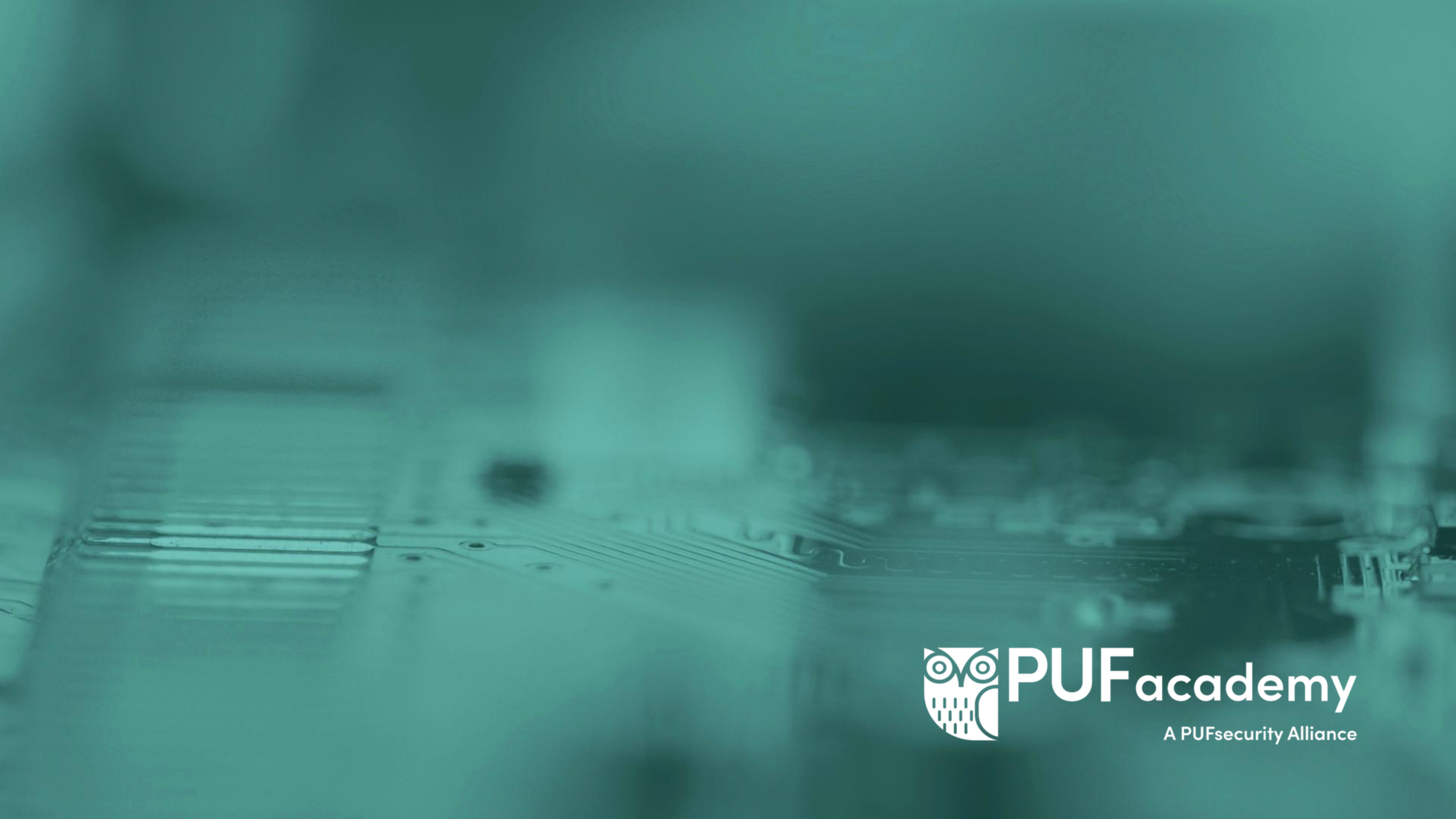


PUFacademy

A PUFsecurity Alliance

Digital Logic Design
- Lecture 5
- Testbench .

2025 Spring

PUFacademy

A PUFsecurity Alliance

Agenda

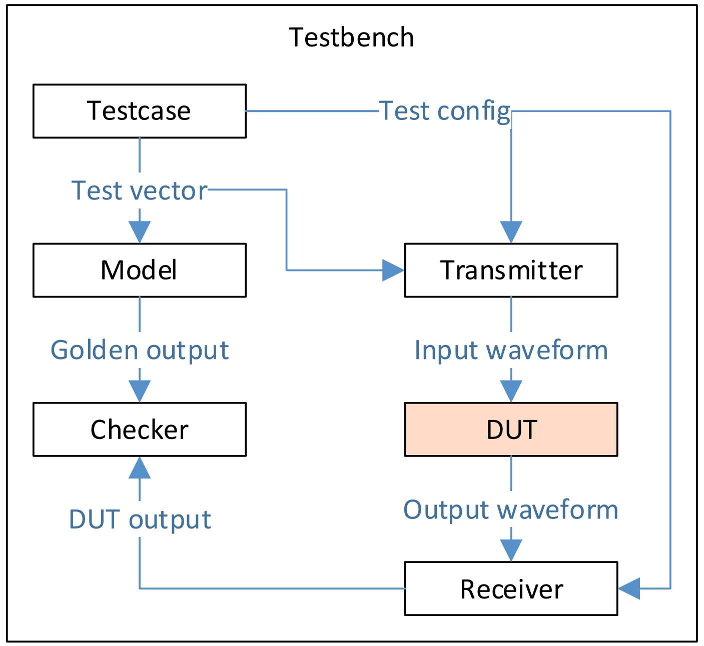
1. Testbench overview
2. Code syntax
3. Testbench components
4. Code Coverage

Testbench intent

- Verify that all designed features have the expected behavior
  - **Algorithm** (data path)
    - Random value
    - Critical value (algorithm limitation)
  - **Performance** (control path)
    - Random delay
    - Critical delay (0 delay or large delay)

Testbench Components

- **Test case**
  - Select test case to execute
- **Design Under Test** (DUT)
  - Design to be tested



