Introduction to Verification and SystemVerilog

Delivering KnowHow

Introduction to Verification and SystemVerilog

www.doulos.com

Introduction to Verification and SystemVerilog

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

Tel: +44 (0) 1425 471223

Email: info@doulos.com

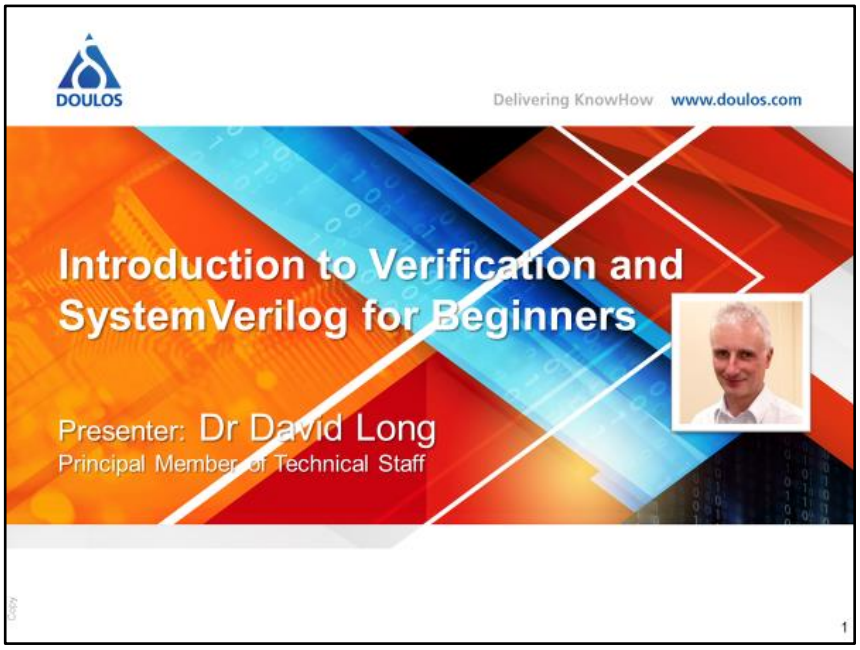
Doulos
6203 San Ignacio Avenue, Suite 110,
San Jose, CA 95119, USA

