

EGC 455
SOC Design & Verification

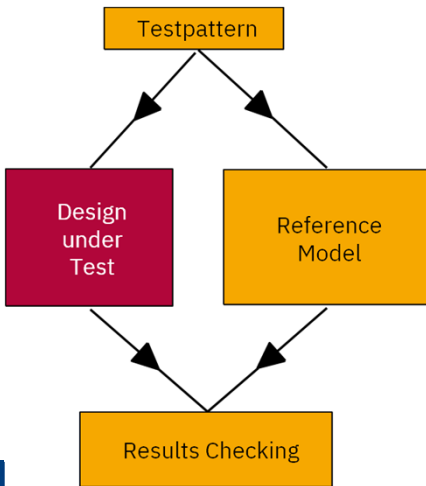
Verification Using Conventional Test Bench



Baback Izadi
Division of Engineering Programs
bai@engr.newpaltz.edu

1


Basic Verification Environment



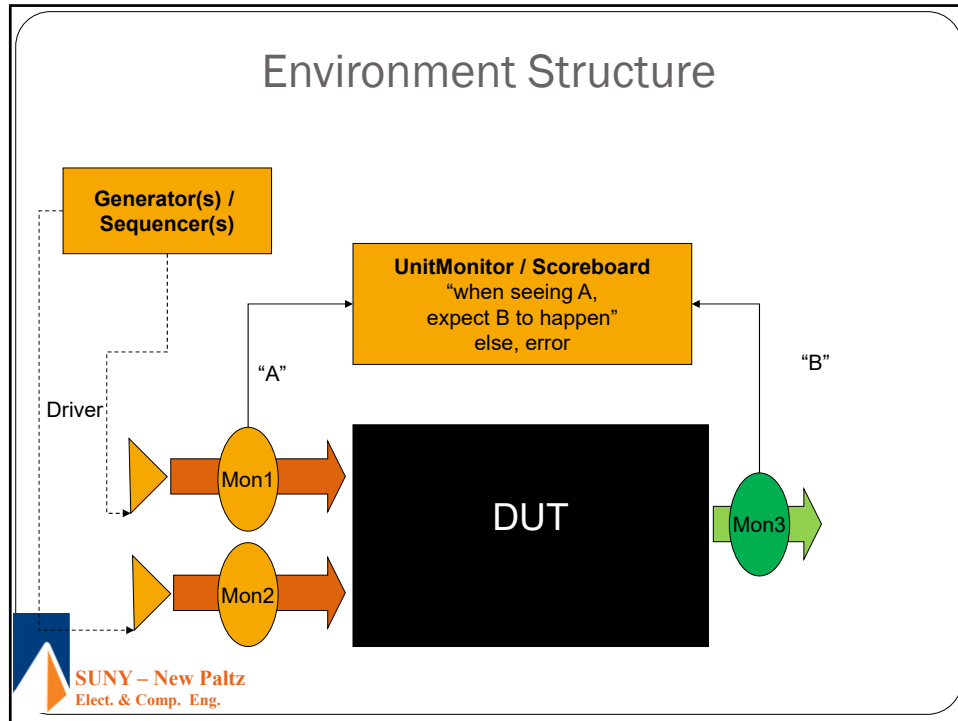
```
graph TD; TP[Testpattern] --> DUT[Design under Test]; TP --> RM[Reference Model]; DUT --> RC[Results Checking]; RM --> RC;
```

Verification methodologies differ mostly in:

