

Introduction to Verification and SystemVerilog



Delivering KnowHow

Introduction to Verification and SystemVerilog



www.doulos.com

Introduction to Verification and SystemVerilog



Copyright © 2015-2023 by Doulos. All Rights Reserved

All intellectual property rights, including copyright, patents, design rights and know-how in or relating to the course or course materials provided or made available in connection with the course remain the sole property of Doulos Ltd or their respective owners and no copies may be made of course materials unless expressly agreed in writing by Doulos Ltd.

All trademarks acknowledged.

Doulos takes great care in developing and maintaining materials to ensure they are an effective and accurate medium for communicating design know-how. However, the information provided on a Doulos training course may be out of date or include omissions, inaccuracies or other errors. Except where expressly provided otherwise in agreement between you and Doulos, all information provided directly or indirectly through a Doulos training course is provided "as is" without warranty of any kind.

Doulos hereby disclaims all warranties with respect to this information, whether express or implied, including the implied warranties of merchantability, satisfactory quality and fitness for a particular purpose. In no event shall Doulos be liable for any direct, indirect, incidental, special or consequential damages, or damages for loss of profits, revenue, data or use, incurred by you or any third party, whether in contract, tort or otherwise, arising for your access to, use of, or reliance upon information obtained from or through a Doulos training course. Doulos reserves the right to make changes, updates or corrections to the information contained in its training courses at any time without notice.

Doulos Limited
Church Hatch, 22 Market Place,
Ringwood, Hampshire, BH24 1AW, UK

Tel: +44 (0) 1425 471223
Email: info@doulos.com

Doulos
6203 San Ignacio Avenue, Suite 110,
San Jose, CA 95119, USA

Tel: 1-888-GO DOULOS
Email: info.usa@doulos.com

www.doulos.com



Contents

Contents.....	7
Introduction to SystemVerilog	9
Verification Approaches	9
Simulation and Testbenches.....	13
Coverage	19
Formal Verification	22
Introduction to SystemVerilog	24
What is SystemVerilog	25
SystemVerilog Classes	29
Use of Interfaces	36
Constraints and Functional Coverage.....	38



Notes

Introduction to SystemVerilog

Introduction to Verification & SystemVerilog

- Introduction to Verification
 - Verification Approaches
 - Simulation and Testbenches
 - Coverage
 - Formal Verification
- Introduction to SystemVerilog

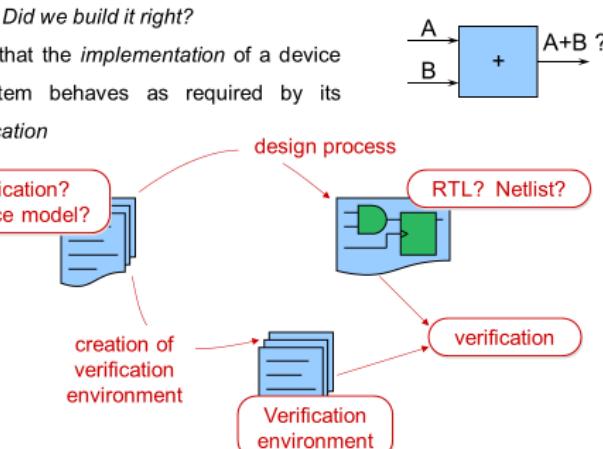
 2

Verification Approaches

What is Verification?

• Verification: *Did we build it right?*

- Check that the *implementation* of a device or system behaves as required by its *specification*

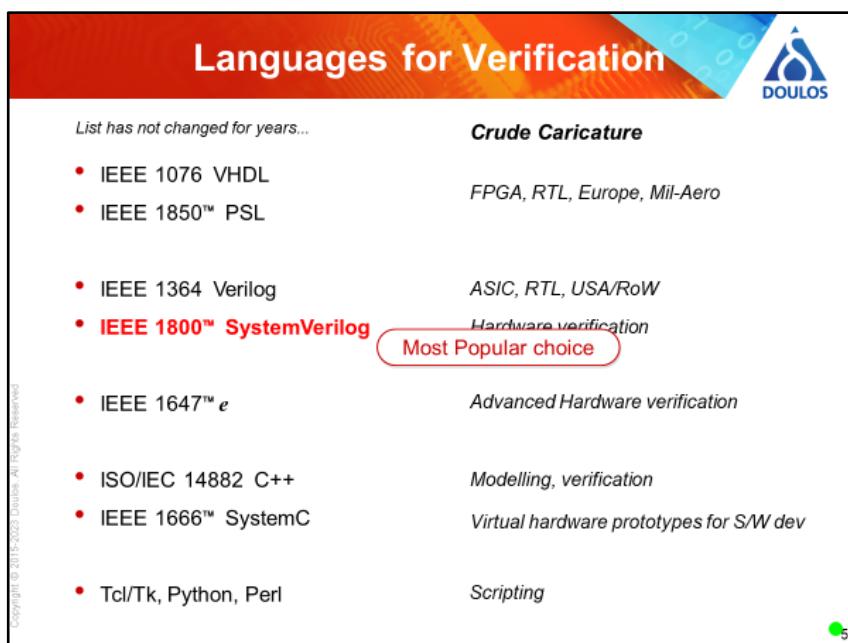
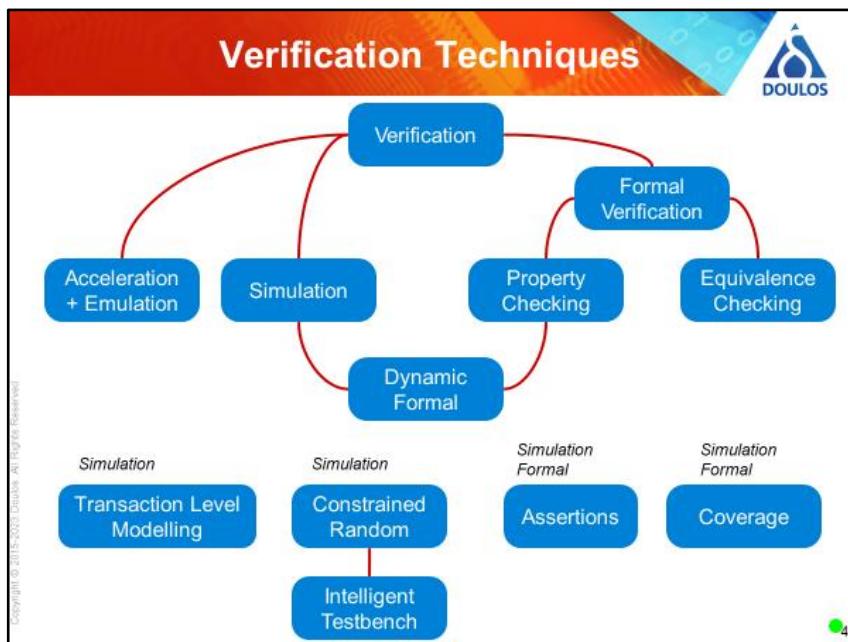