Tel: 1-888-GO DOULOS

Email: info.usa@doulos.com

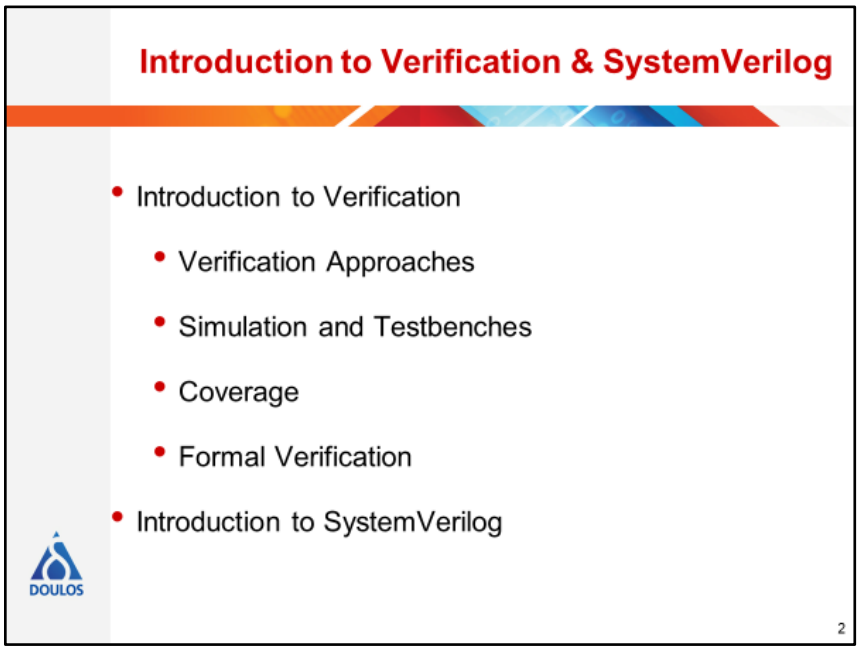
www.doulos.com

Contents

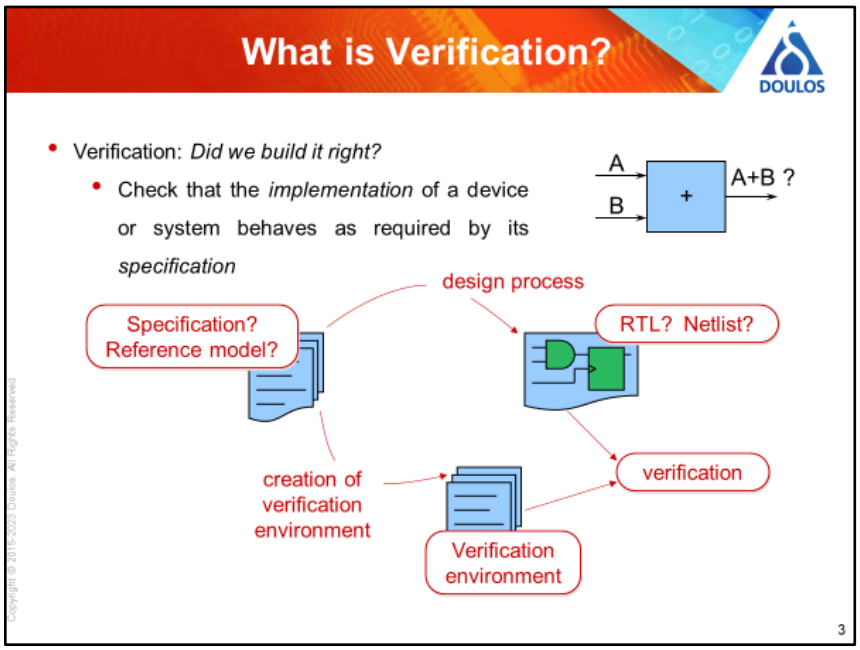


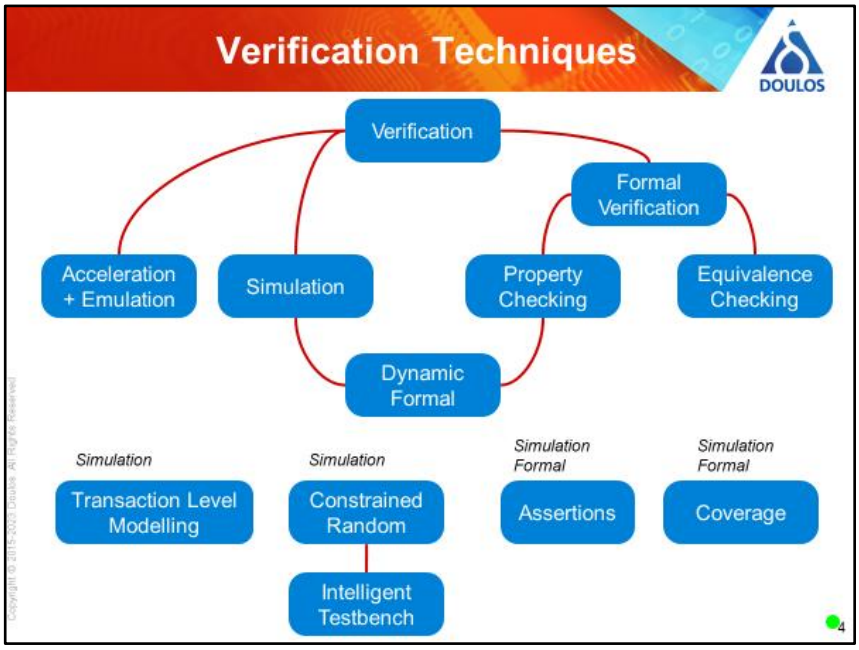
Notes

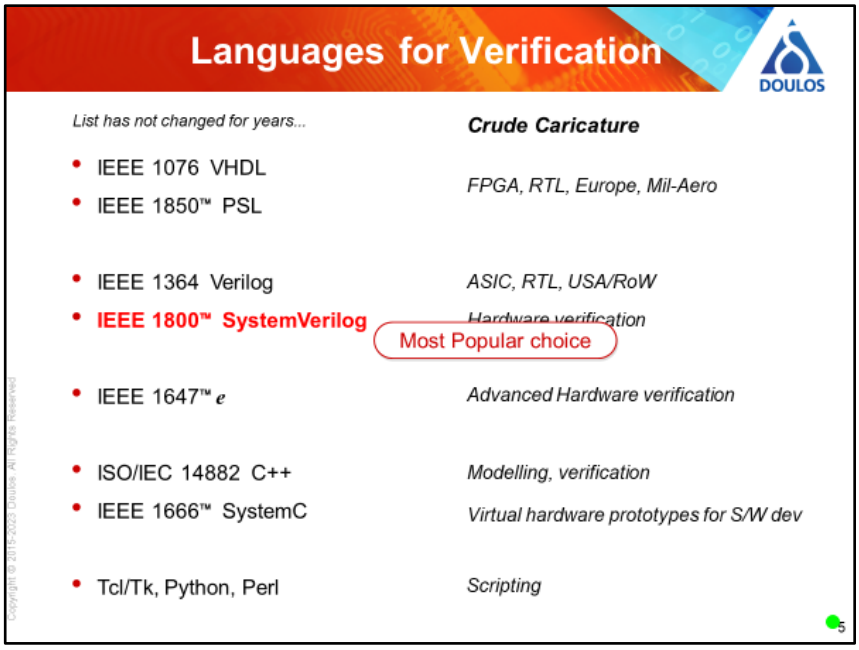
Introduction to SystemVerilog



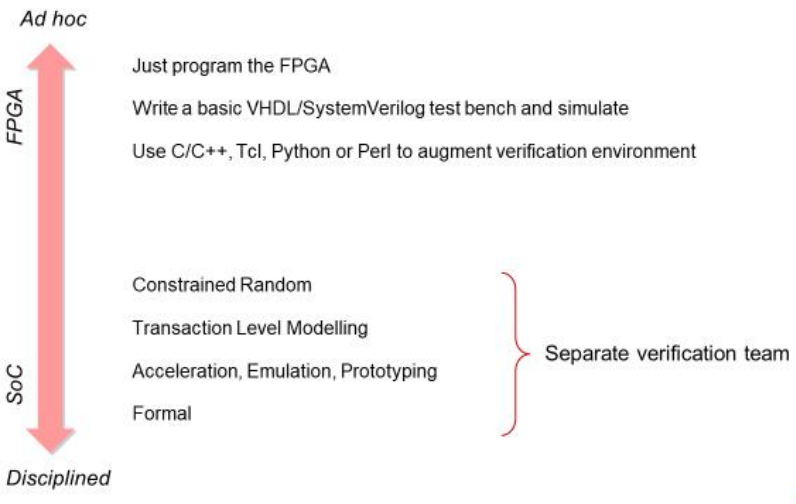
Verification Approaches





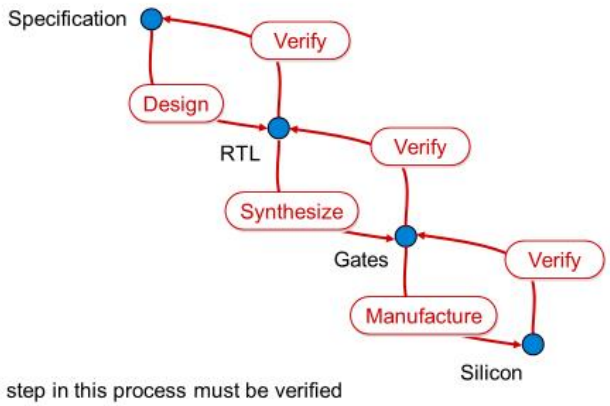


Approaches to Verification



6

Verification in the Design Process



- Each step in this process must be verified

7

Verification Plan

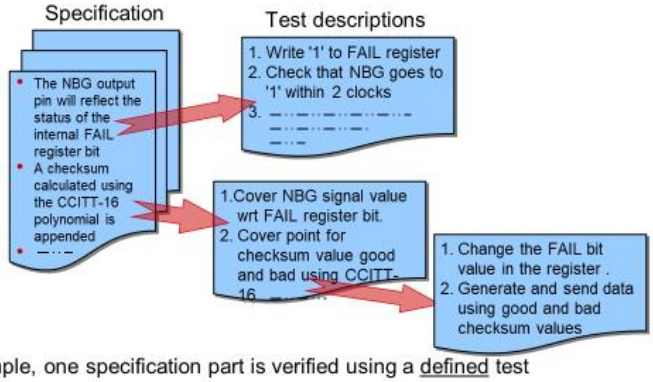
- What are we going to verify? — Requirements, objectives
- How are we going to do it? — Design of test stimulus
- How do we measure success? — Observations

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



- 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

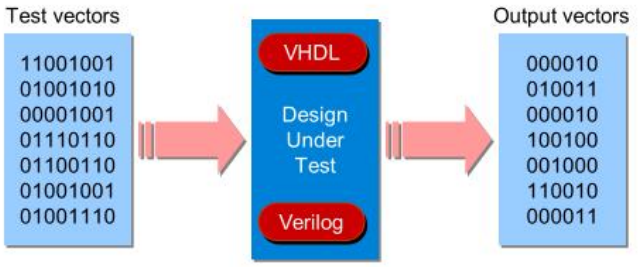
Simulation

- 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

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

11

Simulating a digital system



- #1 tool for functional verification
- Vectors must be created:
  - write a program to do it ("directed")

  OR
  - compute them using constrained-random techniques

    (requires automated checking of output)

12

Results from Simulation

- Only an approximation of reality
- Semantics of HDL and simulator define simulation behaviour
- Simulation requires stimulus
  - quality of test is determined by quality of stimulus

Results are determined by both the design and the stimulus

- Response must be verified against expected behaviour
- General problem: Simulators are never fast enough!

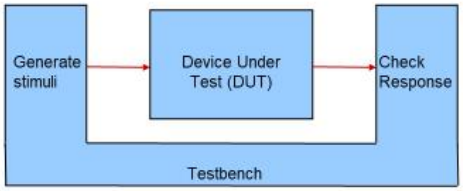
13

Basic Testbench

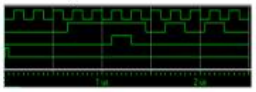
Often written by the designer

Unstructured grey box test

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



Inspect waveform?