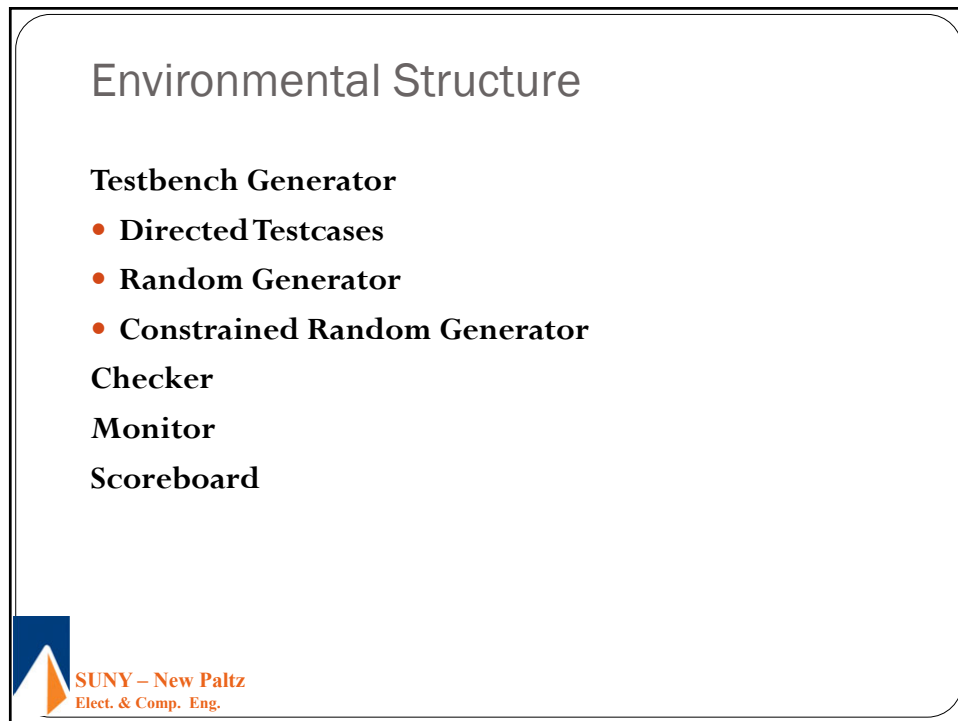
- How and when (online vs. offline) testcases / test patterns are generated
- How results are predicted and when these are compared to actual results from device under test
- Choice of underlying tools and infrastructure
- Abstraction level of reference model
- Completeness of state space exploration
- Choice of programming language

 SUNY – New Paltz
Elect. & Comp. Eng.

2



3



4

Functional verification approaches

Now that you've seen the basic structure....

- Black-Box approach
- White-Box approach
- Grey-Box approach



5

Develop a Test Plan

- The Test Plan is your key to a successful environment. This is developed by you, based on your interpretation of the specification(s).
- It must be detailed to reflect your understanding of the design, and how that design can be stressed.
- It should be written in a way to allow for comprehensive review and feedback from your peers.
- This is your guidebook for developing the environment and executing tests.



6

Test Plan: Content Requirements

- Define the boundaries of the DUT.
 - What blocks of logic will be included? Will a different environment cover the logic that's not included?
 - Hardware arrays/memories, or use a software behavioral?
 - Any short-cuts should be defined and listed. Maybe a behavioral doesn't act like the real HW. How can you validate that?
- Generation: What features will be exercised? What are the min and max values for that stimulus? Do features need to interact depending on configuration?
- Checkers: List out all the checks for validating each feature. Be detailed.



7

Test Plan

If it wasn't verified, it probably has defects.

- What logic will you be testing?
- What logic will not be tested (by you)?
- Is there additional logic being left out?

But wait, there's more!

- Environment coverage:
 - Are generators being fully utilized?
 - Did checkers get executed?
 - Are all testcases still running?
 - Did something in the environment stop working?




8

Three Simulation Commandments

- Thou shalt stress thine logic harder than it will ever be stressed again

Thou shalt stress thine logic harder than it will ever be stressed again	Thou shalt place checking upon all things	Thou shalt not move onto a higher platform until the bug rate has dropped off
--------------------------------------------------------------------------	-------------------------------------------	-------------------------------------------------------------------------------




SUNY - New Paltz
Elect. & Comp. Eng.

