read in the process, just like *always* @* in Verilog-2001.

```
always_comb
  if (sel)
    f = x;
  else
    f = y;
```

n addition to creating a complete sensitivity list automatically, it recursively looks into function bodies and inserts any other necessary signals into the sensitivity list. It also is defined to enforce at least some of the rules for combinational logic, and it can be used as a hint (particularly by synthesis tools) to apply more rigorous synthesis style checks.Finally, always_comb adds new semantics: it implicitly puts its sensitivity list at the end of the process, so that it is evaluated just once at time zero and therefore all its outputs take up appropriate values before simulation time starts to progress.

always_latch and *always_ff* are used for infering transparent latches and flip-flops respectively. Here is an example of *always_ff*:

```
always_ff @(posedge clock iff reset == 0 or posedge reset)
  if (reset)
    q <= 0;
  else if (enable)
    q++;
```

The advantage of using all these new styles of always is that the synthesis tool can check the design intent.

Unique and Priority

Another common mistake in RTL Verilog is the misuse of the *parallel_case* and *full_case* pragmas. The problems arises because these are ignored as comments by simulators, but they are used to direct synthesis. SystemVerilog addresses this with two new keywords: *priority* and *unique.*

Unlike the pragmas, these keywords apply to *if* statements as well as *case* statements. Each imposes specific simulation behaviour that is readily mapped to synthesised hardware. *unique* enforces completeness and uniqueness of the conditional; in other words, exactly one branch of the conditional should be taken at run-time. If the specific conditions that pertain at run-time would allow more than one

branch of the conditional, or no branch at all, to be taken, there is a run-time error. For example, it is acceptable for the selectors in a case statement to overlap, but if that overlap condition is detected at runtime then it is an error. Similarly it is okay to have a unique case statement with no default branch, or an if statement with no else branch, but at run time the simulator will check that some branch is indeed taken. Synthesis tools can use this information, rather as they might a full_case directive, to infer that no latches should be created.

priority enforces a somewhat less rigorous set of checks, checking only that at least one branch of the conditional is taken. It therefore allows the possibility that more than one branch of the conditional could be taken at run-time. It licenses synthesis to create more extravagant priority logic in such a situation.

SystemVerilog Interfaces Tutorial

Interfaces are a major new construct in SystemVerilog, created specifically to encapsulate the communication between blocks, allowing a smooth refinement from abstract system-level through successive steps down to lower RTL and structural levels of the design. Interfaces also facilitate design re-use. Interfaces are hierarchical structures that can contain other interfaces.

There are several advantages when using an Interface:

- They encapsulate connectivity: an interface can be passed as a single item through a port, thus replacing a group of names by a single one. This reduces the amount of code needed to model port connections and improves its maintainability as well as readability.
- They encapsulate functionality, isolated from the modules that are connected via the interface. So, the level of abstraction and the granularity of the communication protocol can be refined totally independent of the modules.
- They can contain parameters, constants, variables, functions and tasks, processes and continuous assignments, useful for both system-level modelling and testbench applications.
- They can help build applications such as functional coverage recording and reporting, protocol checking and assertions.
- They can be used for port-less access: An interface can be instantiated directly as a static data object within a module. So, the methods used to access internal state information about the interface may be called from different points in the design to share information.
- Flexibility: An interface may be parameterised in the same way as a module. Also, a module header can be created with an unspecified interface instantiation, called a Generic Interface. This interface can be specified later on, when the module is instantiated.

At its simplest, an interface is a named bundle of wires, similar to a struct, except that an interface is allowed as a module port, while a struct is not.

The following example shows the definition and use of a very simple interface:

```
// Interface definition
interface Bus;
  logic [7:0] Addr, Data;
  logic RWn;
endinterface

// Using the interface
module TestRAM;
  Bus TheBus();                    // Instance the interface
```

```
  logic[7:0] mem[0:7];
  RAM TheRAM (.MemBus(TheBus));   // Connect it

  initial
  begin
    TheBus.RWn = 0;               // Drive and monitor the bus
    TheBus.Addr = 0;
    for (int I=0; I<7; I++)
      TheBus.Addr = TheBus.Addr + 1;
    TheBus.RWn = 1;
    TheBus.Data  = mem[0];
  end
endmodule

module RAM (Bus MemBus);
  logic [7:0] mem[0:255];

  always @*
    if (MemBus.RWn)
      MemBus.Data = mem[MemBus.Addr];
    else
      mem[MemBus.Addr] = MemBus.Data;
endmodule
```

Interface Ports

An interface can also have input, output or inout ports. Only the variables or nets declared in the port list of an interface can be connected externally by name or position when the interface is instantiated, and therefore can be shared with other interfaces. The ports are declared using the ANSI-style.

Here is an example showing an interface with a clock port:

```
interface ClockedBus (input Clk);
  logic[7:0] Addr, Data;
  logic RWn;
endinterface

module RAM (ClockedBus Bus);
  always @(posedge Bus.Clk)
    if (Bus.RWn)
      Bus.Data = mem[Bus.Addr];
    else
      mem[Bus.Addr] = Bus.Data;
endmodule

// Using the interface
```

```
module Top;
  reg Clock;

  // Instance the interface with an input, using named connection
  ClockedBus TheBus (.Clk(Clock));
  RAM TheRAM (.Bus(TheBus));
  ...
endmodule
```

Parameterised Interface

This is a simple example showing a parameterised interface:

```
interface Channel #(parameter N = 0)
    (input bit Clock, bit Ack, bit Sig);
  bit Buff[N-1:0];
  initial
    for (int i = 0; i < N; i++)
      Buff[i] = 0;
  always @ (posedge Clock)
   if(Ack = 1)
     Sig = Buff[N-1];
   else
     Sig = 0;
endinterface

// Using the interface
module Top;
  bit Clock, Ack, Sig;
  // Instance the interface. The parameter N is set to 7using named
  // connection while the ports are connected using implicit connection
  Channel #(.N(7)) TheCh (.*);
  TX TheTx (.Ch(TheCh));
  ...
endmodule
```

Modports in Interfaces

A new construct related to Interface is also added: Modport. This provides direction information for module interface ports and controls the use of tasks and functions within certain modules. The directions of ports are those seen from the module.

This example includes modports, which are used to specify the direction of the signals in the interface. The directions are the ones seen from the module to which the modport is connected, in our case the RAM:

```
interface MSBus (input Clk);
  logic [7:0] Addr, Data;
```

```
  logic RWn;
  modport Slave (input Addr, inout Data);
endinterface

module TestRAM;
  logic Clk;
  MSBus TheBus(.Clk(Clk));
  RAM TheRAM (.MemBus(TheBus.Slave));
  ...
endmodule

module RAM (MSBus.Slave MemBus);
  // MemBus.Addr is an input of RAM
endmodule
```

Tasks in Interfaces

Tasks and functions can be defined in interfaces, to allow a more abstract level of modelling.

The next example shows two tasks in an interface being used to model bus functionality. The tasks are called inside the testRAM module:

```
interface MSBus (input Clk);
  logic [7:0] Addr, Data;
  logic RWn;

  task MasterWrite (input logic [7:0] waddr,
                    input logic [7:0] wdata);
    Addr = waddr;
    Data = wdata;
    RWn = 0;
    #10ns RWn = 1;
    Data = 'z;
  endtask

  task MasterRead (input  logic [7:0] raddr,
                   output logic [7:0] rdata);
    Addr = raddr;
    RWn = 1;
    #10ns rdata = Data;
  endtask
endinterface

module TestRAM;
  logic Clk;
  logic [7:0] data;
  MSBus TheBus(.Clk(Clk));
```

```
  RAM TheRAM (.MemBus(TheBus));
  initial
  begin
    // Write to the RAM
    for (int i = 0; i<256; i++)
      TheBus.MasterWrite(i[7:0],i[7:0]);

    // Read from the RAM
    for (int i = 0; i<256; i++)
    begin
      TheBus.MasterRead(i[7:0],data);
      ReadCheck : assert (data === i[7:0])
        else $error("memory read error");
    end
  end
endmodule
```

SystemVerilog Clocking Tutorial

Clocking blocks have been introduced in SystemVerilog to address the problem of specifying the timing and synchronisation requirements of a design in a testbench.

A clocking block is a set of signals synchronised on a particular clock. It basically separates the time related details from the structural, functional and procedural elements of a testbench. It helps the designer develop testbenches in terms of transactions and cycles. Clocking blocks can only be declared inside a module, interface or program.