- **Reference Model** (prepare golden output)
  - Read from files (generated by 3rd party, c, python, matlab …)
  - Runtime calculation (DPI)
- **Transmitter** (Driver)
  - Transform test vector to waveform, send to DUT
- **Receiver** (Monitor)
  - Receive waveform from DUT
- **Checker** (Scoreboard)
  - Check DUT output with test vector

Testbench types

- **Direct** testbench
  - Verilog
  - Focus on signal level
  - Use to verify the simple module or submodule

- **BFM** testbench
  - Verilog
  - Focus on the bus function level (bus protocol)
  - Use to verify the general design

- **UVM** testbench
  - SystemVerilog
  - Focus on the transaction level
  - Use to verify the system level design

Direct Testbench

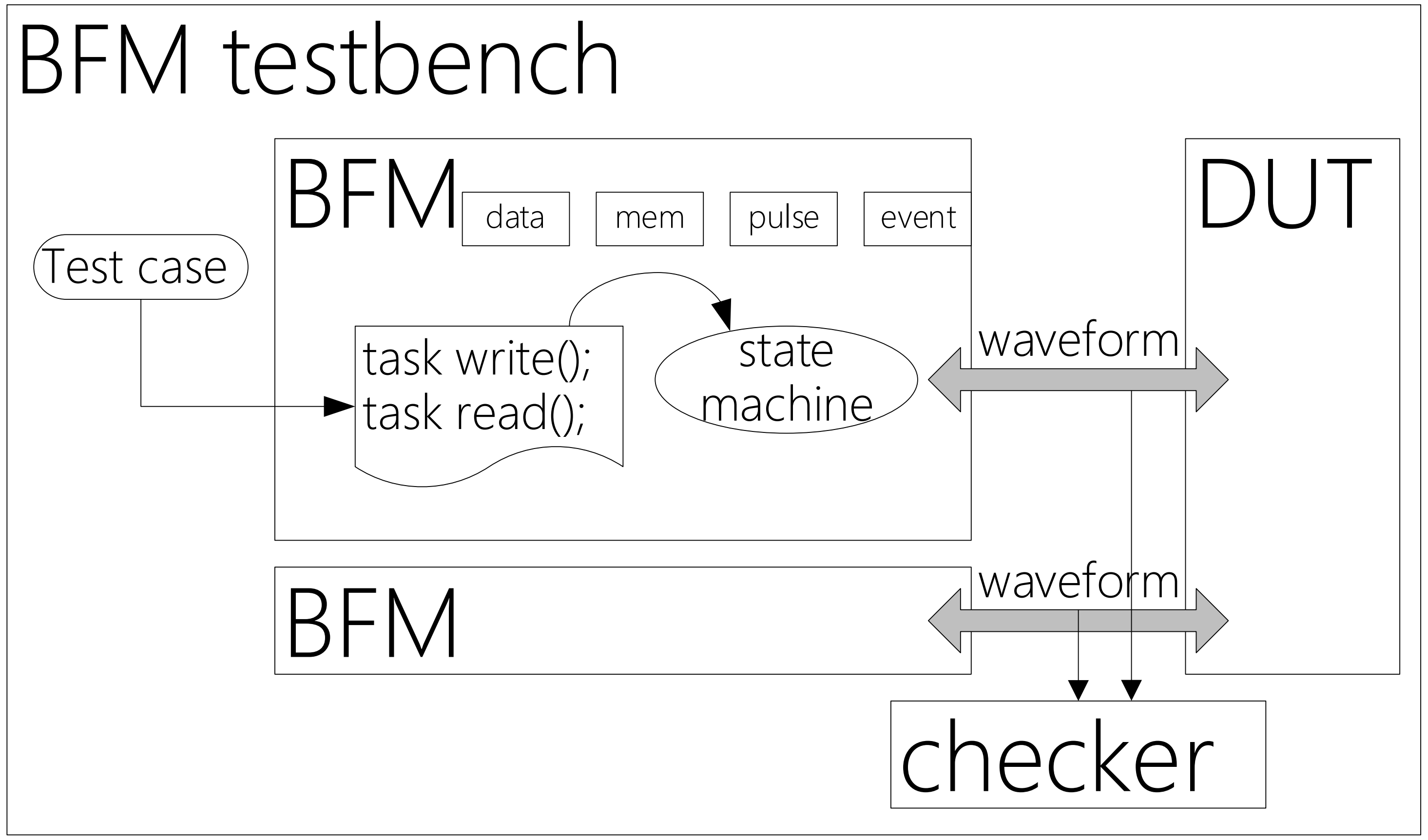
- **Direct** Testbench
  - Focus on signal level
    - Directly drive the input signal of DUT
    - Receive and check the output signal of DUT
  - Hard to re-use
    - Hard to understand the test sequences…
    - Hard to modify for new features…

```
module tb_top;

   dut I_DUT(
      ...
   );

   initial begin
      #100 data = ...;
      #100 data = ...;
      #100 data = ...;
      $finish;
   end

endmodule
```