9

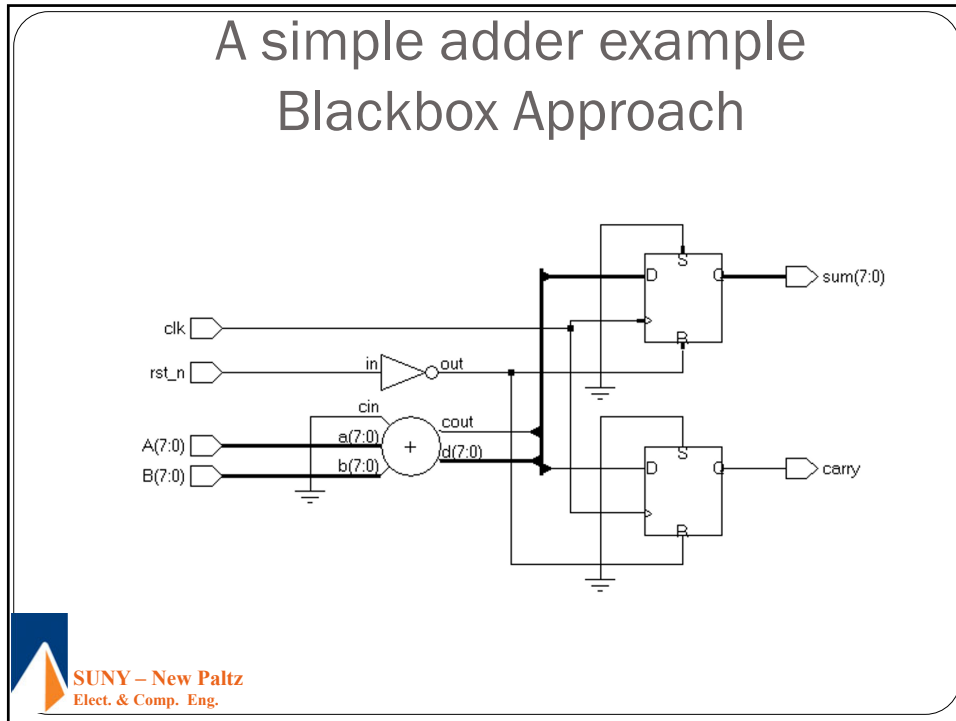
Multiple Environments

- All the planning has been for a single environment
- Projects typically employ multiple environments to stress the design using several methods.
- As we covered earlier:
 - Designer Sim
 - Unit Sim
 - Element Sim (multiple units)
 - Chip / System Sim
- To reduce chance for human error, we want overlapping environments.

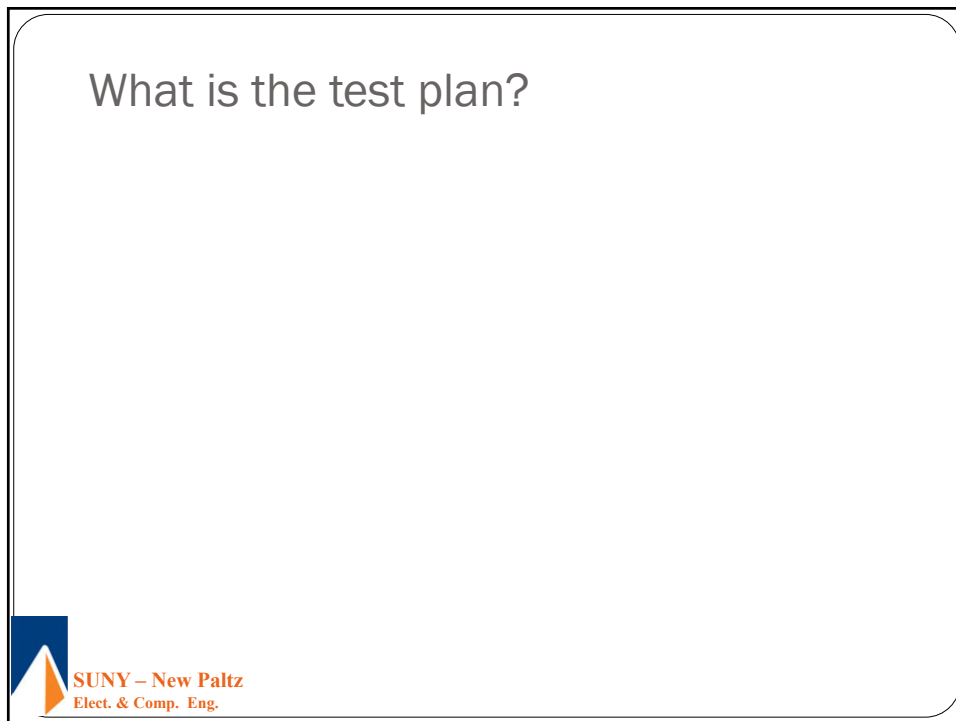


SUNY - New Paltz
Elect. & Comp. Eng.

10



11



12

A simple adder example Whitebox Approach

```
// Design file
module adder
  ( input [7:0] A,
    input [7:0] B,
    input clk,
    input rst_n,
    output reg [7:0] sum,
    output reg carry);

  always @(posedge clk or negedge rst_n)
    if (!rst_n)
      {carry,sum} <= 0;
    else
      {carry,sum} <= A + B;

endmodule
```



13

Any change to the test plan?