First Example

Here is a simple example to illustrate how SystemVerilog's clocking construct works. Consider a loadable, up/down binary counter:

```
module COUNTER (input Clock, Reset, Enable, Load, UpDn,
                input [7:0] Data, output reg[7:0] Q);
  always @(posedge Clock or posedge Reset)
    if (Reset)
      Q <= 0;
    else
      if (Enable)
        if (Load)
          Q <= Data;
        else
          if (UpDn)
            Q <= Q + 1;
          else
            Q <= Q - 1;
endmodule
```

The testbench to test this counter, without using the clocking construct, might look like this:

```
module Test_Counter;
  timeunit 1ns;

  reg Clock = 0, Reset, Enable, Load, UpDn;

  reg [7:0] Data;
  wire [7:0] Q;
  reg OK;

  // Clock generator
  always
  begin
    #5 Clock = 1;
    #5 Clock = 0;
  end

  // Test stimulus
  initial
  begin
    Enable = 0;
    Load = 0;
    UpDn = 1;
    Reset = 1;
    #10; // Should be reset
    Reset = 0;
    #10; // Should do nothing - not enabled
    Enable = 1;    #20; // Should count up to 2
    UpDn = 0;
    #40; // Should count downto 254
    UpDn = 1;

    // etc. ...
  end

  // Instance the device-under-test
  COUNTER G1 (Clock, Reset, Enable, Load, UpDn, Data, Q);

  // Check the results
  initial
  begin
    OK = 1;
    #9;
    if (Q !== 8'b00000000)
      OK = 0;
```

```
    #10;
    if (Q !== 8'b00000000)
      OK = 0;
    #20;
    if (Q !== 8'b00000010)
      OK = 0;
    #40;
    if (Q !== 8'b11111110)
      OK = 0;
    // etc. ...
  end
endmodule
```

The testbench using clocking will look like this:

```
module Test_Counter_w_clocking;
  timeunit 1ns;

  reg Clock = 0, Reset, Enable, Load, UpDn;
  reg [7:0] Data;
  wire [7:0] Q;

  // Clock generator
  always
  begin
    #5 Clock = 1;
    #5 Clock = 0;
  end

  // Test program
  program test_counter;
    // SystemVerilog "clocking block"
    // Clocking outputs are DUT inputs and vice versa
    clocking cb_counter @(posedge Clock);
      default input #1step output #4;
      output negedge Reset;
      output Enable, Load, UpDn, Data;
      input Q;
    endclocking

    // Apply the test stimulus
    initial begin

      // Set all inputs at the beginning
      Enable = 0;
      Load = 0;
```

```
      UpDn = 1;
      Reset = 1;

      // Will be applied on negedge of clock!
      ##1 cb_counter.Reset  <= 0;
      // Will be applied 4ns after the clock!
      ##1 cb_counter.Enable <= 1;
      ##2 cb_counter.UpDn   <= 0;
      ##4 cb_counter.UpDn   <= 1;
      // etc. ...
    end

    // Check the results - could combine with stimulus block
    initial begin
      ##1
      // Sampled 1ps (or whatever the precision is) before posedge clock
      ##1 assert (cb_counter.Q == 8'b00000000);
      ##1 assert (cb_counter.Q == 8'b00000000);
      ##2 assert (cb_counter.Q == 8'b00000010);
      ##4 assert (cb_counter.Q == 8'b11111110);
      // etc. ...
     end

    // Simulation stops automatically when both initials have been completed

  endprogram

  // Instance the counter
  COUNTER G1 (Clock, Reset, Enable, Load, UpDn, Data, Q);

  // Instance the test program - not required, because program will be
  // instanced implicitly.
  // test_COUNTER T1 ();
endmodule
```

There are a few important things to note: the testbench is implemented as a module, with a nested program that contains the clocking block (the full explanation of the advantages of implementing a testbench using a program can be found in the Program article). Program blocks can be nested within modules or interfaces. This way multiple co-operating programs can share variables local to the scope. Nested programs with no ports or top-level programs that are not explicitly instantiated are implicitly instantiated once. Implicitly instantiated programs have the same instance and declaration name.

The clocking construct is both the declaration and the instance of that declaration. Note that the signal directions in the clocking block within the testbench are with respect to the testbench. So Q is an output of COUNTER, but a clocking input. Note also that widths are not declared in the clocking block, just the directions.

The signals in the clocking block cb_counter are synchronised on the posedge of Clock, and by default all signals have a 4ns output (drive) skew and a #1step input (sample) skew. The skew determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative (i.e. they always refer to a time before the clock), whereas output skews always refer to a time after the clock.

An input skew of #1step indicates that the value read by the active edge of the clock is always the last value of the signal immediately before the corresponding clock edge. A step is the time precision.

The ## operator is used in the testbench to delay execution by a specified number of clocking events, or clock cycles.

Clocking Block Drives

Clocking block outputs and inouts can be used to drive values onto their corresponding signals, at a certain clocking event and with the specified skew. An important point to note is that a drive does not change the clock block input of an inout signal. This is because reading the input always yields the last sampled value, and not the driven value.

Synchronous signal drives are processed as nonblocking assignments. If multiple synchronous drives are applied to the same clocking block output or inout at the same simulation time, a run-time error is issued and the conflicting bits are set to X for 4-state ports or 0 for 2-state ports.

Here are some examples using the driving signals from the clocking block cb:

```
cb.Data[2:0] <= 3'h2;    // Drive 3-bit slice of Data in current cycle
##1 cb.Data <= 8'hz;         // Wait 1 Clk cycle and then drive Data
##2 cb.Data[1] <= 1;         // Wait 2 cycles, then drive bit 1 of Data
cb.Data <= ##1 Int_Data; // Remember the value of Int_Data, and then
                         // drive Data 1 Clk cycle later
cb.Data[7:4] <= 4'b0101;
cb.Data[7:4] <= 4'b0011; // Error: driven value of Data[7:4] is 4'b0xx1
```

Clocking Blocks and Interfaces

This is an example presenting multiple clocking blocks using interfaces. A clocking block can use an interface to reduce the amount of code needed to connect the testbench.

The interface signals will have the same direction as specified in the clocking block when viewed from the testbench side (e.g. modport TestR), and reversed when viewed from the DUT (i.e. modport Ram). The signal directions in the clocking block within the testbench are with respect to the testbench, while a modport declaration can describe either direction (i.e. the testbench or the design under test). To illustrate we will implement two busses, with different clocks, and a testbench separated from the top level. The testbench is implemented as a program.

```
// Interface definitions
interface DataBus (input Clock);
  logic [7:0] Addr, Data;
  modport TestR (inout Addr, inout Data);
  modport Ram (inout Addr, inout Data);
endinterface

interface CtrlBus (input Clock);
  logic RWn;
```

```
  // RWn is output, as it is in the clocking block
  modport TestR (output RWn);
  // RWn is input, reversed than in the clocking block
  modport Ram (input RWn);
endinterface

// Testbench defined as a program, with two clocking blocks
program TestRAM (DataBus.TestR DataInt,
                 CtrlBus.TestR CtrlInt);
  clocking cb1 @(posedge DataInt.Clock);
    inout #5ns DataInt.Data;
    inout #2ns DataInt.Addr;
  endclocking

  clocking cb2 @(posedge CtrlInt.Clock);
    output #10;
    output RWn = CtrlInt.RWn;  //  Hierarchical expression
  endclocking

  initial begin
    cb2.RWn = 0;
    cb1.DataInt.Data = 1;
    ...
  end
endprogram

module RAM (DataBus.Ram DataInt, CtrlBus.Ram CtrlInt);
  logic [7:0] mem[0:255];

  always @*
    if (CtrlInt.RWn)
      DataInt.Data = mem[DataInt.Addr];
    else
      mem[DataInt.Addr] = DataInt.Data;
endmodule

module Top;
  logic Clk1, Clk2;

  // Instance the interfaces
  DataBus TheDataBus(.Clock(Clk1));
  CtrlBus TheCtrlBus(.Clock(Clk2));

  RAM TheRAM (.DataBus.Ram(TheDataBus.Ram),
```

```
              .CtrlBus.Ram(TheCtrlBus.Ram)); // Connect them
  TestRAM TheTest (.DataBus.TestR(TheDataBus.TestR),
                   .CtrlBus.TestR(TheCtrlBus.TestR));
endmodule
```

Clocking block events

The clocking event of a clocking block can be accessed directly by using the clocking block name, e.g. @(cb) is equivalent to @(posedge Clk). Individual signals from the clocking block can be accessed using the clocking block name and the dot (.) operator. All events are synchronised to the clocking block.

Here are some other examples of synchronisation statements:

```
// Wait for the next change of Data signal from the cb clocking block
@(cb.Data);

// Wait for positive edge of signal cb.Ack
@(posedge cb.Ack);

// Wait for posedge of signal cb.Ack or negedge of cb.Req
@(posedge cb.Ack or negedge cb.Req);

// Wait for the next change of bit 2 of cb.Data
@(cb.Data[2]);
// Wait for the next change of the specified slice
@(cb.Data[7:5]);
```

Further examples of clocking block constructs may be found in the Accellera SystemVerilog LRM, section 15,

SystemVerilog Assertions Tutorial

Introduction

Assertions are primarily used to validate the behaviour of a design. ("Is it working correctly?") They may also be used to provide functional coverage information for a design ("How good is the test?"). Assertions can be checked dynamically by simulation, or statically by a separate property checker tool – i.e. a formal verification tool that proves whether or not a design meets its specification. Such tools may require certain assumptions about the design's behaviour to be specified.

In SystemVerilog there are two kinds of assertions: immediate (`assert`) and concurrent (`assert property`). Coverage statements (`cover property`) are concurrent and have the same syntax as concurrent assertions, as do `assume property`statements. Another similar statement – `expect` – is used in testbenches; it is a procedural statement that checks that some specified activity occurs. The three types of concurrent assertion statement and the `expect` statement make use of sequences and properties that describe the design's temporal behaviour – i.e. behaviour over time, as defined by one or more clocks.