BFM Testbench

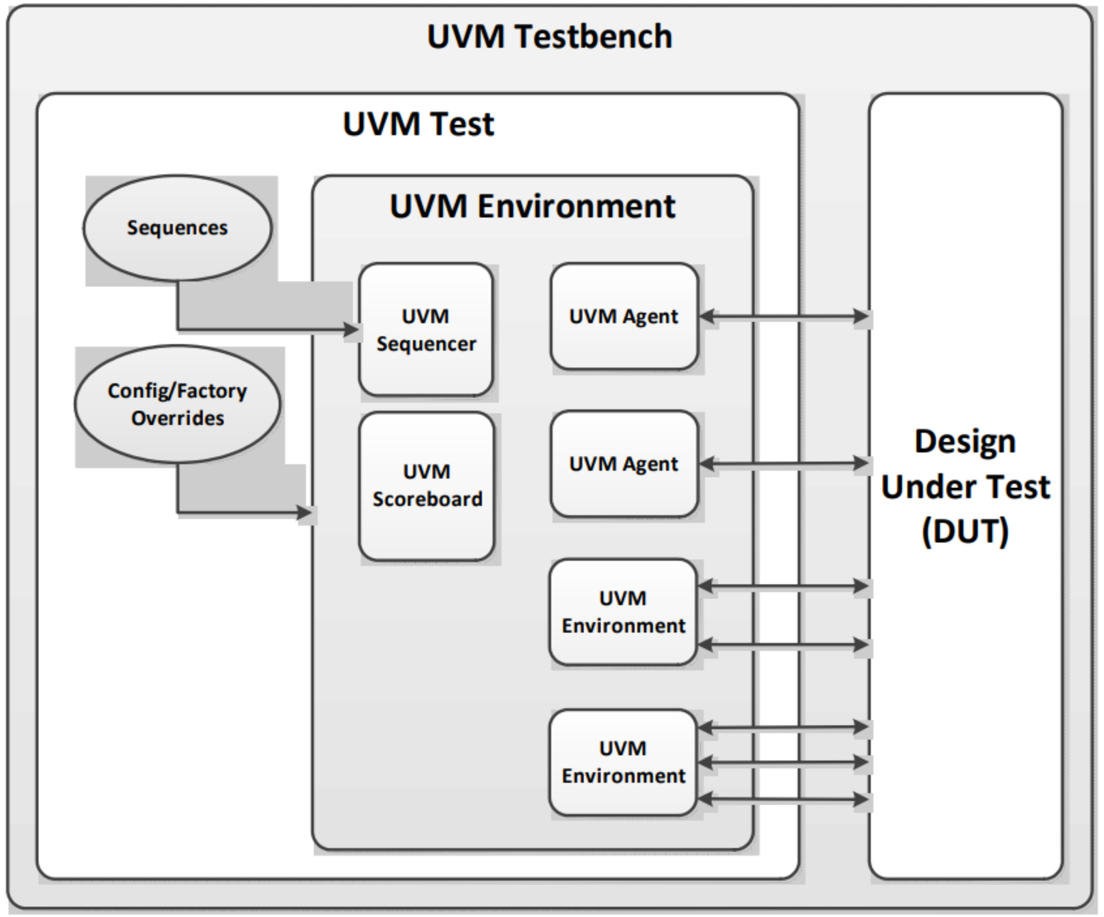
- **BFM** (Bus Functional Modeling)
  - Focus on the bus function level (bus protocol)
    - Modelling bus waveform by task or state machine
  - Test case
    - is derived by task
  - Randomness can be added
    - Whitelist Randomization

```
module tb_top;
   initial begin
      write(…);
      write(…);
      read(…);
      $finish;
   end
endmodule
```



UVM Testbench

- **UVM** (Universal Verification Methodology)
  - Uses **SystemVerilog** to create testbench
  - Focus on the transaction level
    - Modelling the abstract data flows of components in testbench
  - Test case
    - is constructed by sequences
    - Sequence is constructed by transaction
  - The waveform
    - UVM agent received transactions and convert to bus waveform to DUT
  - Constrained random verification
    - Blacklist Randomization
  - Use to verify the system level design



Testbench Coding Guideline

- Writing a testbench has many common aspects with writing a design, both need to do functional partition, and follow some coding style

- Functional partition (Component)
  - initial block
  - always block
  - Submodule ☺

- Coding Style
  - Readability
  - Reusability (Flexibility of controlling DUT input waveforms)

- The following section introduces various syntaxes commonly used in testbench which can be used to form different components. (**Direct** or **BFM**)

Agenda

1. Testbench overview
2. Code syntax
3. Testbench components
4. Code Coverage

Reserved word (keyword)

- initial, always
- begin, end
- #, @, event, ->
- for, while, repeat, forever
- fork, join
- task, function

Structured Procedures

- **Initial** Procedure
  - An initial procedure executes only once in a simulation
  - The activity of initial procedure starts at 0, and ceases when all statements have finished

```
initial begin
       rst_n = 0;
  #100 rst_n = 1;
end
```

- **Always** Procedure
  - An always procedure executes repeatedly as a loop
  - The activity of always procedure starts at 0, and represents a repetitive behavior
  - It may create a simulation deadlock condition without the correct timing control

```
☹ always clk = ~clk; //zero-delay infinite loop (deadlock)
☺ always #half_period clk1 = ~clk1;
```

Procedural Timing Controls

- **Delay** Control
  - Delay control is introduced by the symbol #
  - Delay control delays the amount of simulation time

```
#10        rega = regb; //delay 10 time units
#TIME      rega = regb; //TIME is defined as a parameter
#((A+B)/2) rega = regb; //delay is average of A and B
```

- **Event** Control
  - Event control is introduced by the symbol @
  - Event control waits the occurrence of a declared event
  - Event logical OR operator is used for multiple number of event triggers

```
@regc rega = regb; // controlled by any value change in the regc
@(posedge clock) rega = regb; // controlled by posedge on clock
@(negedge clock) rega = regb; // controlled by negedge on clock
@(posedge clka or posedge clkb) //controlled by two clock sources
```

Named Event and Event Trigger

- Named **Event**
  - An identifier declared as an event data type is called a named event
  - Named event is triggered via the -> operator

```
event ev; //named event

initial begin
   #100 ->ev; //event trigger
end

initial begin
   @(ev) $display($time,,”event is triggered”);
end
```

Sequential Block Statements

- **Sequential** Block
  - The sequential block is delimited by the keywords begin … end
  - The statements within sequential block execute in sequence, one after another

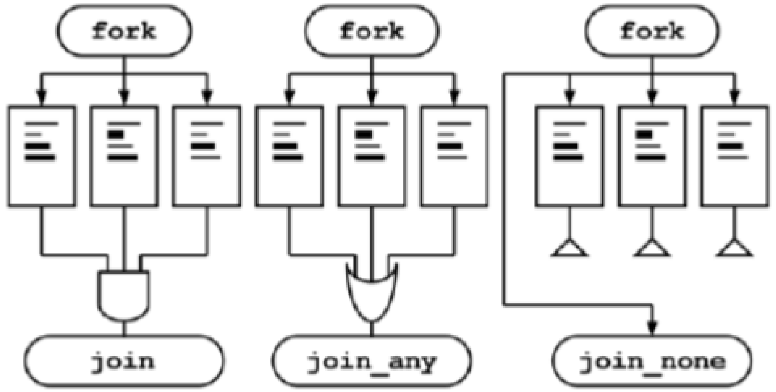
```
begin
   begin #10 $display($time,,“1”); end //time=10, “1” is displayed
   begin #10 $display($time,,“2”); end //time=20, “2” is displayed
end
```