14


```
// Testbench file
module top (
output reg[7:0] A,
output reg[7:0] B,
output reg      clk,
output reg      rst_n,
input  wire      carry,
input  wire [7:0] sum);

adder_sv ADD (.*) ;

initial begin
  clk = 0;
  rst_n = 0;
  A = 0;
  B = 0;
  @(posedge clk);
  @(posedge clk);
  rst_n = 1;
end

always #10 clk = ~clk;


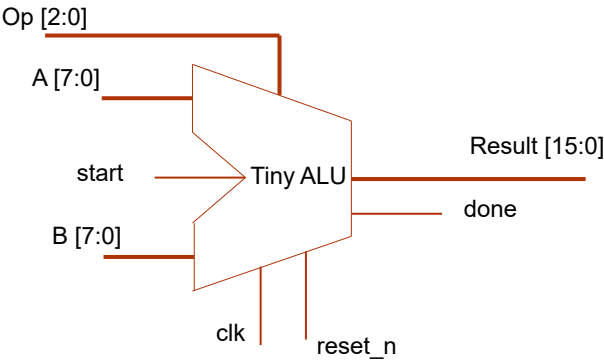
initial begin
  @(posedge rst_n);
  for (A = 0; A <= 25; A++) begin
    for (B = 0; B <= 10; B++) begin
      @(posedge clk);
      @(negedge clk);
      assert({carry,sum} == A + B)
    end
  end
  $display("%0d + %0d = %0d not %0d",A, B,A+B,{carry, sum});
  $stop;
end
endmodule // adder_tester
```



15

Tiny ALU Specification and Test Plan

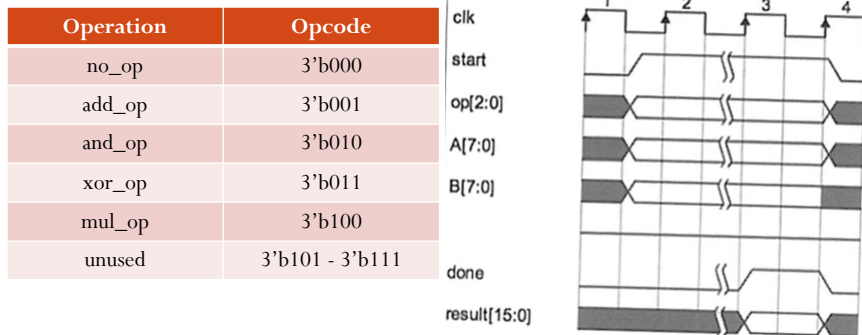
- ALU works at the rising edge of the clock. When `start` is active, it reads the operands. Operations can take any number of cycles. It rises `done` one clock cycle when the operation is complete. The `reset_n` is active low. The `start` must remain high until `done` signal is raised. No `done` signal on `nop`.



16

Test Plan for Tiny ALU

- <https://www.youtube.com/watch?v=iX7-41uG8uE>



17

What is the test plan?



18

```
module top;
    typedef enum bit[2:0] {no_op = 3'b000,
                           add_op = 3'b001,
                           and_op = 3'b010,
                           xor_op = 3'b011,
                           mul_op = 3'b100,
                           rst_op = 3'b111} operation_t;

    byte      unsigned    A;
    byte      unsigned    B;
    bit        clk;
    bit        reset_n;
    wire [2:0] op;
    bit        start;
    wire       done;
    wire [15:0] result;
    operation_t op_set;
    assign op = op_set;
    tinyalu DUT (.A, .B, .clk, .op, .reset_n, .start, .done, .result);
endmodule
```



SUNY - New Paltz
Elect. & Comp. Eng.

19

```
covergroup op_cov;
    coverpoint op_set {
        bins single_cycle[] = {[add_op : xor_op],
                               rst_op,no_op};
        bins multi_cycle = {mul_op};
        bins opn_rst[] = ([add_op:mul_op] => rst_op);
        bins rst_opn[] = (rst_op => [add_op:mul_op]);
        bins sngl_mul[] = ([add_op:xor_op],no_op => mul_op);
        bins mul_sngl[] = (mul_op => [add_op:xor_op], no_op);
        bins twoops[] = ([add_op:mul_op] [* 2]);
        bins manymult = (mul_op [* 3:5]);
    }
endgroup
```



SUNY - New Paltz
Elect. & Comp. Eng.

