Learning system verilog

- System verilog is an extension of verilog language
- Developed to make verification of designs easier
- Supports OOP features
- Also has some improvements for synthesis, but mostly intended to improve the verification process

Index

- Learning system verilog
- Index
- Data types
  - Literals
  - Arrays
  - Type casting
  - String
  - Enumeration
  - Arrays
    - Static array
    - Dynamic array
    - Associative array
    - Queues
    - Array methods
      - Ordering methods
      - Reduction methods
      - Locator methods
      - Index querying
  - Structures
  - Typedef
  - Operators
- Control Flow
  - Loops

    - Break, continue
    - If and else
    - Case
        - Case inside
        - Unique case
        - Priority case
    - Events
    - Function
    - Tasks
- Processes
- Inter-process communication
    - Events
    - Semaphore
    - Mailboxes
- Interfaces
    - Virtual interface
    - Clocking blocks
- Classes
    - Inheritance
    - Polymorphism
    - Virtual classes
    - Static members
    - Copy
    - Parameterized classes
    - Virtual interface
    - Extra info
- Constraints and randomization
    - Random variables
    - Constraint types
        - Simple expressions
        - Inside operator
        - Weighted distribution
        - Implications
        - foreach
        - Array reduction
    - Solve before
    - Constraint mode
    - At randomization stage

        - In-line constratints
        - `pre_randomize()` and `post_randomize()`
        - Disabling randomization
        - Seeding
        - randomize function
    - randcase
    - Inheritance and overriding
- Functional coverage
  - Covergroup
  - Coverpoint
  - Cross coverage
  - Bins
- Assertions
    - Immediate assertion
    - Sequences
        - Cycle delay
        - Timing window
        - Implication
        - Repetition
    - Built-in functions
    - Property
    - Providing trigger event for concurrent accesses
    - Error classes
- Miscellaneous
    - System verilog scheduler
    - Program block
- References

Data types

- Verilog had only `reg`, `wire`, `integer`, ... This was insufficient for performing more complicated verification. System verilog has the following types:

Data type	x and z states	Bits	signed/unsigned	C equivalent
reg	y	>= 1	u	

Data type	x and z states	Bits	signed/unsigned	C equivalent
wire	y	>= 1	u	
integer	y	32	u	
real				double
time				
realtime				double
logic	y	>= 1	u	
bit	n	>= 1	u	
byte	n	8	s	char
shortint	n	16	s	short int
int	n	32	s	int
longint	n	64	s	long int
shortreal				float

- `logic` and `bit` are the most commonly used types.
- `logic` has `z` and `x` states, `bit` does not.
- When `z` or `x` is converted to `bit`, they become `0`

Literals

- To define real values,
  - `real pi  = 3.14;` : 3.14
  - `real var = 5e-6;` : 5 * 10 ^ -6
- Strings
  - `string str = "Test";`

Arrays

- System verilog supports multi dimensional arrays

```
int test_array[2][3];   // 2 x 3 array
test_array = {{1, 2, 3}, {4, 5, 6}};
```

Type casting

- 2 ways
  - Using casting

    ```
    int'(2.5 * 3.5);    // Performs rounding
    ```

  - Using system tasks

    ```
    $rtoi(2.54);        // Performs truncation
    ```

String

- We can use actual dynamic strings in system verilog.

```
string str = "Hello!";
```

String also has various functions. They are

- `str.len()`
- `str.putc(idx, char)`
- `str.getc(idx)`
- `str.tolower()`
- `str.toupper()`
- `str.compare(str2)`
- `str.substr(i, j)`
- Conversion methods
  - `str.atoi()`
  - `str.atohex()`
  - `str.atooct()`
  - `str.atobin()`
  - `str.atoreal()`
  - `str.itoa(inp)`
  - `str.hextoa(inp)`
  - `str.octtoa(inp)`

- `str.bintoa(inp)`
- `str.realtoa(inp)`

Refer Chip verify tutorial on strings

Enumeration

Like enum in C/C++.

```
// Integer type
enum {
  RED,
  BLUE,
  GREEN
} colors;

// Bit type
enum bit[1:0] {
  s_IDLE,
  s_ACTIVE,
  s_DISABLED
} states;
```

Can also use `typedef` to create multiple objects of the enum type

```
// s_IDLE : 0, s_ACTIVE : 1, s_DISABLED : 2
typedef enum bit[1:0] {
  s_IDLE,
  s_ACTIVE,
  s_DISABLED
} states;
```

We can define the start of an enum

```
// s_IDLE : 1, s_ACTIVE : 2, s_DISABLED : 3
enum bit[1:0] {
  s_IDLE = 1,
  s_ACTIVE,
  s_DISABLED
} states;
```

Or, we can even change the number used mid-sequence

```
// s_IDLE : 0, s_ACTIVE : 2, s_DISABLED : 3
enum bit[1:0] {
  s_IDLE,
  s_ACTIVE = 2,
  s_DISABLED
} states;
```

Enumeration methods :

- `first()`
- `last()`
- `next(offset)`
- `prev(offset)`
- `num()`
- `name()`

Arrays

Arrays in system verilog can have the types

- Static array
- Dynamic array
- Associative array
- Queues

Static array

Array size is known at compilation.

```
bit[7:0] arr;
```

Further classified into

- Packed (or vector)
  - `bit[7:0]  m_data;`
- Unpacked
  - `bit       m_data[7:0];`