14

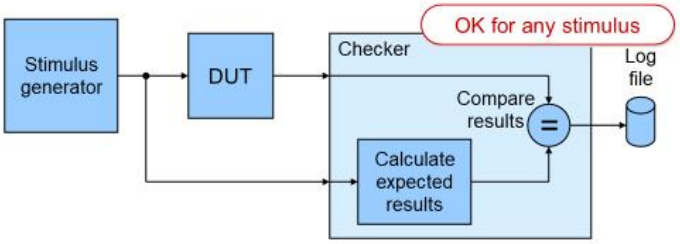
Testbenches

- 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

- 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



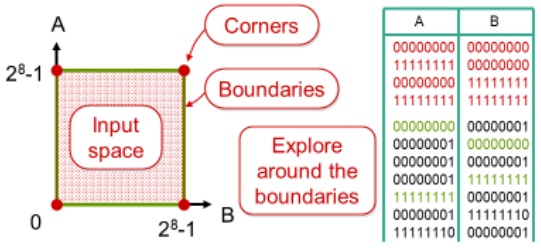
16

Boundary Conditions & Corner Case

- 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

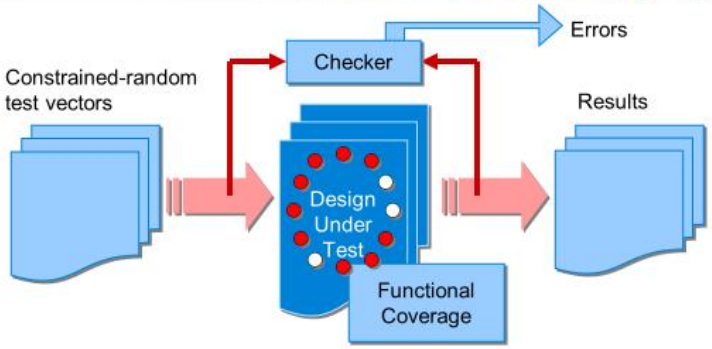
Constrained Random Tests

- 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

"Directed" tests

18

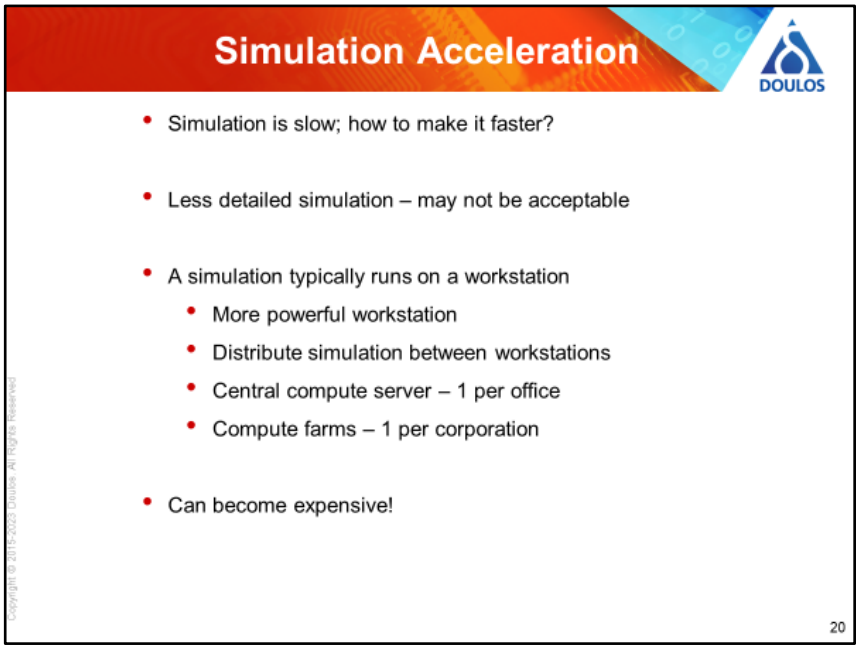
Testbench Automation

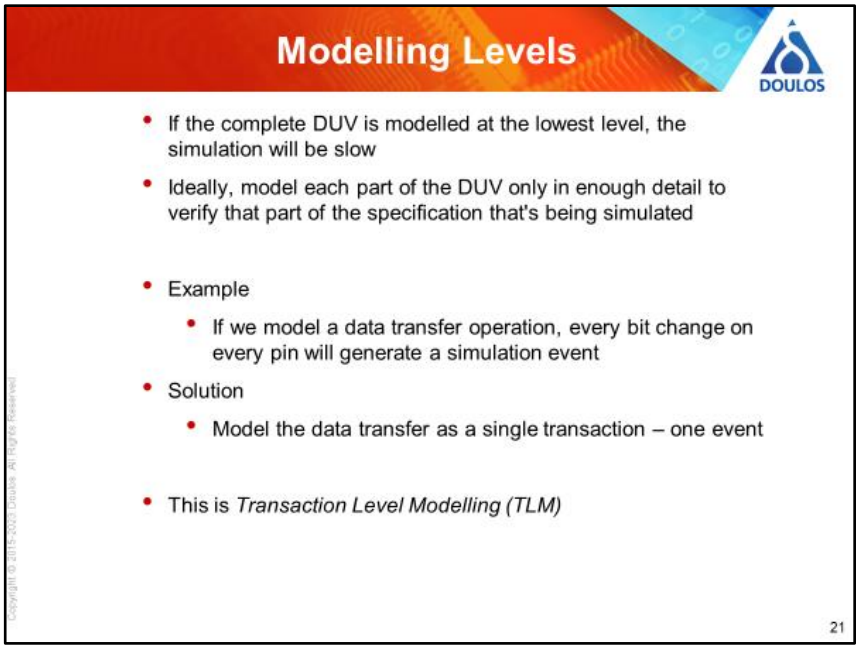


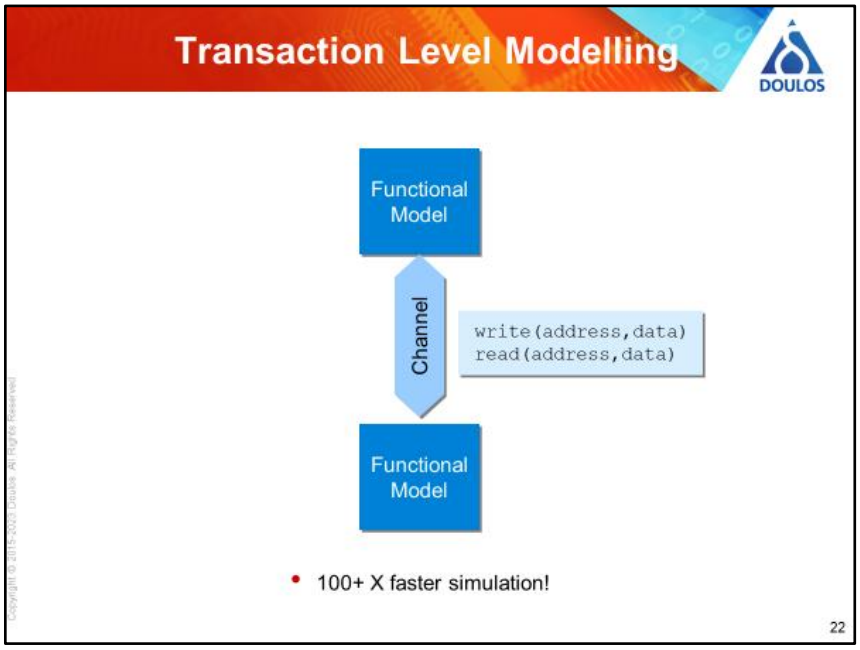
Principles of testbench automation:

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

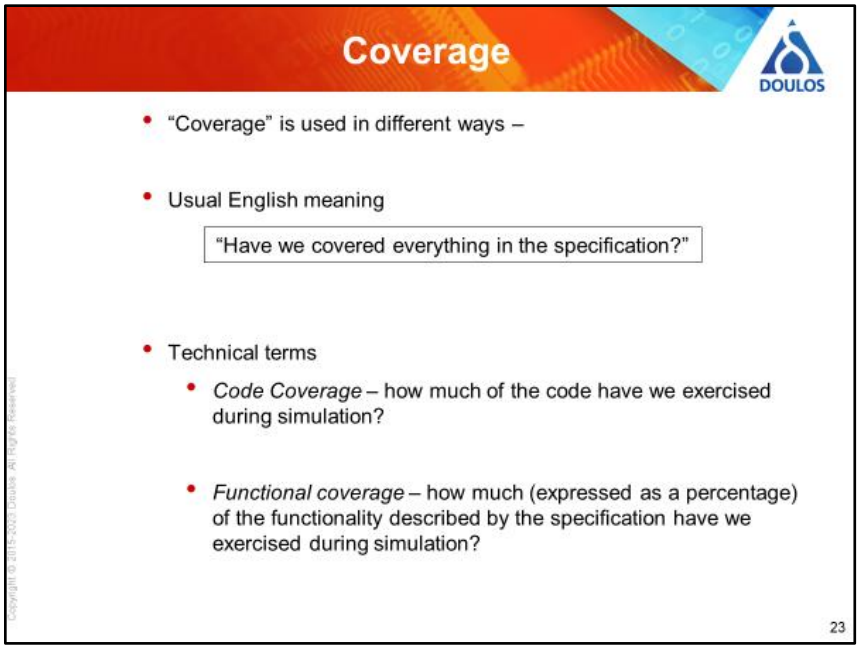
19

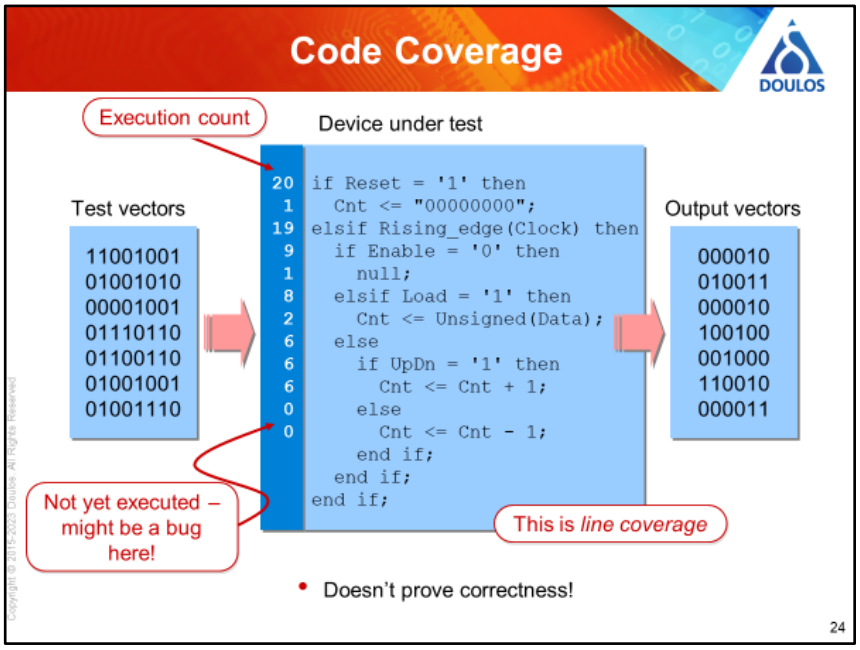


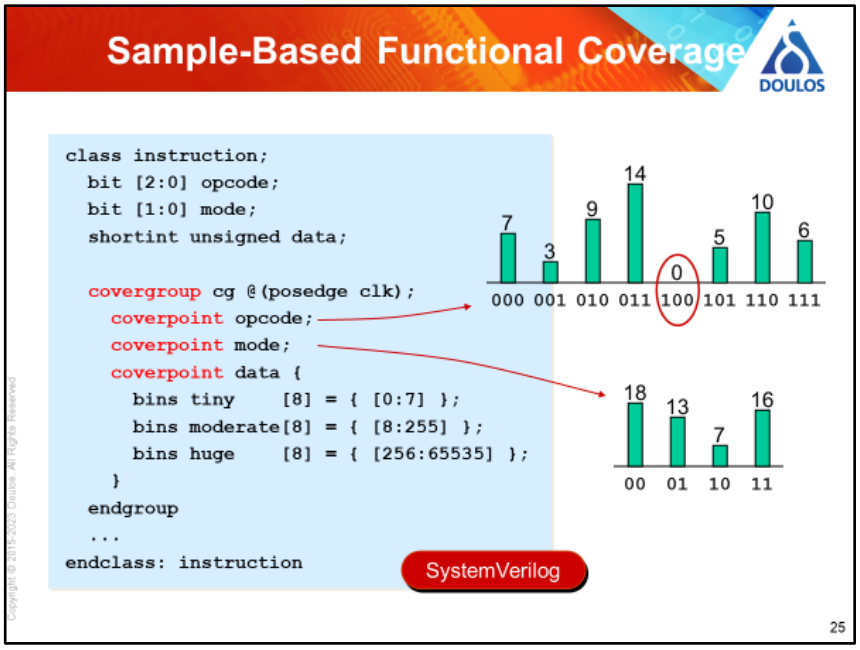




Coverage



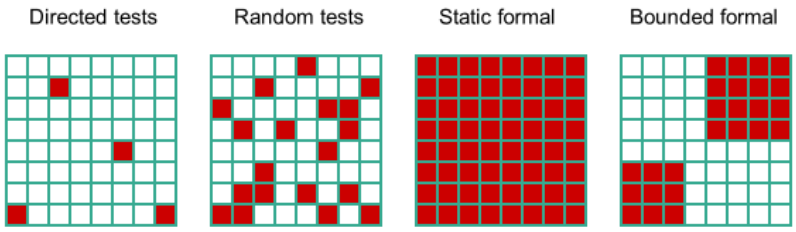




Code vs. Functional Coverage

26

Verification State Space Coverage



- Directed testing relies on the patient engineer
- Random tests use CPU time/knowledgeable engineer to increase coverage
- Static formal verification (property checking) is exhaustive of the state space
  - Control circuits best use
- Bounded formal verification is locally exhaustive, but incomplete

27

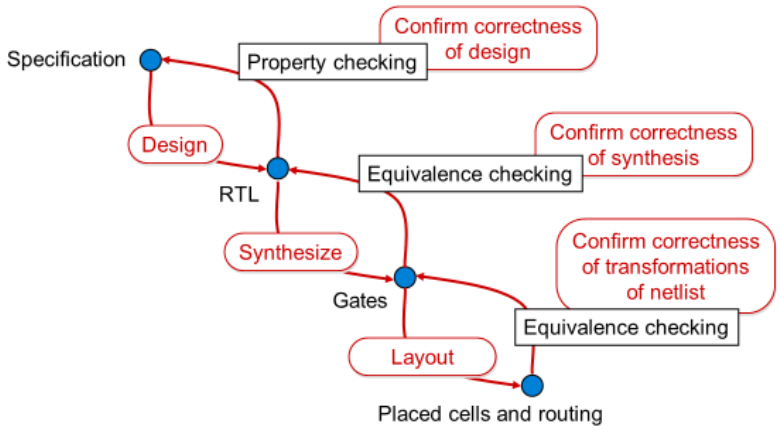
Formal Verification

Formal Verification

- 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?