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to the `if` statement. The difference is that an `if` statement does not assert that an expression is true, it simply checks that it is true, e.g.:

```
if (A == B) ...  // Simply checks if A equals B

assert (A == B); // Asserts that A equals B; if not, an error is generated
```

If the conditional expression of the immediate `assert` evaluates to X, Z or 0, then the assertion fails and the simulator writes an error message.
An immediate assertion may include a pass statement and/or a fail statement. In our example the *pass statement* is omitted, so no action is taken when the `assert` expression is true. If the pass statement exists:

```
assert (A == B) $display ("OK. A equals B");
```

it is executed immediately after the evaluation of the `assert` expression. The statement associated with an `else` is called a *fail statement* and is executed if the assertion fails:

```
assert (A == B) $display ("OK. A equals B");

  else $error("It's gone wrong");
```

Note that you can omit the pass statement and still have a fail statement:

```
assert (A == B) else $error("It's gone wrong");
```

The failure of an assertion has a severity associated with it. There are three severity system tasks that can be included in the fail statement to specify a severity level: `$fatal`, `$error` (the default severity) and `$warning`. In addition, the system task `$info`indicates that the assertion failure carries no specific severity.
Here are some examples:

```
ReadCheck: assert (data === correct_data)

           else $error("memory read error");

Igt10: assert (I > 10)

        else $warning("I is less than or equal to 10");
```

The pass and fail statements can be any legal SystemVerilog procedural statement. They can be used, for example, to write out a message, set an error flag, increment a count of errors, or signal a failure to another part of the testbench.

```
AeqB: assert (a === b)

  else begin error_count++; $error("A should equal B"); end
```

Concurrent Assertions

The behaviour of a design may be specified using statements similar to these:

"The Read and Write signals should never be asserted together."

"A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."

Concurrent assertions are used to check behaviour such as this. These are statements that assert that *specified properties* must be true. For example,

```
assert property (!(Read && Write));
```

asserts that the expression `Read && Write` is never true at any point during simulation.

Properties are built using sequences. For example,

```
assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

where `Req` is a simple sequence (it's just a boolean expression) and `##[1:2] Ack` is a more complex sequence expression, meaning that `Ack` is true on the next clock, or on the one following (or both). `|->` is the implication operator, so this assertion checks that whenever `Req` is asserted, `Ack` must be asserted on the next clock, or the following clock.

Concurrent assertions like these are checked throughout simulation. They usually appear outside any initial or always blocks in modules, interfaces and programs. (Concurrent assertions may also be used as statements in initial or always blocks. A concurrent assertion in an initial block is only tested on the first clock tick.)

The first assertion example above does not contain a clock. Therefore it is checked at every point in the simulation. The second assertion is only checked when a rising clock edge has occurred; the values of `Req` and `Ack` are sampled on the rising edge of Clock.

Implication

The *implication* construct (`|->`) allows a user to monitor sequences based on satisfying some criteria, e.g. attach a precondition to a sequence and evaluate the sequence only if the condition is successful. The left-hand side operand of the implication is called the *antecedent sequence expression*, while the right-hand side is called the *consequent sequence expression*.

If there is no match of the *antecedent sequence expression*, implication succeeds vacuously by returning true. If there is a match, for each successful match of the *antecedent sequence expression*, the *consequent sequence expression* is separately evaluated, beginning at the end point of the match.

There are two forms of *implication*: overlapped using operator `|->`, and non-overlapped using operator `|=>`.

For overlapped implication, if there is a match for the *antecedent sequence expression*, then the first element of the *consequent sequence expression* is evaluated on the same clock tick.

```
s1 |-> s2;
```

In the example above, if the sequence `s1` matches, then sequence `s2` must also match. If sequence `s1` does not match, then the result is true.

For non-overlapped implication, the first element of the *consequent sequence expression* is evaluated on the next clock tick.

```
s1 |=> s2;
```

The expression above is basically equivalent to:

```
`define true 1

s1 ##1 `true |-> s2;
```

where `` `true `` is a boolean expression, used for visual clarity, that always evaluates to true.

Properties and Sequences

In these examples we have been using, the properties being asserted are specified in the `assert property` statements themselves. Properties may also be declared separately, for example:

```
property not_read_and_write;

  not (Read && Write);

endproperty assert property (not_read_and_write);
```

Complex properties are often built using sequences. Sequences, too, may be declared separately:

```
sequence request

  Req;

endsequence



sequence acknowledge

  ##[1:2] Ack;

endsequence



property handshake;

  @(posedge Clock) request |-> acknowledge;

endproperty



assert property (handshake);
```

Assertion Clocking

Concurrent assertions (`assert property` and `cover property` statements) use a generalised model of a clock and are only evaluated when a clock tick occurs. (In fact the values of the variables in the property are sampled right at the end of the previous time step.) Everything in between clock ticks is ignored. This model of execution corresponds to the way a RTL description of a design is interpreted after synthesis.

A clock tick is an atomic moment in time and a clock ticks only once at any simulation time. The clock can actually be a single signal, a gated clock (e.g. `(clk && GatingSig)`) or other more complex expression. When monitoring asynchronous signals, a simulation time step corresponds to a clock tick.

The clock for a property can be specified in several ways:

o Explicitly specified in a sequence:

```
sequence s;
  @(posedge clk) a ##1 b;
endsequence
property p;
  a |-> s;
endproperty
assert property (p);
```

o Explicitly specified in the property:

```
property p;
  @(posedge clk) a ##1 b;
endproperty
assert property (p);
```

o Explicitly specified in the concurrent assertion:

```
assert property (@(posedge clk) a ##1 b);
```

o Inferred from a procedural block:

```
property p;
  a ##1 b;
endproperty
always @(posedge clk) assert property (p);
```

o From a `clocking` block (see the Clocking Blocks tutorial):

```
clocking cb @(posedge clk);

  property p;

    a ##1 b;

  endproperty

endclocking

assert property (cb.p);
```

o From a default clock (see the Clocking Blocks tutorial):

```
default clocking cb;
```

Handling Asynchronous Resets

In the following example, the `disable iff` clause allows an asynchronous reset to be specified.

```
property p1;

  @(posedge clk) disable iff (Reset) not b ##1 c;

endproperty



assert property (p1);
```

The `not` negates the result of the sequence following it. So, this assertion means that if `Reset` becomes true at any time during the evaluation of the sequence, then the attempt for `p1` is a success. Otherwise, the sequence `b ##1 c` must never evaluate to true.

Sequences

A `sequence` is a list of boolean expressions in a linear order of increasing time. The `sequence` is true over time if the boolean expressions are true at the specific clock ticks. The expressions used in sequences are interpreted in the same way as the condition of a procedural `if` statement.

Here are some simple examples of sequences. The `##` operator delays execution by the specified number of clocking events, or clock cycles.

```
a ##1 b             // a must be true on the current clock tick

                   // and b on the next clock tick

a ##N b             // Check b on the Nth clock tick after a

a ##[1:4] b          // a must be true on the current clock tick and b
```

```
                    // on some clock tick between the first and fourth

                    // after the current clock tick
```

The * operator is used to specify a consecutive repetition of the left-hand side operand.

```
a ##1 b [*3] ##1 c    // Equiv. to a ##1 b ##1 b ##1 b ##1 c

(a ##2 b) [*2]        // Equiv. to (a ##2 b ##1 a ##2 b)

(a ##2 b)[*1:3]       // Equiv. to (a ##2 b)

                    // or (a ##2 b ##1 a ##2 b)

                    // or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

The $ operator can be used to extend a time window to a finite, but unbounded range.

```
a ##1 b [*1:$] ##1 c  // E.g. a b b b b c
```

The [-> or *goto repetition* operator specifies a non-consecutive sequence.

```
a ##1 b[->1:3] ##1 c  // E.g. a !b b b !b !b b c
```

This means a is followed by any number of clocks where c is false, and b is true between 1 and three times, the last time being the clock before c is true.
The [= or *non-consecutive repetition* operator is similar to *goto repetition*, but the expression (b in this example) need not be true in the clock cycle before c is true.

```
a ##1 b [=1:3] ##1 c // E.g. a !b b b !b !b b !b !b c
```

Combining Sequences

There are several operators that can be used with sequences:
The binary operator and is used when both operand expressions are expected to succeed, but the end times of the operand expressions can be different. The end time of the end operation is the end time of the sequence that terminates last. A sequence succeeds (i.e. is true over time) if the boolean expressions containing it are true at the specific clock ticks.

```
s1 and s2      // Succeeds if s1 and s2 succeed. The end time is the

               // end time of the sequence that terminates last
```

If s1 and s2 are sampled booleans and not sequences, the expression above succeeds if both s1 and s2 are evaluated to be true.
The binary operator intersect is used when both operand expressions are expected to succeed, and the end times of the operand expressions must be the same.

```
s1 intersect s2 // Succeeds if s1 and s2 succeed and if end time of s1 is

                // the same with the end time of s2
```

The operator `or` is used when at least one of the two operand sequences is expected to match. The sequence matches whenever at least one of the operands is evaluated to true.

```
s1 or s2       // Succeeds  whenever at least one of two operands s1

               // and s2 is evaluated to true
```

The `first_match` operator matches only the first match of possibly multiple matches for an evaluation attempt of a sequence expression. This allows all subsequent matches to be discarded from consideration. In this example:

```
sequence fms;

  first_match(s1 ##[1:2] s2);

endsequence
```

whichever of the `(s1 ##1 s2)` and `(s1 ##2 s2)` matches first becomes the result of sequence `fms`.
The `throughout` construct is an abbreviation for writing:

```
(Expression) [*0:$] intersect SequenceExpr
```

i.e. `Expression throughout` *`SequenceExpr`* means that *`Expression`* must evaluate true at every clock tick during the evaluation of *`SequenceExpr`*.
The `within` construct is an abbreviation for writing:

```
(1[*0:$] ##1 SeqExpr1 ##1 1[*0:$]) intersect SeqExpr2
```

i.e. *`SequenceExpr1`* `within` *`SequenceExpr2`* means that *`SeqExpr1`* must occur at least once entirely within *`SeqExpr2`* (both start and end points of *`SeqExpr1`* must be between the start and the end point of *`SeqExpr2`*).