20

```
covergroup zeros_or_ones_on_ops;
  all_ops : coverpoint op_set {
    ignore_bins null_ops = {rst_op, no_op};}
  a_leg: coverpoint A {
    bins zeros = {'h00};
    bins others= {'h01:'hFE}};
    bins ones  = {'hFF};
  }
  b_leg: coverpoint B {
    bins zeros = {'h00};
    bins others= {'h01:'hFE}};
    bins ones  = {'hFF};
  }
}
```




21

```
op_00_FF: cross a_leg, b_leg, all_ops {
  bins add_00 = binsof (all_ops) intersect {add_op} &&
    (binsof (a_leg.zeros) || binsof
(b_leg.zeros));
  bins add_FF = binsof (all_ops) intersect {add_op} &&
    (binsof (a_leg.ones) || binsof
(b_leg.ones));
  bins and_00 = binsof (all_ops) intersect {and_op} &&
    (binsof (a_leg.zeros) || binsof
(b_leg.zeros));
  bins and_FF = binsof (all_ops) intersect {and_op} &&
    (binsof (a_leg.ones) || binsof
(b_leg.ones));
  bins xor_00 = binsof (all_ops) intersect {xor_op} &&
    (binsof (a_leg.zeros) || binsof
(b_leg.zeros));
}
```




22

```
bins xor_FF = binsof (all_ops) intersect {xor_op} &&
                (binsof (a_leg.ones) || binsof
(b_leg.ones));
        bins mul_00 = binsof (all_ops) intersect {mul_op} &&
                (binsof (a_leg.zeros) || binsof
(b_leg.zeros));
        bins mul_FF = binsof (all_ops) intersect {mul_op} &&
                (binsof (a_leg.ones) || binsof
(b_leg.ones));
        bins mul_max = binsof (all_ops) intersect {mul_op} &&
                (binsof (a_leg.ones) && binsof
(b_leg.ones));
        ignore_bins others_only =
                binsof(a_leg.others) &&
binsof(b_leg.others);
    }
endgroup
```



23

```
initial begin
    clk = 0;
    forever begin
        #10;
        clk = ~clk;
    end
end
op_cov oc;
zeros_or_ones_on_ops c_00_FF;
initial begin : coverage
    oc = new();
    c_00_FF = new();
    forever begin @(negedge clk);
        oc.sample();
        c_00_FF.sample();
    end
end : coverage
```



24

```
function operation_t get_op();  
    bit [2:0] op_choice;  
    op_choice = $random;  
    case (op_choice)  
        3'b000 : return no_op;  
        3'b001 : return add_op;  
        3'b010 : return and_op;  
        3'b011 : return xor_op;  
        3'b100 : return mul_op;  
        3'b101 : return no_op;  
        3'b110 : return rst_op;  
        3'b111 : return rst_op;  
    endcase // case (op_choice)  
endfunction : get_op
```



25

```
function byte get_data();  
    bit [1:0] zero_ones;  
    zero_ones = $random;  
    if (zero_ones == 2'b00)  
        return 8'h00;  
    else if (zero_ones == 2'b11)  
        return 8'hFF;  
    else  
        return $random;  
endfunction : get_data
```



26

```
always @(posedge done) begin : scoreboard
    shortint predicted_result;
    #1;
    case (op_set)
        add_op: predicted_result = A + B;
        and_op: predicted_result = A & B;
        xor_op: predicted_result = A ^ B;
        mul_op: predicted_result = A * B;
    endcase // case (op_set)
    if ((op_set != no_op) && (op_set != rst_op))
        if (predicted_result != result)
            $error ("FAILED: A: %0h B: %0h op: %s result: %0h",
                A, B, op_set.name(), result);
end : scoreboard
```




27

```
initial begin : tester
    reset_n = 1'b0;
    @(negedge clk);
    @(negedge clk);
    reset_n = 1'b1;
    start = 1'b0;
    repeat (1000) begin
        @(negedge clk);
        op_set = get_op();
        A = get_data();
        B = get_data();
        start = 1'b1;
        case (op_set) // handle the start signal
            no_op: begin
                @(posedge clk);
                start = 1'b0;
            end
        endcase
    end
end
```