Parallel Block Statements

- **Parallel** Block
  - The parallel block is delimited by the keywords fork … join
  - The statements within parallel block execute concurrently

```
fork
   begin #10 $display($time,,“1”); end //time=10, both “1” and “2”
   begin #10 $display($time,,“2”); end //are displayed
join
```

  - fork … join
  - fork … join_any
  - fork … join_none



https://blog.csdn.net/a52228254/article/details/106184602

Loop Statements

- **Forever** Loop
  - The forever loop repeatedly executes statements

```
clk = 0
forever begin
   #10 clk = ~clk;
end
```

- **Repeat** Loop
  - The repeat loop executes statements a fixed number of times

```
repeat (size) begin
   if (shift_opb[1]) result = result + shift_opa;
   shift_opa = shift_opa << 1;
   shift_opb = shift_opb >> 1;
end
```

Loop Statements

- **While** Loop
  - The while loop repeatedly executes statements as long as a control expression is true

```
@(posedge clk) req <= 1;
@(posedge clk);
while(~ack) begin
   @(posege clk);
end
req <= 0;
```

- **For** Loop
  - The for loop is executed by a three-step process

```
for(int i=0; i<100; i=i+1) begin
   data[i] = $random;
end
```

Remark:
- use **for** in testbench
- use **generate for** in design

User Defined Task

- **Task**
  - A task can have timing controls inside it
  - A task can enable other tasks
  - A task can define input or inout arguments to receive the variables
  - A task can return result by using output or inout arguments

```
//style 1
task   TASK_NAME;
input  PORT_LIST;
output PORT_LIST;
   begin
      ...
   end
endtask
```

```
//style 2 (ANSI-C style)
task TASK_NAME(input PORT_LIST, output PORT_LIST);
   begin
      ...
   end
endtask
```

Remark:
- Testbench can also use **function** with no timing control

- System Tasks

  $

Utility System Tasks

- **$finish** ~ cause the simulator to exit

```
initial begin
   ...
   $finish;
end
```

- **$random** ~ return a random value

```
rega =  $random  % 60; //value of rega between -59 to 59
regb = {$random} % 60; //value of regb from 0 to 59
```

- **$time** ~ return a 64-bit integer value of time
- **$stime** ~ return a 32-bit integer value of time
- **$realtime** ~ return a real value of time

```
@(posedge clk) time_str = $time;
@(posedge clk) time_end = $time;
clk_period = time_end - time_str;
```

Memory Array System Tasks

- **$readmemh**, **$readmemb** (file_name, mem_name, start_addr, end_addr);
  - Load data from a specified memory file into a specified memory array
  - The address information for these system tasks is optional

- **$writememh**, **$writememb** (file_name, mem_name, start_addr, end_addr);
  - Dump memory array contents to a text file

- Memory Files
  - Addresses appear in memory file is an at character (@) followed by a hexadecimal number
  - Data in memory file can be binary or hexadecimal numbers

```
//mem.dat
@0000 65d44e2a
@0001 78f0d55d
…
@00ff fee58612
```

```
reg [31:0] mem [0:255];
initial begin
   $readmemh(“./mem.dat”, mem);
end
```

Display System Tasks

- **$display**(list_of_arguments);
  - the main system task routine for displaying information
  - $display adds a newline character to the end of its output automatically

Argument	Description
%h or %x	Display in hexadecimal format
%d	Display in decimal format
%b	Display in binary format
%s	Display as a string
%t	Display in current time format

```
always@(posedge clk) begin
   if(en) $display($time,,“addr=%h, data=%h”, addr, data);
   else   $display($time,,“no data input”);
end
```

Dump Waveform System Tasks

- **$dumpfile**(“FILE_NAME”); ~ generate the value change dump files (**VCD**)(**IEEE standard**)
- **$dumpvars**(level, list_of_variable); ~ list which variables to dump into the file

```
//dump all
initial begin
   $dumpfile(“wave.vcd”);
   $dumpvars();
end
```

```
//dump partial design
initial begin
   $dumpfile(“wave.vcd”);
   $dumpvars(1, tb_top);
   //dump current level of tb_top
   $dumpvars(0, tb_top.dut.core);
   //dump all inside and below module core
end
```

- **$fsdbDumpfile**(“FILE_NAME”); ~ generate the fast signal database files (**FSDB**)(**SYNOPSYS**)
- **$fsdbDumpvars**(level, list_of_variable); ~ list which variables to dump into the file

```
initial begin
   $fsdbDumpfile(“wave.fsdb”);
   $fsdbDumpvars();
end
```

Compiler Directives

`

Compiler Directives

- **`timescale** time_unit/time_precision
  - This directive specifies the time unit and time precision
  - The time_unit argument specifies the unit of measurement for times and delays
  - The time_precision argument specifies how delay values are rounded

```
`timescale 1ns/1ps

initial begin
   clk = 0
   forever #(5/2.000) clk = ~clk;
end
```

What happen if timescale is set to 1n/1n

Compiler Directives (cont.)

- **`include** “filename”
  - This directive insert the entire contents of a file in another file during compilation

```
`include “dut_include.v”

module tb_top;
   ...
   `include “test_include.v”
   ...
endmodule
```

```
//dut_include.v
`include “top.v”
`include “subbk1.v”
`include “subbk2.v”
...
```

```
//top.v
module top#(
...
endmodule
```

```
//test_include.v
`include “test_xxx.v”
`include “test_yyy.v”
`include “test_zzz.v”
...
```

```
//test_xxx.v
initial begin
   if($test$plusargs(“test_xxx”)) begin
      ...
   end
end
```

Compiler Directives (cont.)

- **`define** text_macro_name macro_text
  - This directive activates the conditional compilation compiler directives (`ifdef, `ifndef, …)
  - This directive creates a macro for the text substitution

```
`define SHOW_PASS
`define SYS_ADDR_WIDTH 10
`define SYS_DATA_WIDTH 32
`define SYS_MEM  mem
`define WRITE(A, D) `SYS_MEM.write(A, D);
`define READ(A, D)  `SYS_MEM.read(A, D);
`define CHECK(A, B) \
   if(A!==B) $display("FAIL, read(%h) != golden(%h)", A, B); \
   `ifdef SHOW_PASS \
   else      $display("PASS, read(%h) == golden(%h)", A, B); \
   `endif

initial begin
   `WRITE(32'h0, 32'haabbccdd);
   `READ (32'h0, rd);
   `CHECK(rd, 32'haabbccdd);
end
```

Compiler Directives (cont.)

