UVM Basics

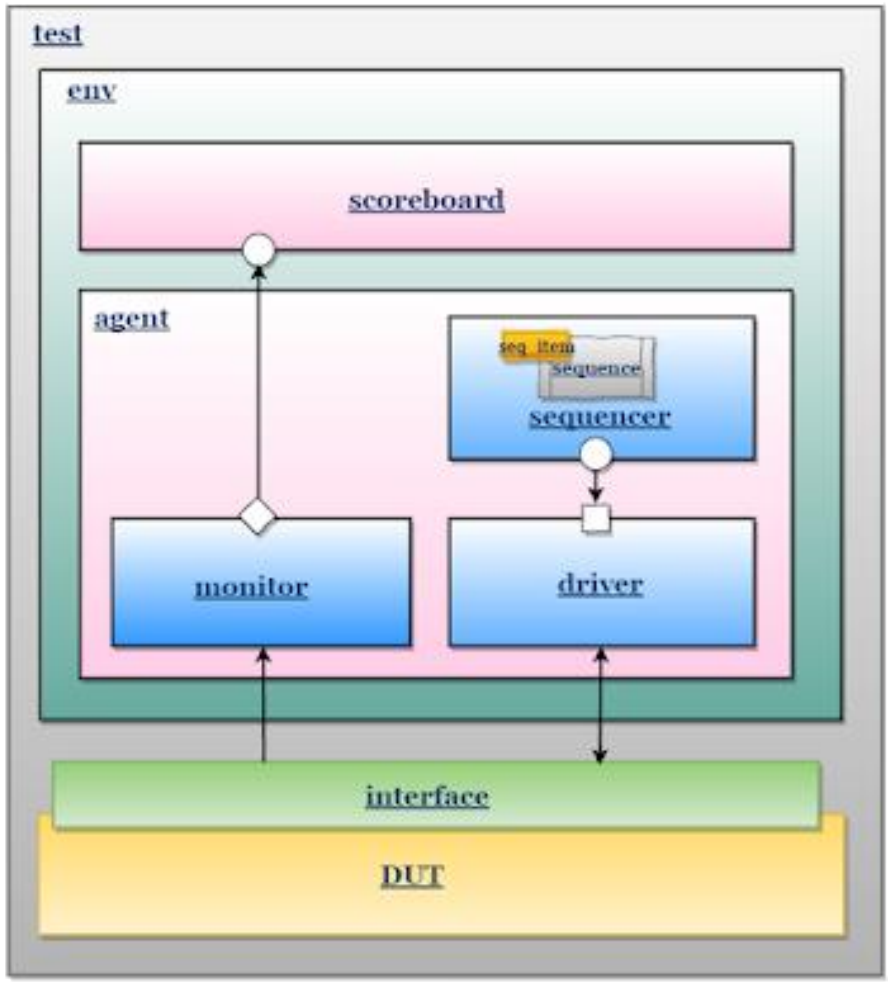
Mitesh Khadgi

For queries, contact: mitesh.khadgi2025@gmail.com

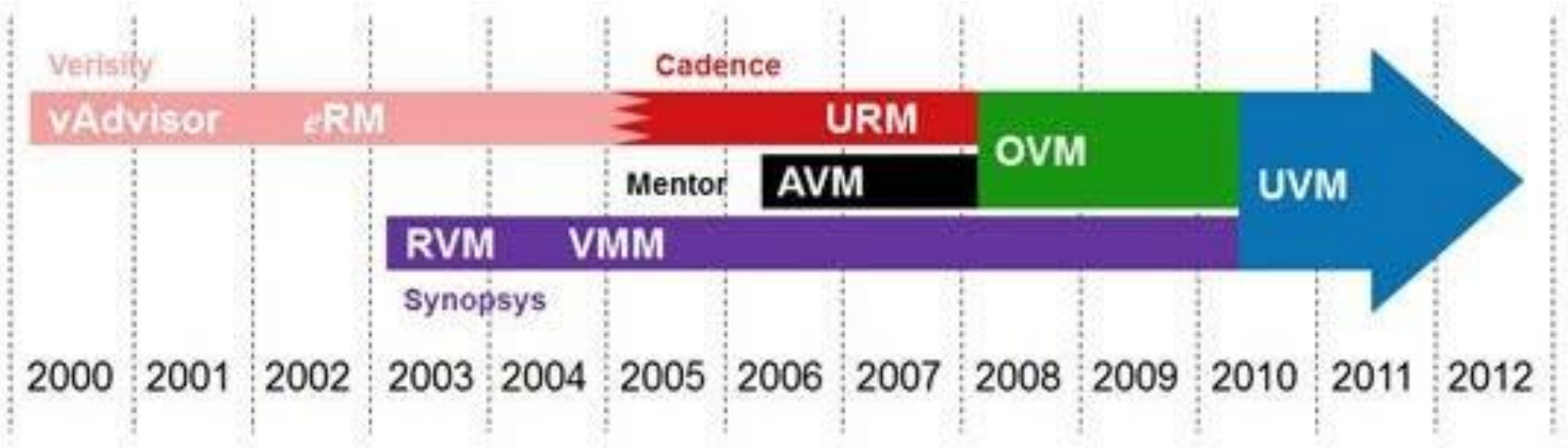
RTL Verification with Verilog

- Register Transfer Level (RTL) designing and its verification is also most important part of circuit design. The intent behind verification is to check that design meets all system specification and requirements or not.

Testbench Architecture to verify complex RTL designs



Evolution of UVM



History of Verilog Language

- Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems.
- It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction.
- It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits.
- In 2009, the Verilog standard (IEEE 1364-2005) was merged into the SystemVerilog standard, creating IEEE Standard 1800-2009. Since then, Verilog is officially part of the SystemVerilog language.
- The current version is IEEE standard 1800-2017.

History of System Verilog

- SystemVerilog started with the donation of the Superlog language to Accellera in 2002 by the startup company Co-Design Automation.
- The bulk of the verification functionality is based on the OpenVera language donated by Synopsys.
- In 2005, SystemVerilog was adopted as IEEE Standard 1800-2005.
- In 2009, the standard was merged with the base Verilog (IEEE 1364-2005) standard, creating IEEE Standard 1800-2009.
- The current version is IEEE standard 1800-2017.

Data types in Verilog and System Verilog

	Data-type	2-state/4-state	# Bits	signed/unsigned	C equivalent	
SYSTEM VERILOG	reg	4	>= 1	unsigned		VERILOG
	wire	4	>= 1	unsigned		
	integer	4	32	signed		
	real				double	
	time					
	realtime				double	
	logic	4	>= 1	unsigned		
	bit	2	>= 1	unsigned		
	byte	2	8	signed	char	
	shortint	2	16	signed	short int	
	int	2	32	signed	int	
	longint	2	64	signed	long int	
	shortreal				float	

Testbench for simple RTL Design

testbench	dut

module tb;
  reg clock, reset;
 	
  dut(clock, reset);
 	
  initial begin
  clock = 0;
  forever #5 clock = ~clock;
  end
 	
  initial begin
  reset = 1;
  #20;
  reset = 0;
  end
 	
endmodule
	module dut (clock, reset);
	  input clock;
	  input reset;
	
	  reg flop;
	
	  always@(posedge reset or posedge clock)
	  begin
	  if(reset) // reset active high
	  begin
	  flop <= 0;
	  end
	  else
	  begin
	  flop <= 1;
	  end
	  end
	
	endmodule

Why System Verilog is used for verification?

- System Verilog is an extension of Veilog with many verification features that allow engineers to verify the design using complex testbench structures and random stimuli in simulation.
- SystemVerilog, standardized as IEEE 1800, is a hardware description and hardware verification language used to model, design, simulate, test and implement electronic systems. SystemVerilog is based on Verilog and some extensions, and since 2008, Verilog is now part of the same IEEE standard. It is commonly used in the semiconductor and electronic design industry as an evolution of Verilog.

What is verification?

- Verification is the process of ensuring that a given hardware design works as expected. An environment called testbech is required for the verification of a given verilog design and is usually in System Verilog these days. The idea is to drive the design with different stimuli to observe its outputs and compare it with expected values to see if the design is behaving the way it should.

What is setup time?

- Setup time is defined as the minimum amount of time BEFORE the clock’s active edge by which the data must be stable for it to be latched correctly. Any violation in this required time causes incorrect data to be captured and is known as a setup violation.

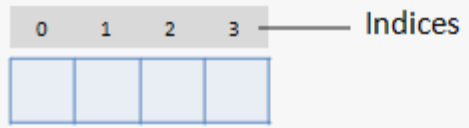
What is hold time?

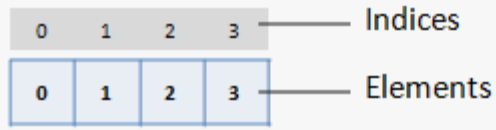
- Hold time is defined as the minimum amount of time AFTER the clock’s active edge during which the data must be stable. Any violation in this required time causes incorrect data to be latched and is known as a hold violation.

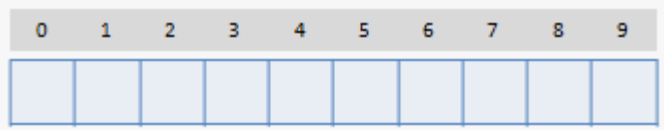
Associative Vs Dynamic Array

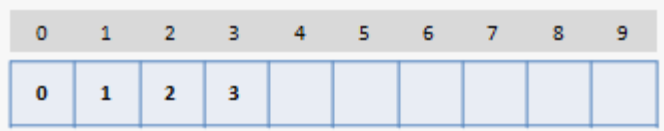
Array Type	Declaration	Features
Fixed	bit [63:0] ddr_mem_f [0:`MEM_SIZE]	Fixed size array at compile time. If all elements are not used waste of memory & performance.
Dynamic Array	bit [63:0] ddr_mem_d []; ddr_mem_d = new[`MEM_SIZE];	No size defined at compile time Size is defined at run time as needed & can resize as needed Good for contiguous data
Queues	bit [63:0] ddr_mem_q[$] ddr_mem_q.push_back(wdata) rdata = ddr_mem_q.pop_front()	No size defined at compile time Elements are added on top/bottom using push_front/back Elements are removed from top/bottom using pop_front/back Good for out-of-order and one-time checking.
Associative Array	bit [63:0] ddr_mem_a[bit [33:0]] ddr_mem_a[address] = wdata if(ddr_mem_a.exists(address)) rdata = ddr_mem_a[address]	No size defined at compile time Elements are added as per key [address] If key exists, elements can be accessed & delete Good for non-contiguous data and multiple access

```
//dynamic array declaration
bit [7:0] d_array[];

//memory allocation
d_array = new[4];

//array initialization
d_array = {0,1,2,3};

// Increasing the size by overriding the old values
d_array = new[10];

//Increasing the size by retaining the old values
d_array = new[10](d_array);
```









www.verificationguide.com

Associative Array

Associative Array

Method	Description
num(), size()	Returns number of entries in associative array
delete()	Removes entry at specific index
exists()	Checks whether element exists at specific index
first()	Returns the value of first index
last()	Returns the value of last index
next()	Returns the smallest index whose value is greater than given index
prev()	Returns the largest index whose value is lesser than given index

When to use Associative and dynamic array?

- Associative arrays can be indexed with strings instead of integers. They are useful for modeling sparse memory.
- Dynamic arrays don’t need their size when being defined and the size can be adjusted multiple times.
- Queues : As name suggest it is useful for FIFO kind of application. It has push and pull methods inbuilt.
- Fixed arrays : Arrays whose size is decided at time of declaration and remain same in simulation.
- The **dynamic array** has a **continuous memory allocation** whereas the **Associative array** has a **non-continuous memory allocation.**
- Memory gets allocated for all the **declared locations** of dynamic array at run-time, whereas memory for **associative array** will be created only for the locations in which **data is written.**

Dynamic Arrays

- When the size of an array is determined only during runtime it is called a Dynamic Array.
- The syntax is:

```
int dyn_array [ ];
```

- The advantage of the dynamic array is
  - The memory is created during run time
  - The size of the array can be modified as per the requirement during run time.
- The size can be provided as shown here:

```
initial
     begin
          dyn_array = new [10];
          ...
          ..
          dyn_array = new [20];
     end
```

Inter-process synchronization

- **semaphore :**
  - as a bucket with a fixed number of “keys.”
  - built-in methods : new(), get(), put() and try_get().
- **mailbox**
  - allows messages to be exchanged between processes. A message can be added to the mailbox at anytime by one process, and retrieved anytime later by another process.
  - Mailboxes behave like FIFOs (First-In, First-Out).
  - built-in methods: new(), put(), tryput(), get(), peek(), try_get() and try_peek().
- **Event**
  - The Verilog "event" type is a momentary flag that has no logic value and no duration. If a process is not watching when the event is triggered, the event will not be detected.
  - SystemVerilog enhances the event data type by allowing events to have persistence throughout the current simulation time step. This allows the event to be checked after it is triggered.

Mailbox

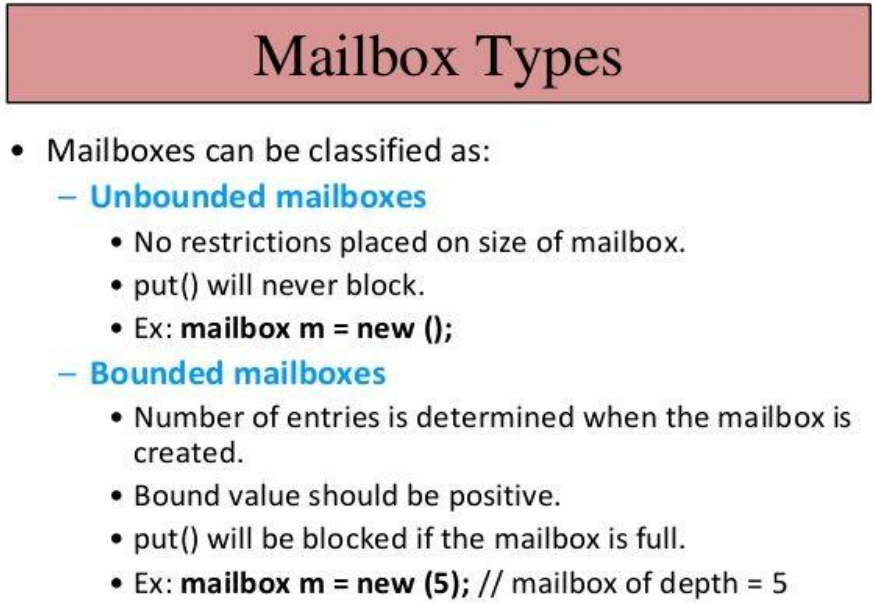
- A System Verilog mailbox is a way to allow different processes to exchange data between each other. It is similar to a real postbox where letters can be put into the box and a person can retrieve these letters later on.