Dynamic array

Size is not known at compile time.

```
// Define a dynamic array
int m_data[];

// Initialize memory
m_data = new[5];

// Print size of array
$display(m_data.size());

// Resize the array to total 10 size (preserve existing values)
m_data = new[10](m_data);

// Delete all entries and de-allocate
m_data.delete();
```

Associative array

Arrays indexed by a key

```
int m_data[string];

m_data["test"] = 0;

m_data = `{
    "ashwin" : 1,
    "rakesh" : 10
  };
```

Queues

We can push back and pop from queues

```
// Define queue
int m_data[$];

// Push data to end of queue
m_data.push_back(200);

// Push data to start of queue
m_data.push_front(10);

// View last element in queue
$display(m_data[$]);

// Pop from start of queue
$display(m_data.pop_front());

// Pop from end of queue
$display(m_data.pop_back());
```

Array methods

Ordering methods

- `arr.reverse()`
- `arr.sort()`
  - Ascending order
- `arr.rsort()`
  - Descending order
- `arr.shuffle()`

Sort functions can use `with` clause like below

```
class packet;
  byte id;
  ...
endclass

packet list[$];

intial begin
  ...

  list.sort with (item.id);
end
```

Reduction methods

- `arr.sum()`
- `arr.product()`
- `arr.and()`
- `arr.or()`
- `arr.xor()`

Locator methods

- `arr.min()`
- `arr.max()`
- `arr.unique()`
- `arr.unique_index()`
- `arr.find with (item > 3)`
- `arr.find_first with (item > 3)`
- `arr.find_last with (item > 3)`

All these functions return lists. Functions that depend on a value like `unique`, `min`, `max` can always use the `with` clause to be used on classes and structs

Index querying

- `arr.find_index with (item > 3)`
- `arr.find_first_index with (item > 3)`
- `arr.find_last_index with (item > 3)`

For array functions, refer : https://verificationguide.com/systemverilog/systemverilog-array-methods/

Structures

- Similar to C struct

```
typedef struct {
    int count_1, count_2;
    string name;
    bit flag;
} counter;
```

```
typedef struct {
    int A, B, C;
} integers;
```

- To create struct objects,

```
counter count_a;
```

- To assign values,

```
// By order of original declaration
count_a = '{0, 0, "count_a", 1};

// By using name
counter count_b = '{count_1:0, count_2:10, flag:1'b1, name:"count_a"};

// Assign all to zero
counter count_c = '{default:0};

// Assign values of type int to 0
counter count_d = '{int:0, flag:1'b1, string:"str"};
```

- They can also use this syntax

```
struct {
    int A, B, C;
} test = {3{0}};
```

- We can define a single structure with typedef using

```
struct {
  int A, B, C;
} obj;
```

- We can create packed structs that can be accessed like a vector

```
typedef struct packed {
  bit[1:0]  state;
  bit[2:0]  count;
  bit       flag;
} ctrl_reg;

ctrl_reg reg1 = 6'b010001;
// state  = 01, count = 000, flag = 1
```

Typedef

Similar to typedef in C

```
typedef bit[7:0] u_byte;
typedef bit[7:0] mem_block[127:0];
```

Operators

`==` and `!=` are used for 2-state values. If there is `z` or `x`, result is `x`.

`===` and `!==` check for `z` and `x` too.

`=?=` and `!?=` use `z` and `x` in the RHS as wildcards. `z` and `x` in the LHS can cause an `x` result.

Control Flow

Loops

- `for`

```
int i;
...
for(i = 0; i < 10; i = i + 1) begin
  ...
end
```

- `while`

```
int i = 0;
...
while(i < 10) begin
  ...
end
```

- `do while`

```
int i = 0;
...
do begin
  ...
end while(i < 10);
```

- `foreach` : Meant for arrays and iterables

```
int l[10];
...
foreach(l[i]) begin
  ...
  $display("%d", l[i]);
end
```

- `forever` : Repeat ... forever!

```
forever begin
  ...
end
```

- `repeat` : Repeat n number of times

```
repeat(5) begin
...
end
```

Break, continue

- Break breaks the loop and exits
- Continue goes to the next iteration

```
forever begin
  ...
  if(i == 0)
    break;
  ...
end
```

```
repeat(10) begin
  ...
  if(i == 0)
    continue;
  ...
end
```

If and else

If and else in system verilog works like normal verilog. Special if statements exist that add runtime checks :

- `priority if` : Throws an error if
  - None of the statements trigger for some input

```
priority if(i == 3) begin
  ...
end else if(i < 5) begin
  ...
end
// Input  | Error message
// i >= 5 : No condition is true
// Others : No error
```

- `unique if` : Throws an error if
  - Any 2 statements trigger for the same input.
  - None of the statements trigger for some input.

```
unique if(i == 3) begin
  ...
end else if(i < 5) begin
  ...
end
// Input  | Error message
// i =  3 : The if/case statement is not unique
// i >= 5 : No condition is true
// Others : No error
```

- No condition is true error can be elimintaed using `else` statement at the end

So, `priority if` is a version of `unique if` without check for duplicate triggers.