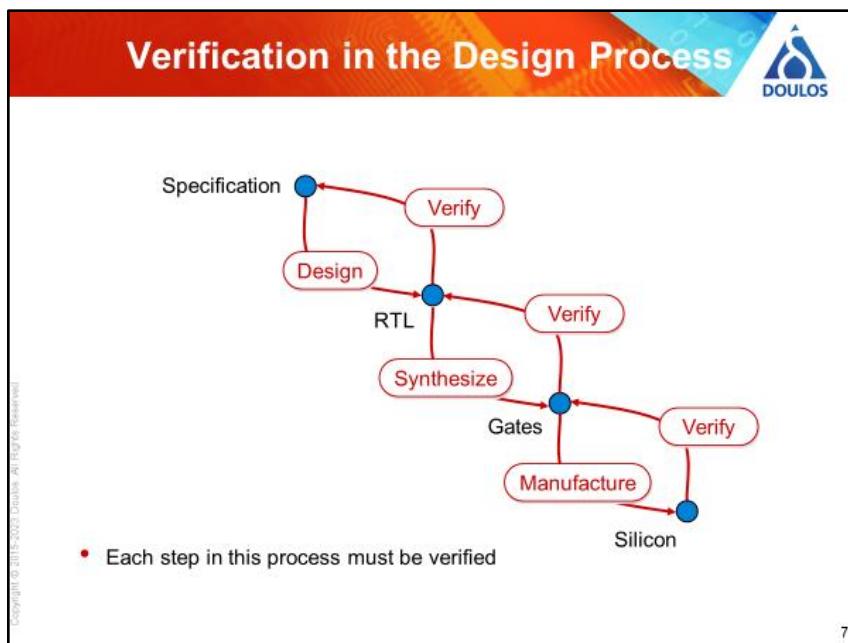
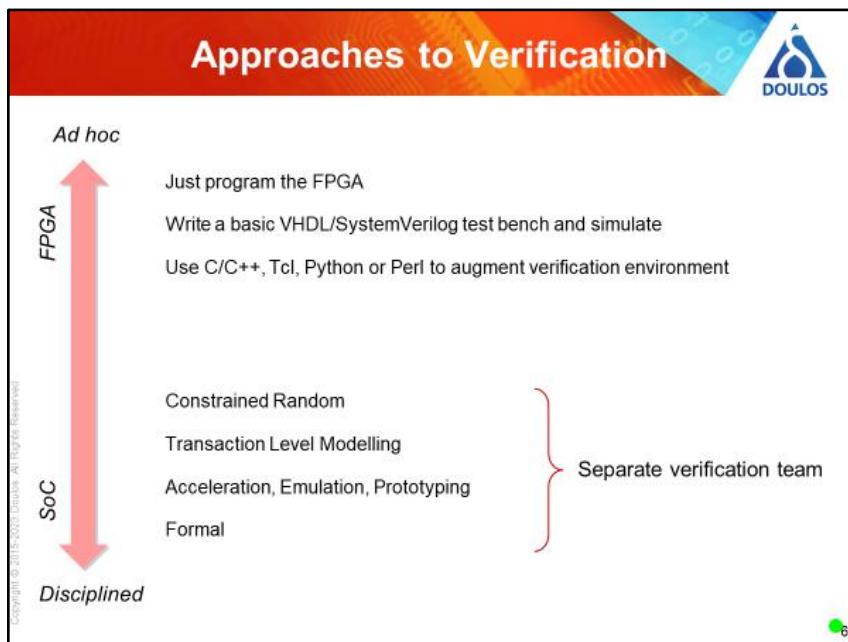
creation of verification environment

design process

verification

3





Verification Plan

 DOULOS

- What are we going to verify? Requirements, objectives
- How are we going to do it? Design of test stimulus
- How do we measure success? Observations

The Verification Plan specifies the verification tasks and effort

- If it's specified it should be verified!
- Identify an appropriate way to test each feature or statement in the specification
 - *directed* or *constrained-randomised* testing
- Create constrained random stimulus that will execute that feature and the data 'self checking' (aka scoreboard) is correctly implemented by the design

8

Example Verification Plan

 DOULOS

Specification

- The NBG output pin will reflect the status of the internal FAIL register bit
- A checksum calculated using the CCITT-16 polynomial is appended
- ...

Test descriptions

- 1. Write '1' to FAIL register
2. Check that NBG goes to '1' within 2 clocks
3. ...
- 1. Cover NBG signal value wrt FAIL register bit.
2. Cover point for checksum value good and bad using CCITT-16
- 1. Change the FAIL bit value in the register.
2. Generate and send data using good and bad checksum values

- In this example, one specification part is verified using a defined test
 - This could be done by simulation or by formal methods
- For the second part, (constrained) random data is generated

9

Linting Tools

- Locating errors or potential errors in HDL code can save a lot of verification effort later
- Simulators and formal tools should find errors ... eventually
- A *Linting Tool* finds common errors quickly and automatically
- Example:

```
always @(Select)
  if (Select)
    Y= A;
  else
    Y=B;
```

Warning: Incomplete event list

- Verilint
- HAL
- LEDA
- ...