- System Verilog mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox can only store a limited amount of data, and if a process attempts to store mroe messages into a full mailbox, it will be suspended until there's enough room in the mailbox. However, an unbounded mailbox has unlimited size.

- Two types of mailbox: generic and parameterized mailbox

Where to use mailbox?

- A Systen Verilog mailbox is typically used when there are multiple threads running in parallel, and want to share data for which a certain level of determinism is required.



Functions/Methods in Mailbox

Method	Description	Syntax
`new()`	Mailbox constructor which optionally specifies maximum size	`function new(int bound = 0);` **Note by default no maximum size.**
`num()`	Returns the number of messages currently in the mailbox	`function int num();`
`put()`	Places a message in the mailbox – blocks if mailbox full	`task put (<message>);`
`try_put()`	Places a message in a mailbox – returns 0 if mailbox full	`function int try_put (<message>);`
`get()`	Retrieves a message from the mailbox – blocks if mailbox empty – error if type mismatch	`task get (ref <variable>);`
`try_get()`	Retrieves a message from the mailbox – returns +int if successful – returns 0 if empty – returns -int if type mismatch	`function int try_get (ref <variable>);`
`peek()`	*Copies* a message from the mailbox – blocks if mailbox empty – error if type mismatch	`task peek (ref <variable>);`
`try_peek()`	*Copies* a message from the mailbox – returns +int if successful – returns 0 if empty – returns -int if type mismatch	`function int try_peek (ref <variable>);`

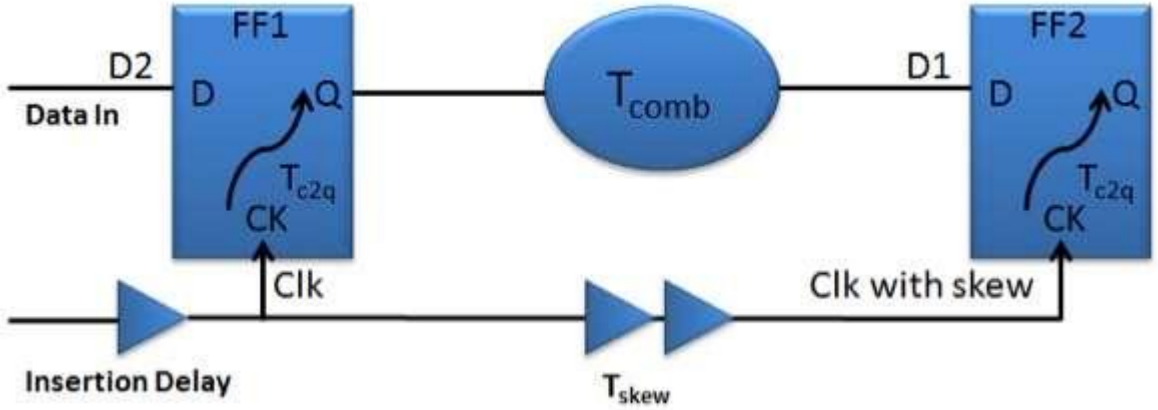
Semaphore

- Semaphore is just like a bucket with a fixed number of keys. Process that use a semaphore must first get a key from the bucket before they can continue to execute. Other processes must wait until keys are available in the bucket for them to use. In a sense, they are best used for mutual exclusion, access control to shared resources and basic syncronization.

Semaphore

Method	Description	Syntax
`new()`	Semaphore constructor which specifies number of keys	`function new(int keyCount = 0);` **Note default number of keys set is 0.**
`get()`	Extracts a set number of keys from the semaphore – blocks if the keys are not available	`task get(int keyCount = 1);`
`try_get()`	Extracts a set number of keys from the semaphore without blocking – returns 0 if the keys are not available	`function int try_get(int keyCount=1);`
`put()`	Returns a set number of keys to the semaphore	`task put(int keyCount=1);`

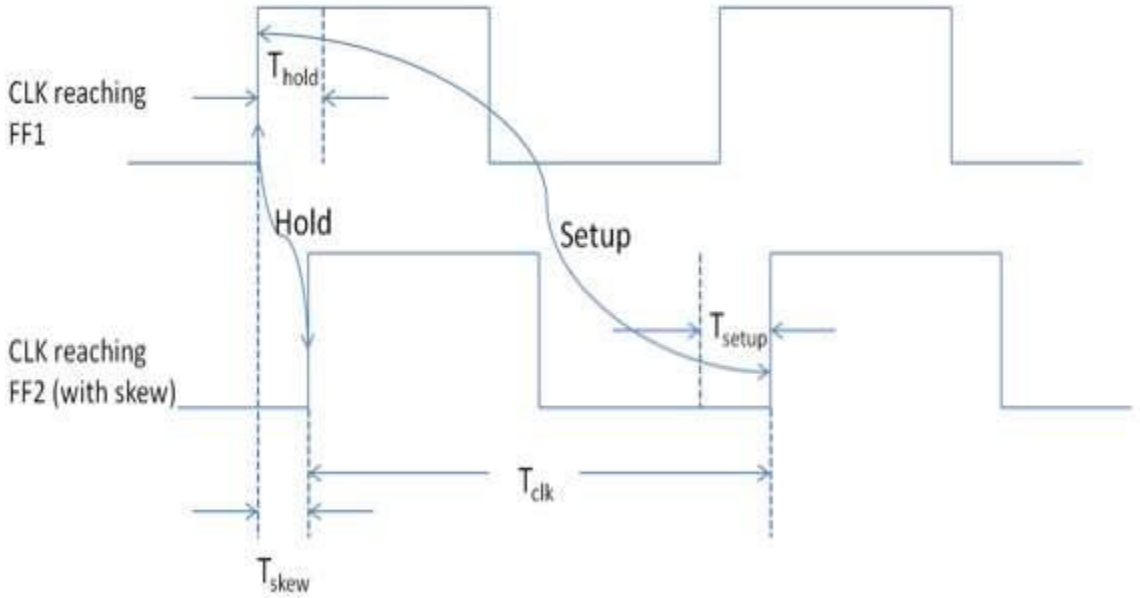
Combinational circuit



- Tcq, Thold, Tsetup
- In the diagram above, at time zero FF1 is to process D2 and FF2 is to process D1. Time taken for the data D2 to propagate to FF2, counting from the clock edge at FF1, is invariably = T_{c2q} +T_{comb} and for FF2 to successfully latch it, this D2 has to be maintained at D of FF2 for T_{setup} time before the clock tree sends the next positive edge of the clock to FF2. Hence to fulfill the setup time requirement, the formula should be like the following.
    - $T_{c2q} + T_{comb} + T_{setup} \leq T_{clk} + T_{skew}$

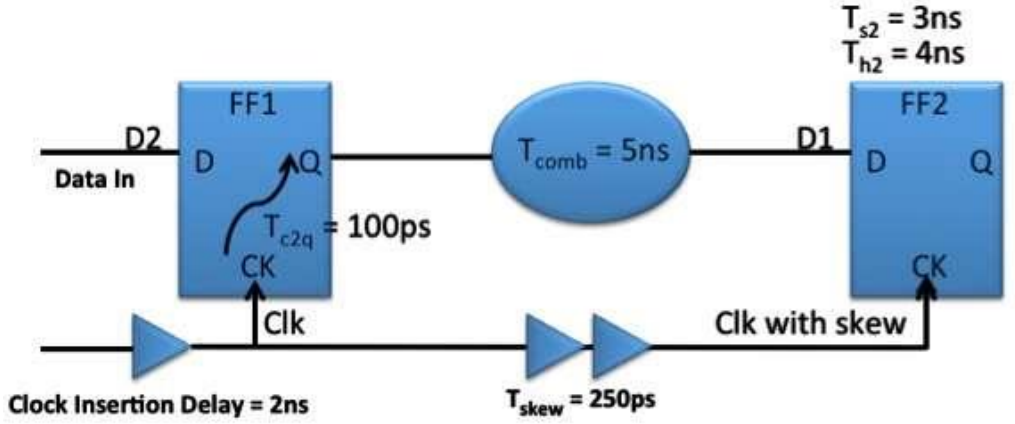
How to avoid hold time violation?

- Now, to avoid the hold violation at the launching flop, the data should remain stable for some time (T_{hold}) after the clock edge. The equation to be satisfied to avoid hold violation looks somewhat like below:
  - $T_{c2q} + T_{comb} \geq T_{hold} + T_{skew}$



Setup and Hold Forumlas

- (Time available for data to travel from FF1 to FF2) ≥ (Time needed for data to travel from FF1 to FF2)
  Tclk + Tskew ≥ Tc2q + Tcomb + Ts2
  Tclk + 0.25ns ≥ 0.1ns + 5ns + 3ns
  Tclk ≥ 7.85ns
- Thus a minimum Clock Period of **7.85 ns** is required to prevent setup violation. This translates to a maximum operating frequency of **127.4 MHz**.



Improve maximum operating frequency

- From the above example it is clear that to improve the maximum operating frequency, any of the following steps can be taken:
  1. Decrease Tcomb between talking flops.
  2. Increase Tskew if there is scope to do so.
  3. Select flops with lower Tc2q and Tsetup.

$$\textbf{Setup slack} = T_{period} - (T_{ck->q} + T_{prop} + T_{setup} - T_{skew})$$

$$\textbf{Hold slack} = T_{ck->q} + T_{prop} - T_{hold} - T_{skew}$$

Mitigating violations

- **Mitigating setup violation**: Thus, we can meet the setup requirement, if violating, by
  - Decreasing clk->q delay of launching flop
  - Decreasing the propagation delay of the combinational cloud
  - Reducing the setup time requirement of capturing flop
  - Increasing the skew between capture and launch clocks
  - Increasing the clock period
- **Mitigating hold violation**: We can meet the hold requirement by:
  - Increasing the clk->q delay of launching flop
  - Decreasing the hold requirement of capturing flop
  - Decreasing clock skew between capturing clock and launching flip-flops

What is task and function?

Task	Function
Can call another task and functions	Unable to call a task, but can call functions
Can contain delay, event, and timing control statements	Cannot contain delay, event, and timing control statements
Task can perform operation in both zero simulation time and non-zero simulation time	Function performs it operation in zero simulation time
Allowed to use zero or more arguments which can be of type output. input, or inout	Atleast one argument needs to be passed
Cannot return a value, but can pass multiple values via the output and inout statements	Can only return a single value, doesn't have output or inout statements
For example: task sum (input [31:0] a, input [31:0] b, output [31:0] b); c = a + b; Endtask	For example: function bit [31:0] sum (input bit [31:0] a, input bit [31:0] b); c = a + b; return c; endfunction

What is initial and always?

initial	always
Assignments in an initial block begin to execute from time 0 in simulation and proceed in the specified sequence	Assignments in an always block also begin from time 0, and repeat forever as a function of the changes on the blocks sensivity list
Execution of statements in an initial begin-end block stops when the end of the block is reached,i.e., executed only once during simulation	Execution continuously repeats from the begin to the end of the process unless held by a wait construct throughout the simulation session
Non-synthesizable construct	Synthesizable construct

For example:
 	
reg [1:0] out1, out2;
initial begin
  out1 = 2'b10;
  #5 out2 = 2'b01;
end
	For example:
	
	reg [1:0] out1, out2;
	always @(posedge clk)
	begin
	  out1 <= in1;
	  out2 <= out1 & in2;
	end

Blocking and Non-blocking

Blocking assignments	Nonblocking assignments
In a blocking assignment, the evaluation of the expression on the RHS is updated to the LHS variable autonously based on the delay value (either 0 if no delay specified, or scheduled as a future event if a non-0 value is specified)	Nonblocking assignment to LHS is scheduled to occur when the next evaluation cycle occurs in simulation and not immediately within the same time unit
When multiple blocking assignments are present in a process, the trailing assignments are blocked from occuring until the current assignment is completed	Multiple nonblocking assignments can be scheduled to occur concurrently on the next evaluation cycle in simulation
These is a possibility of race conditions on the variables of blocking assignments if assignments happen to it from two processes concurrently	The race conditions are avoided as the updated value is assign after evaluation
Recommended to use within combinatorial always blocks	Recommended to use within the sequential always blocks
Can be used in procedural assignments like initial, always, and continious assignments to nets like assign statements	Can be used only in the procedural block like initial, and always; Continuous assignment to nets like the assign statement is not permitted
Represented by "=" operator sign between LHS and RHS	Representated by "<=" operator sign between LHS and RHS
For example: #10 a = b; #15 b = c;	For example: #10 a <= b; #15 b <= c;

Program vs Module

- A program is similar to module.
- Program and Module can contain ports, interfaces, final and initial statements.
- Program block can not contain always block
- A module (design) can not call task/function inside a program block.
- Module is basically used for your RTL design while Program block is used for RTL verification. Program block is used as a divider between the dut and the testbench
- Every program needs to finish with no always or forever kind of loops, whereas module can run forever without any finish statement.