Case

Case statements are similar to normal verilog

```
case(statement)
case_1: begin
        ...
        end

case_2: begin
        ...
        end

        ...

default:begin
        ...
        end
endcase
```

Case inside

We can also define ranges and wildcards in case statements using `case inside` . Example :

```
bit[3:0] i;

case(i) inside
  4'b00?? : begin         // Matches first 2 bits = 0
            ...
            end
  [4'b0100 : 4'b1000] :   // Matches 4-8 (inclusive)
            begin
            ...
            end
  default : begin         // Match anything else
            ...
            end
endcase
```

We also have `casez` and `casex` like verilog.

We can also use `unique` and `priority` cases.

Unique case

```
unique case(i) inside
  4'b00?? : begin         // Matches first 2 bits = 0
            ...
            end
  [4'b0010 : 4'b1000] :   // Matches 2-8 (inclusive)
            begin
            ...
            end
endcase
```

Will throw errors for

- "Case statemetns are not unique" for `2` , `3`
- "No condition is true" for `9 - 15`

Priority case

If above code was a priority case, errors would happen for `9 - 15` for "No condition is true", but runs the first case statement (higher priority) for inputs `2` and `3`

`unique0` is a variant of `unique` modifier that does not throw an error for "No condition is true".

Events

Used to synchronize between two or more processes. One process will trigger it and the other will wait for the event.

Events are created using `event` keyword

```
event event_a;
event event_b = event_a;  // Alias to event_a
event event_c = null;
```

Trigger events using `->event_name` or `->>event_name` operators

```
->event_a;

// OR

->>event_a;
```

Wait on events using `@(event_name)` or `wait(event_name.triggered)`

```
@(event_a)
...

// OR

wait(event_a.triggered)
...
```

If an `->` and `@` statement are encountered on same timestep, a race condition occurs, so the `@` might never be triggered. Here, we use the `wait(event_name.triggered)` functionality.

Function

Functions cannot consume simulation time, so they cannot use `#`, `@`, `fork ... join`, `wait` or call tasks.

They can have zero or more inputs but only one output

There are 2 styles for defining inputs :

```
// Inside the ()
function bit[3:0] sum(
  bit[2:0] a,
  bit[2:0] b
);
  ...
endfunction

// Or, outside it, with input keyword
function bit[3:0] sum;
  input bit[2:0] a;
  input bit[2:0] b;
  ...
endfunction
```

Return value can also be defined in 2 ways

```
// Using return keyword
function static integer fun();
  ...
  return a;
endfunction

// Or, by assigning to the function name
function static integer fun();
  ...
  fun = a;
  ...
endfunction
```

In the 1st format, execution stops on `return` statement, but it continues till the end in the 2nd statement

```
// Invoking a function
$display(sum(r_reg1, r_reg2));
```

Return type can also be `void` .

`static` v/s `automatic` functions : Static tasks and functions have only one instance which is invoked again and again. For automatic tasks and functions, a new one is invoked every time.

```
// Static function (default is also static)
function static integer fun();
  integer a = -1;
  a = a + 1;
  $display(a);
endfunction


// Automatic function
function automatic integer fun();
  integer a = -1;
  a = a + 1;
  $display(a);
endfunction


// Call the function 10 times
initial begin
  repeat(10) fun();
end
```

- The first function will return `0, 1, 2, ... 9`

- The second function will return `0, 0, 0, ... 0`

We can also pass values by reference to (only) automatic functions using `ref` keyword.

```
function void increment(
  ref int a
);
  a = a + 1;
endfunction
```

There can also be default values for the functions

```
// Default value of argument is 1
function int increment(
  int inc = 1
);
  int i = 0;
  i = i + inc;
  return i;
endfunction

// Call using default argument
initial begin
  repeat(10) #1 $display(increment());
end
// Call using argument value 2
initial begin
  repeat(10) #1 $display(increment(2));
end
```

Functions can call other functions

Tasks

Tasks, unlike functions, can consume simulation time and can also have multiple outputs.

```
// Style 1
task task_1;
  input   int i_port;
  ...
  output bit[3:0] o_port;

  begin
    ...
  end
endtask

// Style 2
task task_2 (
  input   int i_port,
  ...
  output bit[3:0] o_port
);
  begin
    ...
  end
endtask

// Calling a task
task_1(
  .i_port(w_porta),
  ...
  .o_port(w_portb)
);
```

The work happens inside the `begin ... end` block. We can have multiple named `begin ... end` blocks too. The outputs are assigned their final values after everything inside `begin ... end` and not continuously driven. Task calls are blocking.

Tasks can also be `static` or `automatic`

Recursion is supported. Tasks can call other functions and tasks

Processes

- `fork ... join` is used for launching multiple processes.

```
initial begin
  fork
    // Thread 1 : Generate counter
    repeat(10) begin
      #1 count = count + 1;
    end

    // Thread 2 : Print counter and val
    repeat(10) #1 $display("%d %d", count, val);

    // Thread 3 : Control val
    begin
      val = 1;
      #5 val = 0;
    end
  join

  $display("Completed!");
end
```

All 3 threads execute in parallel. The join statement ends only when all 3 threads have completed execution.