10

Simulation and Testbenches

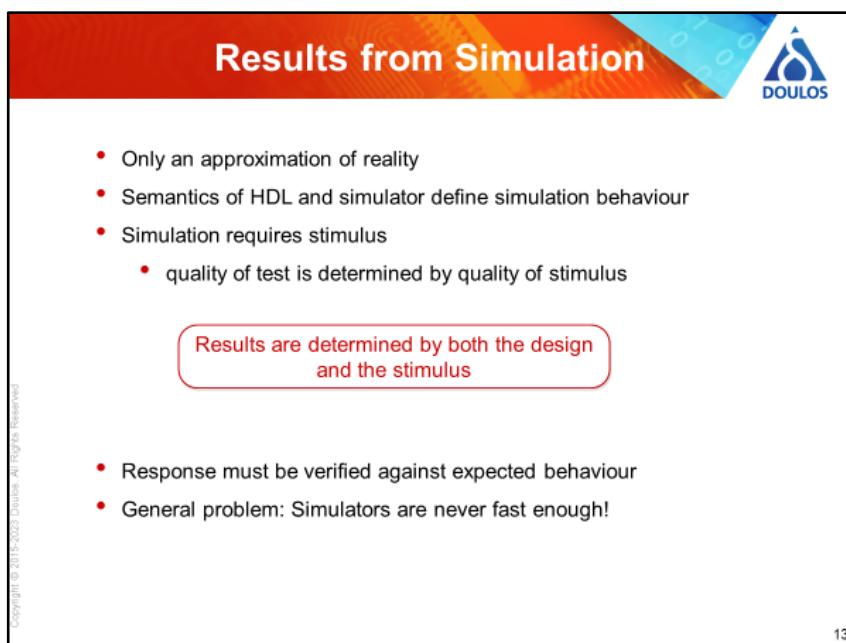
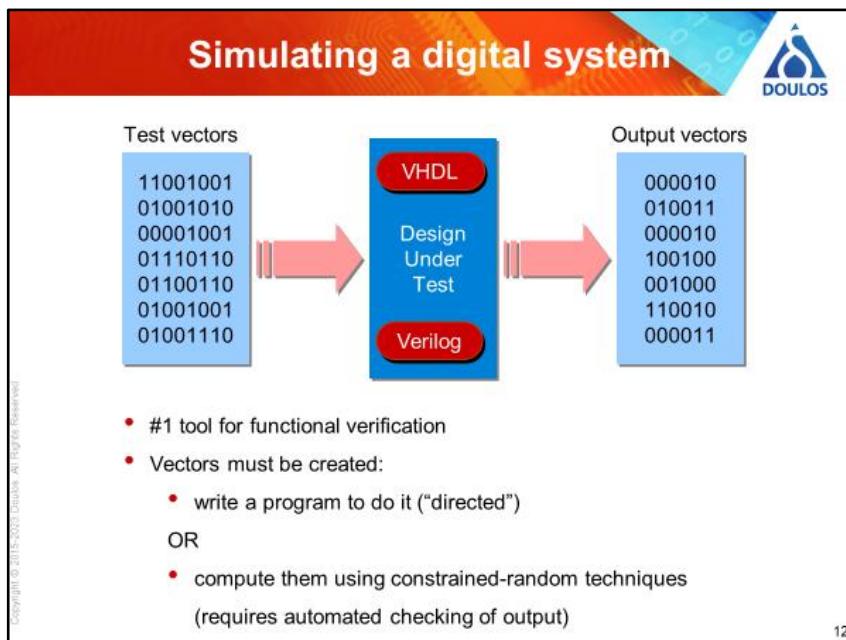
Simulation

- Execute a model of the system
- A simulation is only as good as the model
 - The more accurate the model, the longer the simulation time
- Cannot simulate everything – not enough time

Speed  Accuracy 

- Widely used:
 - simulation is intuitively attractive, looks like the real thing
 - mature, familiar tools
 - excellent debugging

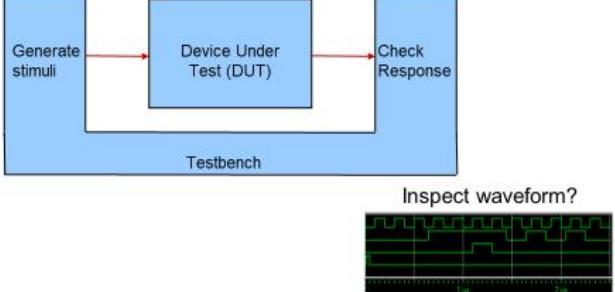
11



Basic Testbench

Often written by the designer
Unstructured grey box test

```
#10 selA = 1'b1;
#10 A = 1'b0;
#10 selA = 1'b0;
```



Inspect waveform?

14

Testbenches

- A *testbench* sends stimuli to the Device under Test (DUT) and collects responses from the DUT
 - Usually designed to be *self-checking*, i.e. it automatically checks the DUT's responses for correctness
- A testbench is therefore the environment seen by the DUT
 - The testbench must provide all signals needed for the DUT to operate
- The testbench can be written in:
 - the same HDL as the DUV
 - another HDL, or
 - a specialised Hardware Verification Language (HVL)
- Testbenches can use the entire range of the language – no need to stick to a synthesis subset

15

Structured Testbench

Copyright © 2015-2023 Doulos. All Rights Reserved

- Stimulus can be read from a file or can be created using constrained-random generation
- Whatever stimulus is applied, the DUT's output should be checked automatically
- Prefer monitoring/checking processes that work correctly for *any* stimulus

```

graph LR
    SG[Stimulus generator] --> DUT[DUT]
    DUT --> C[Checker]
    C --> Log[Log file]
    C --> Calc[Calculate expected results]
    C --> Comp[Compare results]
    Comp --> Log
    
```

OK for any stimulus

16

Boundary Conditions & Corner Cases

Copyright © 2015-2023 Doulos. All Rights Reserved

- Exhaustive testing is impractical, so which tests to include?
- Example - an arithmetic function:

```

entity Arithmetic is
  port (
    A, B : in  UNSIGNED(7 downto 0);
    F     : out UNSIGNED(7 downto 0));
end entity Arithmetic;
  
```

• Verification engineer's experience is important in choosing corner cases

• When using formal verification, the story is very different - all cases explored automatically

A	B
00000000	00000000
11111111	00000000
00000000	11111111
11111111	11111111
00000000	00000001
00000001	00000000
00000001	00000001
00000001	11111111
11111111	00000001
00000001	11111110
11111110	00000001

17

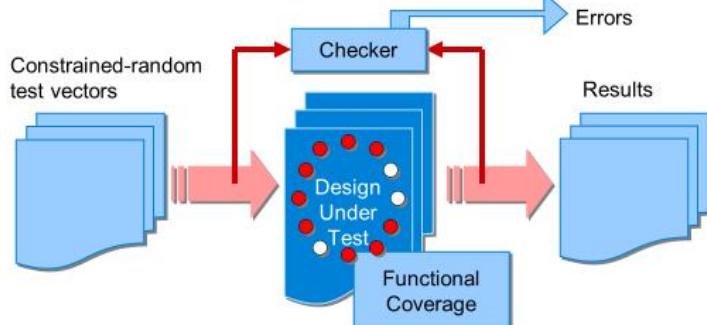
Constrained Random Tests



- Writing many different tests is difficult
 - In the worst case, need every combination of inputs
 - Choose a representative sample – how?
 - Unconscious bias towards good data – may ignore unlikely but fatal combinations
- Random test generation is easy
 - Random number generation built in to most languages
 - But... many random combinations should not occur
- Constrained random generation
 - Random, but subject to certain constraints
- Example
 - Meaningful sequence of opcodes, but random data and random addresses within a certain range

18

Testbench Automation



Principles of testbench automation:

- Constrained-random generation of test vectors
- Automatic checking of results
- Functional coverage to measure verification completeness

19

Simulation Acceleration



Copyright © 2015-2023 Doulos. All Rights Reserved

- Simulation is slow; how to make it faster?
- Less detailed simulation – may not be acceptable
- A simulation typically runs on a workstation
 - More powerful workstation
 - Distribute simulation between workstations
 - Central compute server – 1 per office
 - Compute farms – 1 per corporation
- Can become expensive!