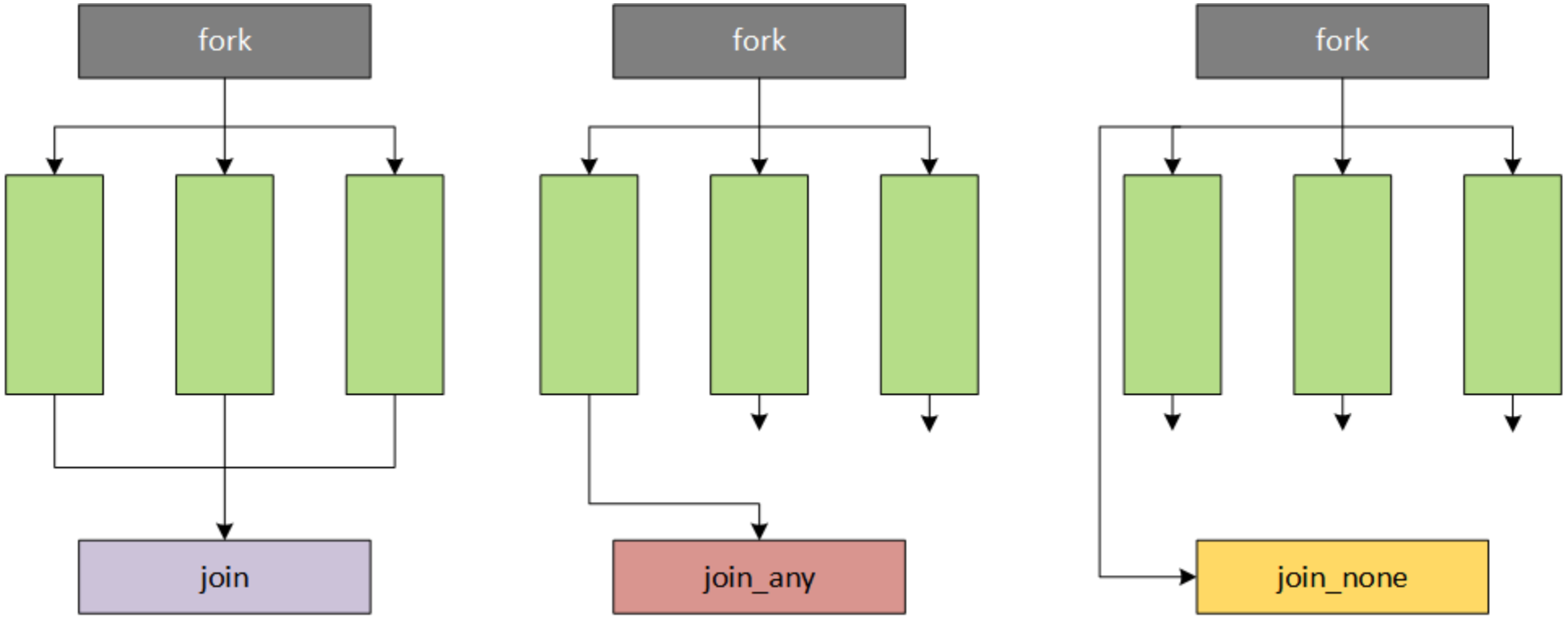
Control flow loops in System Verilog

- while/do-while
- foreach
- for
- forever
- repeat

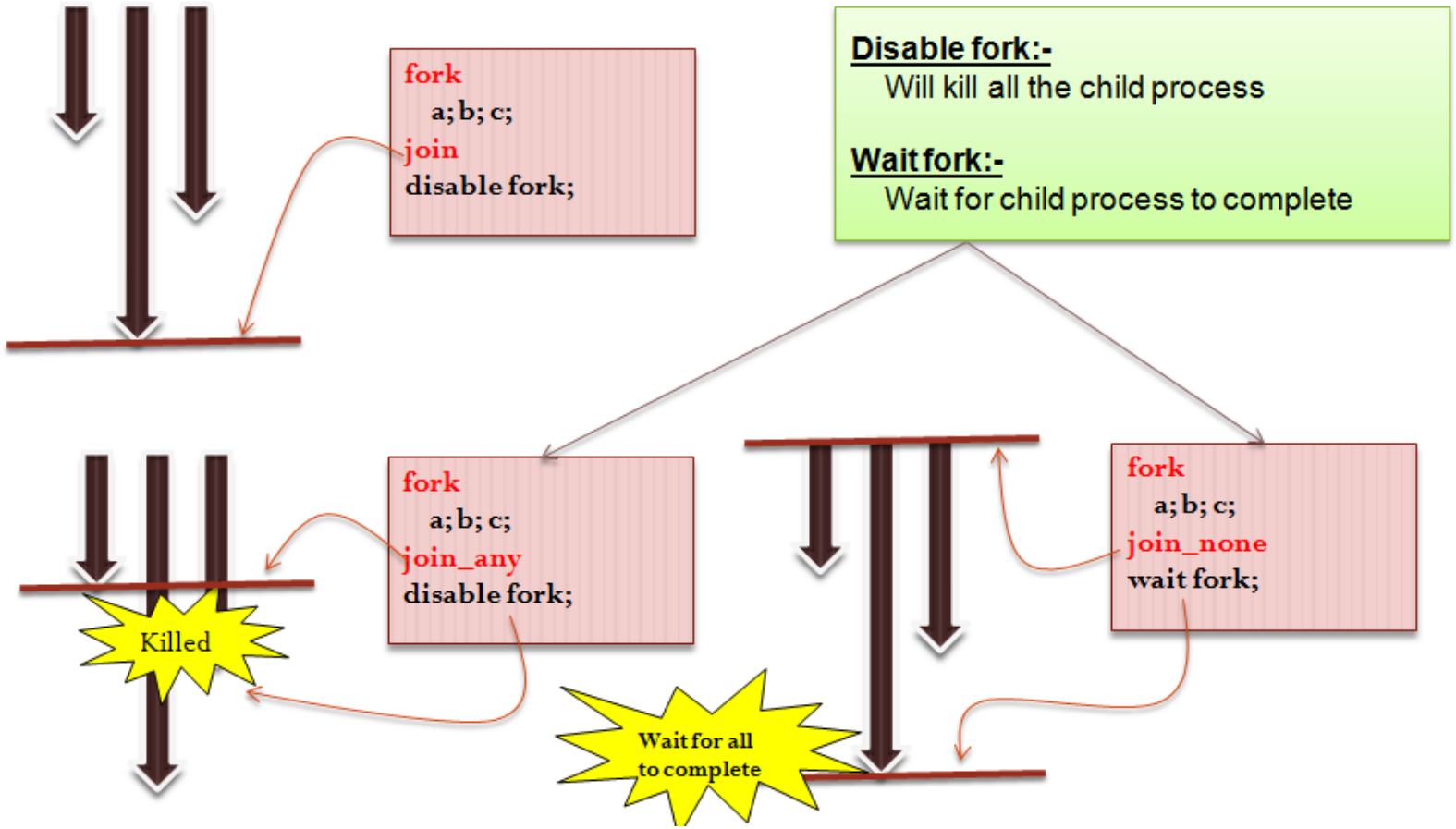
Fork-Join statements in System Verilog

- Thread 1; // finishes at 10 ns
- Thread 2; // finishes at 5 ns
- Thread 3; // finishes at 2 ns
- Fork Join concept is used when we need to execute all the threads parallely
- fork join: It will execute all the threads until it finishes at 10 ns
- fork join_any: It will execute only one thread out of 3 which takes least amount of time, in our case it is Thread 3
- fork join_none: It will start executing all the Threads in background, without waiting for threads to complete, main thread resumes execution of statements after the fork join_none block

Visualization of fork join cases



SystemVerilog fork/join



Difference between .new(), new(), and create

- Every Class has a built-in method called new() which is a must to call to create an Object of that Class type. The new() method is also known as the “Constructor“. Constructing an Object allocates the space in the memory needed to hold the Properties of an Object. We can define our own Constructor that overrides the built-in Constructor & put whatever code we want inside it. We can also define arguments to pass to the Constructor. The function new() does not have any return type. But new() method implicitly returns the Handle to an Object of the Class type.
- create() vs new() in UVM
  - Create() is a factory method which construct an object. To override an object you need to construct it using create(). if you use set_type_override before run then, factory replaces constructed object with derived object (specified in override). if you use new() then you can't override.

create() vs new() in UVM

- new is used for constructor functions and Object. create() is used to inherit Objects
- A new() constructor will only create an object of a given type and therefore using a new() will not allow run-time changing of class types. Hence, using a new() means the testbench code will need to change based on the different types to be used.
- The create function goes through the UVM factory and checks for registered type or instance overrides. This allows for objects or components to be replaced by derived types using the factory. I would suggest reading up on Factory Pattern, which is a key concept in UVM.

System tasks

- $display – Print to screen a line followed by an automatic newline.
- $write – Print to screen a line without the newline.
- $swrite – Print to variable a line without the newline.
- $sscanf – Read from variable a format-specified string. (*Verilog-2001)
- $fopen – Open a handle to a file (read or write)
- $fdisplay – Print a line from a file followed by an automatic newline.
- $fwrite – Print to file a line without the newline.
- $fscanf – Read from file a format-specified string. (*Verilog-2001)
- $fclose – Close and release an open file handle.
- $readmemh – Read hex file content into a memory array.
- $readmemb – Read binary file content into a memory array.
- $monitor – Print out all the listed variables when any change value.
- $time – Value of current simulation time.
- $dumpfile – Declare the VCD (Value Change Dump) format output file name.
- $dumpvars – Turn on and dump the variables.
- $dumpports – Turn on and dump the variables in Extended-VCD format.
- $random – Return a random value.

OOPs(Object Oriented Programming) concept in System Verilog

Encapsulation:

- Creating containers of data along with the associated behaviors. Basically the code that operates on that data.

Inheritance:

- The ability to extend or override those containers with additional data and new behaviors.

Abstraction:

- The process of abstraction in System Verilog is used to hide certain details and only show the essential features of the object. In other words, it deals with the outside view of an object.

Data Hiding:

- Hiding the implementation details to reduce the complexity as well as raising the abstraction level.

Generic Programming (Parametrization):

- The ability to write code that can be reused across wide range of applications. Its like in Verilog overriding parameters on a Module instance.

Polymorphism:

- Its the simple key concept. This is the reuse of same code to take on many different behaviors based on the type of object at hand.

Encapsulation

- Many times we use the base class provided by third party sources. By default, these class members are public in nature. It means these class members can be accessed directly from outside of that class. But sometimes base class providers may restrict how others can access the class members as a safety/security measure, preventing corruption of internal states and logic.
- Local: It is available only to methods inside the same class. Further, these local members are not visible within subclasses.
  - For example: local bit [31:0] tmp_addr;
- Protected: It is a property or method that has all of the characteristics of a local member, except that it can be inherited, and is visible to only subclasses.
  - For example: protected bit [31:0] tmp_addr;

Scope of local variable

```
class parent_class;
  local bit [31:0] tmp_addr;
  function new(bit [31:0] r_addr);
    tmp_addr = r_addr + 10;
  endfunction
  function display();
    $display("tmp_addr = %0d", tmp_addr);
  endfunction
endclass
module encapsulation;
  initial begin
    parent_class p_c = new(5);
    p_c.tmp_addr = 20;  // Accessing local variable outside the class
    p_c.display();
  end
endmodule
```

OUTPUT:
Error- Illegal class variable access
Local member 'tmp_addr' of class 'parent_class' is not visible to scope 'encapsulation'.

Scope of protected variable

```
class parent_class;
  protected bit [31:0] tmp_addr;
  function new(bit [31:0] r_addr);
    tmp_addr = r_addr + 10;
  endfunction
  function display();
    $display("tmp_addr = %0d",tmp_addr);
  endfunction
endclass
class child_class extends parent_class;
  function new(bit [31:0] r_addr);
    super.new(r_addr);
  endfunction
  function void incr_addr();
    tmp_addr++;
  endfunction
endclass
```

```
//   module
module encapsulation;
 initial begin
   parent_class p_c = new(5);
   child_class  c_c = new(10);

   // variable declared as protected cannot be accessed
outside the class
   p_c.tmp_addr = 10; // this doesn't work
   p_c.display();

   c_c.incr_addr();  //Accessing protected variable in
extended class
   c_c.display();
 end
endmodule
```

Inheritance

```
class parent_class;
  bit [31:0] addr;
endclass

class child_class extends parent_class;
  bit [31:0] data;
endclass

module inheritance;
  initial begin
    child_class c = new();
    c.addr = 10;
    c.data = 20;
    $display("Value of addr = %0d, data = %0d", c.addr, c.data);
  end
endmodule
```

Output:
Value of addr = 10, data = 20

Polymorphism

```
// base class
class base_class;
  virtual function void display();
    $display("Inside base class");
  endfunction
endclass
// extended class 1
class ext_class_1 extends base_class;
  function void display();
    $display("Inside extended class 1");
  endfunction
endclass
// extended class 2
class ext_class_2 extends base_class;
  function void display();
    $display("Inside extended class 2");
  endfunction
endclass
```

```
module class_polymorphism;

 initial begin

   //declare and create extended class
   ext_class_1 ec_1 = new();
   ext_class_2 ec_2 = new();
   //base class handle
   base_class b_c[2];

   //assigning extended class to base class
   b_c[0] = ec_1;
   b_c[1] = ec_2;
   //accessing extended class methods using base class
handle
   b_c[0].display();
   b_c[1].display();
 end
endmodule
```

OUTPUT:
Inside extended class 1
Inside extended class 2

Another example of Polymorphism

- class base;
- int a = 10;
- int b = 20;
- virtual function display;
- $display("Printing values from base class = %d, %d", a, b);
- endfunction
- endclass
- class child_1 extends base;
- function display;
- $display("Printing values from child class = %d, %d", a, b);
- endfunction
- endclass

OUTPUT for D.display():
Printing values from child class = 10, 20

```
module tb;

initial begin
base A;
child_1 B;
child_1 D;
base D;

A = new();
B = new();
D = new();
A.display();
B.display();
D.display();
End

endmodule
```

Abstraction in System Verilog

```
//abstract class
virtual class packet;
  bit [31:0] addr;
endclass
module virtual_class;
  initial begin
    packet p;
    p = new();
  end
endmodule
```

OUTPUT:

```
// virtual_class, "p = new();"
// Instantiation of the object 'p' can not be done because its type
'packet' is an abstract base class.
// Perhaps there is a derived class that should be used.
```

Using virtual class to hide and use derived class

```
//abstract class
virtual class packet;
  bit [31:0] addr;
endclass
class extended_packet extends packet;
  function void display;
    $display("Value of addr is %0d", addr);
  endfunction
endclass
module virtual_class;
  initial begin
    extended_packet p;
    p = new();
    p.addr = 10;
    p.display();
  end
endmodule
```

OUTPUT:

// Value of addr is 10

System Verilog Assertions

- The behavior of a system can be written as an assertion that should be true at all times. Hence assertions are used to validate the behavior of a system defined as properties, and can also be used in functional coverage.
- For example, assume the design requests for grant and expects to receive an ack within the next 4 cycles. But, if the design gets an ack on the 5th cycle, the property that an ack should be returned within 4 clocks is violated and the assertion fails.

- // A propert writin in Verilog/System Verilog
- always @ (posedge clock) begin
- if(!(a && b))
- $display("Assertion failed");
- end

- // The property above written in System Verilog Assertions syntax
- assert property(@(posedge clock) a && b);

Types of System Verilog Assertions

- Immediate assertion
  - 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.

  - assert (A == B) $display ("OK. A equals B");
  - else $error("It's gone wrong");
- Concurrent assertion

Concurrent Assertion

- module tb;
- bit a, b, c, d;
- bit clock;

- always #10 clock = ~clock;

- initial begin
- for(int i = 0; i < 20; i++)
- {a, b, c, d} = $random;
- $display("%0t a = %0d, b = %0d, c = %0d, d = %0d", $time, a, b, c, d);
- @(posedge clock);
- end
- #10 $finish;
- end

```
sequence s_ab;
  a ##1 b;
endsequence

sequence s_cd;
  c ##2 d;
endsequence

property p_expr;
  @(posedge clock) s_ab ##1 s_cd;
endproperty

assert property (p_expr);
endmodule
```

Explanation for concurrent assertion

- The first sequence s_ab validates that b is high the next clock when a is high, and the second sequence s_cd validates that d is high 2 clocks after c is found high. The property asserts that the second sequence is on the next cycle after the first sequence.