There are different types of join.=

- `join`
  - All threads must execute
  - Prints all values till 10 and then `completed`
- `join_any`
  - Any one thread must exit. Other threads continue executing.
  - Prints till `c = 4` then `completed` then prints the rest
- `join_none`
  - Launches the threads and joins immediately. Other threads continue executing
  - Prints `completed` then prints all the values till 10
- We can kill threads already running (from `join_any` and `join_none`) by using `disable fork` statement.

```
initial begin
  fork
    // Thread 1 : Generate counter
    repeat(10) begin
      #1 count = count + 1;
    end

    // Thread 2 : Print counter and val
    repeat(10) #1 $display("%d %d", count, val);

    // Thread 3 : Control val
    begin
      val = 1;
      #5 val = 0;
    end
  join_any

  $display("Completed!");

  #2 disable fork;
end
```

Will only print count till (and including) 6.

Similarly, we can also wait for all processes to complete

```
initial begin
  fork
    // Thread 1 : Generate counter
    repeat(10) begin
      #1 count = count + 1;
    end

    // Thread 2 : Print counter and val
    repeat(10) #1 $display("%d %d", count, val);

    // Thread 3 : Control val
    begin
      val = 1;
      #5 val = 0;
    end
  join_any

  $display("Completed!");

  wait fork;

  $display("exit");
end
```

Will print `Completed!` after 4 count and `exit` after 10 counts

Inter-process communication

Inter-process communication is done by 3 mechanism :

Events

Definition :

```
event event_name;
```

Triggering :

```
->event_name;
// OR
->>event_name;
```

Waiting

```
@(event_name);
// OR
wait(event_name.triggered);
```

System verilog events can also be passed as arguments to tasks

Events can be merged using `=` assignment

```
event evt_a, evt_b;
...

evt_b = evt_a;
...

->evt_a; // Triggers both a and b
...

->evt_b; // Triggers both a and b
```

Semaphore

Semaphores have counts and we can put objects to the semaphore and get from it. It will block the get request till the count is >= requested value.

Definition and initialization (argument to new is initial number of keys)

```
semaphore sem;
sem = new(5);
// OR
semaphore sem = new(5);
```

Getting (try to get 2 keys). Is a task that blocks till we get the specified number

```
sem.get(2);
```

Try getting it. Is a function (so, is non-blocking) that returns if we were able to get it

```
// Returns 0 or 1
$display(sem.try_get(2));
```

Putting (try to put back 2 keys). Is a function

```
sem.put(2);
```

You can put back more keys than you took. Its perfectly legal.

Mailboxes

We can send messages using mailboxes.

Functions :

- function `new(int size)`
  - Create mailbox with given size
- function `num()`
  - Return number of messages in mailbox
- task `put(msg)`
  - Send an object as message. Blocks if mailbox is full
- task `get(ref msg)`

    - Get an object. Blocks if mailbox is empty
- task `peek(reg msg)`
    - Gets last object without consuming it. Blocks if empty
- function `try_put(msg)`
    - Non-blocking put. Returns if successfull
- function `try_get(ref msg)`
    - Non-blocking get attempy. Returns if successfull
- function `try_peek(ref msg)`
    - Non-blocking function that returns mailbox entry if mailbox is not empty, without consuming it

Mailboxes are typeless and can transport any type. However, this can cause bugs. We can restrict mailboxes to types using parameters

```
mailbox #(int) mbox;
```

Interfaces

Way to encapsulate related signals together into a block

```
interface mybus(input clk);
  logic [31:0]  addr;
  logic [31:0]  wdata;
  logic [31:0]  rdata;
  logic         enable;
  logic         write;
endinterface
```

Interfaces only define the signals related, but not their directions. Interfaces can have multiple `modports` which define the directions with **respect to the module we are connecting to**

Interfaces can also be parameterized

Simple example using parameterization and modport :

```
// Data interface
interface data_if #(
  parameter p_WIDTH = 4   // Paramter for data width
) (
  input logic clk         // Clock signal
);
  logic[p_WIDTH - 1:0] data;

  // Modport for generator and consumer of data
  modport gen(
    input clk,
    output data
  );
  modport cons(
    input clk,
    input data
  );
endinterface

// Module to generate data
module generator(
  data_if.gen data
);
  always @(posedge data.clk)
    data.data = $random;
endmodule

// Module to consume data
module consumer(
  data_if.cons data
);
  always @(posedge data.clk)
    $display("%h", data.data);
endmodule

// Toplevel test module
module test;
  // Define clock signal
  reg r_clk = 0;
  always #1 r_clk = ~r_clk;

  // Initialize interface
  data_if #(8) data(r_clk);

  // Define generator
  generator gen(data.gen);
  // Define consumer
  consumer cons(data.cons);

  // Finish after 100 timesteps
```

```
  initial #100 $finish;
endmodule
```

Virtual interface

Interfaces are allocated statically, but classes are allocated dynamically. So, classes cannot have interfaces. We use `virtual` keyword to store reference to an actual interface.

```
class test;
  virtual bus_if interf;
  ...
  function new(virtual bus_if if_inp);
    interf = if_inp;
  endfunction
  ...
endclass
```

Clocking blocks

Clocking blocks are made to make it easier to drive signals relative to clock signals in test benches. We can set the signals and they will be actually driven relative to the clock.