20

Modelling Levels

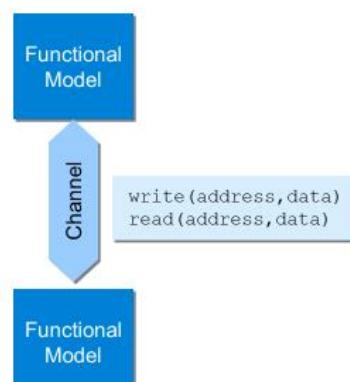


Copyright © 2015-2023 Doulos. All Rights Reserved

- If the complete DUV is modelled at the lowest level, the simulation will be slow
- Ideally, model each part of the DUV only in enough detail to verify that part of the specification that's being simulated
- Example
 - If we model a data transfer operation, every bit change on every pin will generate a simulation event
- Solution
 - Model the data transfer as a single transaction – one event
- This is *Transaction Level Modelling (TLM)*

21

Transaction Level Modelling



- 100+ X faster simulation!

22

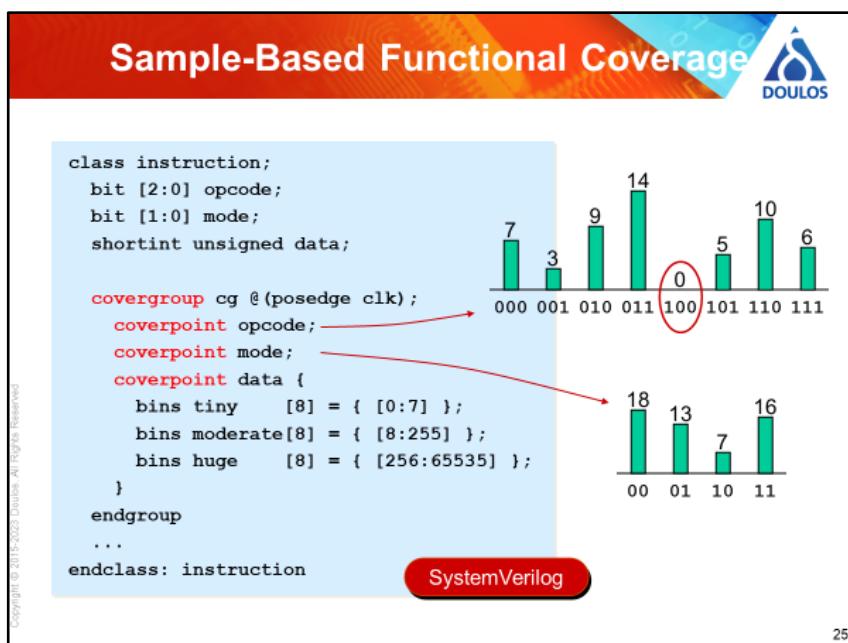
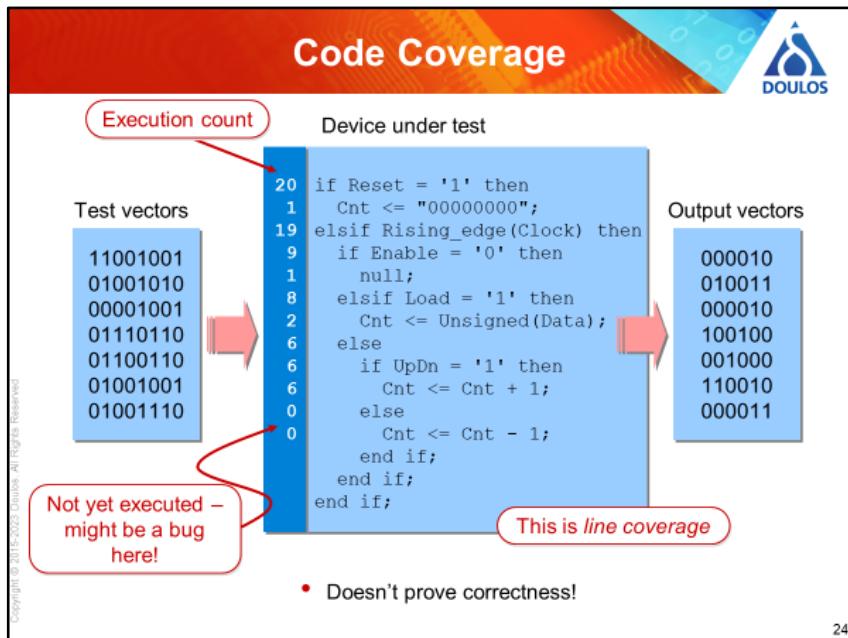
Coverage

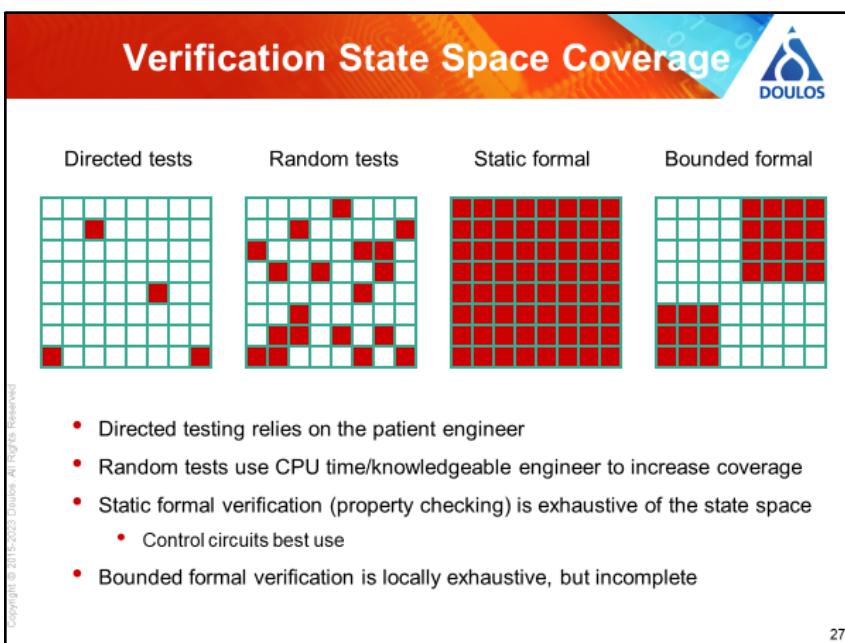
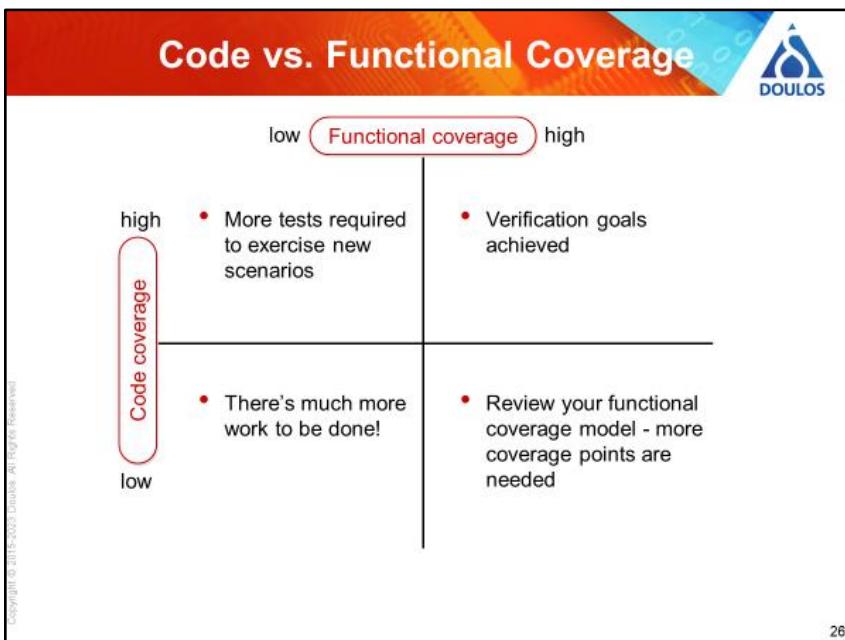
Coverage