Operator	Use
[*n] [*m:n]	Continuous repetition operator. The expression repeats continuously for the specified range of cycles.
[->n] [->m:n]	Go to repetition operator. Indicates there's one or more delay cycles between each repetition of the expression. a[->3] is equivalent to (!a[*0:$] ##1 a) [*3]
[=n] [=m:n]	Non-consecutive implication. a[=3] is equivalent to (!a[*0:$] ##1 a ##1 !a[*0:$]) [*3]

Operators and its usage

Operator	Use
##n ##[m:n]	Delay operators - Fixed time interval and Time inteval range
!, ||, &&	Boolean operators
|->	Overlapping implication
|=>	Nonoverlapping implication
not, or, and	Property operators

In-built functions and its usage

Function	Use
$rose	returns true if the LSB of the expression changed to 1. Otherwise, it returns false.
$fell	returns true if the LSB of the expression changed to 0. Otherwise, it returns false.
$stable	returns true if the value of the expression did not change. Otherwise, it returns false.
$past(expression, num_cycles)	Returns value of expression from num_cycles ago
$countones	Returns the number of 1s in an expression
$onehot	Returns true if exactly one bit is 1. if no bits or more than one bit is 1, it returns false.
$onehot0	Returns true is no bits or just 1 bit in the expression is 1
$isunknown	Returns true if any bit in the expression is 'X' or 'Z'

Implication Operators

- 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;

Example

- sequence request
- Req;
- endsequence

- sequence acknowledge
- ##[1:2] Ack;
- endsequence

- property handshake;
- @(posedge Clock) request |-> acknowledge;
- endproperty

- assert property (handshake);

Direct and Randomized tests

- Verification engineers will first create something known as a verification plan that details every feature of the design required to be tested in RTL simulations and how each test will create independent scenarios that target a particular feature.

- Complex designs have a lot of scenarios and many corner cases that are better verified by randomized tests and take much less effort and time. For example, a test will configure the peripheral having registers with random values every time the test is run with a different seed thereby achieving different scenarios for every run. This will ensure that we hit corner cases and uncover any hidden bugs.

System Verilog Constraints

- System Verilog allows users to specify constraints in a compact, declarative way which are then processed by an internal solver to generate random values that satisfy all conditions. Basically constraints are nothing more than a way to let us define what legal values should be assigned to the random variables. A normal variable is declared to be random by the keyword rand.
  - class packet;
  - rand bit [7:0] addr;
  - rand bit [7:0] data;
  - constraint addr_limit { addr <= 8'hB; }
  - endclass
- Types of constraints:
  - Common Constraints: constraint addr_range { addr > 6; }
  - inside constraint: constraint addr_range { addr inside {[6:100]}; }
  - Implication Constraint: constraint addr_range { addr == 2 -> var > 10; }
  - foreach Constraint: constraint addr_range { foreach (addr[i]) {addr[i] == i;} }
  - solve before Constraint: constraint addr_range { a -> b == 3'h3; solve a before b; }
  - Soft constraints: constraint addr_range { soft addr > 6; }
  - Inline constraints: packet.randomize() with { addr < 10; }
  - Disable constraints: packet.addr_range.constraint_mode(0);
  - Disable randomization: packet.addr_range.rand_mode(0);

How to write constraints?

- class packet;
- rand int i;
- constraint c_legal_value {
- i > 0;
- i < 100;
- }
- constraint c_by_5 {
- (i%5) == 0;
- }

```
constraint c_dist {
  i dist {
    [0:10] := 2,
    [11:20] := 4,
    [21:30] := 6,
    [31:40] := 8,
    [41:50] := 10,
    [51:60] := 10,
    [61:70] := 8,
    [71:80] := 6,
    [81:90] := 4,
    [91:100] := 2
  };
 };
endclass
```

Explanation

- The example above declares a class called packet with a constraint on its address field. Note that both its properties are prefixed with the rand keyword which tells the solver that these variables should be randomized when asked to. The constraint is called addr_limit and specifies that the solver can allot any random value for the address that is below or equal to 8'hB. Since, the 8-bit variable addr is of type but, it can have any value from 0 to 255, but with the constraint valid values will be limited to 11.
- This powerful feature will allow is to create variables that are constrained within ranges that are valid for the design and will have a much better verification impact.

Covergroups and coverpoints in System Verilog

- covergroup can be defined in either a package, module, program, interface, or class and usually encapsulates the following information:
  - A set of coverage points
  - Cross coverage between coverage points
  - An event that defines when the covergroup is sampled
  - Other options to configure coverage object
- bit [1:0] mode;
- covergroup cg @ (posedge clock);
- label: coverpoint mode;
- endgroup
- A covergroup can contain one or more coverage points. Evaluation of the coverpoint expression happens when the covergroup is sampled. The covergroup is specified to be sampled at every occurence of a positive edge of the clock.mode can have values from 0 to 3, since mode is of 2 bits.
- The question of coverage is: How much of the total values each variable can have has actually happened?

How to write covergroup for functional coverage inside packet?

- class packet;
- rand bit [3:0] mode;
- rand bit [1:0] key;
- function display();
- $display();
- endfunction

```
covergroup CovGrp;
   coverpoint mode {
     bins featureA = {0};
     bins featureB = {[1:3]};
     bins common [] = {4:$};
     bins reserve = default;
   }
   coverpoint key;
 endgroup
endclass
```

How to write covergroup for functional coverage in separate class?

- class myCov;
- covergroup CovGrp;
- endgroup
- function new();
- CovGrp = new;
- endfunction
- endclass

```
module tb;
  myCov coverage_inst = new();
  initial begin
    coverage_inst.CovGrp.sample();
    covergoup CovGrp @ (posedge clk); // Sample coverpoints at
posedge clk
    covergroup CovGrp @ (eventA); // eventA can be triggered
with -> eventA;

    #1 if(reset == 0)
      coverage_inst.stop();
    #10 if(reset = 1)
      coverage_inst.start();
  end
endmodule
```

How to write covergroup and how to use different options?

- options:
- weight = 1;
- name = "This is a name";
- per_instance = boolean; // default value = false/0
- goal = 90;
- comment = "This is a comment";
- at_least = 1;
- detect_overlap = boolean; // default value = false/0
- auto_bin_max = 64;
- cross_auto_bin_max = integer value;
- cross_num_print_missing = 0;

- type_options:
- strobe = 0;

```
covergroup c_group;
   option.per_instance = 1;
   option.comment = "This is the comment";

   cp1: coverpoint addr {
     option.weight = 2;
     option.auto_bin_max = 32;
   }
   cp2: coverpoint data;
   cp1_X_cp2: cross cp1, cp2 {
     option.cross_auto_bin_max = 32;
   }
  endgroup : c_group
```

Types of coverage bins in System Verilog

- implicit bins
  - if you do not specify any bins, then Implicit bins are created. The number of bins creating can be controlled by auto_bin_max parameter.
- explicit bins
  - Not all values are interesting or relevant in a cover point, so when the user knows the exact values he is going to cover, he can use explicit bins. You can also name the bins.
- ignore bins
  - A set of values or transitions associated with a coverage-point can be explicitly excluded from coverage by specifying them as ignore_bins.
- illegal bins
  - A set of values or transitions associated with a coverage-point can be marked as illegal by specifying them as illegal_bins. All values or transitions associated with illegal bins are excluded from coverage. If an illegal value or transition occurs, a runtime error is issued.

Transition bins

- Transitional functional point bin is used to examine the legal transitions of a value. SystemVerilog allows to specifies one or more sets of ordered value transitions of the coverage point.
- Type of Transitions:
  - Single Value Transition
  - Sequence Of Transitions
  - Set Of Transitions
  - Consecutive Repetitions
  - Range Of Repetition
  - Goto Repetition
  - Non Consecutive Repetition

Wildcard bins

- By default, a value or transition bin definition can specify 4-state values.
- When a bin definition includes an X or Z, it indicates that the bin count should only be incremented when the sampled value has an X or Z in the same bit positions.
- The wildcard bins definition causes all X, Z, or ? to be treated as wildcards for 0 or 1 (similar to the ==? operator).
- For example:
  - wildcard bins g12_16 = { 4'b11?? };

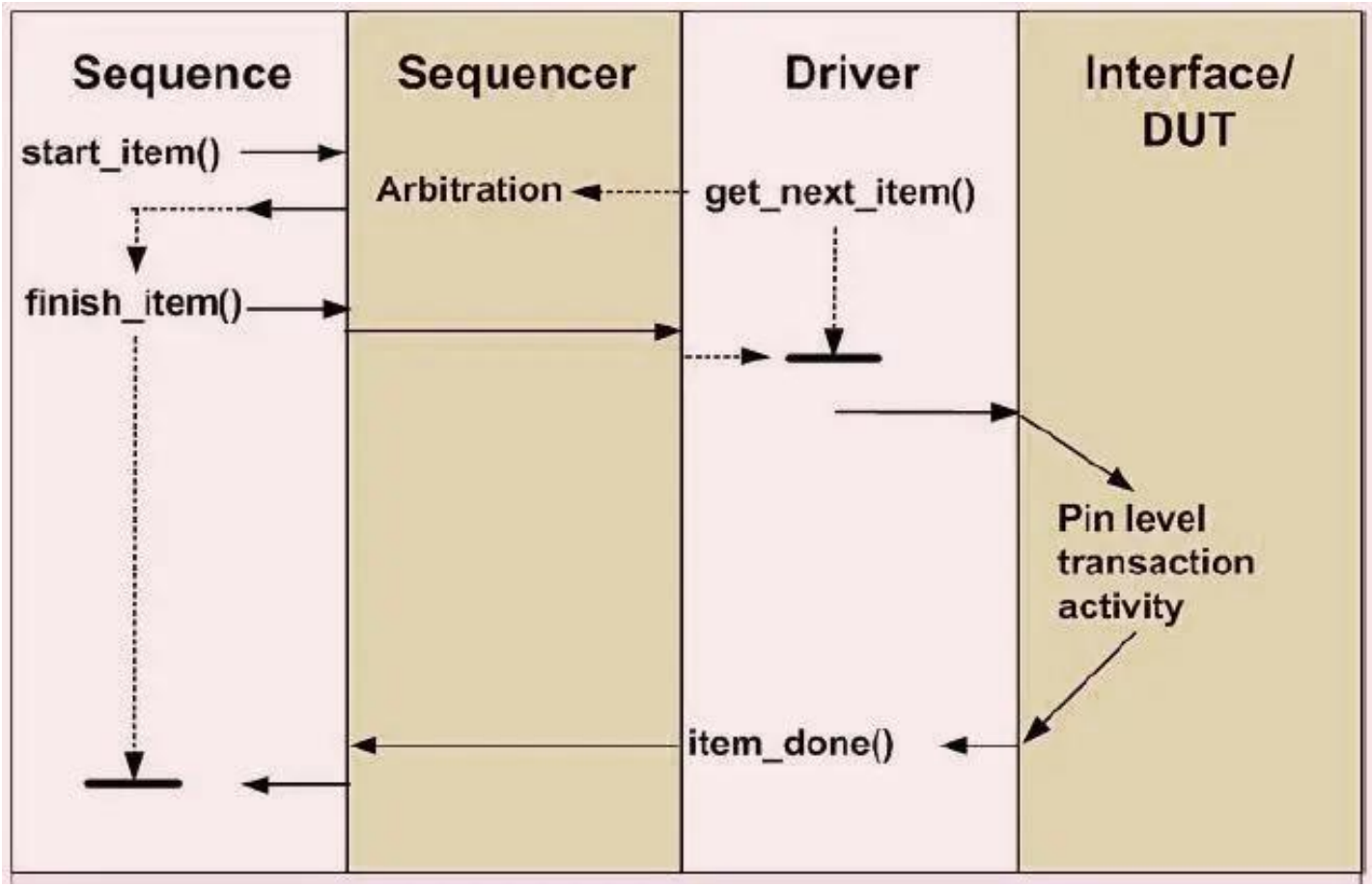
Types of Code Coverage

- There are different categories of code coverage.
  - Statement coverage
  - Block coverage
  - Conditional/Expression coverage
  - Branch/Decision coverage
  - Toggle coverage
  - FSM coverage