Variables in Sequences and Properties

Variables can be used in sequences and properties. A common use for this occurs in pipelines:

```
`define true 1



property p_pipe;

  logic v;

  @(posedge clk) (`true,v=DataIn) ##5 (DataOut === v);
```

```
endproperty
```

In this example, the variable `v` is assigned the value of `DataIn` unconditionally on each clock. Five clocks later, `DataOut` is expected to equal the assigned value. Each invocation of the property (here there is one invocation on every clock) has its own copy of `v`. Notice the syntax: the assignment to `v` is separated from a sequence expression by a comma, and the sequence expression and variable assignment are enclosed in parentheses.

Coverage Statements

In order to monitor sequences and other behavioural aspects of a design for *functional coverage*, `cover property` statements can be used. The syntax of these is the same as that of `assert property`. The simulator keeps a count of the number of times the property in the cover property statement holds or fails. This can be used to determine whether or not certain aspects of the designs functionality have been exercised.

```
module Amod2(input bit clk);

  bit X, Y;

  sequence s1;

    @(posedge clk) X ##1 Y;

  endsequence

  CovLavel: cover property (s1);

  ...

endmodule
```

SystemVerilog also includes `covergroup` statements for specifying functional coverage. These are introduced in the Constrained-Random Verification Tutorial.

Assertion System Functions

SystemVerilog provides a number of system functions, which can be used in assertions.

`$rose`, `$fell` and `$stable` indicate whether or not the value of an expression has changed between two adjacent clock ticks. For example,

```
assert property

  (@(posedge clk) $rose(in) |=> detect);
```

asserts that if in changes from 0 to 1 between one rising clock and the next, detect must be 1 on the following clock.

This assertion,

```
assert property

  (@(posedge clk) enable == 0 |=> $stable(data));
```

states that `data` shouldn't change whilst `enable` is 0.

The system function `$past` returns the value of an expression in a previous clock cycle. For example,

```
assert property

  (@(posedge clk) disable iff (reset)

                enable |=> q == $past(q+1));
```

states that `q` increments, provided `reset` is low and `enable` is high.

Note that the argument to `$past` may be an expression, as shown above.

The system functions `$onehot` and `$onehot0` are used for checking one-hot encoded signals. `$onehot(expr)` returns true if exactly one bit of `expr` is high; `$onehot0(expr)` returns true if at most one bit of `expr` is high.

```
assert property (@(posedge clk) $onehot(state));
```

There are other system functions.

Binding

We have seen that assertions can be included directly in the source code of the modules in which they apply. They can even be embedded in procedural code. Alternatively, verification code can be written in a separate program, for example, and that program can then be bound to a specific module or module instance.

For example, suppose there is a module for which assertions are to be written:

```
module M (...);

  // The design is modelled here

endmodule
```

The properties, sequences and assertions for the module can be written in a separate program:

```
program M_assertions(...);

  // sequences, properties, assertions for M go here

endprogram
```

This program can be bound to the module `M` like this:

```
bind M M_assertions M_assertions_inst (...);
```

The syntax and meaning of `M_assertions` is the same as if the program were instanced in the module itself:

```
module M (...);

  // The design is modelled here
```

```
  M_assertions M_assertions_inst (...);

endmodule
```

SystemVerilog Classes Tutorial

SystemVerilog introduces classes as the foundation of the testbench automation language. Classes are used to model data, whose values can be created as part of the constrained random methodology.

A class is a user-defined data type. Classes consist of data (called *properties*) and tasks and functions to access the data (called*methods*). Classes are used in object-oriented programming. In SystemVerilog, classes support the following aspects of object-orientation – encapsulation, data hiding, inheritance and polymorphism.

Class Declaration

Here is a simple class declaration:

```
class C;
  int x;
  task set (int i);
    x = i;
  endtask
  function int get;
    return x;
  endfunction
endclass
```

This class has a single data member, x, and two methods, set() and get(). To use the class, an object must be created:

```
C c1;
c1 = new;
```

The first of these statements declares c1 to be a C. In other words, the variable c1 can contain a *handle* to an *object* (i.e. an instance) of the class C. The second statement creates an object and assigns its handle to c1. The two statements could be replaced by the following statement, which declares a variable, creates a class object and initialises the variable:

```
C c1 = new;
```

Having created a class object, we can use the class methods to assign and look at the data value, x:

```
initial
begin
  c1.set(3);
  $display("c1.x is %d", c1.get());
end
```

Data Hiding

Although the task set() and the function get() were intended to be the means by which the class's member

variable x was assigned and retrieved, it would be possible to do this directly:

```
initial
begin
  c1.x = 3;
  $display("c1.x is %d", c1.x);
end
```

This is because all class members are, by default, publicly visible. To hide x, it must be declared local:

```
local int x;
```

It is now illegal to access c1.x outside the class, except using the class methods.

Parameterised Classes

Classes may be parameterised in the same way that modules may.

```
class Register #(parameter int N = 1);
  bit [N-1:0] data;
  ...
endclass
```

The default parameter value can be overridden when the class is instantiated.

```
Register #(4) R4;                    // data is bit [3:0]
Register #(.N(8)) R8                 // data is bit [7:0]
Register R;                          // data is bit [0:0]
```

It is also possible to pass a data type to a class:

```
class Register #(parameter type T = int);
  T data;
  ...
endclass

Register Rint;                       // data is int
Register #(bit [7:0]) Rint;          // data is bit [7:0]
```

Extending Classes – Inheritance

One of the key features of object-oriented programming is the ability to create new classes that are based on existing classes. A derived class by default inherits the properties and methods of its parent or base class. However, the derived class may add new properties and methods, or modify the inherited properties and methods. In other words, the new class is a more specialised version of the original class.

In SystemVerilog the syntax for deriving or inheriting one class from another is this:

```
class Derived extends BaseClass;
  // New and overridden property and method declarations.
endclass
```

Consider the example of the class Register, which was used earlier. This could represent a general-purpose data register, which stores a value. We could derive a new class, ShiftRegister, from this that

represents a specialised type of a register.

```
class ShiftRegister extends Register;
  task shiftleft;  data = data << 1; endtask
  task shiftright; data = data >> 1; endtask
endclass
```

Objects of the class ShiftRegister can be manipulated using the original set() and get() methods, as well as the shiftleft and shiftright methods. However, there is a problem if the data property in Register was declared as local – it would not be visible in the extended class! So instead, it would need to be declared protected:

```
class Register;
  protected int data;
  ...
endclass
```

A *protected* member is one that is not visible outside the class, but is visible in derived classes; local members are not visible, except in the class in which they are declared.

Virtual Classes and Methods

Sometimes, it is useful to create a class without intending to create any objects of the class. The class exists simply as a base class from which other classes can be derived. In SystemVerilog this is called an *abstract class* and is declared by using the word *virtual*:

```
virtual class Register;
  ...
endclass
```

Methods, too, may be declared virtual. This means that if the method is overridden in a derived class, the signature (the return type, the number and types of its arguments) must be the same as that of the virtual method. This provides a mechanism for saying, “I want all derived classes to have a method that looks exactly like this.” A virtual method in an abstract class need not have a body – this will have to be defined in a non-abstract derived class.

Note that methods in a virtual class need not be virtual and that virtual methods may be declared in non-abstract classes.

Other Features of Classes

This tutorial has provided an introductory overview of classes in SystemVerilog. Refer to the SystemVerilog LRM for more details.

Classes can include random variables and constraints. The tutorial on Constrained-Random Testing explains these features and contains further examples of classes.

Testbench Automation and Constraints Tutorial

In this tutorial we illustrate how to use classes that represent data objects in a constrained-random testbench..This tutorial illustrates the following key points:

- Using classes to represent data structures
- Specifying which data values should be random
- Specifying constraints
- Generating directed-random values in a testbench
- Using coverage to measure and guide verification

Directed-Random Verification

Traditionally, simulation-based verification has used a directed testing approach. In other words, a testbench implements tests using specific data values.Consider the example of a memory system. It is not possible to test such a system exhaustively – it would be impractical to write every possible data value to every possible address in every possible sequence.

1. In a directed testing approach, you might select some appropriate data values and write them into some selected memory locations and then read them out again. One problem with this approach is that you could miss certain types of system error – for example errors with certain addresses or when using certain data values.
2. Using a random testing approach, you might find more errors, but unless you run the simulations for long periods of time, you still might not detect certain problems.
3. In a directed random test, you control how random the data values are using constraints. For example, you might want to make sure that some memory locations are tested exhaustively, and that “corner cases” (i.e. significant cases such as the minimum and maximum address values) are definitely tested. You might want to write to an ascending or descending sequence of addresses.

SystemVerilog supports all three paradigms: directed, random and directed random testing. It does this by providing for random data value generation under the control of constraints.

In order to measure how good a test is, SystemVerilog provides constructs for specifying functional coverage models and measuring the coverage during simulation. By analysing the coverage data, tests can be directed to ensure they do indeed test the design adequately.

Using classes to represent data structures

Most practical verification problems require you to implement some kind of transaction in which a collection of data is transferred into or out of the design under test (DUT). This collection of data may be as simple as the address and data being transferred on a system bus, or something much more elaborate like a complete image represented as video data. In any case, it is appropriate to create an abstract data structure that can be used to represent this information as it moves through the verification system and the DUT.

As an example of this kind of data modelling we will consider messages in a CANbus network (CANbus is a networking system used for in-vehicle data buses described in ISO standard 11898).