28

When Is Formal Verification Used?



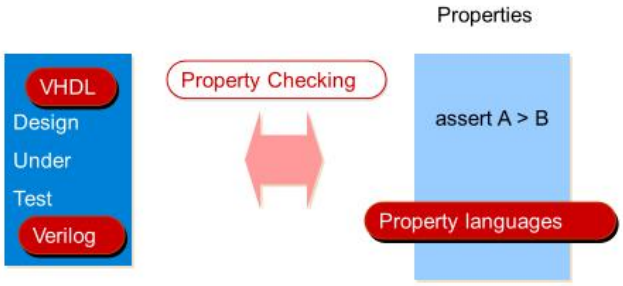
29

Equivalence Checking

- Using formal methods – no simulation

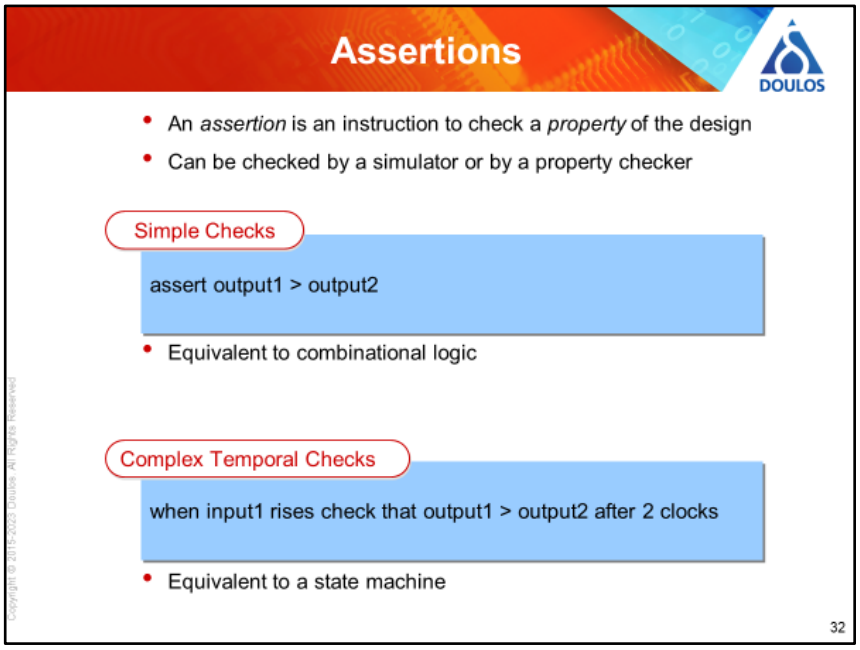
30

Property Checking

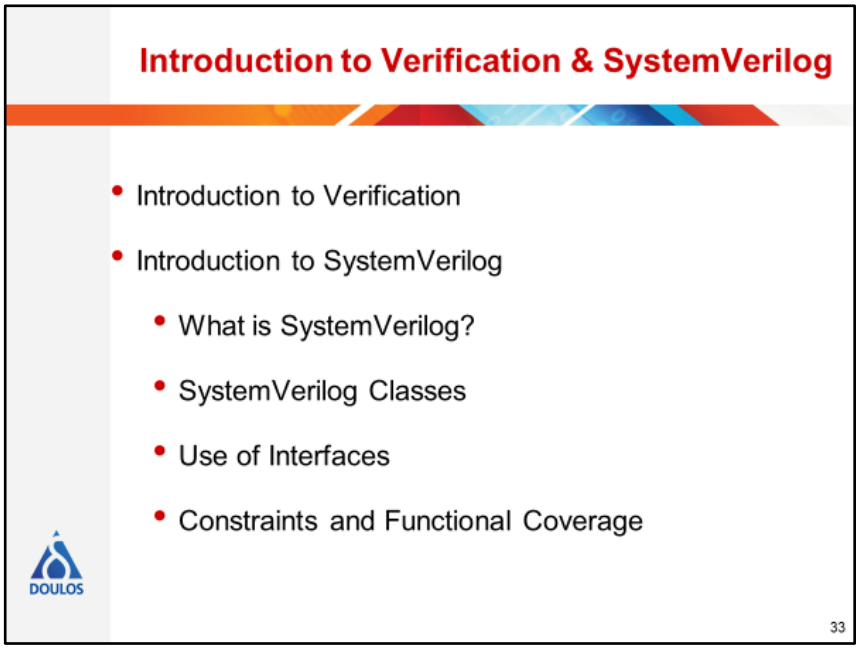


- Using formal methods – no simulation
- Exhaustive state space coverage

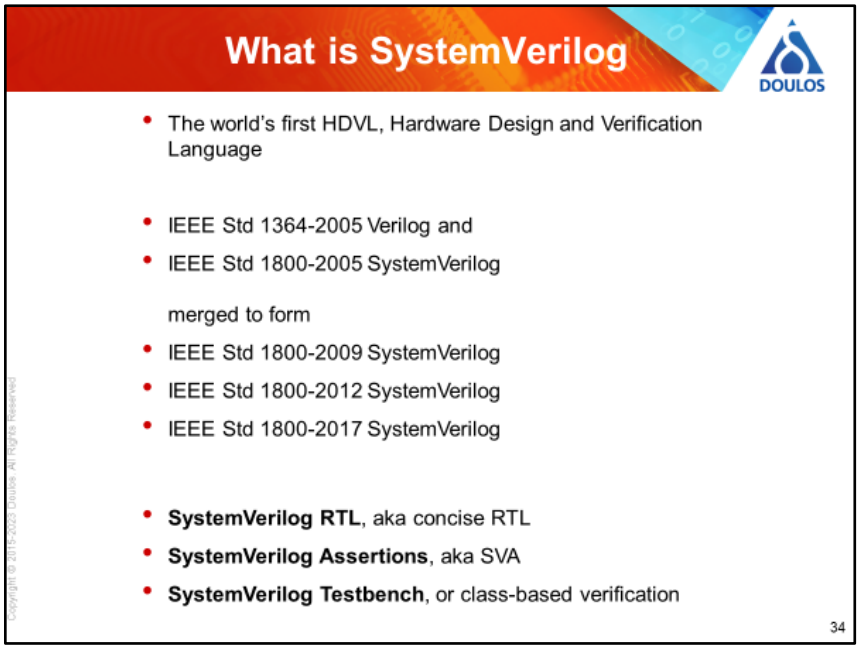
31

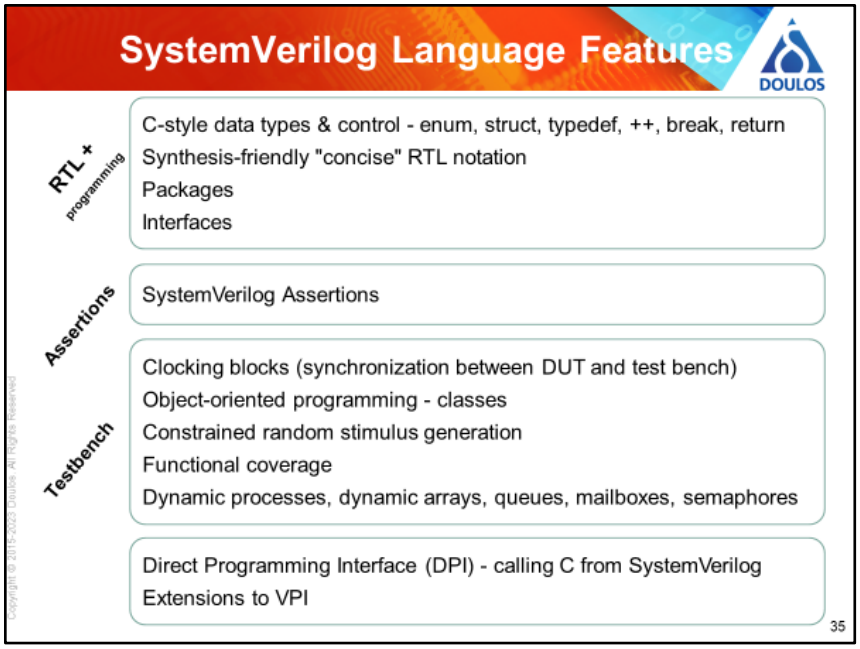


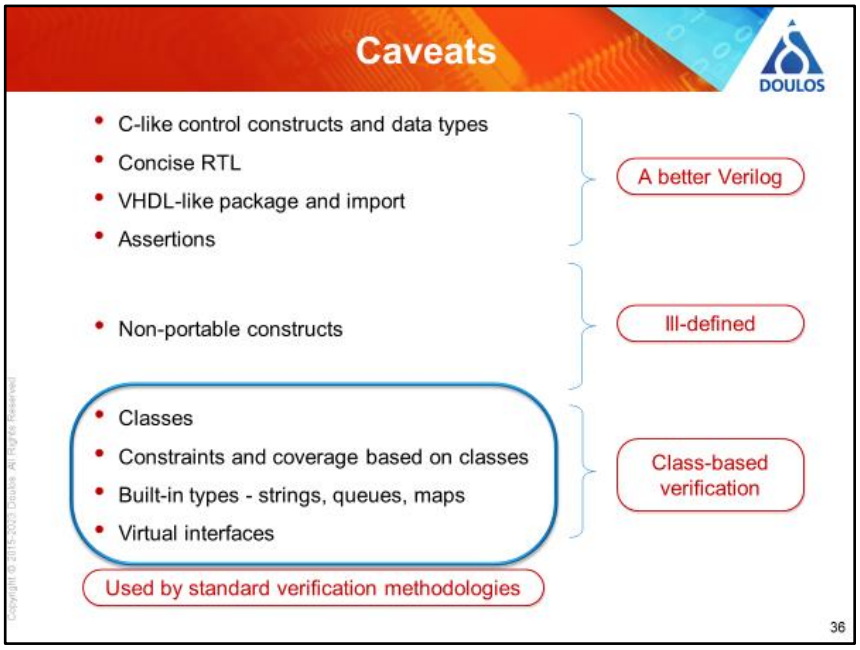
Introduction to SystemVerilog

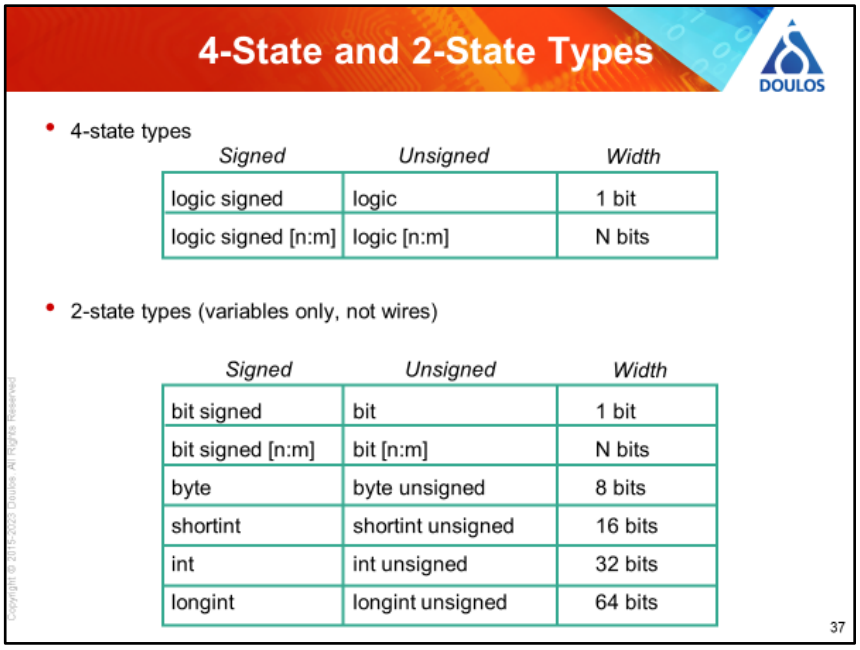


What is SystemVerilog









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

s = '{1, -8, 8'hff};
```

Variable

Assignment pattern

38

Interfaces

- Simple interface = bundle of wires/vars

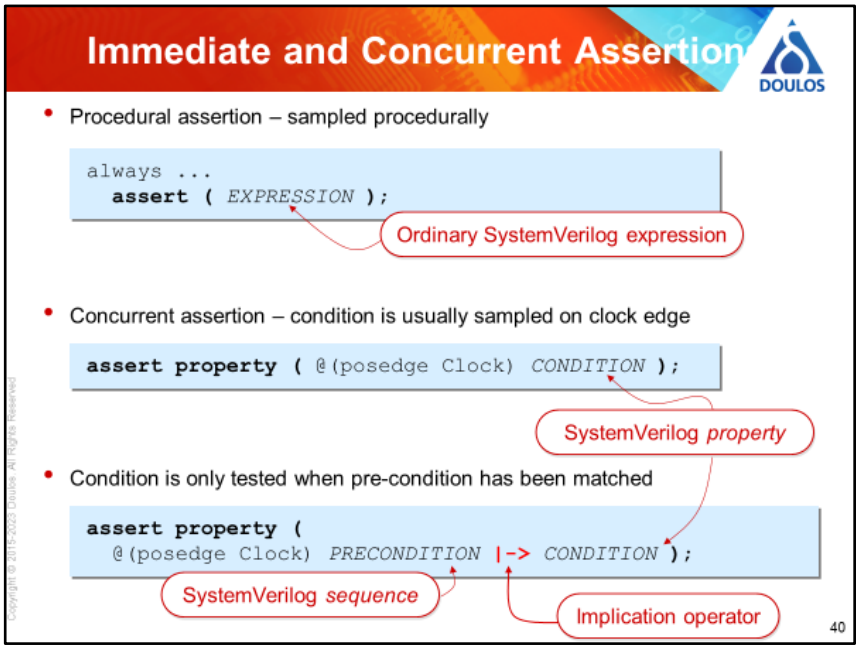
```
interface APB;
  logic PCLK, PSEL, PENABLE, PWRITE;
  logic [15:0] PADDR;