- **`ifdef … `ifndef … `else … `endif**
  - This directive includes or excludes optionally lines of source description during compilation
  - These directives may appear anywhere in the source description

```
//tb_top.v
`define NEST_ONE
...

`ifdef NEST_ONE
   initial $display("nest_one is defined");
   `ifdef NEST_TWO
      initial $display("nest_two is defined");
   `else
      initial $display("nest_two is not defined");
   `endif
`else
   initial $display("nest_one is not defined");
`endif
```

`define can be written inside the source code

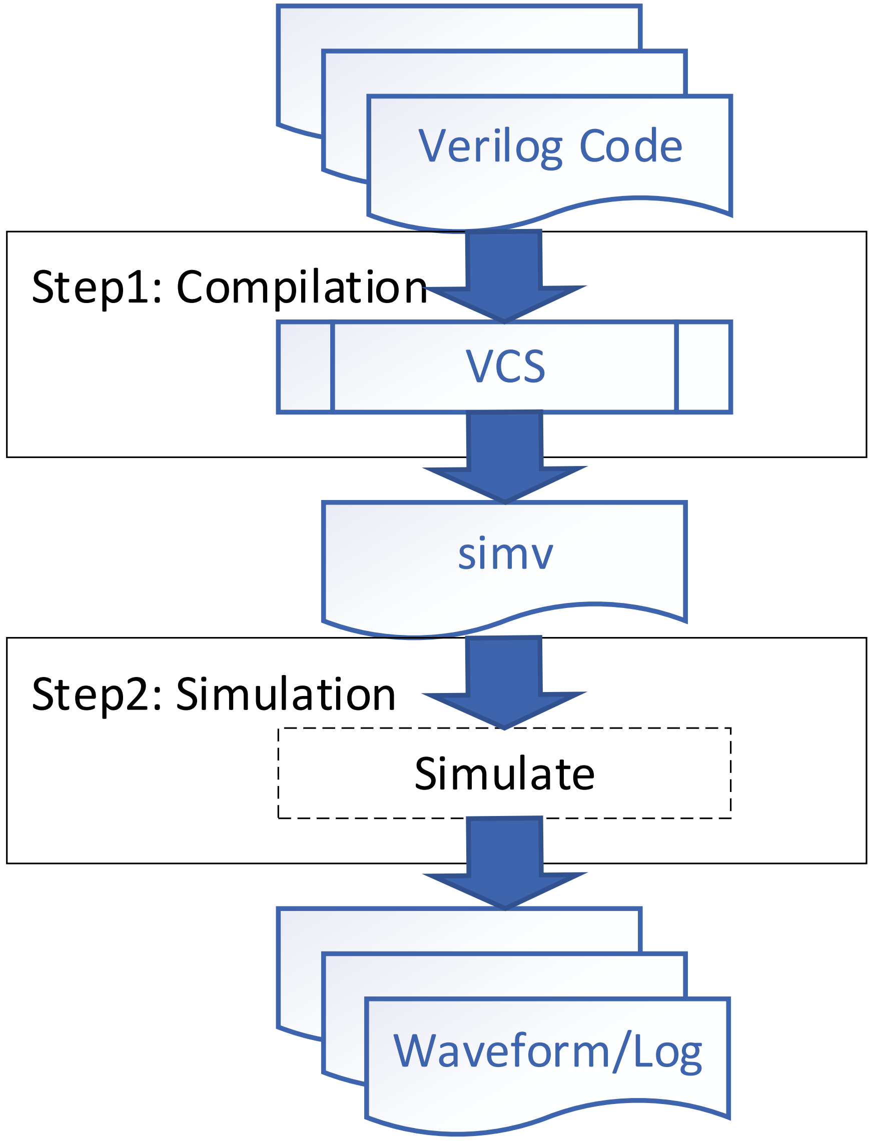
```
//command line
[xxx@wsxx]$vcs tb_top.v +define+NEST_TWO
```

`define can be added in the compilation

Command Line Input

+

VCS 2-Step flow



- Step1: Compilation
  - Command: vcs   +define…
  - Compiler directives are enabled for the sub-sequent phases till the end of the simulation
- Step2: Simulation
  - Command: ./simv   +plusarg…
  - Process the test case and generate the test result (waveform or log files)
- --
- 1-step flow (Compile-and-Simulation)
  - vcs **–R**   +define…   +plusarg…

Command Line Input Define

- **+define+**MACRO_NAME
- **+define+**MACRO_NAME=value

```
//command line
[xxx@wsxx]$vcs +define+NEST_ONE
```

```
//tb_top.v
...
`ifdef NEST_ONE
   ...
`endif
```

```
//command line
[xxx@wsxx]$vcs +define+SYS_DATA_WIDTH=32
```

```
//tb_top.v
...
Parameter DW = `SYS_DATA_WIDTH;
```

Command Line Input System Tasks

- **$test$plusargs**(string);
  - These task is specified the information for use in the simulation
  - A string with the plus (+) character is provided an optional argument to the simulator
- **$value$plusargs**(string, variable);
  - These task has an ability to get the value from a specific user string.

```
//command line
[xxx@wsxx]$./simv +test1
```

```
initial begin
   if($test$plusargs(“test1”)) begin
      …
   end
end
```