- “Coverage” is used in different ways –
- Usual English meaning

“Have we covered everything in the specification?”
- Technical terms
 - *Code Coverage* – how much of the code have we exercised during simulation?
 - *Functional coverage* – how much (expressed as a percentage) of the functionality described by the specification have we exercised during simulation?

23





Formal Verification

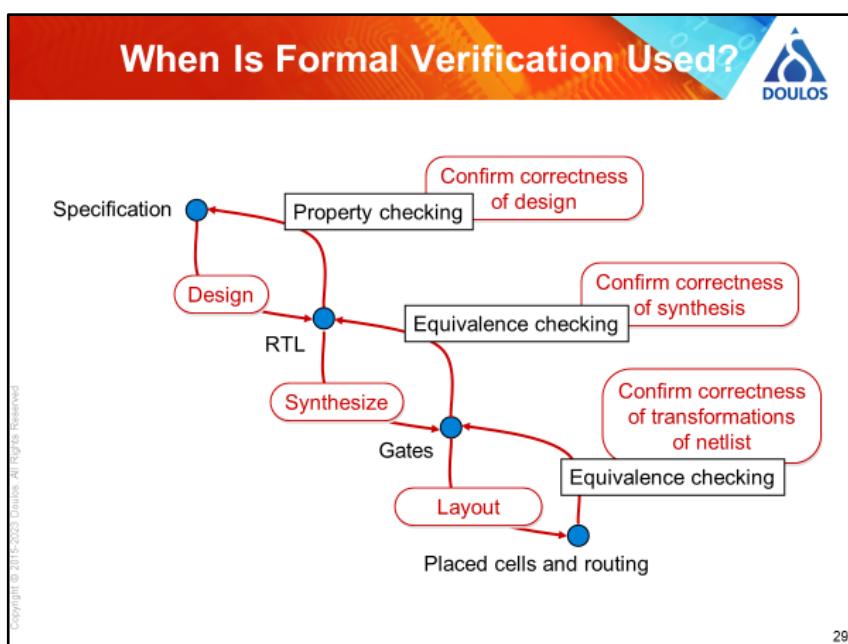
Formal Verification

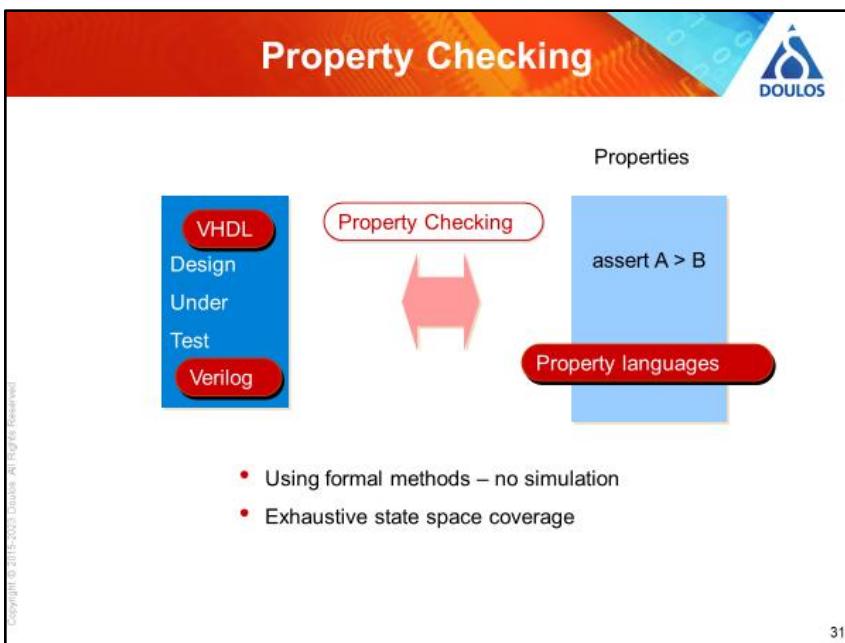
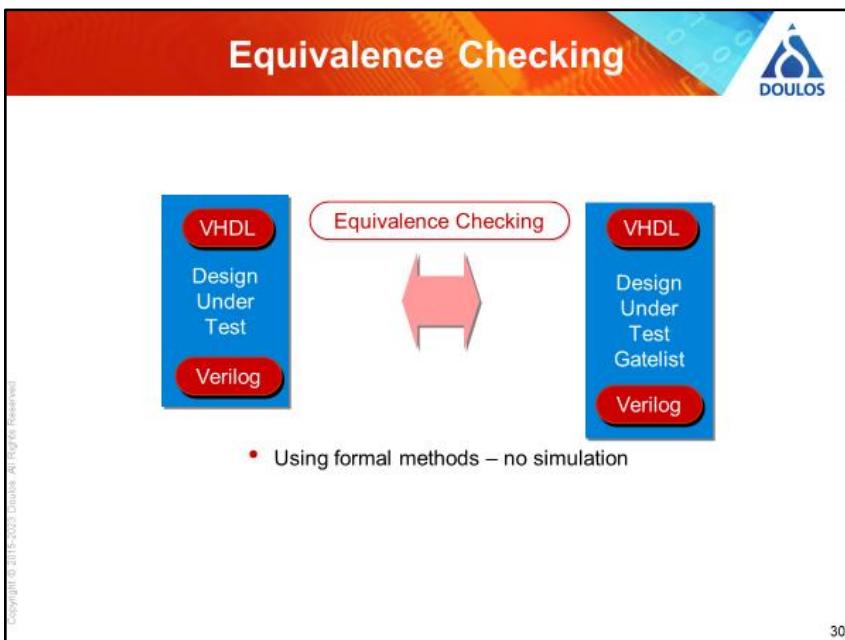
DOULOS

- The alternative to simulation is mathematical proof
 - Usually known as *Formal Methods*
 - Covers a wide variety of techniques
- Formal methods do not need stimuli – can be considered to be exhaustive
 - But not everything can be checked – best suited to state machines
- Why not always use formal methods?
 - State explosion problem – every possible sequence of states
- Two basic techniques:
 - Equivalence checking* – do two versions of the system have the same functionality
 - Property checking* – does a system satisfy certain properties?