  logic [31:0] PWDATA;
  logic [31:0] PRDATA;
endinterface
```

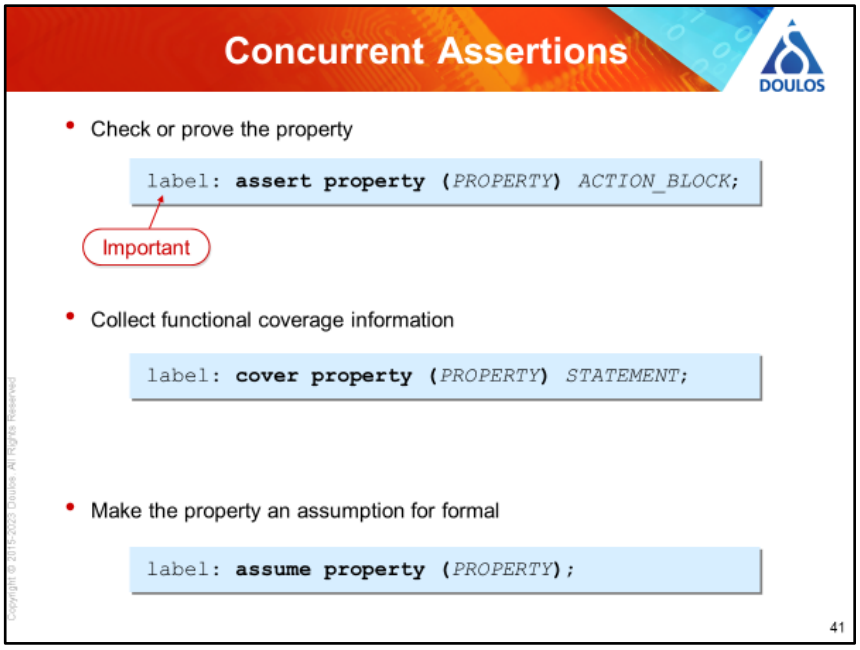
```
module Master (APB iport, ...);
  ...
endmodule
```

Interface port

Must use ANSI-style port list

39





Temporal Behaviour

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

Concurrent assertion

Sequences

```
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

Always put classes in a package

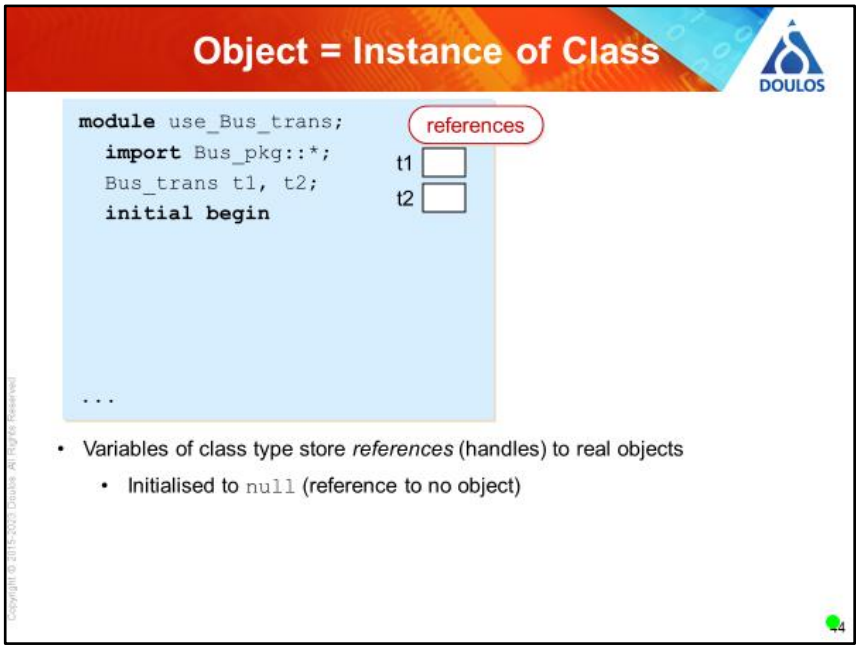
```
package Bus_pkg;
  typedef logic [15:0] T_addr;
  typedef logic [15:0] T_data;
  typedef enum bit {dir_Rd, dir_Wr} T_dir;
  class Bus_trans;
    int    ID;
    T_dir  dir;
    T_addr addr;
    T_data data;
    function void print;
      string kind = (dir==dir_Rd) ? "Read"  : "Write";
      $display("%s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

endpackage : Bus_pkg
```