The CANbus message format has two possible versions. The simpler 2.0A format, which we will use for this example, has the following fields:

- An 11-bit "identifier" (address)
- A single-bit field known as "RTR" indicating whether a reply is expected
- Two "reserved" bits, fixed at zero in the 2.0A format
- A 4-bit "data length" field, containing a binary value in the range 0 to 8
- A data payload consisting of 0 to 8 bytes, as indicated by the "data length" field
- A 15-bit CRC (checksum) field

We can easily create a struct to represent this data structure. Each field in the data structure is directly represented by a field in our struct. Those fields can be given bit widths using an appropriate SystemVerilog data type, such as bit or logic. For an eight-bit field, the type byte is used. bit [7:0] could have been used instead – the choice is a matter of style and convenience. (byte is a signed type, but that is not relevant here.)

```
typedef struct {
```

```
  rand bit [10:0] ID;      // 11-bit identifier
  rand bit        RTR;     // reply required?
       bit  [1:0] rsvd;    // "reserved for expansion" bits
  rand bit  [3:0] DLC;     // 4-bit Data Length Code
  rand byte       data[];  // data payload
       bit [14:0] CRC;     // 15-bit checksum
} message_t;
```

In the case of a system where data is sent serially, a packed struct could be convenient - a packed struct means that the data structure can be packed into a single vector, making it easier to use in a system where information is sent serially. However in this case, we want to randomize both the contents and the length of the payload data, and that is more easily done using an unpacked array - hence we have to use an unpacked struct.

Note also the use of typedef - this allows us to create a re-usable name for our struct data type message_t

To get the full benefits of the SystemVerilog constrained random generation facilities, it is convenient to use a class. This allows us to associate functions with our data (class methods), and also to benefit from other built-in methods such as pre_randomize() and post_randomize(). So for maximum flexibility, our next step is to create a class:

```
class CAN_Message;

  rand message_t message;



  // Class methods go here



endclass: CAN_Message
```

Class methods

Now we have defined our CAN_message class, we need to add methods to the struct that can modify or inspect it. We would need to add methods to this class for many purposes, including calculating the correct 15-bit CRC. For example, consider this very straightforward method to set or clear the RTR (reply request) bit in a message structure, ensuring that there is no payload data if the RTR bit is set:

```
class CAN_Message;

  rand message_t message;
```

```
  // Class methods go here


  task set_RTR (bit new_value);
    // Set the RTR bit as requested
    message.RTR = new_value;
    if (message.RTR) begin
      // Messages with the RTR bit set should have no data.
      message.DLC = 0;
      clear_data();  // make the data list empty
    end
  endtask


  task clear_data;
    message.data.delete();
  endtask


endclass: CAN_Message
```

Note that this method is itself a part of the CAN_message class, so that it can directly access any fields of the struct, using the .member notation.

Generation (randomize)

The idea of pseudo-random stimulus generation is central to the directed random verification methodology. It's obviously ridiculous to use random numbers for every part of every struct. You need control over the random generation process. SystemVerilog provides this control using constraints.

A constraint is a Boolean expression describing some property of a field. Constraints direct the random generator to choose values that satisfy the properties you specify in your constraints. Within the limits of your constraints, the values are still randomly chosen. The process of choosing values that satisfy the constraints is called solving. The verification tool that does this is called the solver; the solver may be embedded in a simulator or be part of a separate testbench generator program.

For example, the four-bit DLC field in our CAN_message.message struct can hold values in the range 0 to 15, but the CANbus message specifications require its value to be restricted to a maximum of 8. We can express this constraint as the numerical inequality

```
DLC <= 8
```

or, perhaps more clearly, using SystemVerilog's range-membership operator inside

```
 DLC inside {[0:8]}
```

These are both Boolean expressions and therefore they can be used in a constraint using the constraint keyword. Constraints are class members, just like fields and methods. They can be written either in the original class, or in derived classes. In this example we are modifying the original class definition. The example also shows how you can control the number of elements in a dynamic array by using the dynamic_array.size()method as part of a constraint.

```
class CAN_Message;

 //...



 constraint c1 { message.DLC inside {[0:8]}; }

 constraint c2 { message.data.size() == message.DLC; }



endclass: CAN_Message
```

The random generator will always attempt to honour your constraints. It is sometimes possible to write conflicting constraints, in which case the generator will fail.

Writing the Testbench

Now that we've completed this class definition, we need to be able to make use of it in the testbench. As a simple example of this process, suppose we want to build a test that needs ten distinct messages to do its work. We would create an unpacked array of 10 CAN_message objects:

```
CAM_message test_message[10];
```

We could then initialize the messages with random data like this:

```
for (int i = 0; i < 10; i++)

 test_message[i].randomize();
```

We could also provide additional constraints using the with construct:

```
test_message[0].randomize() with { message.DCL == 4; };
```

This is the same as writing a constraint in the CAN_message class like this

```
constraint c3 { message.DCL == 4; }
```

Alternatively, we could use the class inheritance mechanism to create a subclass, where the message length is fixed:

```
class CAN_message_4 extends CAN_message;

  // ...

  constraint c1 { message DCL == 4; }    // Overload c1

endclass
```

Suppose that the DUT has a serial input for receiving CAN messages. In order to drive the abstract class data into the DUT, the message struct will need to be serialised. To do this we can write a method in the class.

```
class CAN_Message;

  rand message_t message;



  // Class methods go here

  // ...



  task getbits(ref bit data_o, input int delay=1);

    bit [17:0] header;

    bit [14:0] tail;

    header = {message.ID,message.RTR,message.rsvd,message.DLC};

    tail = message.CRC;

    $display("tail=%0b",tail);

    //step through message and output each bit (from left to right)

    foreach(header[i]) #delay data_o = header[i];

    foreach(message.data[i,j]) #delay data_o = message.data[i][j];

    foreach(tail[i]) #delay data_o = tail[i];

  endtask
```

```
  //...


endclass: CAN_Message
```

This getbits task updates the output data_o by using a ref argument. An input delay is specified as a simple model of the bit period. Here is an example of calling this function:

```
module top();

  // declaration of CAN_message and message_t omitted...

  bit data_o;

  const int bit_interval = 1;

  CAN_Message test_message[10];

  int interval=10;



  initial

  message_gen: begin

    for (int i = 0; i < 10; i++) begin

      std::randomize(interval) with {interval>0;interval<6;};//random interval

      $display("interval=%0d",interval);

      #interval;

      $display("time = %0t",$time);

      test_message[i] = new;

      test_message[i].randomize();

      test_message[i].print();

      test_message[i].getbits(data_o,bit_interval);

      #bit_interval $display("time = %0t",$time);

    end

    $finish;
```

```
  end:message_gen



endmodule : top
```

So far in this tutorial we have looked at how random variables and constraints in classes are used to create tests. SystemVerilog also provides a number of other constructs that are not covered here, including the ability to create random sequences of tokens..

Functional Coverage

Having seen how to write tests using SystemVerilog, we shall now consider how we can measure their effectiveness. One way to do this is to measure the functional coverage. This is a user-defined metric of how much of the design has been tested. (SystemVerilog also includes the concurrent cover property statement, which is used to count the number of times a particular sequence or property occurs. For further information see the Assertion-Based verification Tutorial.)

As an example of functional coverage, consider a variable of a user-defined enumerated type:

```
enum {Red, Green, Blue} Colour;
```

It would be useful to know whether or not the variable Colour has been set to all the possible values at some point during simulation. To do this you would define a covergroup containing a single coverpoint:

```
covergroup cg_Colour @(posedge Clock);

  coverpoint Colour;

endgroup
```

Next you must create an instance of the covergroup. This is like creating a class object:

```
cg_Colour = new cg_inst;
```

During simulation, the simulator will count the number of times that Colour takes each of the values, Red, Green and Blue. The value of Colour is sampled on every rising edge of Clock. (You don't have to specify a sampling event; if you don't then you must sample the values explicitly, using the covergroup's sample method – cg_inst.sample());

Bins

In the example we have just used, the simulator will create three bins for the coverpoint - one for each value of the enumerated type. Suppose we are covering a variable of an integer type:

```
bit [15:0] i;

covergroup cg_Short @(posedge Clock);

  c : coverpoint i;

endgroup
```

The simulator could potentially create 2^16 bins for the coverpoint. (In fact, there is a default of a maximum of 64 automatically created bins.) It would probably be more useful to define some bins to hold specific values or ranges of values:

```
covergroup cg_Short @(posedge Clock);

  coverpoint i {

    bins zero     = { 0 };

    bins small    = { [1:100] };

    bins hunds[3] = { 200,300,400,500,600,700,800,900 };

    bins large    = { [1000:$] };

    bins others[] = default;

  };

endgroup
```

This creates one bin, “zero”, for the value of i being 0; one bin, “small”, for all values of i between 1 and 100, inclusive; three bins, for the eight values listed, with the first holding 200 and 300, the next 400 and 500 and the last 600, 700, 800 and 900; one bin for values 1000 and above, and one bin for every other value.

Cross Coverage

It is often useful to know how often two (or more) variables have specific pairs (triples etc.) of values. This is achieved using cross coverage:

```
logic [3:0] x, y;



covergroup cg_xy @(posedge Clock);

  X  : coverpoint x;

  Y  : coverpoint y;

  XY : cross X, Y;

endgroup
```

This will create 16 bins for each of the coverpoints X and Y and 256 bins for XY – one for each possible pair of values. Note that SystemVerilog coverpoints only operate on 2 state values: values x or z are excluded.