Copyright © 2015-2023 Doulos. All Rights Reserved

28







Assertions

- An *assertion* is an instruction to check a *property* of the design
- Can be checked by a simulator or by a property checker

Simple Checks

```
assert output1 > output2
```

- Equivalent to combinational logic

Complex Temporal Checks

```
when input1 rises check that output1 > output2 after 2 clocks
```

- Equivalent to a state machine

Copyright © 2015-2023 Doulos. All Rights Reserved

32

Introduction to SystemVerilog



Introduction to Verification & SystemVerilog

- Introduction to Verification
- Introduction to SystemVerilog
 - What is SystemVerilog?
 - SystemVerilog Classes
 - Use of Interfaces
 - Constraints and Functional Coverage

33

What is SystemVerilog

What is SystemVerilog

The world's first HDVL, Hardware Design and Verification Language

- IEEE Std 1364-2005 Verilog and
- IEEE Std 1800-2005 SystemVerilog

merged to form

- IEEE Std 1800-2009 SystemVerilog
- IEEE Std 1800-2012 SystemVerilog
- IEEE Std 1800-2017 SystemVerilog

- SystemVerilog RTL**, aka concise RTL
- SystemVerilog Assertions**, aka SVA
- SystemVerilog Testbench**, or class-based verification

34

SystemVerilog Language Features

RTL + Programming	Assertions	Testbench
C-style data types & control - enum, struct, typedef, ++, break, return Synthesis-friendly "concise" RTL notation Packages Interfaces	SystemVerilog Assertions	Clocking blocks (synchronization between DUT and test bench) Object-oriented programming - classes Constrained random stimulus generation Functional coverage Dynamic processes, dynamic arrays, queues, mailboxes, semaphores
		Direct Programming Interface (DPI) - calling C from SystemVerilog Extensions to VPI

35

Caveats

Copyright © 2015-2023 Doulos. All Rights Reserved

- C-like control constructs and data types
- Concise RTL
- VHDL-like package and import
- Assertions

A better Verilog

- Non-portable constructs

III-defined

- Classes
- Constraints and coverage based on classes
- Built-in types - strings, queues, maps
- Virtual interfaces

Class-based verification

Used by standard verification methodologies

36

4-State and 2-State Types

Copyright © 2015-2023 Doulos. All Rights Reserved

- 4-state types

<i>Signed</i>	<i>Unsigned</i>	<i>Width</i>
logic signed	logic	1 bit
logic signed [n:m]	logic [n:m]	N bits

- 2-state types (variables only, not wires)

<i>Signed</i>	<i>Unsigned</i>	<i>Width</i>
bit signed	bit	1 bit
bit signed [n:m]	bit [n:m]	N bits
byte	byte unsigned	8 bits
shortint	shortint unsigned	16 bits
int	int unsigned	32 bits
longint	longint unsigned	64 bits

37

Struct

- Aggregate of dissimilar data items, just like C
- Best used with typedef

```

typedef struct {
    bit b;
    int i;
    logic [7:0] v;
} mystruct_t;

mystruct_t s;           ← Variable
s.b = 1;
s.i = -8;
s.v = 8'hff;

s = '{1, -8, 8'hff};   ← Assignment pattern

```

38

Interfaces

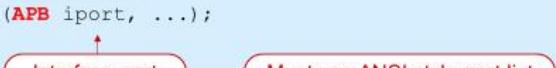
- Simple interface = bundle of wires/vars

```

interface APB;
    logic PCLK, PSEL, PENABLE, PWRITE;
    logic [15:0] PADDR;
    logic [31:0] PWDATA;
    logic [31:0] PRDATA;
endinterface

module Master (APB iport, ...);
    ...
endmodule

```




39

Immediate and Concurrent Assertion

DOULOS

- Procedural assertion – sampled procedurally

```
always ...
  assert ( EXPRESSION );
```

Ordinary SystemVerilog expression

- Concurrent assertion – condition is usually sampled on clock edge

```
assert property ( @ (posedge Clock) CONDITION );
```

SystemVerilog property

- Condition is only tested when pre-condition has been matched

```
assert property (
  @ (posedge Clock) PRECONDITION I-> CONDITION );
```

SystemVerilog sequence Implication operator

Copyright © 2015-2023 Doulos. All Rights Reserved

40

Concurrent Assertions

DOULOS

- Check or prove the property

```
label: assert property ( PROPERTY ) ACTION_BLOCK;
```

Important

- Collect functional coverage information

```
label: cover property ( PROPERTY ) STATEMENT;
```

- Make the property an assumption for formal

```
label: assume property ( PROPERTY );
```

Copyright © 2015-2023 Doulos. All Rights Reserved

41

Temporal Behaviour

DOULOS

- Properties and sequences describe temporal behaviour
- "Temporal" means the sequence spans more than one clock cycle

```
assert property (
    @posedge Clock) (a ##1 b) |-> (d ##1 e)
);
```

Property

- Termination mid-way through matching a sequence
 - (Discharges property's obligation to hold for PROPERTY)

```
assert property (
    @posedge Clock) disable iff (TERMINATE) PROPERTY;
```

Expression

Termination operator

42

SystemVerilog Classes

SystemVerilog Classes

DOULOS

```
package Bus_pkg;
    typedef logic [15:0] T_addr;
    typedef logic [15:0] T_data;
    typedef enum bit (dir_Rd, dir_Wr) T_dir;
    class Bus_trans;
        int ID;
        T_dir dir;
        T_addr addr;
        T_data data;
    endclass : Bus_trans
    function void print();
        string kind = (dir==dir_Rd) ? "Read" : "Write";
        $display("%s cycle #%0d: A=%h, D=%h",
            kind, ID, addr, data);
    endfunction : print
endpackage : Bus_pkg
```

Always put classes in a package

Class defines a transaction object

Data members or class properties

Method

43

Object = Instance of Class

DOULOS

```
module use_Bus_trans;
import Bus_pkg::*;
Bus_trans t1, t2;
initial begin
    ...

```

t1 [] references
t2 []

- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)

4

Object = Instance of Class

DOULOS

```
module use_Bus_trans;
import Bus_pkg::*;
Bus_trans t1, t2;
initial begin
    t1 = new;
    ...

```

t1 [] references
t2 []

Bus_trans object

ID	0
dir	dir_Rd
addr	xxxx
data	xxxx

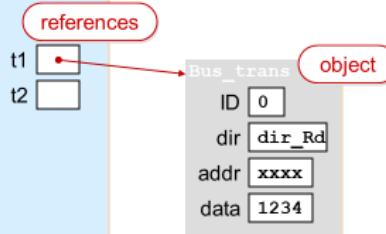
- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values

45

Object = Instance of Class

```
module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
  ...