The directions are relative to how they will be accessed. `output` ports are driven in sync with the clock and `input` ports are sampled by the clock.

```
// Defining clock
logic clk       = 0;
always #10 clk  = ~clk;

// Defining data lines
logic[3:0] data_1 = 0;
logic[3:0] data_2 = 0;

// Define clocking block
clocking clock_block @(posedge clk);
  output        data_1;
  input         data_2;
endclocking

initial begin
  #1  clock_block.data_1  <= 1;
  $display("Data 1 : %d", data_1);                // Output : 0

  #10;
  $display("Data 1 : %d", data_1);                // Output : 1

  data_2                  <= 1;

  #1;
  $display("Data 2 : %d", data_2);                // Output : 1
  $display("Clocked 2 : %d", clock_block.data_2); // Output : 0

  #20;
  $display("Clocked 2 : %d", clock_block.data_2); // Output : 1

  $finish;
end
```

Here, data_1 waits till the first `posedge` of `clk` at 10 timesteps to be driven to 1. Similarly, data_2 was ready at time 11, but was sampled only at timestep 30.

We can also set delays and timing information on when to drive output signals and when to sample input signals.

- Input skew : `x` Time before sampling event
- Output skew : `x` Time after sampling event
- `##` operator can be used to delay event by certain number of clock cycles

```
clocking clocking_name @(
  posedge clk           // Event or signal transition
);
  default input #1 output #3;
  input       data_1;   // Sampled 1 before trigger
  input #2    data_2;   // Sampled 2 before trigger
  output      data_3;   // Driven 3 after trigger
  output #2   data_4'   // Driven 2 after trigger
endclocking
```

- Clocking block name can be used like an event : `@(clocking_name)`
- Clocking blocks can be used within an interface too
- A clocking block in a scope can be declared as `default clocking ... endclocking` to define it as the default clocking block. Then, all sequences and operators like `##` in that scope will be with reference to this block unless specified otherwise.

Classes

- Encapsulates data, functions and tasks
- Also supports inheritance
- No function overloading, though `:(`

Example :

```
class test;
  int val;
  bit[3:0] val2;

  // new function should not have return value
  function new();
    val = 10;
    val2 = 10;
  endfunction

  task displ();
    $display("val : %d, val2 : %h", val, val2);
  endtask
endclass
```

Data members can be declared as `const` , meaning they can only be assigned in the `new()` function once.

Inheritance

```
class test2 extends test;
  int val3;

  function new(int init_val);
    super.new();
    val3 = init_val;
  endfunction

  task displ();
    $display("val : %d, val2 : %h, val3 : %d", val, val2, val3);
  endtask
endclass
```

`this` and `super` keyword work as expected, and can be used to refer to data members and functions

By default, all members are public and inherited. We can use the `local` access specifier which is essentially the equivalent of `private` in C++. We also have `protected` access specifier.

Polymorphism

Polymorphism is possible. Example using above classes :

```
test2 derv_obj = new(100);
test  base_obj;

base_obj = derv_obj;

base_obj.displ();   // Calls base class version
derv_obj.displ();   // Calls derived class version
```

We can define `test::displ()` as `virtual task` so that `base_obj.displ()` will refer to `derv_obj.displ()` .

Virtual classes

Virtual classes also exist. These classes cannot be instantiated and serve as base classes.

```
// No objects can be created.
virtual class base_cls;
  ...

  // This function must be implemented, else an error is thrown
  pure virtual function int fun(...);
endclass

class derv_cls extends derv_cls;
  ...

  // Overloading the pure virtual function
  function int fun(...);
    ...
  endfunction
endclass
```

Static members

We can also have `static members` and functions which are shared between objects. They can be accessed outside using `class_name::function(...)` or `class_name::data` syntax.

```
class class_name;
  ...
  static int val = 0;

  static function disp_val();
    $display(val);
  endfunction

endclass

...

class_name::val = 100;
class_name::disp_val;
```

Copy

```
// obj2 is a reference
obj2 = obj;

// obj3 is a copy, but objects inside obj are linked
obj3 = new obj;

// obj4 is completely free copy. copy is a new function we need to write
obj.copy(obj4);
```

Parameterized classes

Like template classes in C++.

We can parameterize values and also types

```
class test_cls #(
  parameter p_WIDTH = 8,
  type      t       = string
);
  ...
  t val;
  ...
endclass
```

Virtual interface

- Interfaces and signals are static. Classes are dynamic
- To use interfaces in classes we need to use `virtual` keyword.

```
class test_cls;
  // Virtual interface
  virtual amba_if bus;
  ...

  // Attach the interface at initialization
  function new(virtual amba_if bus_in);e
    bus = bus_in;
    ...
  endfunction
endclass

...

program main;
  ...
  // The actual interface being passed
  amba_bus bus1(clk);
  ...
  initial begin
    // Pass the interface to constructor
    test_cls obj = new(bus1);
    ...
  end
endprogram
```

Extra info

- If we need to use a class before defining it, we need to typedef it. This could be unavoidable if two classes store members of each other.

Example

```
typedef class cls2;

class cls1;
  cls2 obj;
  ...
endclass

class cls2;
  cls1 obj;
  ...
endclass
```

- We can use `extern` keyword to declare a method inside a class and implement it outside using `::` operator.

- We can randomize objects using `.randomize()` function. This takes into account the constrints defined on the class. The values to be randomized need to be defined using `rand` keyword. More in constraints part

Constraints and randomization

Class values can be randomly assigned. However, not all values might be valid. So, we need to assign constraints to the random values that can be generated. This is done using `constraint` blocks inside classes.