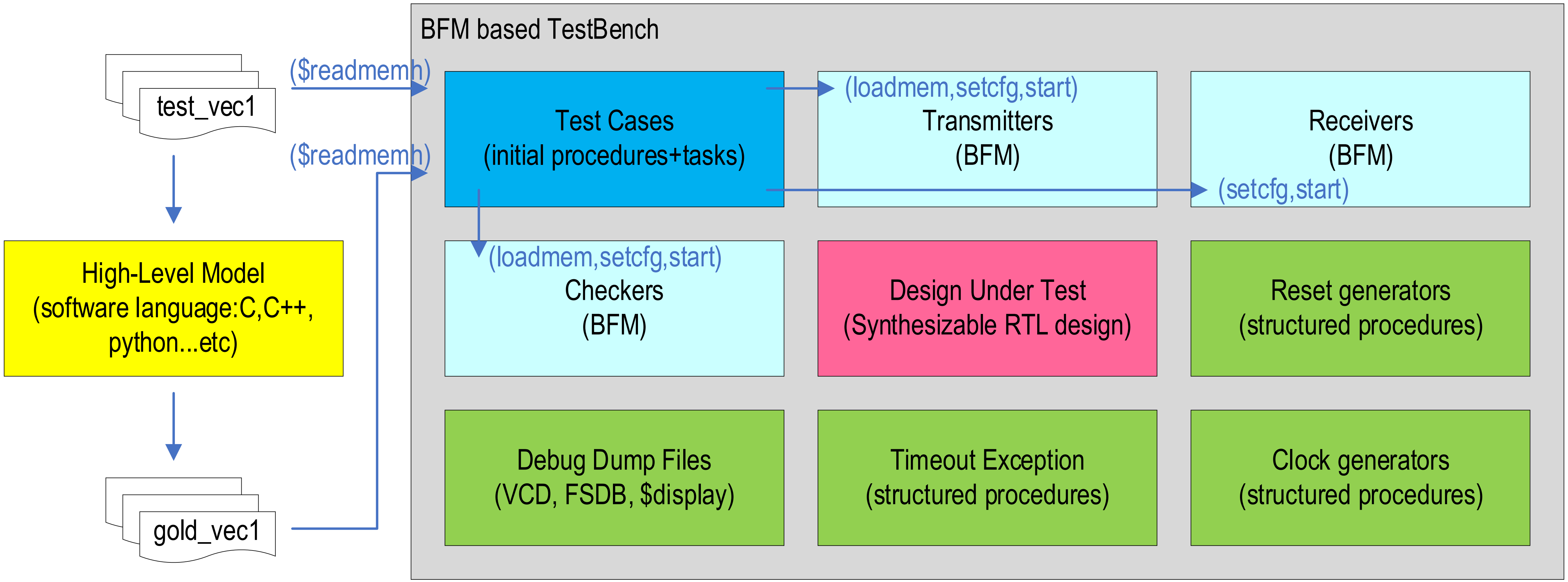
```
//command line
[xxx@wsxx]$./simv +test1 +time_out=2000
```

```
initial begin
   if(!$value$plusargs(“time_out=%d”, time_out))begin
      time_out = 10000;//default
   end
   #time_out $finish;
end
```

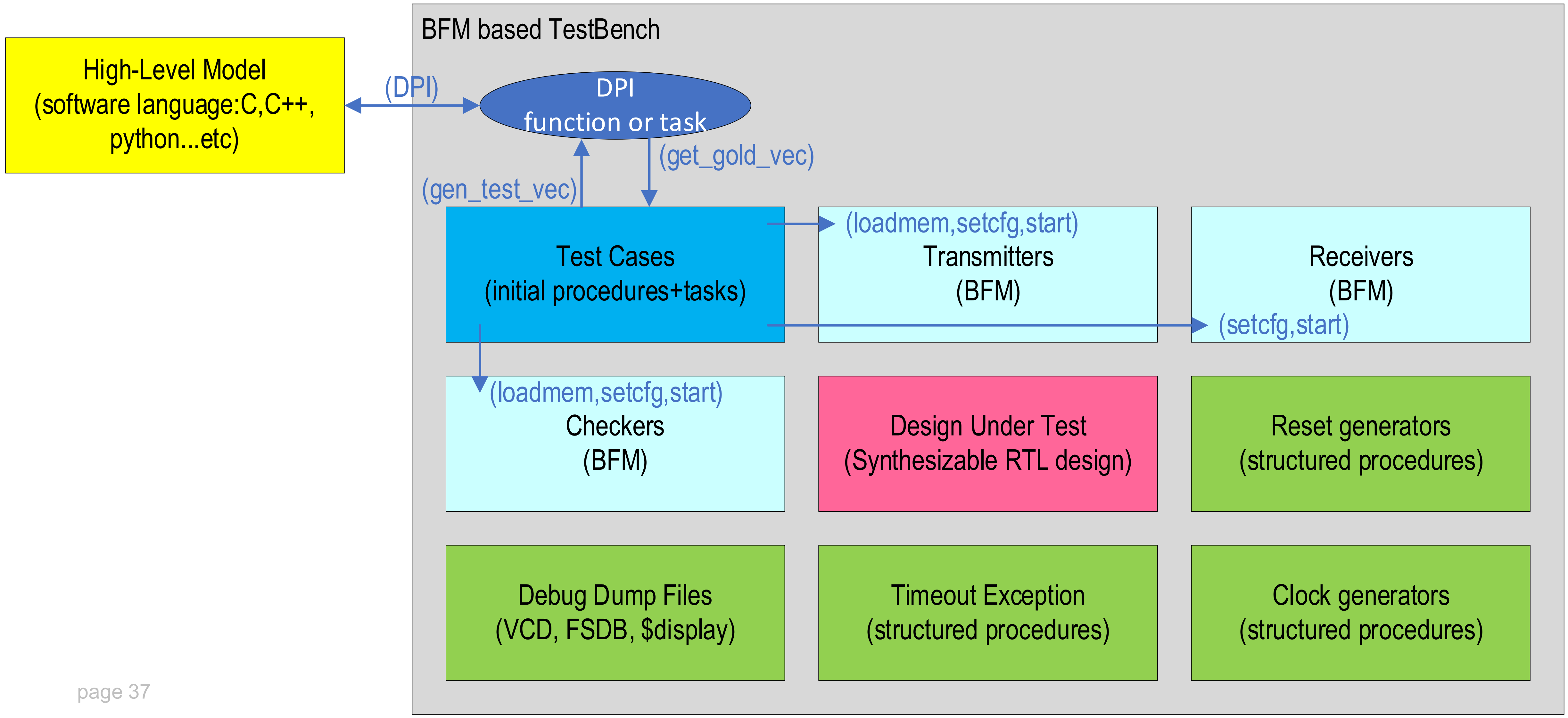
Agenda

1. Testbench overview
2. Code syntax
3. Testbench components
4. Code Coverage

Test Bench Components with $readmem



Test Bench Components with DPI



Example BFM module

- Transmitter
  - io: connect to the upstream of DUT (flow control and send data)
  - task: set_test_vector
  - task set_timing_config
  - task start (to start the FSM)
- Receiver
  - io: connect to the downstream of DUT (flow control)
  - task set_timing_config
  - task start (to start the FSM)
- Checker
  - io: connect to the downstream of DUT (collect data)
  - task: set_golden_vector
  - task: set_compare_config
  - task: get pass/fail count
  - Task: start (to start the FSM)