```



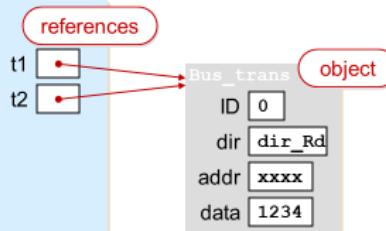
- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

46

Object = Instance of Class

```
module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
  ...

```



- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

47

Object = Instance of Class

Copyright © 2015-2023 Doulos. All Rights Reserved

The diagram illustrates the relationship between variables and objects. On the left, a Verilog module code snippet is shown:

```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
    t2.data = 16'habcd;
  ...

```

Variables `t1` and `t2` are declared as `Bus_trans` type. In the `initial` block, `t1` is created using `new`, which initializes its data member to `16'h1234`. Then, `t2` is assigned the value of `t1`, and its data member is updated to `16'habcd`.

On the right, a state diagram shows the `Bus_trans` object with the following fields:

- `ID`: 0
- `dir`: `dir_Rd`
- `addr`: `xxxx`
- `data`: `abcd`

Two boxes labeled `t1` and `t2` represent references to the object. Arrows point from these boxes to the `object` box, indicating that `t1` and `t2` store references to the same object instance.

List of Points:

- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

48

Object = Instance of Class

Copyright © 2015-2023 Doulos. All Rights Reserved

The diagram illustrates the relationship between variables and objects, similar to the previous slide, but includes a method call example.

The Verilog module code is identical to the one in the previous slide:

```

module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;
    t2 = t1;
    t2.data = 16'habcd;
    t1.print(); // Method call
  ...

```

In addition to the variable assignments and object creation, the line `t1.print();` is highlighted with a yellow box and labeled `Method call`. A yellow callout box to the right of the object state diagram displays the output of the `print` method: `Read cycle #0: A=xxxx, D=abcd`.

List of Points:

- Variables of class type store *references* (handles) to real objects
 - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation
 - Methods act on the object through which they were called

49

Initializing Objects

DOULOS

```

module use_Bus_trans;
    import Bus_pkg::*;

    Bus_trans t3 = new;           Create object with default initial values

```

- **new** allocates memory and calls default constructor

```

class Bus_trans;
    ...
    function new;             ← Explicit constructor new. No return type
        addr = 0;
        dir = dir_Wr;
        $write("Created new ");
        print();
    endfunction : new          ← Call a method from within the class
                                ← print myself
    ...

```

50

Constructor Arguments

DOULOS

- Like any function in SystemVerilog, constructors may have arguments

```

function new (T_dir direction);
    addr = 0;
    dir = direction;
    $write("Created new "); print();
endfunction : new

```

```

Bus_trans t4 = new;           ERROR
Bus_trans t4 = new(dir_Rd); Read

```

- Arguments may have default values (no overloading, though)

```

function new (T_dir direction = dir_Rd); ...

```

```

Bus_trans t5 = new;           Read
Bus_trans t6 = new(dir_Wr); Write

```

51

Randomized Data Members

DOULOS

```

class Bus_trans;
    static int next_ID;
    const int ID;
    rand T_dir dir;      any data member can be declared rand
    rand T_addr addr;
    rand T_data data;
    function new;
        ID = next_ID++;
    endfunction : new
    function void print; ...
endclass : Bus_trans

Bus_trans tR;      tR
repeat (3) begin
    tR = new;
    void'( tR.randomize() );
    tR.print();
end

```

unique serial number

Write cycle #0: A=35e7, D=4a8f
Write cycle #1: A=b267, D=04e3

52

Randomized Data Members

DOULOS

```

class Bus_trans;
    static int next_ID;
    const int ID;
    rand T_dir dir;      any data member can be declared rand
    rand T_addr addr;
    rand T_data data;
    function new;
        ID = next_ID++;
    endfunction : new
    function void print; ...
endclass : Bus_trans

Bus_trans tR;      tR
repeat (3) begin
    tR = new;
    void'( tR.randomize() );
    tR.print();
end

```

randomize an existing object

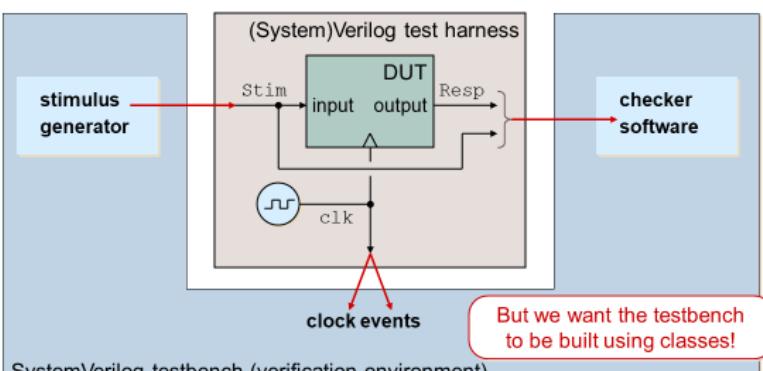
Write cycle #0: A=35e7, D=4a8f
Write cycle #1: A=b267, D=04e3
Read cycle #2: A=1040, D=93c2

53

Test Harness and Testbench



- Test harness is a module containing:
 - the DUT instance and connections to its ports
 - clock generator and other support structures



SystemVerilog testbench (verification environment)

54

Lifetime and Persistence



- Module, interface and program instances:
 - created at elaboration, before simulation begins
 - hierarchy structure controlled by parameters
 - structure/instances cannot be changed dynamically

- Objects of class type:
 - created dynamically, during simulation, using new
 - structure controlled by run-time activity
 - can be created and destroyed at any time

verification environment	transaction data objects
--------------------------	--------------------------

- typically constructed at time zero
- structure probably remains unchanged throughout simulation
- created in large numbers during the simulation
- destroyed after use (unless logged)

55

Creating the Testbench

Copyright © 2015-2023 Doulos. All Rights Reserved

```

module TB_top;
  import TB_pkg::*;
  TB_env tb; tb = new();
  initial begin
    tb.run();
  end
endmodule : TB_top

class TB_env;
  ...
  task drive_Stim(input bit data);
    @(posedge harness.clk)
    harness.Stim <= data;
  endtask
  ...

```

```

module harness;
  logic Stim, Resp;
  bit clk;
  Sys_Top DUT (.%);
  ...
endmodule

```

- Our entire testbench class is hard-coded for the name of the test harness!

56

Use of Interfaces

Virtual Interface

Copyright © 2015-2023 Doulos. All Rights Reserved

```

class TB_env;
  virtual TB_hook hook;
  function new(virtual TB_hook h);
    hook = h;
  ...
  endfunction : new
  task drive_Stim(input bit data);
    @(posedge hook.clk)
    hook.Stim <= data;
  endtask
  ...