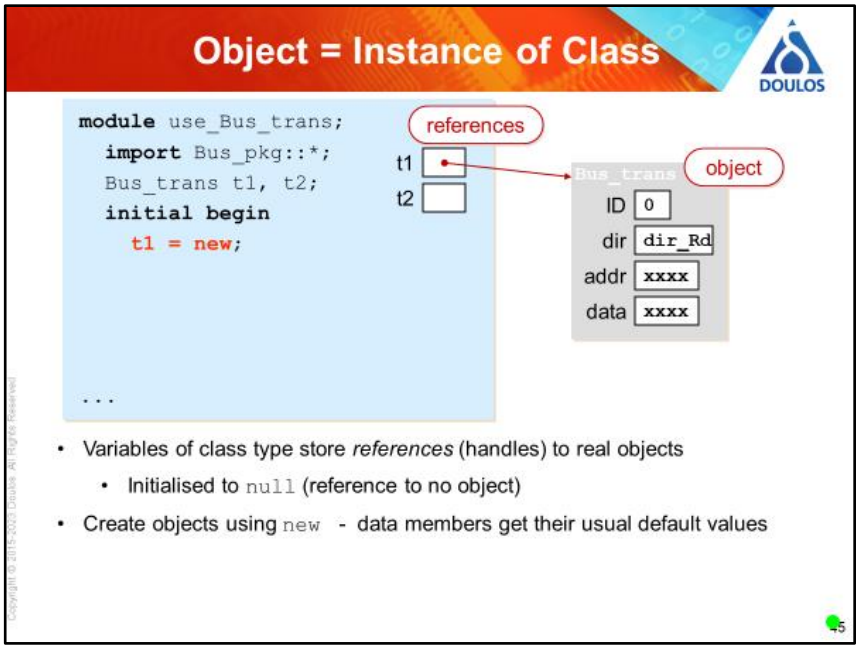
Class defines a transaction object

Data members or class properties

Method

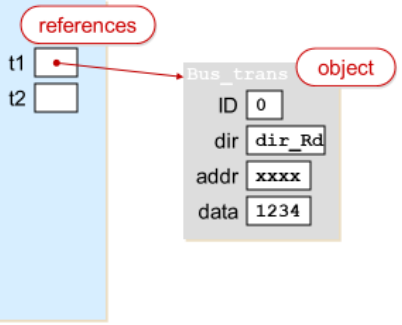
43





Object = Instance of Class

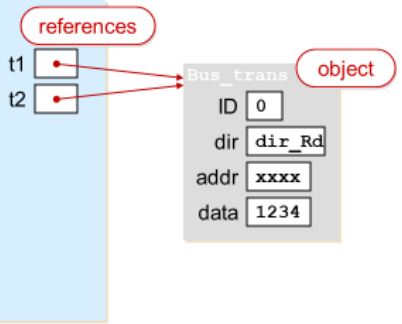
```
module use_Bus_trans;
  import Bus_pkg::*;
  Bus_trans t1, t2;
  initial begin
    t1 = new;
    t1.data = 16'h1234;


...
```



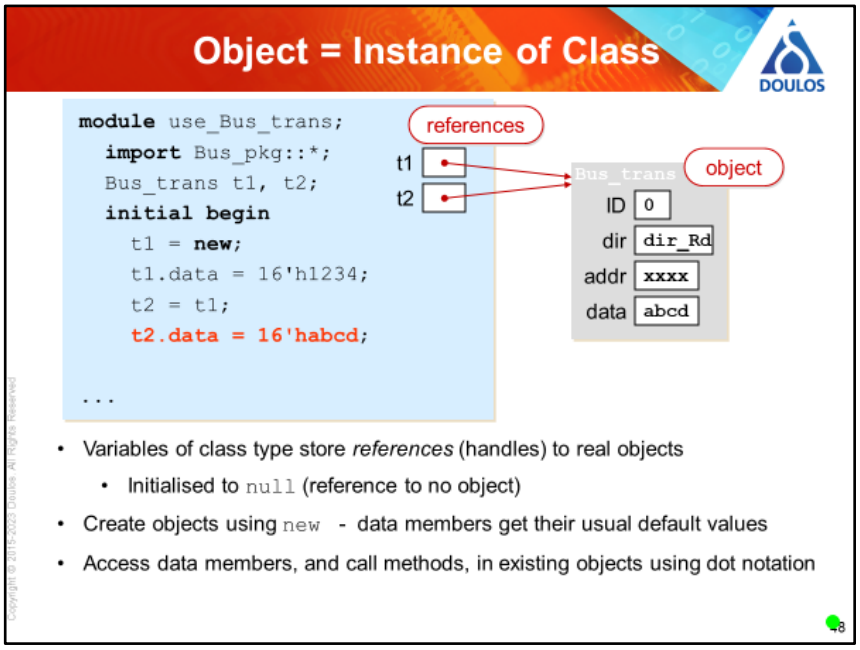
- Variables of class type store *references* (handles) to real objects
  - Initialised to `null` (reference to no object)
- Create objects using `new` - data members get their usual default values
- Access data members, and call methods, in existing objects using dot notation

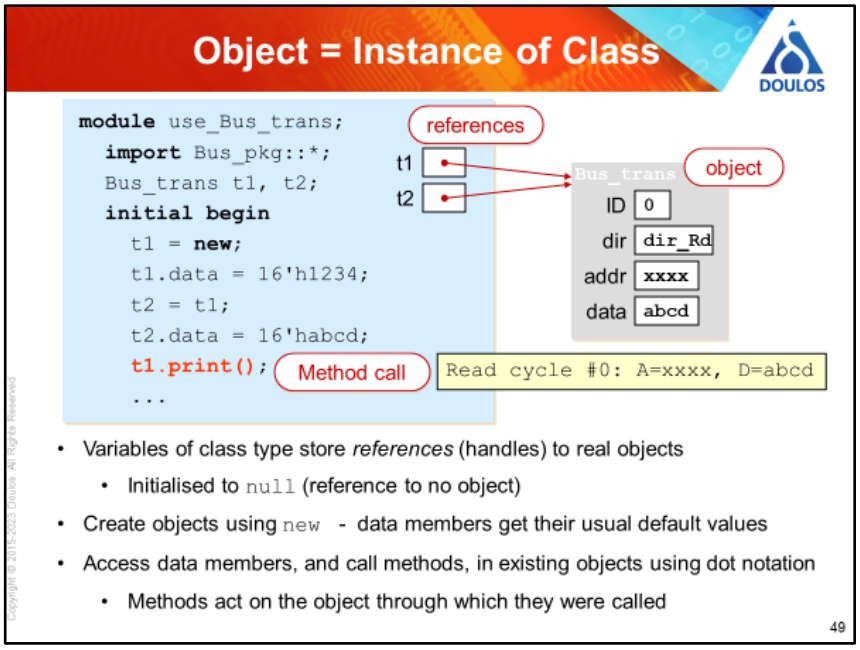
Object = Instance of Class

```
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





Initializing Objects

```
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;
    addr = 0;
    dir = dir_Wr;
    $write("Created new ");
    print();
  endfunction : new
  ...
```

Explicit constructor new. No return type

Call a method from within the class

print myself

50

Constructor Arguments

- 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;
Bus_trans t4 = new(dir_Rd);
```

ERROR

Read

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

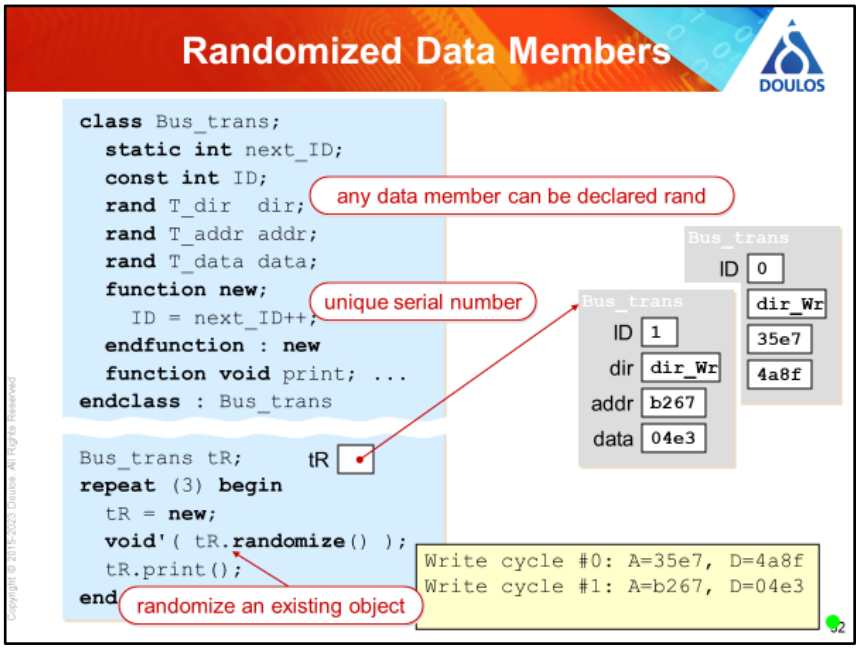
```
function new (T_dir direction = dir_Rd); ...
```