Covering Transitions

Coverage of transitions may also be collected. An example where this may be used is for finite state machines. Consider a state machine with three states, Idle, State1 and State2, where the only legal transitions are those to and from Idle. In addition, the state machine should only remain in the Idle state for a maximum of 4 clocks.

```
enum {Idle, State1, State2} State;



covergroup cg_State @(posedge Clock);

  states     : coverpoint State;

  state_trans : coverpoint State {

    bins legal[] = ( Idle => State1, State2 ),

                 ( State1, State2 => Idle);

    bins idle[] = ( Idle [* 2:4] );

    bins illegal = default sequence;

  }

endgroup
```

This would create a separate bin for each legal transition – including remaining in Idle – and one bin for all the illegal transitions.
SystemVerilog also provides the illegal_bins construct, which causes the simulator to stop with an error if an illegal value or transition occurs:

```
covergroup cg_State @(posedge Clock);

  ...

    illegal_bins illegal = default sequence;

  }

endgroup
```

Coverage options

Options control the behaviour of covergroups and coverpoints. For example, the coverage results for a particular covergroup or coverpoint may be weighted, or a maximum number of automatically created bins could be specified. Options such as these can be set in the covergroup, or procedurally after the covergroup has been instanced.

```
int i_a, i_b, i_c;
```

```
covergroup cg @(posedge Clock);
  option.auto_bin_max = 10;
  a : coverpoint i_a;
  b : coverpoint i_b;
  c : coverpoint i_c { option.auto_bin_max = 20; }
endgroup

cg cg_inst = new;
cg_inst.a.option.weight = 2;
```

In this example, 10 bins are created for the coverpoints cg_inst.a and cg_inst.b and 20 bins are created for cg_inst.c. cg_inst.a is assigned a weight of 2, whereas the other coverpoints each have a weight of 1 (the default weight).

There are many other options – refer to the SystemVerilog LRM for details of these.

Other Features of Coverage

The other functional coverage features that have not been covered in this tutorial are covergroup arguments; wildcard bins and block execution events. For details of these, please refer to the SystemVerilog LRM .

SystemVerilog DPI Tutorial

The SystemVerilog Direct Programming Interface (DPI) is basically an interface between SystemVerilog and a foreign programming language, in particular the C language. It allows the designer to easily call C functions from SystemVerilog and to export SystemVerilog functions, so that they can be called from C.

The DPI has great advantages: it allows the user to reuse existing C code and also does not require the knowledge of Verilog Programming Language Interface (PLI) or Verilog Procedural Interface (VPI) interfaces. It also provides an alternative (easier) way of calling some, but not all, PLI or VPI functions.

Functions implemented in C can be called from SystemVerilog using import "DPI" declarations. We will refer to these functions as imported tasks and functions. All imported tasks and functions must be declared.

Functions and tasks implemented in SystemVerilog and specified in export "DPI" declarations can be called from C. We will refer to these tasks and functions as exported tasks and functions.

An Example

Here an example is presented. A module called Bus contains two functions: write, which is a SystemVerilog function that is also exported to C, and a function called slave_write which is imported from C. Both functions return void.

SystemVerilog:

```
module Bus(input In1, output Out1);
  import "DPI" function void slave_write(input int address,
```

```
                                        input int data);
  export "DPI" function write;  // Note - not a function prototype

  // This SystemVerilog function could be called from C
  function void write(int address, int data);
    // Call C function
    slave_write(address, data); // Arguments passed by copy
  endfunction
  ...
endmodule
```

C:

```
#include "svdpi.h"
extern void write(int, int);    // Imported from SystemVerilog
void slave_write(const int I1, const int I2)
{
  buff[I1] = I2;
  ...
}
```

Note the following points:

- The C function slave_write is called inside the SystemVerilog function, the arguments being passed by value (we will see more detail about this later in the tutorial).
- The function imported from C has two inputs, which in C are declared as const. This is because they shouldn't be changed in the C function.

Both DPI imported and exported functions can be declared in any place where normal SystemVerilog functions can be (e.g. package, module, program, interface, constructs). Also all functions used in DPI complete their execution instantly (zero simulation time), just as normal SystemVerilog functions.

Examples of Importing C Functions

Here are some more examples of imported functions:

```
// User-defined function

import "DPI" function void AFunc();



// Standard C function

import "DPI" function chandle malloc(int size);



// Standard C function

import "DPI" function void free(chandle ptr);
```

```
// Open array of 8-bit

import "DPI" function void OpenF(logic [7:0] Arg[]);
```

chandle is a special SystemVerilog type that is used for passing C pointers as arguments to imported DPI functions.

Including Foreign Language Code

All SystemVerilog applications support integration of foreign language code in object code form. Compiled object code can be specified by one of the following two methods:

- by an entry in a bootstrap file; Its location is specified with one instance of the switch -sv_liblist pathname.
- by specifying the file with one instance of the switch -sv_lib pathname_without_extension (i.e. the filename without the platform specific extension).

Here is an example of a bootstrap file:

```
#!SV_LIBRARIES

myclibs/lib1

proj2/clibs/lib2
```

The first line must contain the string: #!SV_LIBRARIES. Then the following lines hold one and only one library location each. Comment lines can be inserted. A comment line start with a # and ends with a newline.

Here is an example of a switch list:

```
-sv_lib myclibs/lib1

-sv_lib proj2/clibs/lib2
```

The two files above are equivalent, if the pathname root has been set by the switch -sv_root to /home/user and the following shared object libraries are included:

```
/home/user/myclibs/lib1.so

/home/user/proj2/clibs/lib2.so
```

Binary and Source Compatibility

Binary compatibility means an application compiled for a given platform will work with every SystemVerilog simulator on that platform. Source-level compatibility means an application needs to be re-compiled for each SystemVerilog simulator and implementation-specific definitions will be required for the compilation.

Depending on the data types used for imported or exported functions, the C code can be binary-level or source-level compatible. Binary compatible are:

1. Applications that do not use SystemVerilog packed types.
2. Applications that do not mix SystemVerilog packed and unpacked types in the same data type.
3. Open arrays (see Argument Passing below) with both packed and unpacked parts.

Return Value and Argument Data Types

Result types of both imported and exported functions are restricted to small values. Small values include:

- void, byte, shortint, int, longint, real, shortreal, chandle, and string
- packed bit arrays up to 32 bits and all types that are eventually equivalent to packed bit arrays up to 32 bits.
- scalar values of type bit and logic

All SystemVerilog data types are allowed for formal arguments of imported functions.

Imported functions can have input, output and inout arguments. The formal input arguments cannot be modified. In the C code, they must have a const qualifier. Also, the initial values of output arguments are undetermined and implementation-dependent as far as the C function is concerned.

Argument passing

There is no difference in argument passing between calls from SystemVerilog to C and calls from C to SystemVerilog, apart from the fact that functions exported from SystemVerilog cannot have open arrays as arguments. Formal arguments in SystemVerilog can be specified as open arrays only in import declarations. This facilitates writing generalised C code that can handle SystemVerilog arrays of different sizes.

An open array is an array with the packed, unpacked or both dimensions left unspecified. This is indicated using the symbol [ ] for the open array dimensions.

The imported and exported functions' arguments can be passed in several modes, with certain limitations for each mode:

- Argument passing by value: here the following restrictions apply:
  - Only small values of formal input arguments are passed by value.
  - Function results restricted to small values are directly passed by value.
  - The user needs to provide the C-type equivalent to the SystemVerilog type of a formal argument (see below).
- Argument passing by reference (i.e. pointer or handle):
  - Formal arguments (input, output, inout), except for open arrays and small values of input arguments, are passed by direct reference (i.e. C pointer) and are directly accessible in C code.
  - Formal arguments declared in SystemVerilog as open arrays are always passed by a handle (type svOpenArrayHandle) and are accessible via library functions. This is independent of the direction of the SystemVerilog formal argument. Arguments passed by handle must have a const qualifier (the user cannot modify the contents of a handle).
  - If an argument of type T is passed by reference, the formal argument will be of type T*. Packed arrays can also be passed using generic pointers void* (typedef-ed accordingly to svBitPackedArrRef or svLogicPackedArrRef).

C vs SystemVerilog Data Types

A pair of matching type definitions is required to pass a value through DPI: the SystemVerilog definition and the C definition.

SystemVerilog types which are directly compatible with C types are presented in the following table:

SYSTEMVERILOG TYPE	C Type
byte	char
int	int
longint	long long
shortint	short int
real	double
shortreal	float
chandle	void*
string	char*

SystemVerilog and C types

There are SystemVerilog-specific types, including packed types (arrays, structures, unions), 2-state or 4-state, which have no natural correspondence in C. For these the designers can choose the layout and representation that best suits their simulation performance. The representation of data types such as packed bit and logic arrays are implementation-dependent, therefore applications using them are not binary-compatible (i.e. an application compiled for a given platform will not work with every SystemVerilog simulator on that platform).

Packed arrays are treated as one-dimensional, while the unpacked part of an array can have an arbitrary number of dimensions. Normalised ranges are used for accessing all arguments except open arrays. (Normalized ranges mean [n-1:0] indexing for the packed part (packed arrays are restricted to one dimension) and [0:n-1] indexing for a dimension in the unpacked part of an array.) The ranges for a formal argument specified as an open array, are those of the actual argument for a particular call.

If a packed part of an array has more than one dimension, it is transformed to a one-dimensional one, as well as normalised (e.g. packed array of range [L:R] is normalized as [abs(L-R):0], where index min(L,R) becomes the index 0 and index max(L,R) becomes the index abs(L-R)). For example:

```
logic [2:3][1:3][2:0] b [1:8] [63:0]
```

becomes

```
logic [17:0] b[0:7] [0:63]
```

after normalisation.

Enumerated names are not available on the C side of the DPI. enum types are represented as the types associated with them.