```

```

interface TB_hook;
  logic Stim, Resp;
  bit clk;
endinterface

```

```

interface TB_hook
  clk _____
  Stim _____
  Resp _____

```

- Testbench class now OK for any instance of a TB_hook interface
- Link TB object to interface instance at runtime
- Where is this instantiated?
- How does it link to the DUT?

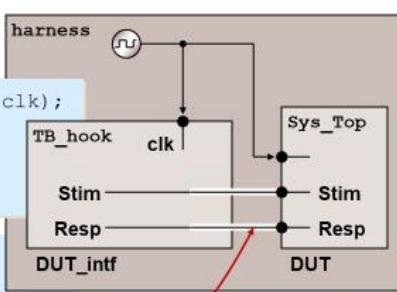
57

Building a test harness

DOULOS

Interface contains all signals required by testbench

```
interface TB_hook (input bit clk);
    wire Stim, Resp;
    ...
endinterface
```



Hierarchical connection

Harness contents

- DUT instance
- DUT connections
- clock generator

58

Connecting the virtual interface

DOULOS

test harness

```
class TB_env;
    virtual TB_hook V;
    function new (virtual TB_hook V, ...);
        this.V = V;
        ...
    endfunction
    ...

```

constructor

Choice of interface type

```
module TB_top;
    TB_env tb;
    ...
    initial begin
        tb = new(harness.DUT_intf, ...);
    end

```

Choice of instance

59

Testbench Static Structure

DOULOS

- Test case (TB_top) and test harness are the only static instances

```
module TB_top;
  import TB_pkg::*;
  TB_env tb;
  initial begin
    tb = new(...);
    tb.run();
  end
endmodule : TB_top
```

The diagram illustrates the static simulation structure. It shows the `TB_top` module containing a `tb` object of type `TB_env`. The `TB_env` class has a `hook` virtual method. The `tb` object's `hook` method is connected to the `Stim` and `Resp` ports of a `DUT_intf`, which in turn connects to the `DUT`. The `DUT` is also connected to `Sys_Top`, which provides `Stim` and `Resp` signals. A `harness` block is shown with a `clk` signal connected to the `tb`'s `hook` method.

```
package TB_pkg;
class TB_env;
  ...
  virtual TB_hook hook;
  ...
endpackage
```

The diagram shows the `TB_env` class definition. It includes a `hook` method and other class members. A red box highlights the `hook` method, with a callout pointing to it labeled "dynamically constructed testbench object".

60

Copyright © 2015-2023 Doulos. All Rights Reserved

Constraints and Functional Coverage

Constrained randomization

DOULOS

```
class Bus_trans;
  rand T_dir dir;
  rand T_addr addr;
  rand T_data data;
  constraint rom_area {
    dir == dir_Rd; addr <= 16'h7FFF;
  }
  ...
endclass : Bus_trans
```

System has ROM at low addresses

	dir	Rd	Wr
FFFF	X	X	
8000		C	X
7FFF		C	
0000	C		X

- But now there will be *no* access to high addresses!
- Solution: use an *implication constraint*

```
constraint low_adrs_is_ROM {
  (addr <= 16'h7FFF) -> (dir == dir_Rd);
}
```

	dir	Rd	Wr
FFFF	C	C	C
8000	C	C	
7FFF	C	C	
0000	C		X

61

Copyright © 2015-2023 Doulos. All Rights Reserved

Creating an Extended Class

```
class Bus_trans;
    rand T_dir dir;
    rand T_addr addr;
    rand T_data data;
```

General, re-usable

Better not to mix
these together...

```
constraint low_adrs_is_ROM {
    (addr <= 16'h7FFF) -> (dir == dir_Rd);
}
```

Specific to the current DUT

- Don't *modify* the original class definition
- Instead, *extend* it:

```
class Mem_map_trans extends Bus_trans;
    constraint low_adrs_is_ROM {
        (addr <= 16'h7FFF) -> (dir == dir_Rd);
    }
    ...

```

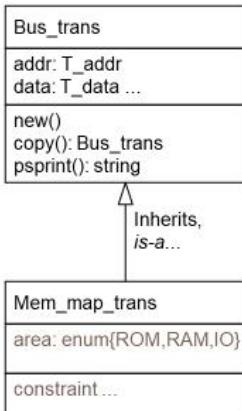
Everything in the base class, plus...

62

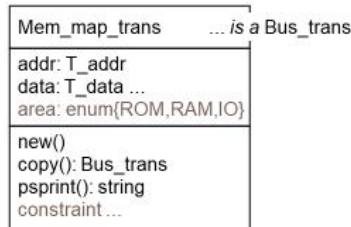
Inheriting Class Members



- What you write



- What you get



- A *Mem_map_trans* object can be used anywhere a *Bus_trans* object is appropriate

63

Functional Coverage

Doulos

- Covergroup can be in a module, interface or class

```

class Monitor;
  rand logic [2:0] opcode;
  rand logic [1:0] mode;

  covergroup cg;
    coverpoint opcode;
    coverpoint mode;
    option.per_instance = 1;
  endgroup

  function new;
    cg = new;
    ...
  endfunction

  task body;
    ... wait for transaction
    cg.sample();
    ...
  endtask

```

Auto-binning

Coverage hole

Does not count X or Z

Must instantiate covergroup in constructor!

Built-in method

64

EDA Playground

Doulos

playground

Bringing you by DOULOS

Languages & Libraries

Testbench • Designs

SystemVerilog/UVM

None

Other Libraries

None

SV 2.3.1

SV 2.11

Enable TL-Verilog

Enable SystemVerilog

Enable VUnit

Tools & Simulators

Aldec Riviera Pro 2022.04

Compile Options

Simulate Testbenches

Run Options

Examples

Community

Log in

Run Copy

KnowHow Dealing with Complexity in Formal SEMINARS

REGISTER NOW

Playgrounds

Practice - Share - Learn

Simulate your code in a web browser

https://www.edaplayground.com/

3642 views and 2 likes

Connecting a SystemVerilog interface to a class-based verification environment using a virtual interface

65



The slide is a promotional graphic for DOULOS. It features a central white area with a red-to-white gradient background. On the left, there are four sections with red headers: "SoC Design & Verification", "FPGA & Hardware Design", "Embedded Software", and "Python & Deep Learning". To the right of each section is a list of technologies. Below these lists are two small icons: Python and Deep Learning. At the bottom of the slide, there is a row of various technology logos, including SystemVerilog, UVM, SystemC, Linux (Penguin), Yocto, RTOS, SELinux, and ARM.

SoC Design & Verification

- » SystemVerilog
- » UVM
- » Formal
- » SystemC
- » TLM-2.0

FPGA & Hardware Design

- » VHDL
- » Verilog
- » SystemVerilog
- » Tcl
- » Xilinx
- » Intel FPGA (Altera)

Embedded Software

- » Emb C/C++
- » Emb Linux
- » Yocto
- » RTOS
- » Security
- » Arm

Python & Deep Learning

SystemVerilog OUT OF THE BOX UVM SYSTEMC     



www.doulos.com