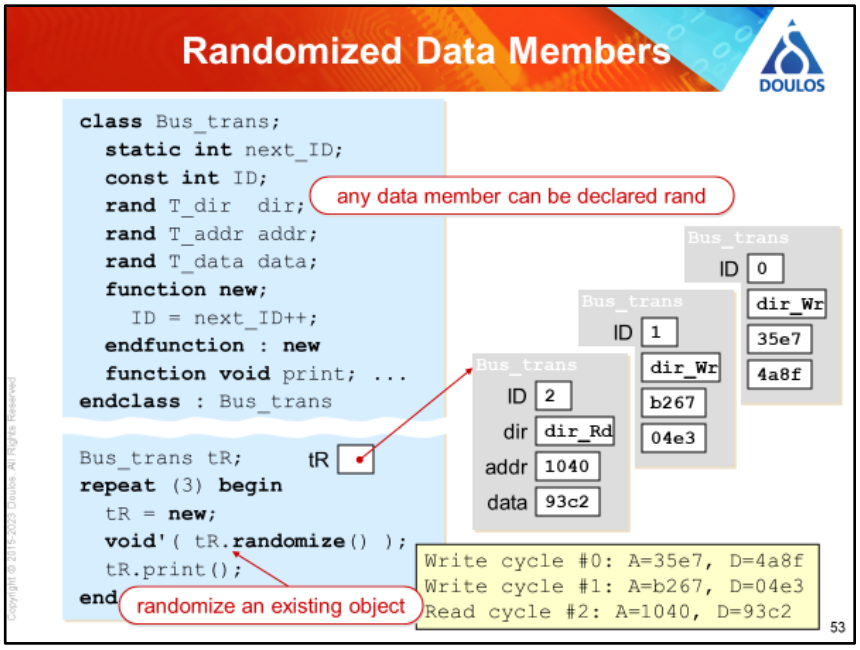
```
Bus_trans t5 = new;
Bus_trans t6 = new(dir_Wr);
```

Read

Write

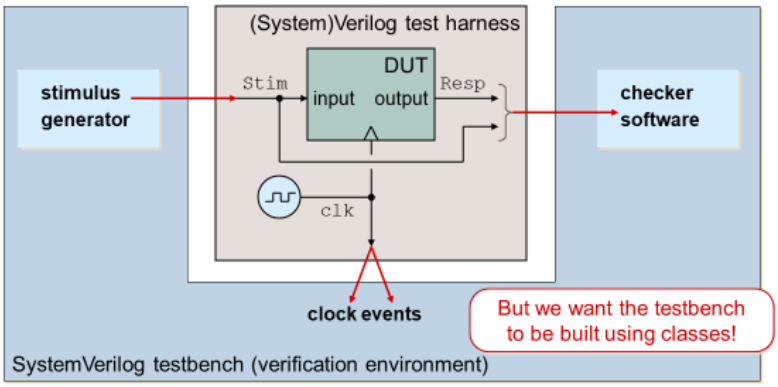
51





Test Harness and Testbench

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



54

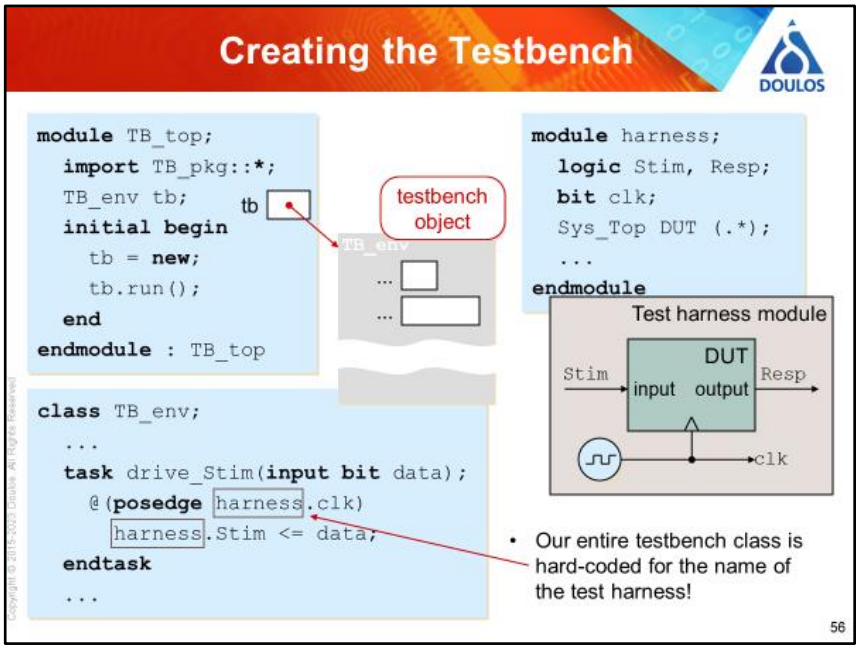
Lifetime and Persistence

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

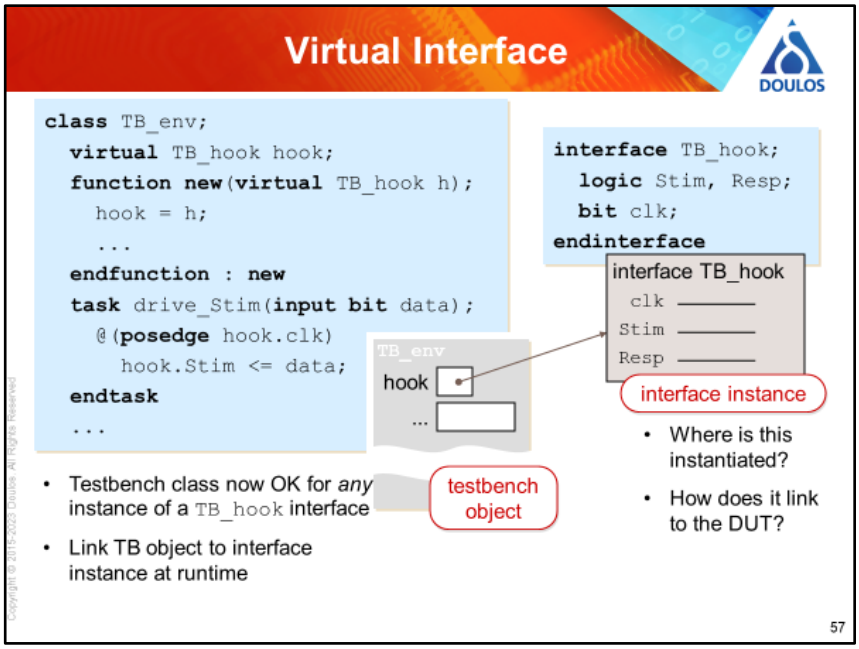
- Objects of class type:
  - created dynamically, during simulation, using new
  - structure controlled by run-time activity
  - can be created and destroyed at any time

verification environment	transaction data objects
• typically constructed at time zero	• created in large numbers during the simulation
• structure probably remains unchanged throughout simulation	• destroyed after use (unless logged)

55



Use of Interfaces

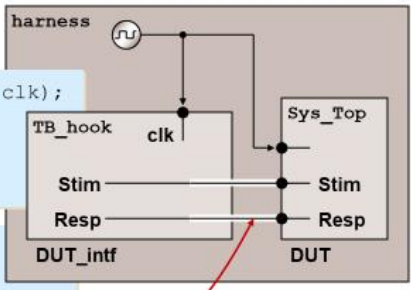


Building a test harness

Interface contains all signals required by testbench

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

```
module harness;
  bit clk;
  TB_hook DUT_intf (.clk);
  Sys_Top DUT (
    .clk  (clk),
    .Stim (DUT_intf.Stim),
    .Resp (DUT_intf.Resp),
    ...
```