The C include files

The C-layer of the DPI provides two include files:

1. svdpi.h is implementation-independent and defines the canonical representation, all basic types, and all interface functions. Applications using only this include file are binary-compatible with all SystemVerilog simulators.
2. svdpi_src.h defines only the actual representation of 2-state and 4-state SystemVerilog packed arrays and hence, its contents are implementation-dependent. Applications that need to include this file are not binary-level compatible, they are source-level compatible

Argument Passing Example 1

This example includes a struct, a function imported from C and a SystemVerilog function exported to C. The struct uses three different types: byte, int (which are small values) and a packed 2-dimensional array. The SystemVerilog struct has to be re-defined in C. Byte and int are directly compatible with C, while the packed array is redefined using the macro SV_BIT_PACKED_ARRAY(width, name). SV_LOGIC_PACKED_ARRAY(width,name) and SV_BIT_PACKED_ARRAY(width,name) are C macros allowing variables to be declared to represent SystemVerilog packed arrays of type bit or logic respectively They are implementation specific, therefore source-compatible, and require "svdpi_src.h" to be included. The SystemVerilog function exported to C has an input of a type int (a small value), and a packed array as an output. The packed array will be passed as a pointer to void. (SvLogicPackedArrRef is a typdef for void *). The SystemVerilog function is called inside the C function, the first argument being passed by value, and the second by reference.

SystemVerilog:

```
typedef struct {
  byte A;
  bit [4:1][0:7] B;
  int C;
} ABC;

// Imported from C
import "DPI" function void C_Func(input ABC S);

// Exported to C
export "DPI" function SV_Func;

function void SV_Func(input int In,
                      output logic[15:0] Out);
  ...
endfunction
```

C:

```
#include "svdpi.h"
#include "svdpi_src.h"
typedef struct {
  char A;
  SV_BIT_PACKED_ARRAY(4*8, B); // Implementation specific
  int C;
} ABC;


SV_LOGIC_PACKED_ARRAY(64, Arr); // Implementation specific

// Imported from SystemVerilog
extern void SV_Func(const int, svLogicPackedArrRef);
void C_Func(const ABC *S)
```

```
       // A
struct is passed by reference
{
  ...
  // First argument passed by value, second by reference
  SV_Func(2, (svLogicPackedArrRef)&Arr);
}
```

Argument Passing Example 2

This is an example with a function imported from C having a 3-dimensional array as argument. The argument is passed by a svOpenArrayHandle handle and has a const qualifier. The function described in C uses several access functions:

```
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1,
```

int indx2, int indx3)

returns a pointer to the actual representation of 3-dimensional array of any type.

```
int svLow(const svOpenArrayHandle h, int d)
```

and

```
int svHigh(const svOpenArrayHandle h, int d)
```

are array querying functions, where h= handle to open array and d=dimension. If d = 0, then the query refers to the packed part (which is one-dimensional) of an array, and d> 0 refers to the unpacked part of an array.

SystemVerilog:

```
// 3-dimensional unsized unpacked array
import "DPI" function void MyFunc(input int i [][][]);

int Arr_8x4x16 [8:0][2:5][17:2];

int Arr_4x16x8 [3:0] [15:0][-1:-8];

MyFunc (Arr_8x4x16);

MyFunc (Arr_4x16x8);
```

C:

```
#include "svdpi.h"

void MyFunc(const svOpenArrayHandle h)
{
```

```
  int Value;
  int i, j, k;
  int lou1 = svLow(h, 1);
  int hiu1 = svHigh(h, 1);
  int lou2 = svLow(h, 2);
  int hiu2 = svHigh(h, 2);
  int lou3 = svLow(h, 3);
  int hiu3 = svHigh(h, 3);
  for (i = lou1; i <= hiu1; i++) {
    for (j = lou2; j <= hiu2; j++) {
      for (k = lou3; k<= hiu3; k++) {
       Value = *(int *)svGetArrElemPtr3(h, i, j, k);
        ...
      }
      ...
    }
  }
}
```

C Global Name Space

By default, the C linkage name of an imported or exported SystemVerilog function is the same as the SystemVerilog name. For example the following export declaration,

```
export "DPI" void function func;
```

Corresponds to a C function called func.

It is possible for there to be another SystemVerilog function with the same name, func. For example, another func could be declared in a separate module. To cater for this, and to provide a means to have different SystemVerilog and C function names for a DPI function, an optional C identifier can be defined in import "DPI" or export "DPI" declarations:

```
export "DPI" Cfunc = function func;
```

The function is called func in SystemVerilog and Cfunc in C.

Pure and Context Functions

It is possible to declare an imported function as pure to allow for more optimisations. This may result in improved simulation performance. There are some restrictions related to this, though. A function can be specified as pure only if:

- Its result depends only on the values of its inputs and has no side-effects.
- It is a non-void function with no output or inout arguments.
- It does not perform any file operations, read/write anything (including I/O, objects from the OS, from the program or other processes, etc.), access any persistent data (like global or static variables).

Here is an example of a pure function from the standard C math library:

```
import "DPI" pure function real sin(real);
```

An imported function that is intended to call exported functions or to access SystemVerilog data objects other then its actual arguments (e.g. via VPI or PLI calls) must be specified as context. If it is not, it can lead to unpredictable behaviour, even crash. Calling context functions will decrease simulation performance. All export functions are always context functions.

If VPI or PLI functions are called from within an imported function, the imported function must be flagged with the context qualifier. Not all VPI or PLI functions can be called in DPI context imported functions, e.g. activities associated with system tasks.

Importing and Exporting Tasks

C functions would usually be imported as SystemVerilog functions. However they can also be imported as SystemVerilog tasks:

```
import "DPI" task MyCTask(input int i, output int j);
```

Similarly, SystemVerilog tasks may be exported:

```
export "DPI" task MySVTask;
```

A SystemVerilog task does not have a return value and is called as a statement – in an initial or always block, for example. An important feature of tasks is that, unlike functions, they may consume simulation time, if they include one or more .timing controls. Now if an imported DPI task calls an exported DPI task that consumes simulation time, the imported task will consume time.

Only context imported tasks may call exported tasks:

```
import "DPI" context task MayDelay();
```

In SystemVerilog, tasks may be disabled, using a disable statement. When this happens, the task exits with its outputs undefined. In order to cater for an imported task (or the exported task it calls) being disabled, the C code must handle this possibility.

Consider an imported DPI task that calls an exported DPI task that does delay:

```
task Delay (output int t);
  #10 t = $stime;
endtask

export "DPI" task Delay;
import "DPI" context task DoesDelay(output int t);
```

The C code will look like this:

```
extern int Delay(int *t);

int DoesDelay (int *t)
{
  ...
```

```
  Delay();
  ...
}
```

Notice that the C functions DoesDelay and Delay return an int, even though they correspond to SystemVerilog tasks. The return value of Delay will be 0, unless the SystemVerilog task DoesDelay is disabled, in which case the C function Delay returns 1. This must be checked in DoesDelay, which must acknowledge that it has seen the disable and also return 1:

```
int DoesDelay (int *t)
{
  ...
  if ( Delay(t) ) {           // Was the task DoesDelay disabled?
    svAckDisabledState();
    return 1;
  }
  ...
  return 0;
}
```

Note that if Delay is disabled whilst DoesDelay is executing, Delay will return 0.

In summary, if a C function that implements an imported DPI task itself calls an exported DPI task, then

1. The imported task must be declared context.
2. The C function must return an int, with 1 indicating a disable.
3. The C function must check for a disable every time it calls an exported DPI task.

Interface Classes in SystemVerilog

In SystemVerilog, an interface class declares a number of method prototypes, data types and parameters which together specify how the classes that need those features can interact. The methods are declared as pure virtual functions - an interface class does not provide an implementation for the prototypes - this is done in a non-interface class (virtual or 'concrete') that *implements* one or more interface classes. In other words, an interface class has neither state nor implementation.

An interface class can be thought of as bringing together a number of qualities that may be needed for some aspect of a class' behaviour.

- An interface class establishes a protocol for how objects can use a certain behavioural feature and therefore interact.
- The methods can be implemented differently in a non-interface class which can implement one or more interfaces according to its requirements. This allows the developer to emulate multiple inheritance - getting base class behaviour from multiple sources.
- As a result of separate implementation, it is possible to develop a leaner and flexible class hierarchy which models objects more accurately and without the overhead of unecessary declarations resulting from single inheritance.
- Using interface classes is potentially more efficient than (for example) using abstract classes as there is no performance cost of method look up associated with virtual classes.
- Interface classes can have type parameters.

Example

In the following example, a channel provides connectivity and messaging APIs for components and the prototypes for the APIs are declared in separate interface classes.

The Messaging API deals with passing messages (transactions) between connected components and consist of both blocking and non-blocking methods:

```
interface class Messaging #(type T = logic);

  pure virtual task         put(T t);

  pure virtual task         get(output T t);

  pure virtual task         peek(output T t);

  pure virtual function bit  try_peek(output T t);

  pure virtual function bit  try_put(T t);

  pure virtual function bit  try_get(output T t);

endclass
```

The Connecting interface declares methods to connect and disconnect components, as well as to introspect and debug the connection (not shown here).

```
interface class Connecting #(type P = logic);

  pure virtual function void connect (P provider);

  pure virtual function int  connected_to();

  pure virtual function void disconnect(P other);

endclass
```

The concept of a channel to pass transactions between components and implement a basic TLM protocol would need to implement these two interface classes. Here, we have a base virtual class to declare the root of the channel based class hierarchy. We have added the notion that the channel will connect objects derived from a base Component type and will pass messages which will also be derived from a base Transaction type

```
virtual class TLM_channel_base #(type Tr = Transaction, C = Component)

  implements Messaging #(Tr), Connecting #(C);



  protected event e_get, e_put;
```

```
  protected int bound;


  pure virtual protected function T pop();

  pure virtual protected function void push(T t);


endclass : TLM_channel_base
```