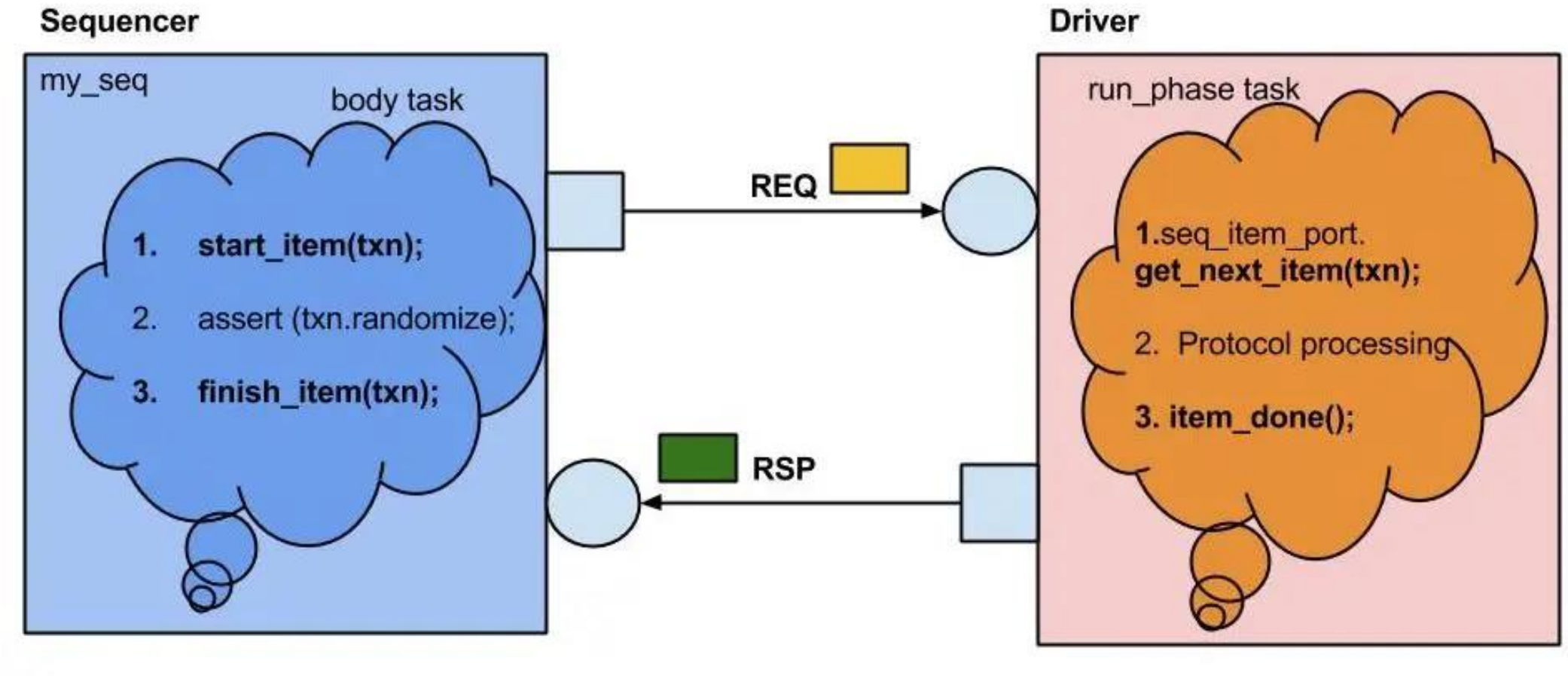
Code vs Functional Coverage

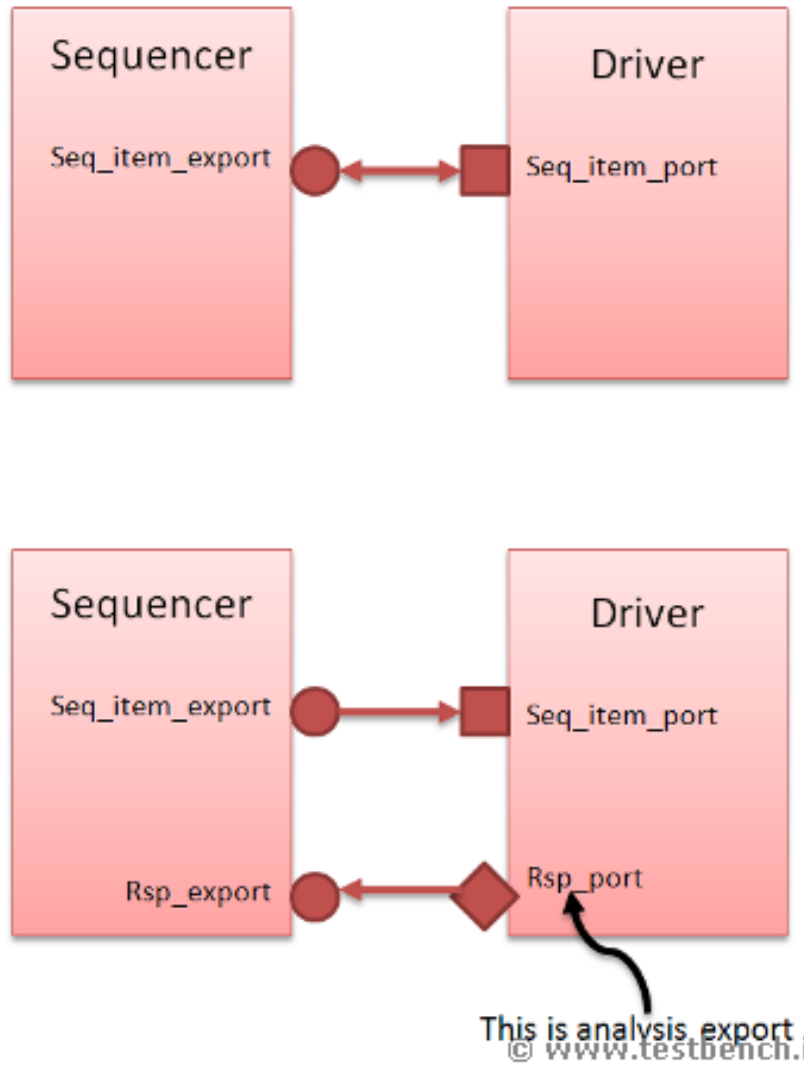
Code Coverage	Functional Coverage
Tells how well HDL code has been exercised by your test bench	Measures how well the functionality of the design has been covered by your test bench. User has to define the functionality to be measured through coverage.
Verifies completeness of verification environment interms of hitting expression lines etc. of RTL code	Verifies completeness of verification environment as per the requirements spec and also functional coverage points
Does not use design specification	Use design specification
Support in all languages	Verilog does not support functional coverage. To do functional coverage, SystemVerilog, Specman E or Vera are needed.

Driver and Sequencer Communication in UVM

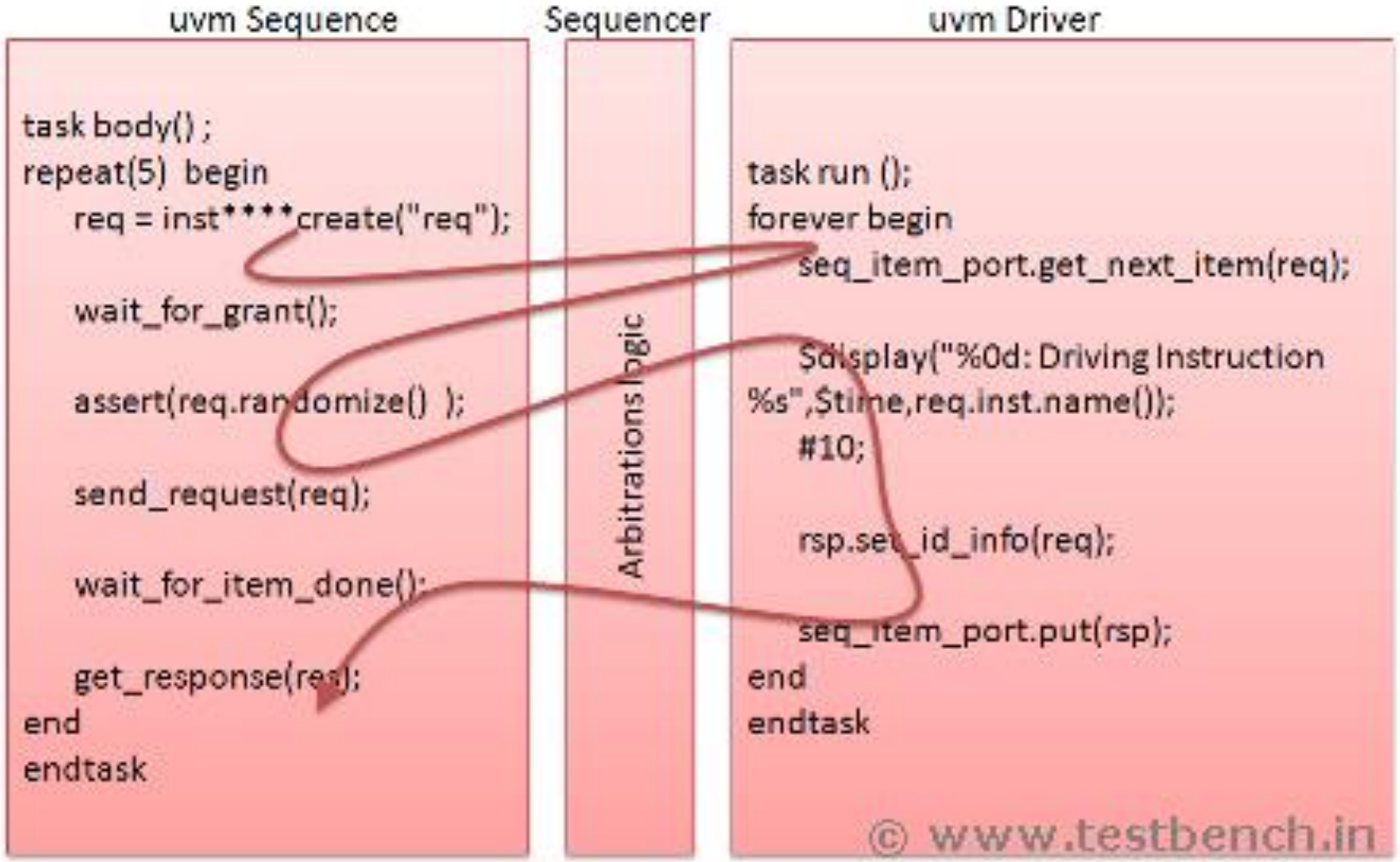


Simpler explanation for sequencer and driver communication

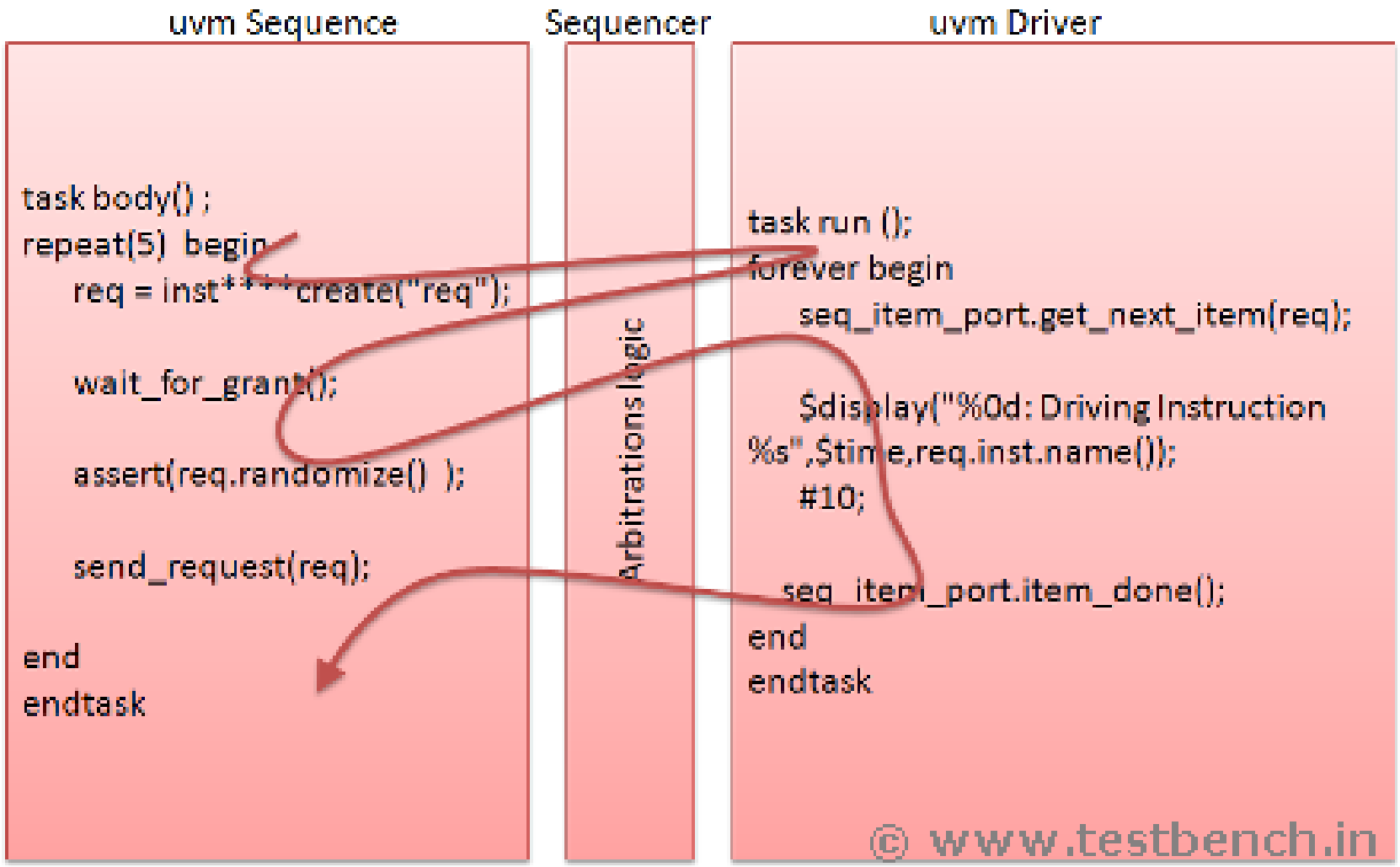



Sequencer
Seq_item_export
Driver
Seq_item_port
Sequencer
Seq_item_export
Rsp_export
Driver
Seq_item_port
Rsp_port
This is analysis export
© www.testbench.in

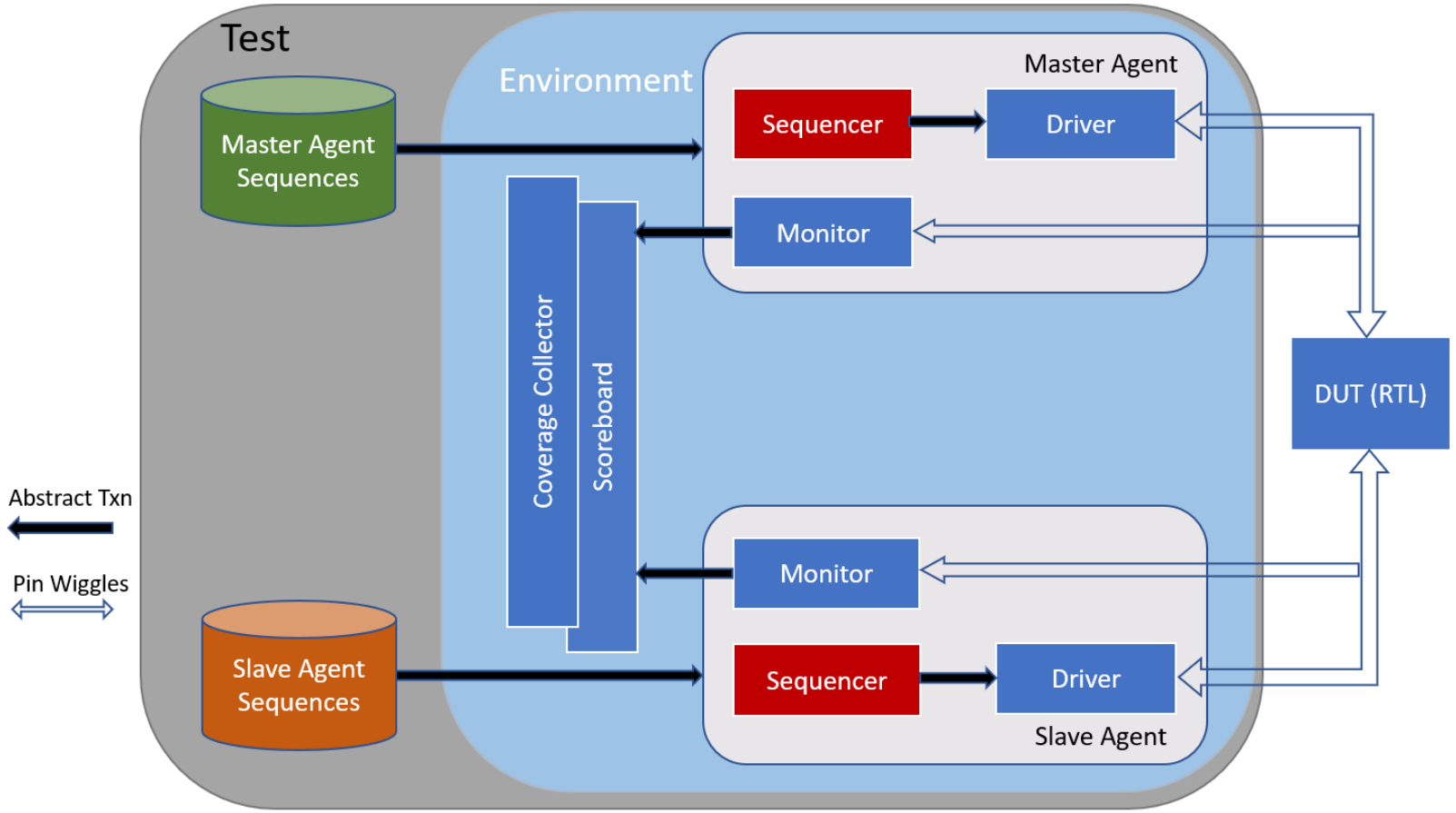
With grant and get_response()