Example of testcase

- Initial begin
  - if($test$plusargs(“test1”)) begin
    - //prepare test vector
      - Read test vector from files, or call DPI to runtime calculate
      - Set test vector to transmitter and checker
    - //timing config
      - Set timing config to transmitter and receiver
    - //compare config
      - Set compare config to checker
    - //start
      - Start transmitter, receiver, checker
    - //wait done
      - Wait transmitter and receiver done signal
  - end
- end

Example of testcase

```
...
...
//load vectors
$readmemh("test_vec1.dat", test_vec);
$readmemh("gold_vec1.dat", gold_vec);
for(i=0;i<xmit_testlen;i=i+1) xmit.loadmem(i,test_vec[i]);
for(i=0;i<rcvr_testlen;i=i+1) chkr.loadmem(i,gold_vec[i]);

//config bfms
xmit.setcfg_length(xmit_testlen);
xmit.setcfg_pause(1);
xmit.setcfg_pause_rate(30);

rcvr.setcfg_length(rcvr_testlen);
rcvr.setcfg_pause(1);
rcvr.setcfg_pause_rate(10);

chkr.setcfg_length(rcvr_testlen);
chkr.setcfg_check_unknown(1);
chkr.setcfg_check_overflow(1);
chkr.setcfg_pass_showmsg(1);//fail msg is always showed
chkr.setcfg_fail_stopsim(1);//fail then call $finish
```

```
//start the test
@(posedge rst_n);
xmit.start(1, 100);//delay 100 cycle then start bfm
rcvr.start(1,   0);
chkr.start(1,   0);

//wait done
@(posedge xmit_busy);
while(xmit_busy | rcvr_busy) begin//check bfm busy status
   @(posedge clk);
end

//check the result
chkr.get_pass_cnt(pass_cnt);
chkr.get_fail_cnt(fail_cnt);
if(fail_cnt > 0 || (pass_cnt != rcvr_testlen)) begin
     $display($time,,"FAIL, test case is failed");
end
else $display($time,,"PASS, test case is passed");
```

Direct Programming Interface

- Example of C-Code

```
int cipher32_en(unsigned char *ct, unsigned char *pt, unsigned char *key)
{
   ...
   return 1;
}
```

Use pointer of the char data type to access the SystemVerilog data structures

- Compile the C-Code to shared library

```
//command line
[xxx@wsxx]$gcc -fPIC -g -shared -o c_model.so C_MODEL/*
```

- Compile the C shared library and the Verilog design by VCS with –sverilog argument

```
//command line
[xxx@wsxx]$vcs –full64 –sverilog c_model.so design_include.v
```

Direct Programming Interface (cont.)

- Direct Programming Interface (DPI) of SystemVerilog
  - DPI allows direct inter-language function or task calls between the languages
  - Users are responsible for specifying in their foreign code the native types equivalent to the types used in imported declarations
  - The data types of the SystemVerilog are 2-state (0,1) or 4-state (0,1,x,z)

```
import "DPI-C" function int cipher32_en(output byte ct  [4], //cipher-text
                                        input  byte pt  [4], //plain-text
                                        input  byte key [20]);

module tb_top;
   ...
   initial begin
      ...
      flag = cipher32_en(ct,pt,key);
      ...
   end
   ...
endmodule
```

Use byte array for the char data type of C language

Verilog integer is NOT the same as the int data type of C language

Agenda

Code and Functional Coverage

- **Code Coverage**
  - Code coverage measures the degree of the RTL code is tested
  - The coverage metrics can be generated automatically by the simulator

- **Functional Coverage**
  - Functional coverages are user-defined metrics
    - The data-oriented coverage is written by SystemVerilog covergroup
    - The control-oriented coverage is written by SystemVerilog assertion

Code Coverage Metrics

- Code Coverage
  - **Line** Coverage
    - To shows which lines of code are exercised and which ones are not
  - **Branch** Coverage
    - To monitors the execution of conditional statements such as if/else/case and operator ?:
  - **Expression** Coverage
    - To monitors whether the expressions in your code evaluate to true or false
    - For the operations like &&, ||, ==, <=, >=, &, | … etc
  - **Toggle** Coverage
    - To monitors value changes from 0 to 1 and from 1 to 0 on each signal bit
  - **FSM** Coverage
    - To analyzed the states and transitions that occur in the FSM design

Example of VCS coverage

- Add VCS coverage options into script of regression

```
//script for regression
...
   vcs –full64 –R \
       -cm line+branch+cond+fsm \
       -cm_hier dut.hier \
       -cm_line contassign \
       -cm_cond std+allops \
       -cm_noconst \
       -cm_name ${testname} \
       -l ${testname}.log \
       test_inc.v +${testname}
...
```

- View the coverage metrics

```
//command line
[xxx@wsxx]$verdi -cov –covdir simv.vdb
```

VCS Coverage Options

VCS coverage options	Descriptions
-cm line+branch+cond+fsm	Specifies compiling for the specified types of coverage
-cm_line contassign	Enables line coverage for continuous assignments
-cm_noconst	Disable coverage if a signal is permanently at 1 or 0 value
-cm_name <testname>	Specifies unique name for a test during simulation