Hierarchical connection

Harness contents

- DUT instance
- DUT connections
- clock generator

58

Connecting the virtual interface

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

constructor

Choice of interface type

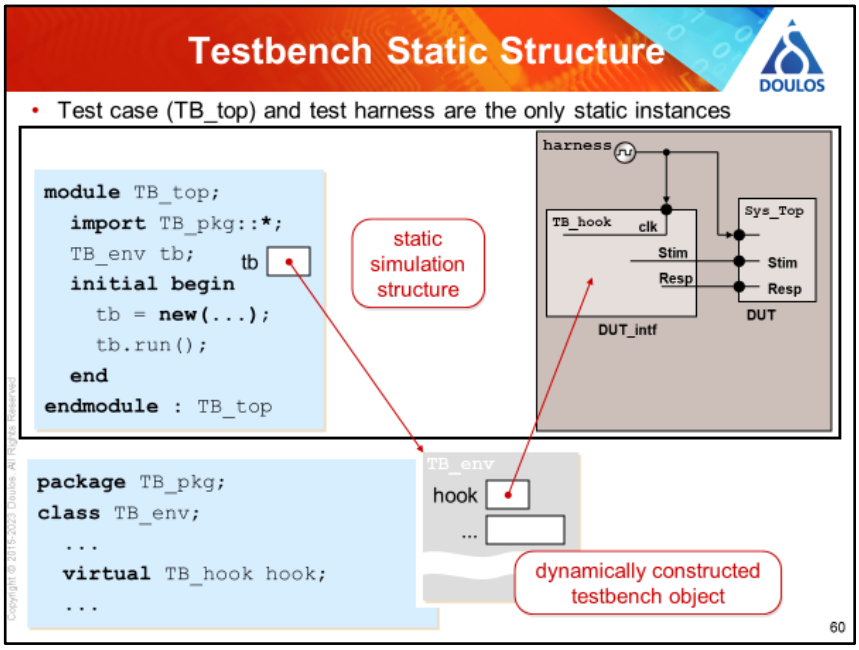
```
module TB_top;
  TB_env tb;
  ...
  initial begin
    tb = new(harness.DUT_intf, ...);
...
```

test harness

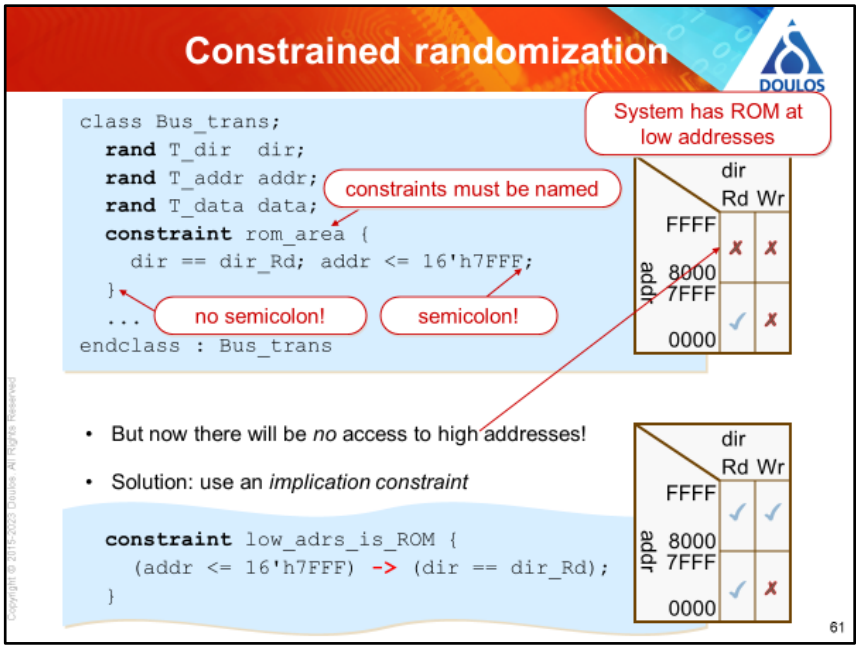
```
module harness;
  bit clk;
  TB_hook DUT_intf (.clk);
  Sys_Top DUT (
    ...
```

Choice of instance

59



Constraints and Functional Coverage



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...

Specific to the current DUT

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

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

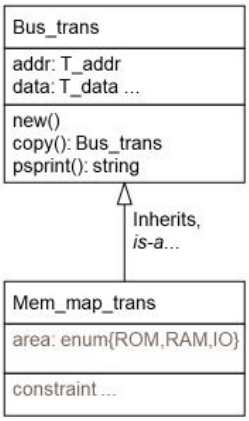
Everything in the base class, plus...

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

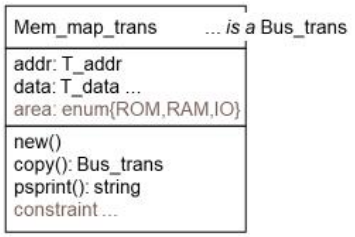

62

Inheriting Class Members

- What you write

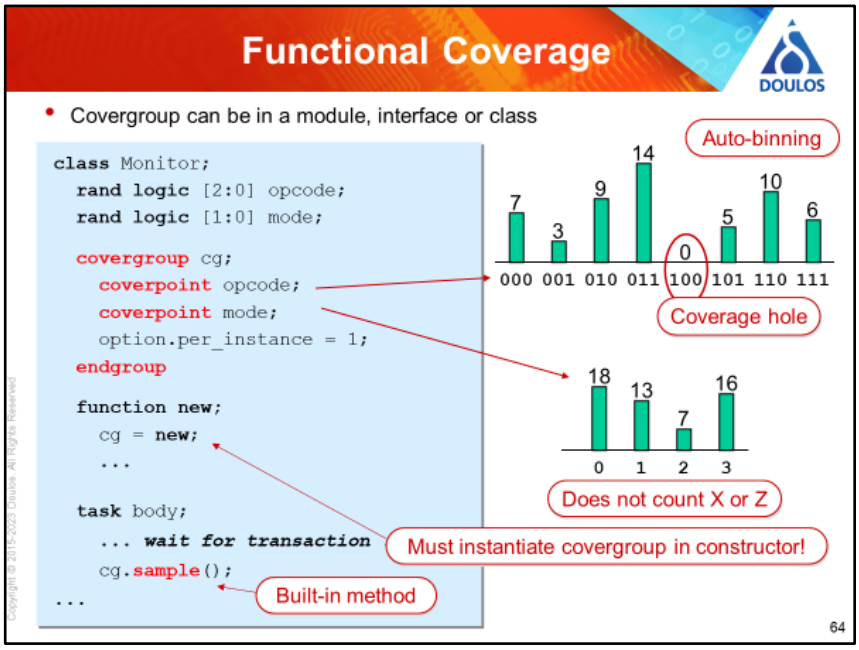


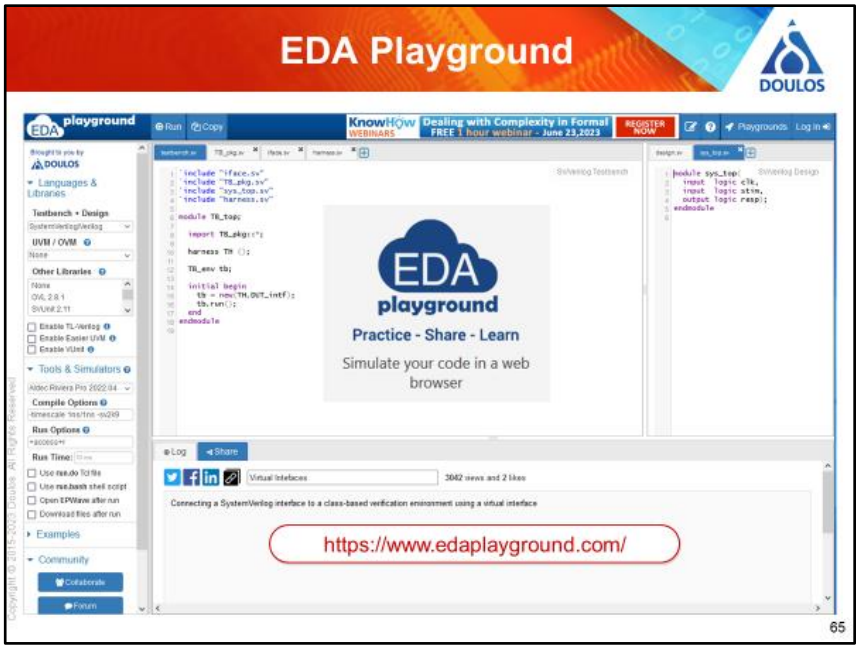
- What you get

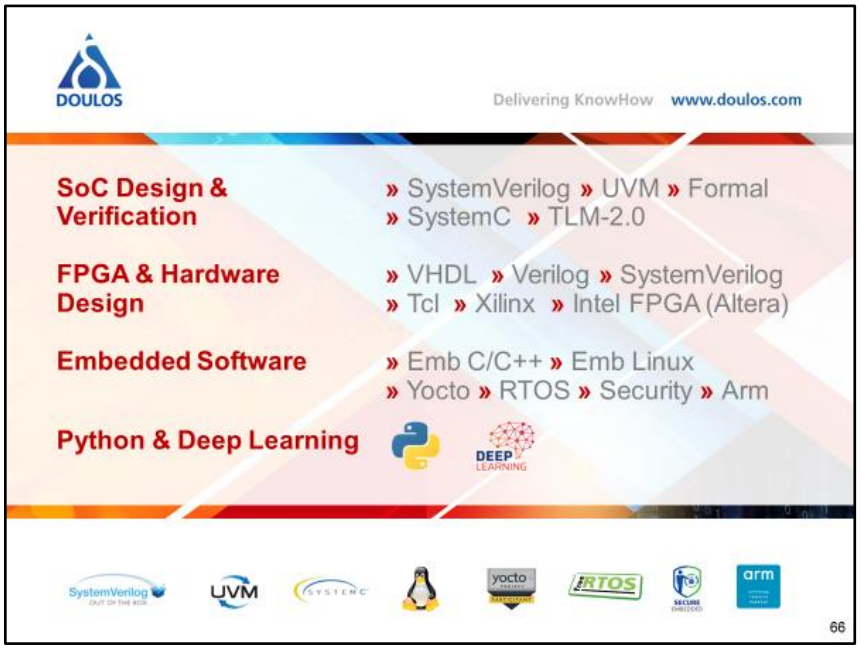


- A *Mem_map_trans* object can be used anywhere a *Bus_trans* object is appropriate

63







DOULOS

www.doulos.com