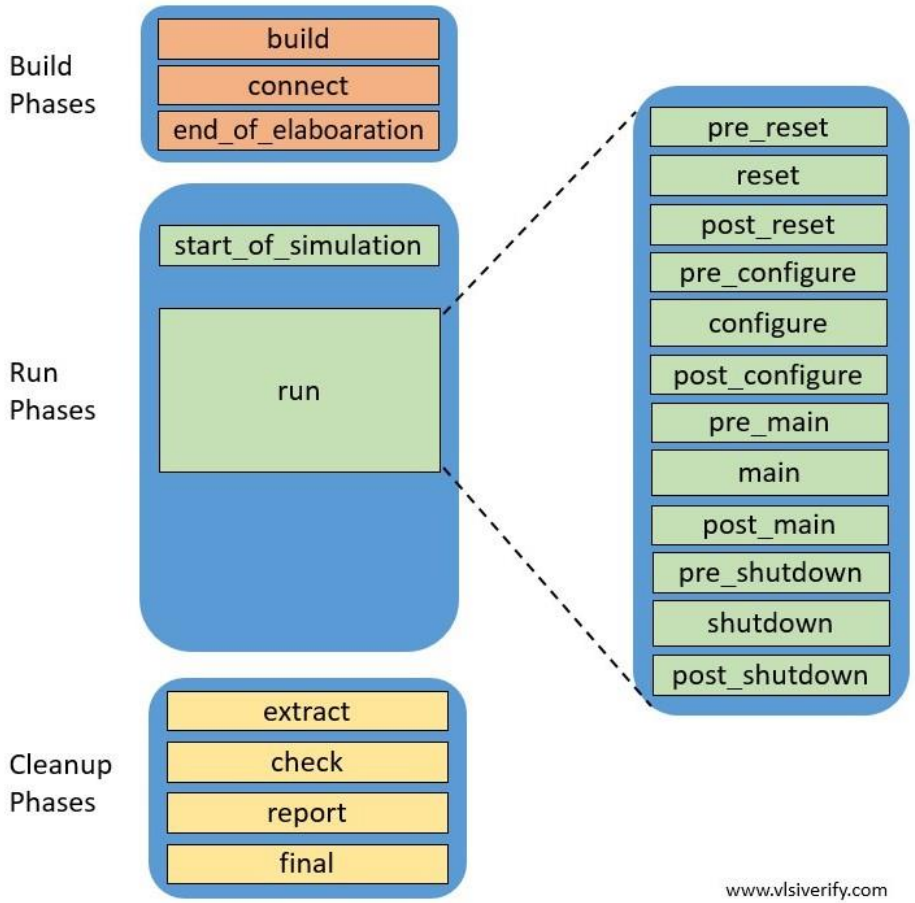
With grant and send_request()



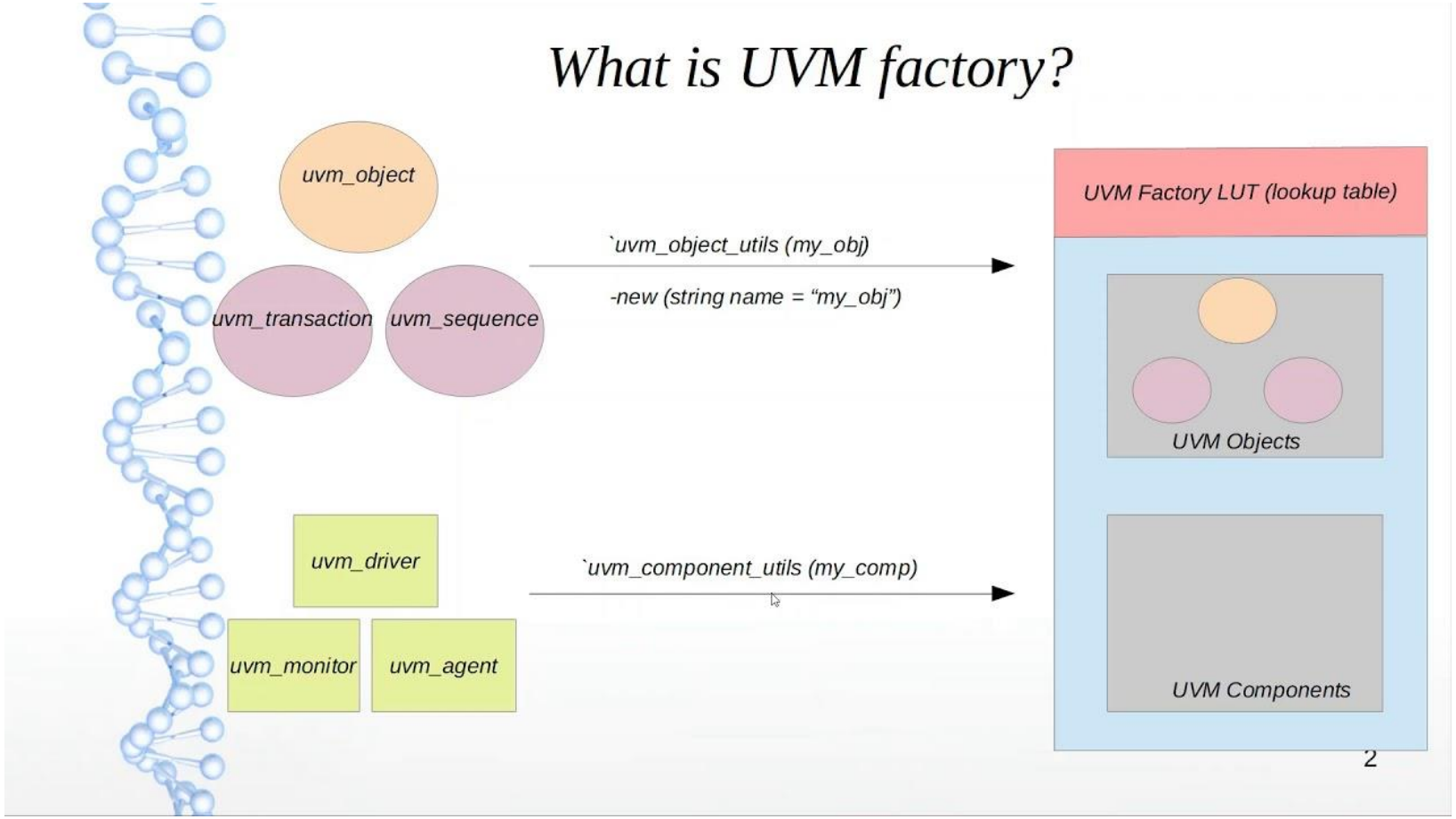
Architecture with multiple agents



UVM Phases


Build Phases
build
connect
end_of_elaboaration
Run Phases
start_of_simulation
run
pre_reset
reset
post_reset
pre_configure
configure
post_configure
pre_main
main
post_main
pre_shutdown
shutdown
post_shutdown
Cleanup Phases
extract
check
report
final
www.vlsiverify.com

UVM Factory


What is UVM factory?
uvm_object
uvm_transaction
uvm_sequence
`uvm_object_utils (my_obj)
-new (string name = "my_obj")
UVM Factory LUT (lookup table)
UVM Objects
uvm_driver
uvm_monitor
uvm_agent
`uvm_component_utils (my_comp)
UVM Components
2

UVM Factory overrides

- UVM Factory facilitates an object of one type to be substituted with an object of derived type without having to change the structure of the Testbench or modify the Testbench code. This behavior is called “overriding” and there are following types of overriding is possible with UVM Factory:
  - Type Overriding(set_type_override)
  - Instance Overriding(set_inst_override)

Factory Registration

- Every UVM component or sequence/transactions must have Factory registration & this registration can be done by using Factory registration macros
- For example:
  - `uvm_component_utils(axi_driver)
  - `uvm_object_utils(packet)

Default Constructor

- As we know that uvm_component and uvm_object constructors are virtual methods hence user have to follow their prototype template. As we know that in UVM components/objects are constructed during build phase but Factory constructor should contain default arguments in the definition of the components/objects. This allows Factory registered component/object to be created inside Factory initially & later to be re-assigned to the class properties passed via the create() command as arguments. The default arguments are different for components and objects.
- For example:
- function new (string name = "axi_txn");
- super.new(name);
- endfunction

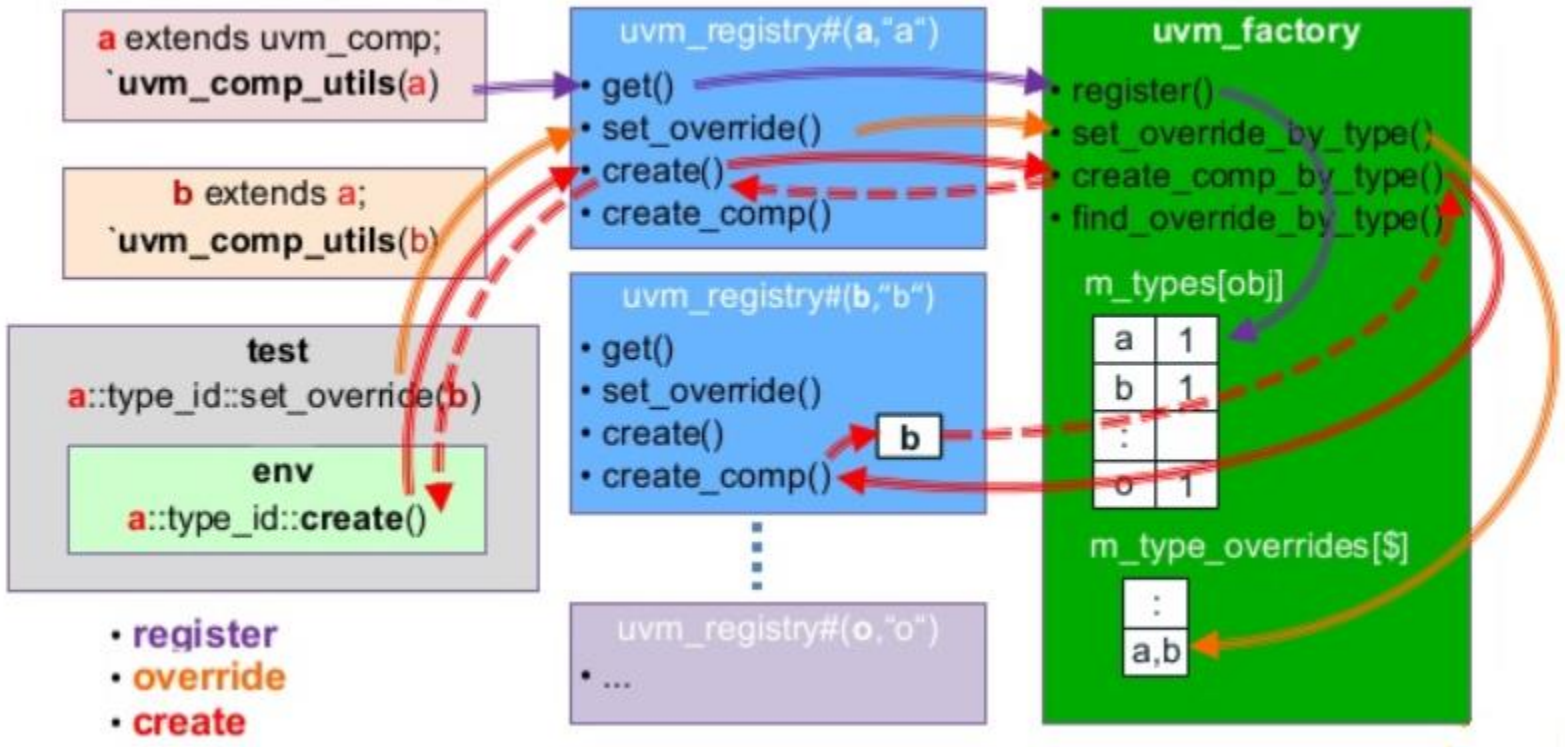
Component & Object Creation

- Using Factory, hierarchically lower components/objects are created by the immediate higher components/objects. Hence, the next goal would be the Factory supported component and object creation code entry for the child components inside the parent components and objects.
- In order to allocate the memory for an object of a class, we call the **new** method. But, we should use **create** method to create the objects in the build_phase of the UVM Simulation. The Create method is the static method uvm_component_registry. It first constructs the class, and assigns the name and parent arguments correctly and assigns the pointer of the class to its declaration handle. The create method is different for Component creation and object creation.

Factory vs Callbacks

UVM Factory	UVM Callbacks
Cannot change the behavior once the object is built.	Can add or remove functionality after the object has been built.
Not Suitable for VIP Components.	The callbacks are more popular with VIPs.
Not much planning is required if the modularity is maintained factory can be used easily provided that all components are created using factory.	Callbacks have to be planned and placed at strategic locations in the code.
Suitable when feature enhancement is a major one.	Suitable for minimal feature addition.
Multiple copies are required.	Easy to Maintain.