Random variables

We need to first define which variables are random. We use `rand` and `randc` keywords for this. `rand` says that the member is random and `randc` assigns that values must be cycled, meaning a value is repeated only if all other values have been exhausted.

We then define constraint blocks which define which values are valid. A valid object must satisfy all the statemetns in all the constraint blocks.

Example

```
class packet;
  rand  int      size;
  randc bit[3:0] flag;
  rand  byte     list[$];

  // Flag must take only 0, 1, 4, 5, 8, 9
  constraint flag_constr {
    flag[1] == 0;
    flag    < 10;
  }

  // External constraint
  constraint size_constr;
endclass

// Size must be less than 10
constraint packet::size_constr {
  list.size() < 10;
  list.size() == size;
}

packet pkt = new();

initial begin
  repeat(10) begin
    pkt.randomize();
    $display("%p", pkt);
  end
end
```

Constraint types

These are the constraints that can be defined in theconstraint blocks.

Simple expressions

Simple relational operators like `==` , `<` , `>` , `<=` , `>=`

These statements themselves can use indexing, arithmentic operations, functions, etc

We can also combine and modify statements using `||` , `&&` and `!`

Inside operator

We can specify a lower and upper limit to the values or from a list. Both limits are inclusive.

```
// Length can be 2, 3, ... 9, 10
constraint len_constr{
  list.size() inside {[2:10]};
}

// Flag can be 0, 2, 5
constraint flag_constr{
  flag inside {0, 2, 5};
}
```

We can also combine inside statements using `||` `&&` and `!`

```
class packet;
  rand  int      size;
  randc bit[3:0] flag;
  rand  byte     list[$];

  // Flag must take only 0, 1, 4, 5, 8, 9
  constraint flag_constr {
    !(flag inside {1, 2, 3});
  }

  // Size must be less than 10
  constraint size_constr {
    list.size() inside {[3:8]} || list.size() inside {[15:17]};
    list.size() == size;
  }
endclass
```

Weighted distribution

Each value has a weight. This weight is assigned using

- `:=` : Gives given weight to every item in the list
- `:/` : Divides the given weight equally among every item in the list

We cannot use weighted distribution on `randc` variables. So, here we need to make `flag` to a normal `rand` variable.

```
constraint flag_constr {
  // Weight of each number
  // 1, 2, ... 5  : 10
  // 6, 7, ... 10 : 50
  flag dist {[1:5]:/50, [6:10]:=50};
}
```

Implications

We can have if-else logic for defining constraints.

```
constraint size_from_flag_constr {
  if(flag == 2)
    size < 5;
  else if(flag == 3)
    size < 6;
}
```

A single if statement can also be defined using `->` operator.

```
constraint size_from_flag_constr {
  flag == 2 -> size < 5;
  flag == 3 -> size < 6;
}
```

foreach

Foreach can be used to iterate and check inside constraints

```
constraint list_constr {
  foreach(list[i])
    list[i] <= i * i * i + 1;
  foreach(list[i])
    list[i] >= 0;
}
```

Array methods using the `with` clause can also be used, and

Array reduction

Array reduction functions discussed in Array reduction section can be used for constraints

```
constraint list_sum {
  list.sum() > 10;
}
```

Solve before

The system verilog constraint solver tries to give a uniform value for all valid combinations. This could be a problem if we have implications. For example, certain modes can have lower probability if they have more possible values in other fields.

Instead, if we want each constraint to be solved independently before going to the next, we use `solve a before b;` as an additional statement in the constraint block.

Constraint mode

Constraints can be turned `ON` or `OFF` using the `constraint_mode()` function.

```
// Turn off flag constraint for obj
obj.flag_constr.constraint_mode(0);
```

This only turns the constraint off for that object. If we want to turn it off for all objects off the class, we need to declare the constraint as `static constraint` .

At randomization stage

In-line constratints

Constraints can be added at `randomize()` function call using `with` clause.

For example, if we want only `flag == 3` examples,

```
pkt.randomize() with {
  flag == 3;
};
```

Single variables can be randomized using `std::randomize()` and inline constraints

```
bit[3:0] register;

std::randomize(register) with {register dist {1:=50, 2:=50, [3:10]:/50};};
```

`pre_randomize()` and `post_randomize()`

Member functions in a class

`pre_randomize()` function is called before randomization in `randomize()` function call.

`post_randomize()` member function is called after randomization.

Disabling randomization

Randomization can be disabled individually for each value using `member.rand_mode(mode)` function which takes argument `0` for disabled and `1` for enabled. `member.rand_mode()` returns current randomization mode (0 for disabled, 1 for enabled)

Seeding

The random seed of a class can be changed by calling `obj.srandom(seed)`

randomize function

Randomize function fails if the object does not meet constraints to begin with.