VCS Coverage Options

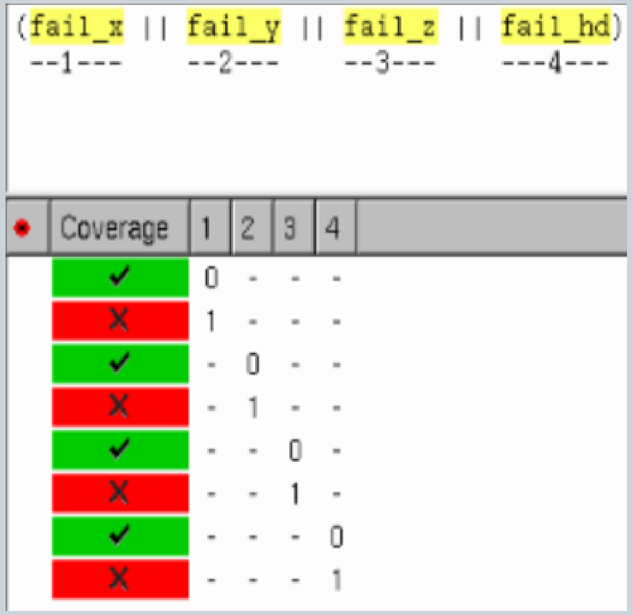
VCS coverage options	Descriptions
-cm_cond <arguments>	Modifies condition coverage as specified by the arguments

Aargument	Descriptions
basic	Only logical conditions and no multiple conditions
std	The default: only logical, multiple, sensitized conditions
full	Logical and non-logical, multiple conditions, no sensitized conditions
allops	Logical and non-logical conditions
event	Signals in event controls in the sensitivity list position are conditions
anywidth	Enables conditions that need more than 32 bits
for	Enables conditions if for loops
tf	Enables conditions in user-defined tasks and functions

VCS Coverage Options

VCS coverage options	Descriptions
-cm_cond <arguments>	Modifies condition coverage as specified by the arguments

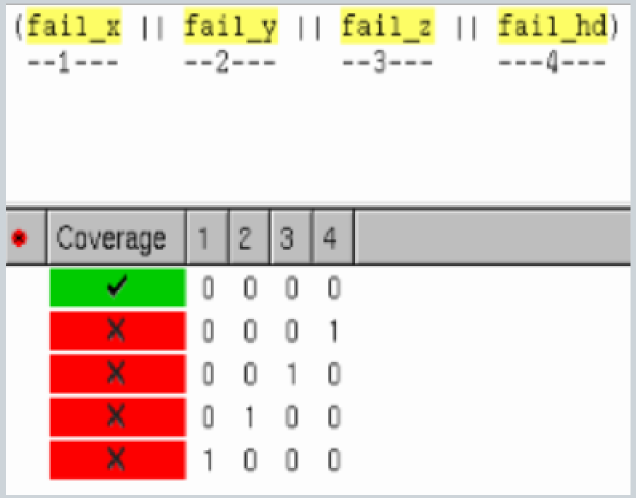
basic



```
(fail_x || fail_y || fail_z || fail_hd)
 --1---    --2---    --3---    ---4---
```

●	Coverage	1	2	3	4
	✓	0	-	-	-
	✗	1	-	-	-
	✓	-	0	-	-
	✗	-	1	-	-
	✓	-	-	0	-
	✗	-	-	1	-
	✓	-	-	-	0
	✗	-	-	-	1

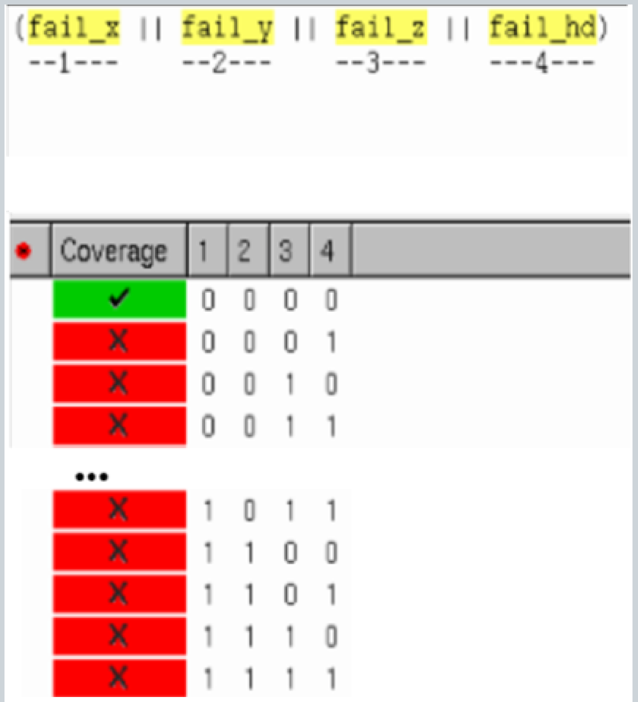
std



```
(fail_x || fail_y || fail_z || fail_hd)
 --1---    --2---    --3---    ---4---
```

●	Coverage	1	2	3	4
	✓	0	0	0	0
	✗	0	0	0	1
	✗	0	0	1	0
	✗	0	1	0	0
	✗	1	0	0	0

full



```
(fail_x || fail_y || fail_z || fail_hd)
 --1---    --2---    --3---    ---4---
```

●	Coverage	1	2	3	4
	✓	0	0	0	0
	✗	0	0	0	1
	✗	0	0	1	0
	✗	0	0	1	1
	•••				
	✗	1	0	1	1
	✗	1	1	0	0
	✗	1	1	0	1
	✗	1	1	1	0
	✗	1	1	1	1

VCS Coverage Options

VCS coverage options	Descriptions
-cm_hier <filename>	Specifies module definitions in a configuration file that you can include or exclude for coverage

```
//<op>moduletree MODULE_NAME <level>
//<op>tree       INST_HIER   <level>
//
//<op>   :
//      +: select
//      -: deselect
//<level>:
//      0: all level
//      1:   1 level (current)
//      2:   2 level
//      n:   n level
//
+moduletree aes_core 0
```

Reference

- IEEE Std 1800™-2017
- VCS/VCSi User Guide
- Coverage Technology Reference Manual
- Universal Verification Methodology User’s Guide
- https://en.wikipedia.org/wiki/Bus_functional_model
- https://en.wikipedia.org/wiki/Code_coverage

Feedback to us

NTHU 晶片安全設計 回饋單

https://forms.office.com/r/DYDu8vLaWN

Thank you!

PUFacademy

A PUFsecurity Alliance

Visit our website: pufacademy.com

Contact us: PUFacademy@pufsecurity.com