Concept of UVM factory and UVM Configuration



Theory about m_sequencer and p_sequencer

- In SystemVerilog based OVM/UVM methodologies, the sequence is an object with limited life time unlike a component which has a lifetime through out simulation.
- The sequence is created , started and once done, de-referenced from memory and this can be any time in the duration of a test.
- So if you want to access anything from the testbench hierarchy (which are components) - the sequence would need a handle to the sequencer on which the sequence is running. (Note that sequencer is a component)
- **m_sequencer** is a handle of type uvm_sequencer_base which is available by default in a sequence.
- The real sequencer on which a sequence is running would normally be derived from the uvm_sequencer_base class.
- Hence to access the real sequencer on which sequence is running , you would need to typecast the m_sequencer to the physical sequencer which is generally called **p_sequencer**

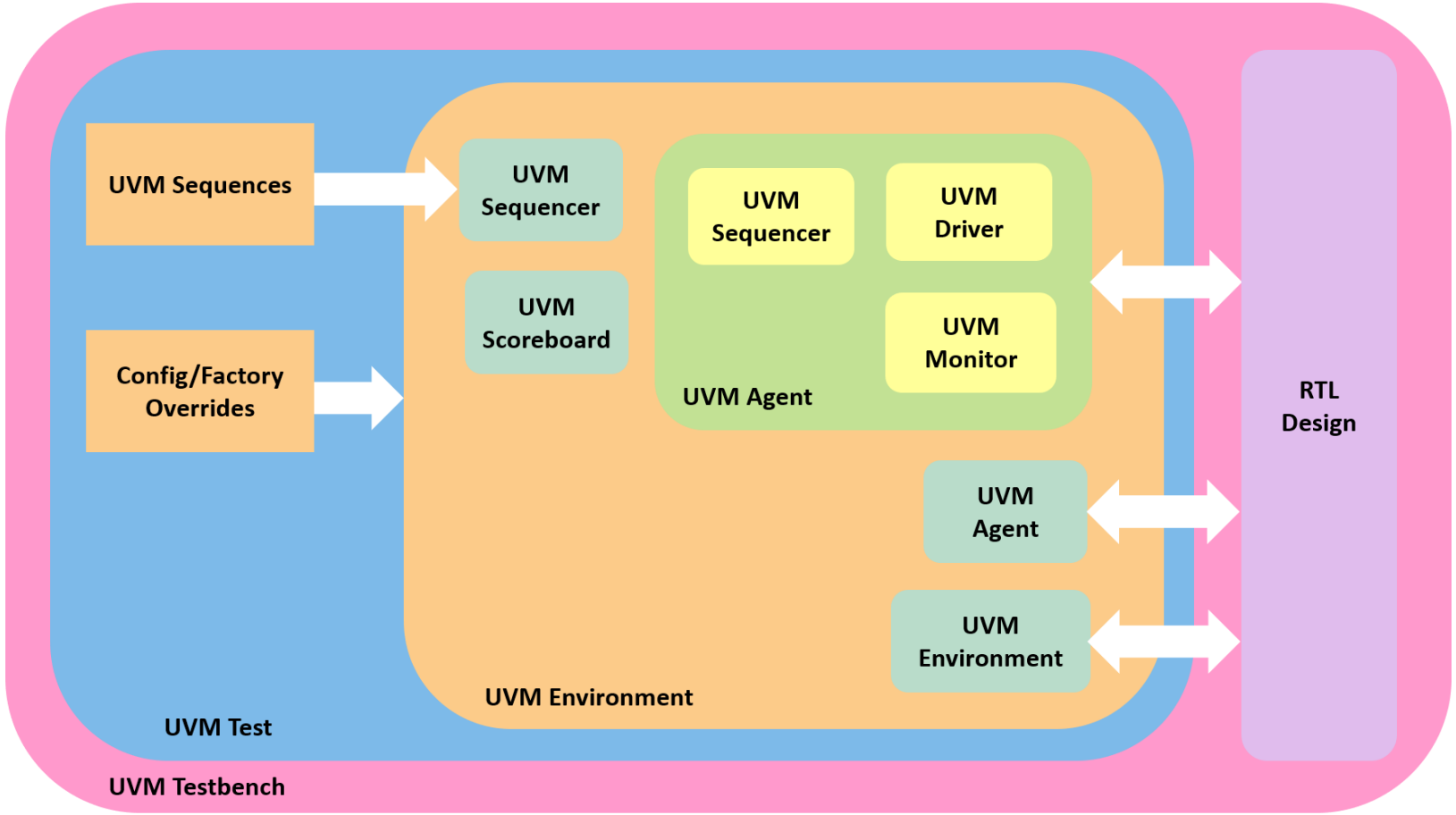
m_sequencer and p_sequencer

- m_sequencer is the generic uvm_sequencer pointer. it will always exist for the uvm_sequence and is initialized when the sequence is started. p_sequencer is a typed-specific sequencer pointer, created by registering the sequence to the sequencer using macros.
- The m_sequencer handle contains the reference to the sequencer(default sequencer) on which the sequence is running. This is determined by:
  - the sequencer handle provided in the start method
  - the sequencer used by the parent sequence
  - the sequencer that was set using the set_sequencer method
- The p_sequencer is a variable, used as handle to access the sequencer properties. p_sequencer is defined using the macro `uvm_declare_p_sequencer(SEQUENCER_NAME)

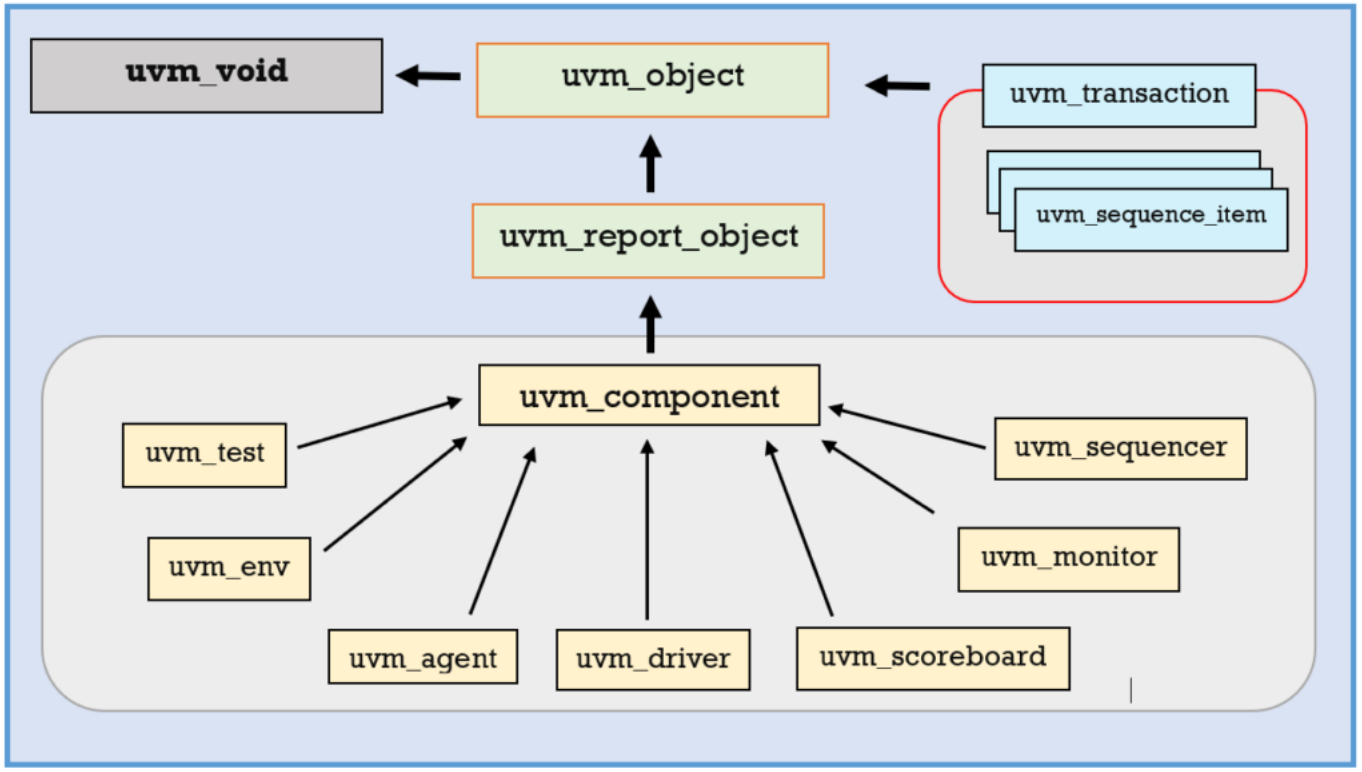
How to use m_sequencer and p_sequencer?

```
//A test_sequencer class derived from base UVM sequencer
//Lets say, it has a clock monitor component to access clock.
class test_sequencer_c extends uvm_sequencer;
clock_monitor_c clk_monitor;
endclas

//Here is a test sequence that runs on the test_sequencer
//Lets say, sequence need access to sequencer to get access to clock monitor

class test_sequence_c extends uvm_sequence;

test_sequencer_c p_sequencer;
clock_monitor_c my_clock_monitor;

task pre_body()
//Typecast the m_sequencer base type to p_sequencer
if(!$cast(p_sequencer, m_sequencer)) begin
`uvm_fatal("Sequencer Type Mismatch:", " Worng Sequencer");
end
//get access to clock monitor
my_clock_monitor = p_sequencer.clk_monitor;
endtask


endclass
```

UVM Adoption and Usage



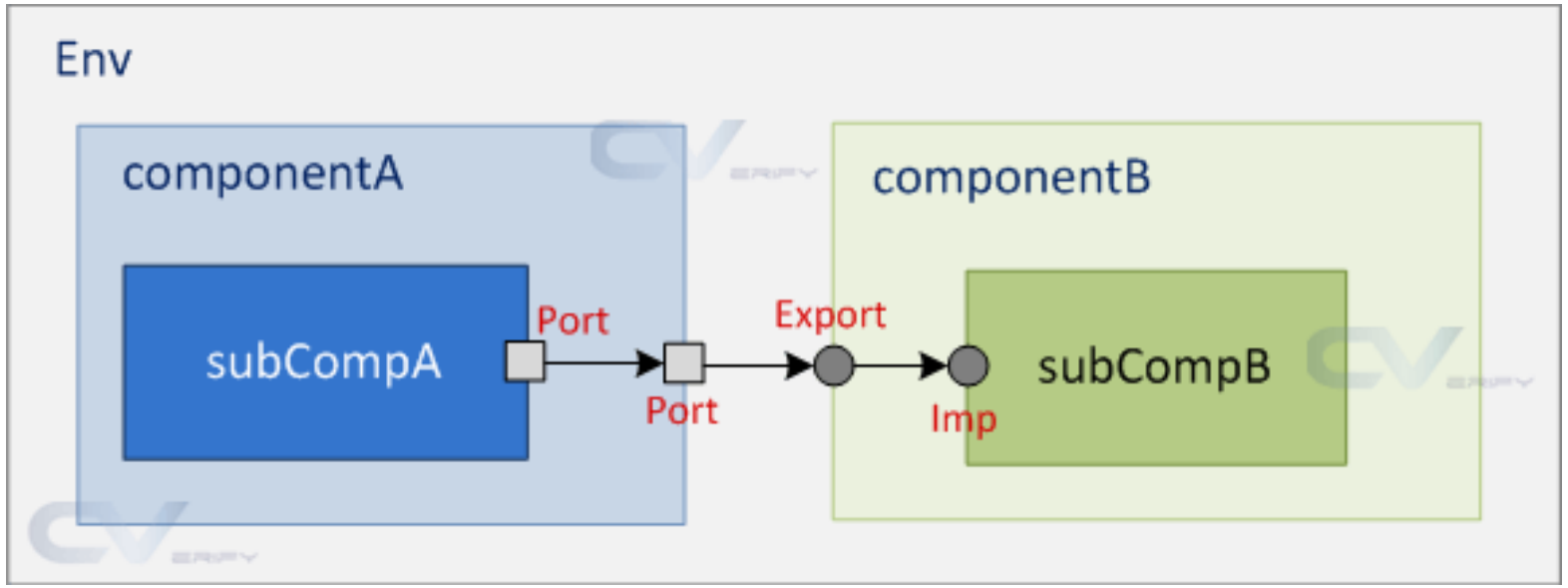
UVM components



What is the need for a virtual interface in SystemVerilog?

- SystemVerilog interface is static in nature, whereas classes are dynamic in nature. because of this reason, it is not allowed to declare the interface within classes, but it is allowed to refer to or point to the interface. A virtual interface is a variable of an interface type that is used in classes to provide access to the interface signals.

Analysis Port, Analysis Export, and Analysis implementation



Basic structure of UVM

- Testbench.svh is module that creates an object of test class passes on interface pointer and start simulation using run_test() call.

- Include all class definition, UVM packages, and macros.

- Clock generation logic
- Instantiate interface and pass to test top as virtual interface
- run_test()

Test.svh

- Instatiate environment (ennv), config, and stimulus.

- Build Phase:
- Construct Env and Config
- Get virtual interface handle from TB and pass it to env

- Run Phase:
- Construct Sequence
- Start Test
- Pass sequence (seq) to sequencer (seqr)
  - -raise objection
  - -seq.start(seqr)
  - -drop objection

Environment and Sequence

- Env.svh:
- Instantiate Agent.
- Build Phase
- -Pass virtual interface handle to agent after getting it from Test.
- Add connect phase if scoreboard is present

- Sequence.svh:
- Parameterize to type "sequence_item(seq_item)"
- task body(): once sequence started it gets executed.
- -instantiate seq_item
- -construct seq_item
- -start_item(seq_item);
- -randomize();
- -finish_item(seq_item);

Sequence item and Agent

- seq_item.svh:
- Declare all transaction variables
- Implement "do_copy", "convert2string" functions

- Agent.svh:
- Instantiate all components to be present in agent like driver, sequencer, and monitor.
- Build Phase
  - -construct all sub components instantiated above
  - -Get virtual interface from env and pass it to all the sub components.
- Connect Phase
  - -connect driver and sequencer port to export

Sequencer and Driver

- Sequencer.svh:
- Parameterize to type "seq_item"
- Build phase
- -Get virtual interface from agent