Arguments can be passed to the randomize function on which members need to be randomized. Argument can even be null, in which case the return value indicates if constraints are met

```
class test;
  rand byte a;
  rand byte b;
  ...
endclass

module tb;
  test t1;
  ...
  t1.randomize(a);  // Only a is randomized
  ...
endmodule
```

randcase

case statement with weighted probability of taking each branch

```
randcase
  1 : $display(" Case 1");    // 1/5th probability
  1 : $display(" Case 2");    // 1/5th probability
  3 : $display(" Case 3");    // 3/5th probability
endcase
```

Inheritance and overriding

Constraints are also inherited and can be overriden like normal

Functional coverage

- Is used to track how many features have been covered in testing
- Defined using `covergroup`
- Variables are defined as `coverpoint`
- Concept of "bins". All bins in a coverpoint have to be filled for full coverage of that feature.

Example :

```
class packet;
  rand  int      size;
  randc bit[3:0] flag;
  rand  byte     list[$];

  // Flag must take only 0, 1, 4, 5, 8, 9
  constraint flag_constr {
    flag[1] == 0;
    flag    < 10;
  }

  // Size must be less than 10
  constraint size_constr {
    list.size() < 10;
    list.size() == size;
  }

  // Coverage group
  covergroup flag_coverage;
    coverpoint flag {
      bins flags[] = {0, 1, 4, 5, 8, 9};
    }
  endgroup

  function new();
        cg = new;
  endfunction
endclass
```

Covergroup

- `covergroup ... endgroup` is used to define coverage group
- We call `.sample()` function of a covergroup to sample values.

```
class packet;
  ...
  covergroup cg;
    ...
  endgroup
  ...
endclass

initial begin
  packet obj;
  ...
  obj.cj.sample();
  ...
end
```

- Or, sampling can be setup on an event or signal transition as

```
covergroup cg @(event);
  ...
endgroup
```

- We can stop and restart coverage check of a covergroup using `.stop()` and `.start()` functions.
- Covergroup must be initialized using `new` like `flag_coverage = new;` . This is usually done inside class constructor

Coverpoint

- `coverpoint var_name {...}` is used to define coverage details for different signals
- `coverpoint var_name;` by default creates independent bins for each possible value of `var_name`
- We can also define coverpoints for expressions, like `coverpoint a+b {...}`
- `iff` can be used for conditional coverage
-

```
covergroup cg;
  ...
  // check coverage only when enable is high
  coverpoint flag iff(w_enable) {
    // Bins for the covergroup
    ...
  }
endgroup
```

- Coverpoints can be named using the syntax `<name> : coverpoint <value> {...}`

Cross coverage

Cross coverage means checking for combinations of all bins inside multiple coverpoints

```
cp_1  : coverpoint val_1 {
  ...
}
cp_2  : coverpoint val_2 {
  ...
}
cross cp_1, cp_2;
```

Bins

- `bins` are defined inside coverpoints to define different coverage categories. All bins must be "hit" or "covered" for full coverage.
- Coverage is tested by the `covergroup` during a process called "sampling". This can happen by either explicitly calling `.sample()` function like
- `ignore_bins` ignore the values from other bins in the coverpoint
- `illegal_bins` throw an error if the bin is hit

Ways to define bins :

```
class packet;
  rand  int      size;
  randc bit[3:0] flag;
  rand  byte     list[$];

  covergroup cg;
    flag_cp : coverpoint flag {
      // Bin for a single value
      bins idle     = {0};

      // One bin each for each value
      bins modes[]  = {[1:10]};

      // A single bin which is triggered if any of 11, ... 15 happens
      bins invalid  = {[11:$]};

      // Change from 0 to 1 to 5
      bins change   = (0 => 1 => 5);

      // Ignore 13 from coverage
      ignore_bins ignore    = {13};

      // Make 14 illegal
      illegal_bins illegal  = {14};
    }

    size_cp : coverpoint size {
      // 0 to 20 range is divided into 4 bins
      bins size_range[4] = {[0:20]};
    }

    // Cross coverage between coverpoints
    cross_cp : cross size_cp, flag_cp;
  endgroup
endclass
```

Assertions

- Types of assertions
  - `assert` : To specify that given property is true in simulation
  - `assume` : To specify that given property is an assumption and to be used by tools to generate input stimulus
  - `cover` : To evaluate property for functional coverage

    - `restrict` : To specify property as a constraint on formal verification
- **Sequence**s give a method to describe sequence of multiple events happening.
- **Property** is a way to define conditions that must be satisfied. This could include simple expressions or sequences.
- Assertions cannot be used by dynamic entities like classes. One way to overcome this is to use static lines and assign them when required from the class variables.

Immediate assertion

Immediate assertions are evaluated when the line is encountered.

```
// Simple assertion
a_1 : assert(<expression>);

// If we want to do something on assert fail/pass
a_2 : assert(<expression>) begin
  ...               // Assertion passed
end else begin
  ...               // Assertion failed
end

// We can also omit the pass portion
a_3 : assert(<expression>) else begin
  ...
end
```

If we provide code to execute when an assertion fails using the `assert() ... else ...` syntax, the assertion error is not thrown. We can manually throw the error using `$error`, `$warning`, `$fatal` or `$info`

Assertion failures are reported but simulation goes ahead.