By using interface classes to specify the API, we have not achieved any functionality that couldn't be done with single class inheritance alone, but what we have done is thought a bit more about how a certain API can be contained to provide a distinct feature. In this example, there is no need to provide a class hierarchical relationship between Messaging and Connecting, which might otherwise be necessary.
The channel root class, above, commits to providing an implementation of the APIs (delegated to a non-virtual channel). It also adds some channel specific (as opposed to messaging related) methods, *pop* and *push*. In the channel implementation considered here, these are the methods that actually do the insertion and retrieval of messages in the channel. The TLM channel declaration, below, allows the user to reuse the same TLM channel base class and API, but with different message and component types.

```
virtual class TLM_Channel extends TLM_channel_base

  #(.Tr(Transaction), .C(Component));


  local Tr fifo[$];

  local C providers[$];

  local int connected;


  // Messaging functionality

  extern function      new(int _bound = 0);

  extern task          put(Tr t);

  extern function bit   try_put(Tr t);

  extern task          get(output Tr t);

  extern function bit   try_get(output Tr t);

  extern task          peek(output Tr t);
```

```
  extern function bit   try_peek(output Tr t);


  extern function Tr    pop();
  extern function void  push(Tr t);
  extern local function bit  empty();
  extern local function bit  full();
  extern local task     await_not_full();
  extern local task     await_not_empty();


  // Connection functionality
  // Connect a provider
  extern virtual function void connect (C provider);
  extern virtual function int  connected_to();
  extern virtual function void disconnect(C other);


endclass
```

So now we have a more specialised specification for the channel which can be used to pass transactions between components. The user then extends from this base class to declare a non-virtual channel for their environments.

```
class APB_component extends Component;

  ...

endclass


typedef TLM_Channel #(APB_transaction, APB_component) APB_channel;
```

A working example of an interface class can be found on **EDA Playground**.

When to Use Interfaces, Inheritance or Aggregationa

An extended class has an *is-a* relationship with the super class and *can-do* many things as well as *has* qualities represented by other classes.

- *is-a* implies *inheritance*
- *can-do* implies *interface*
- *has* implies *aggregation*

References

1. Interface vs Abstract Classes
2. "SystemVerilog Interface Classes – More Useful Than You Thought", Stan Sokorac, ARM
3. IEEE Std 1800-2012, section 8.26, p157
4. Universal Verification Methodology(UVM) 1.2 Class Reference

SystemVerilog Abstract Classes

The name of a class in SystemVerilog declares its type, so that when an object is constructed from the class with **new**, the methods and members of the class then determine how the object interacts with other objects in the test environment. Even though 2 classes may be structurally the same, their compatibility will be determined by the type (class) name. In SystemVerilog, a sub-class can be declared that **extends** a **super** class. This means that the sub-class is a syb-type or specialisation and *inherits* the super class' methods and members as if they were declared in the sub-class itself.

Just using these ideas allows one to model the relationship between objects: for instance, given a declaration for a transaction, one may wish to sub-type the declaration and create a more specialised transaction for an application with a particular address map:

```
class Bus_trans;



  static int next_ID;

  const int ID;



  rand T_dir dir;

  rand T_addr addr;

  rand T_data data;



  function new;

    ID = next_ID++;

  endfunction: new
```

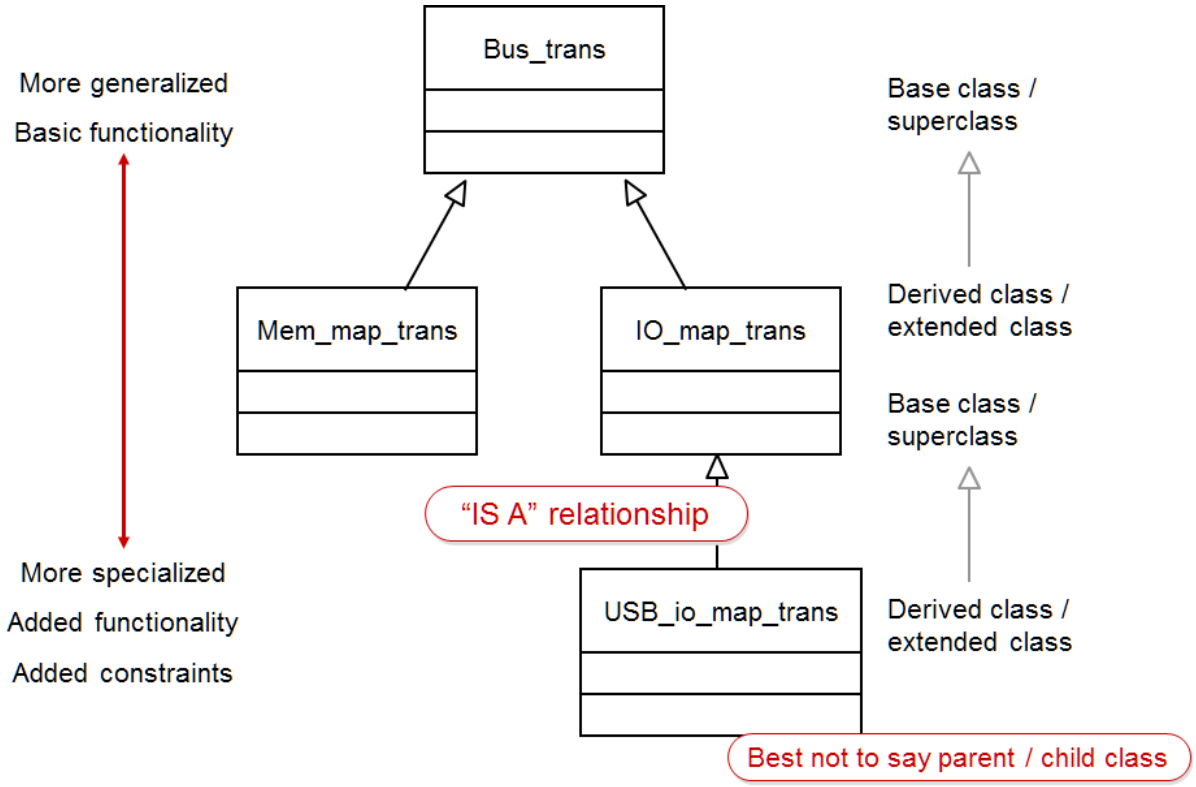
```
  function void print; ...

endclass: Bus_trans
```

This general purpose transaction can be extended:

```
class Mem_map_trans extends Bus_trans;

  rand enum { ROM, RAM, IO } area;


  constraint address_map {

    area == ROM -> addr inside {['h0000:'h7FFF]};

    area == RAM -> addr inside {['h8000:'hDFFF]};

    area == IO -> addr inside {['hFF80:'hFFFF]};

  }


  constraint ROM_read_only { area == ROM -> dir == dir_Rd; }

  constraint IO_byte_wide { area == IO -> data[15:8] == 0; }

  constraint area_choice {

    area dist { ROM := 70, RAM := 20, IO := 10 };

  }


  function string psprint();

    $sformat(psprint, "%s cycle: %s",

    area.name(), super.psprint() );

  endfunction : psprint


  function new (Stream owner = null);

    super.new(owner);

  endfunction : new
```

```
endclass
```

The sub-class (*Mem_map_trans*) can access fields and methods declared in the super (also referred to as **base**) class while adding the abstract control knob, *area*, to represent the address map. The base class can be repeatedly sub-typed, as can the sub-classes so that a **class hierarchy** can be developed with relationships that can be represented in a UML diagram.



The point is that the general class declaration at the root of the hierarchy contains class member declarations for all the sub typed classes. *As well as* method declarations like print and copy. When developing a class hierarchy, one would like the root of the hierarchy to represent a blue-print for sub classes. We can have the concept of a transaction (the abstract, blue-print) and from that extend and build an actual transaction, for instance, the memory mapped transaction. The concept of an abstract class - declared in SystemVerilog as a **virtual** class, allows us to model this type of relationship. So we can just change the declaration for the transaction class:

```
virtual class Transaction;

...

endclass
```

And that would work. But to be useful, we need to know several features of an abstract class:

1. The virtual class can be considered to be an incomplete declaration, because it might be known that certain methods will be needed in the sub classes - it may not be possible to determine *how* those methods will work. Thus the virtual class might declare some **pure virtual** methods.

```
2.  virtual class Transaction;
```

```
3.    ...
4.    pure virtual function string convert2string();
5.    ...
6.  endclass: Transaction
```

7. The virtual class is a template and cannot be instanced. While it is possible (and useful) to declare a variable of the virtual class type, it is not allowed to instantiate an object using **new()**. The reason for this follows on from the previous point - the pure virtual methods declare the prototype for a method that *must* be implemented in a 'real' or concrete class.

```
8.  class Basic_transaction extends Transaction;
9.     rand addr_t address;
10.    rand data_t data;
11.    rand dir_t  dir;
12.
13.    function new(string name = "");
14.       super.new(name);
15.    endfunction
16.
17.    virtual function string convert2string();
18.       return $sformatf("address   = %9x data = %9x dir = %0s", address, data,
    dir.name());
19.    endfunction
20.    ...
21. endclass: Basic_transaction
```

22. A variable (or handle) of the virtual class type can be assigned a reference to an object of any of its sub-types. This is upcasting and allows code to be written using a common base class handle and reused with different actual object references. Downcasting is also allowed in SystemVerilog using **$cast()**, but certain checks are applied before the cast takes place.

```
23. ...
24. Transaction      t;
```

```
25. Basic_transaction b;

26.

27. b = new();    // allowed/necessary, but cannot construct t

28. ...

29. t = b;        // upcast: t holds a reference to an object of type Basic_transaction

30. ...

31. $cast(b, t);  // downcast - with checks
```

The virtual class can declare all the method types that can be used in a concrete class as well as data members, but can also declare a pure virtual method, which can only be declared in a virtual class.