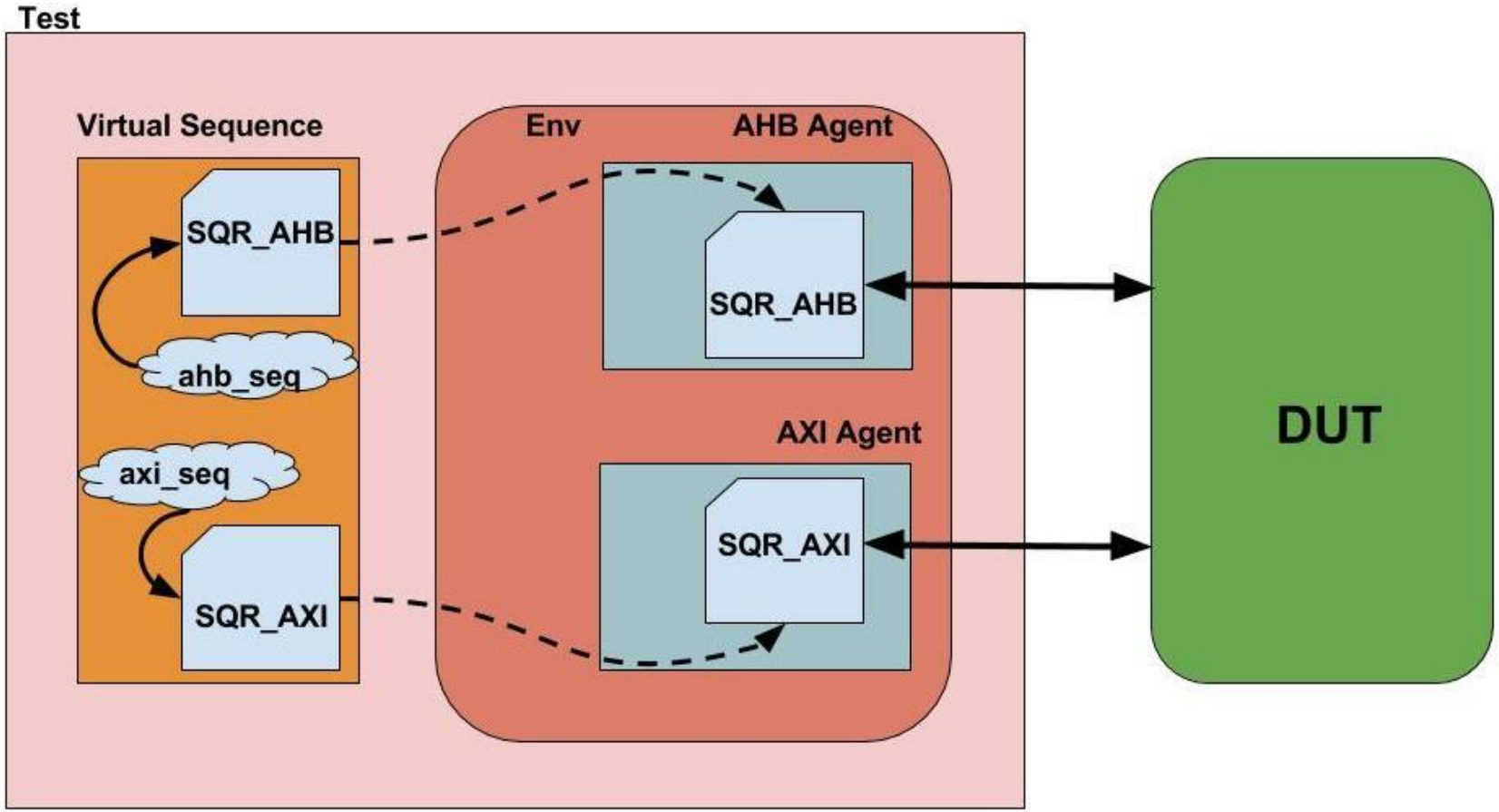
- Driver.svh:
- Build Phase
- -get virtual interface from agent (parent) or config database
- Run Phase:
- -forever begin
- -get_next_item(): (get seq_item from sequencer)
- -item_done(); (handshake done from driver to sequencer)
- -end
- Other API's like "peek()", "try_get()", and "put()" could be used

Monitor and config file

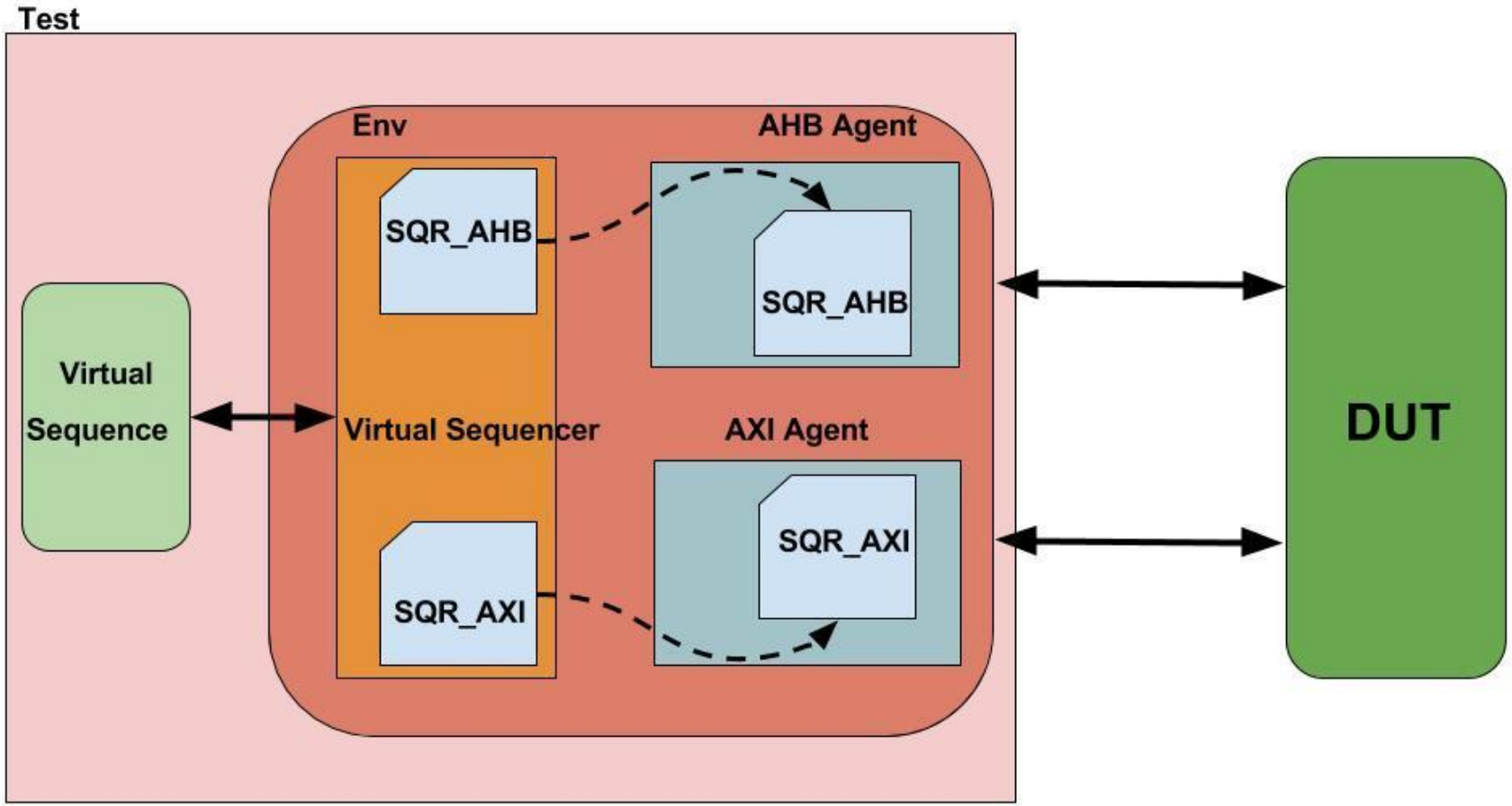
- Monitor.svh:
- Analysis port: monitor writes transaction objects to this port once get detected at the interface.
- Construct analysis port
- Build Phase:
- -get virtual interface handle from agent (parent) or config database.
- Run Phase:
- -Code functionality
- -construct transaction objects (seq_items)
- -tap signal at the interface through mod port.
- -write transactions to analysis port.

- Config.svh: if needed

Virtual Sequence in UVM



Virtual Sequencer in UVM



Abbreviations

- API – Application Programming Interface
- BFM – Bus-Functional Model
- DUT – Design Under Test
- OOP – Object-Oriented Programming
- OVM – Open Verification Methodology (a predecessor of UVM)
- TLM – Transaction Level Modeling (or Transaction Level Model)
- UVM – Universal Verification Methodology
- VIP – Verification Intellectual Property

UVM Glossary

- Agent (class uvm_agent) — a component that contains one sequencer, one driver, and one monitor, and which senses and drivers the signals of one SystemVerilog interface.
- Analysis port (class uvm_tlm_analysis_port) — a specific type of transaction-level port that can be connected to zero, one, or many analysis exports and through which a component may call the method write implemented in another component, specifically a subscriber.
- Analysis export (class uvm_tlm_analysis_export) — a specific type of export that receives an incoming transaction stream from an analysis port.
- Analysis imp (class uvm_tlm_analysis_imp) — a specific type of imp that receives an incoming analysis transaction stream. The component class that instantiates an analysis imp must define the method write required by that imp.
- Barrier (class uvm_barrier) — an object that causes a set of processes to be blocked until a specified number of processes reach the barrier, at which point all processes waiting at the barrier are allowed to proceed.

Continued…

- Build phase — a pre-defined phase and a method of class uvm_component that is called during the phase of that name. The method build_phase may be overridden to get and set information in the configuration database and to create lower-level objects, including ports, exports, and child components.
- Callback — a method of a class that can be overridden by the user when extending that class and which is called automatically by the UVM library under certain well-defined circumstances. UVM offers several distinct categories of callback, including callbacks for sequences, events, objections, registers, register fields, the report catcher, and user-defined callbacks.
- Checking — the process by which the verification environment checks the functional correctness of the DUT, either using either procedural code within a UVM component or using concurrent assertions within a SystemVerilog interface.
- Component (class uvm_component) — the structural building block of a UVM verification environment, conceptually equivalent to a Verilog module though implemented as an object in UVM. Components are constructed during the build phase, connected to other components using ports and exports, and able to be overridden (i.e. replaced) using the factory during the build phase. Components typically override the phase methods that are executed during the standard UVM phases.
- Configuration database — a repository for storing the configuration information used to parameterize the verification environment.

Continued…

- Configuration object — an object, associated with a component and stored in the configuration database, which is used to configure the behavior or structure of its associated component. A configuration object is constructed and initialized by a module, test or env and is read by its associated component during the build phase.
- Connect phase — a pre-defined phase and a method of class uvm_component that is called during the phase of that name. The method connect_phase may be overridden to make connections between ports and exports. The connect_phase methods are called bottom-up.
- Coverage — data collected during simulation and used to determine when verification is complete. Function coverage is user-defined, structural coverage depends solely on the structure of the source code. Functional coverage data is typically collected using the SystemVerilog covergroup statement within subscribers or scoreboards or using the SystemVerilog cover property statement within SystemVerilog interfaces.
- Driver (class uvm_driver) — a component that receives transactions from a sequencer and that actively or reactively drives the signal-level interface of the DUT as directed by the contents of those transactions. A driver may also send information from the signal-level interface back upstream to the sequencer, either by modifying the request object or using a separate response object.
- Env (class uvm_env) — a component that instantiates one or more agents or lower-level envs and is used to add a level of component hierarchy within the verification environment. The top-level env is instantiated from the test.

Continued…

- Export — an object that allows a component to receive an incoming transaction stream that originates in another component. An export must receive its transaction stream either from a port on another component or from an export on a parent component, and must send transactions forward either to an export on a lower-level component or to an imp on a lower-level component.
- Factory (class uvm_factory) — a UVM mechanism that allows a component, transaction, or sequence object to be replaced with an object of another type at the time it is created, thereby providing a general mechanism by which the behavior or structure of a class representing VIP can be modified as it is used.
- Imp — an object that allows a component to implement the methods associated with an incoming transaction stream. Whereas an export must be connected to a lower-level export or imp, the component class that instantiates an imp must define any methods required by that imp.
- Monitor (class uvm_monitor) — a passive component that senses the signal-level interface of the DUT (or, alternatively, that receives transactions from lower layers of the verification environment), gathers data into transactions, and sends those transactions out through an analysis port to the verification environment for checking and coverage collection.
- Phase (class uvm_phase) — a UVM mechanism that subdivides execution into various predefined phases, the purpose of which it to allow verification components to agree on when to build components, connect ports, run simulation, reset the DUT, and so forth.

Continued…

- Port — an object that allows a component to send an outgoing transaction stream to another component. A port must be connected either to an export on another component or to a port on a parent component.
- Register model (class uvm_reg_block) — an object instantiated between sequences or scoreboards and agents that abstracts certain details of the control and status registers in the DUT, thus permitting sequences or scoreboards to access registers by name without having to know their address, endianness, or whereabouts in the DUT.
- Request — a transaction object created by a sequence and sent to a driver.
- Response — a transaction object sent from a driver back to a sequence. The response object may be the request object itself or may be a new object created by the driver.
- Run phase — a pre-defined phase and a method of class uvm_component that is called during the phase of that name. The run_phase method is the only common phase that is a task and may therefore consume time, the other methods being functions.

Continued…

- Sequence (class uvm_sequence) — an object that generates transactions or starts other sequences. A sequence class contains a user-defined task body that is called when the sequence is started. The task body does the work of the sequence. A sequence that directly generates transactions must always execute on a sequencer. A sequence indicates its readiness to generate a transaction by calling method start_item, and delivers the transaction by calling method finish_item. A sequence may retrieve a response to a transaction by calling method get_response. A sequence may be synchronized with other parts of the verification environment using barriers, callbacks, or events.
- Sequencer (class uvm_sequencer) — a component that coordinates and arbitrates between transactions generated by sequences running on that component and that sends those transactions downstream to a single driver or to another sequencer. A sequencer performs arbitration when multiple sequences running on that same sequencer attempt to generate transactions in parallel.
- Scoreboard (class uvm_scoreboard) — a component that receives transactions from multiple agents and typically performs end-to-end checking of DUT functionality, though it may also collect functional coverage information. A scoreboard may or may not incorporate a reference model of DUT functionality.
- Starting phase (methods set_starting_phase, get_starting_phase) — the phase in which a sequence is started. From uvm-1.2 onward, the starting phase can only be accessed using the methods set_starting_phase and get_starting_phase in order to prevent the value from being modified after it has been read.
- Subscriber (class uvm_subscriber) - a component that contains exactly one analysis imp and that implements the method write associated with that analysis imp to process an incoming transaction stream.

Continued…

- Test (class uvm_test) — the top-level user-defined UVM component in the component hierarchy. The test object is instantiated implicitly from uvm_top when method run_test is called. A test may modify the behavior or structure of the verification environment in order to increase functional coverage.
- Transaction (class uvm_sequence_item) — an object that represents a communication abstraction such as a handshake, bus cycle, or data packet. A transaction class contains user-defined data members that represent the properties of the protocol, and user-defined methods to copy, compare, print, pack, unpack, and record those members. Transactions are typically generated by a sequence and passed to a driver or generated by a monitor and passed to zero or more subscribers. Transactions are passed between components using ports and exports.
- Virtual sequence (class uvm_sequence) — a sequence that does not itself generate transactions but that typically starts other sequences on other sequencers. By this definition, because a virtual sequence does not itself interact with a sequencer, it can be started without a sequencer. Virtual sequences are typically used to coordinate the behavior of multiple agents (though strictly speaking a sequence that coordinates multiple agents need not be a virtual sequence).
- Write (method) — a method that is called through an analysis port and implemented within a component that has an analysis imp. The implementation of the method write typically performs checking or functional coverage collection.

Start with UVM_Phases code example

- http://www.testbench.in/UT_02_UVM_TESTBENCH.html

- Download the source code
  - uvm_phases.tar

Thank You ☺