Sequences

Structures to formally define sequence of signals

```
// No timing with arguments
sequence test_seq(a, b);
  a == b;
endsequence

// Calling another sequenence inside a sequence
sequence test_seq_2;
  test_seq(data[0],data[1]);
endsequence

// With a trigger statement
sequence test_seq_3;
  @(posedge clk) test_seq(data[0], data[1]);
endsequence

// We can define in terms of cycles
sequence test_seq_4;
  // data[1] 4 cycles after data[0] goes high
  @(posedge clk) data[0] ##4 data[1];
endsequence
```

Operators we can use inside sequences :

Cycle delay

`##4` : Delay of 4 event/clock cycles

Timing window

`##[1:5]` : Delay of 1, 2, ... or 5 cycles

`##[2:$]` : Delay of 2 or more cycles (indefinite)

Implication

`a |-> b` : Evaluates `b` when `a` is true

`a |=> b` : Evaluates `b` one cycle after `a` is true

Repetition

`a |-> ##1 b ##1 b ##1 b` : `b` must be high for 3 consecutive cycles after `a`

This can also be written as : `a |-> ##1 b [*3]`

Specifies that the confition should be matched for n number of cycles

go to repitition

`a |-> ##1 b [->3] ##1 c` specifies that `b` must be high for 3 cycles, which need not be concecutive, and then `c` must be high on next cycle

non-consecutive repition

`a |-> ##1 b [=3] ##1 c` is similar, but `c` han be high on the any cycle after the 3rd cycle `b` is high. The version before needs `c` to be high on the next cycle.

Only expressions can use non-consecutive and go-to repition. Sequences cannot use them.

Built-in functions

Functions to use in sequences

- `$rose(val)`
  - Did `val` rise from `0` to `1` ?
- `$fell(val)`
  - Did `val` fall from `1` to `0` ?
- `$stable(val)`
  - Is `val` stable compared to the last cycle?
- `$past(val, num)`
  - Returns the value of `val` `num` cycles ago
- `$past(val, num, en)`
  - `$past` , but cycles are counted only when `en` signal was high

Functions helpful for assertions

- `$onehot(val)`
  - Is `val` one-hot encoded?
- `$onehot0(val)`
  - One-hot encoded or all zero
- `$isunknown(val)`
  - Is any bit in `val` unknown? (`x` or `z`)
- `$countones(val)`
  - Return number of ones in vector `val`

Property

Defines conditions that must be satisfied in concurrent assertions. All concurrent assertions must have an associated property, which could be inline as well.

```
sequence seq(clk, a, b);
  @(posedge clk) a |=> b;
endsequence

property example_ptty;
  c |-> seq(clk, a, b);
endproperty
```

We can also disable expressions in properties using `disable iff`

```
property example_ptty;
  disable iff(!en) c |-> seq(clk, a, b);
endproperty
```

sequences have a mmmber, `.ended` which could be usefull for synchronizing different sequences

```
property example_ptty;
  disable seq1.ended |-> ##2 seq2.ended;
endproperty
```

Means seq2 must end 2 cycles after seq1 ends.

Providing trigger event for concurrent accesses

Concurrent assertions are sampled at a clock tick. We need to provide a trigger event for this. This is how it can be done :

- Specify in the sequence

```
sequence seq;
  @(posedge clk) ...
endsequence
```

- Specify in the property

```
property ptty;
  @(posedge clk) seq;
endproperty
```

- When inside a clocking block

```
clocking cb @(posedge clk);
  ...

  property ptty;

  endproprty

  ...
endclocking
```

- Default clocking block

```
default clocking cb @(posedge clk);
endclocking
```

Error classes

The class of errors are

- `$fatal`
- `$error`
- `$warning`
- `$info`

They are called like `$fatal("Fatal error!!")` . We can call these inside the if...else... blocks of the assertions.

Miscellaneous

System verilog scheduler

Every timestep in a system verilog simulation is scheduled in multiple stages :

- preponed
  - Concurrent assertions are sampled
- active
  - Processing continous assignments
  - Processing blocking assignments
  - Evaluation of RHS of non-blocking assignments
  - Evaluate primitives
  - `$display`
- inactive
  - `#0` delay statements
- non-blocking assignments (NBA)
  - Apply evaluated RHS to variables
- observed
  - Assertions are evaluated
- re-active
  - Active region for statements inside program block and assertion action blocks
- re-inavtive
  - `#0` delay statements inside program blocks
- postponed
  - `$monitor` , `$strobe`

- Re-active and re-inactive region are used to execute statements in `program` blocks, to avoid race conditions with statements in modules. Often in race conditions the output was dependent on the simulators.

Program block

To avoid the race condition mentioned before, we use `program` blocks instead of `module` . programs are identical to modules, but are scheduled after all module statements.

```
module design_mod(input clk, output[3:0] data);
  ...
  always @(posedge clk)
    data <= ...
endmodule

module test_mod(input[3:0]  data);
  ...
  always @(posedge clk)
    $display("Data : %h", data);
endmodule

module toplevel;
  ...
endmodule
```

Has a race condition between `$display` in `test_mod` and the assign statement inside `design_mod`. Which executes first is not clear if we dont know the underlying details of the scheduler (`$display` happens in active region and assignment in NBA region which comes later, so display is executed first).

To avoid this, we use program. `program` statements always run after `module` statements. So, `$display` comes after the assignment if we had defined test_mod as a program

```
program test_prog(input clk, input[3:0] data);
  ...
  always @(posedge clk)
    $display("Data : %h", data);
endprogram
```

References

1. Chipverify system verilog tutorial
2. verificationguide system verilog tutorial
3. EDA playground
4. always_ff, always_latch, always_comb blocks