28

```
rst_op: begin
    reset_n = 1'b0;
    start = 1'b0;
    @(negedge clk);
    reset_n = 1'b1;
end
default: begin
    wait(done);
    start = 1'b0;
end
endcase // case (op_set)
end
$stop;
end : tester
endmodule : top
```



29

Object Oriented Approach



30

Terminology

```
1 class rectangle_t;  
2   int length;  
3   int width;  
4  
5   function new(int l, int w);  
6     length = l;  
7     width = w;  
8   endfunction  
9  
10  function integer area;  
11    return length * width;  
12  endfunction  
13 endclass
```

Data members

Constructor – Only necessary if initialization needed. Otherwise use implicit constructor

Method

SUNY – New Paltz
Elect. & Comp. Eng.

31

Differences in SystemVerilog Classes

```
class rectangle_t;  
  int length;  
  int width;  
  
  function new(int l, int w);  
    length = l;  
    width = w;  
  endfunction  
  
  function integer area;  
    return length * width;  
  endfunction  
endclass  
  
module class_example ;  
  rectangle_t rectangle;  
  initial begin  
    rectangle = new(50,20);  
    $display(rectangle.area);  
  end  
endmodule
```

Declared as class

Includes functions and tasks as methods

Memory is **not allocated here**

Memory allocated here

Method called to determine area

SUNY – New Paltz
Elect. & Comp. Eng.

32

Advantage of Classes: Extension

```

class rectangle_t;
int length;
int width;

function new(int l, int w);
length = l;
width = w;
endfunction

function integer area;
return length * width;
endfunction
endclass

class square_t extends rectangle_t;

function new(int side);
super.new(side,side);
endfunction
endclass


module class_example;
rectangle_t rectangle;
square_t square;
initial begin
rectangle = new(50,20);
$display("rectangle area: %0d", rectangle.area);
square = new(50);
$display("square area: %0d", square.area);
end
endmodule
        
```

A Square is a Rectangle with the same length and width

Leverage rectangle_t data and methods

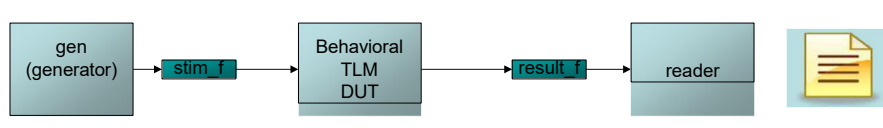
In this case, constructor (new) calls parent's constructor

We can use the method from the parent



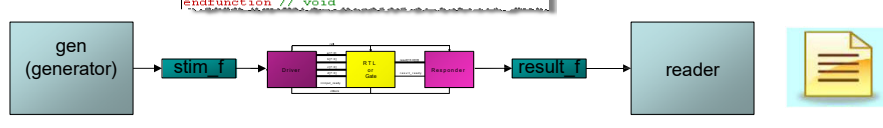
33


What Can I Do with Objects?



```

function void calc_beh_result (input_tran t);
result = (t.a * t.b) + (t.c * t.d);
endfunction // void
        
```





34

Randomizing Objects

```
1 typedef logic[15:0] addr_t;
2 typedef logic[7:0] data_t;
3 typedef enum {read, write} mem_op_t;
4
5 class memory_op;
6     rand addr_t addr;
7     rand data_t data;
8     rand mem_op_t mem_op;
9
10    function new(addr_t a = 0, data_t d = 0, mem_op_t o = read);
11        addr = a;
12        data = d;
13        mem_op = o;
14    endfunction // new
15
16    function string convert2string();
17        string s;
18        $sformat(s,"addr %0h data %0h op %s",addr, data, mem_op);
19        return s;
20    endfunction // convert2string
```

Random Variables



35

Randomizing

```
28 module top;
29     mem_op op;
30
31     initial begin
32         op = new();
33         repeat (5) begin
34             assert(op.randomize());
35             $display("op -> %s",op.convert2string);
36         end
37     end
38 endmodule // top
```

```
# Loading work.top
# op -> addr b619 data 88 op write
# op -> addr 3900 data 3b op write
# op -> addr c1f2 data 50 op write
# op -> addr 0309 data 7d op read
# op -> addr 7e69 data 87 op read
VSIM 3> █
```



36

Constraining Randomization

```

27 initial begin
28     op = new();
29     repeat (5) begin
30         assert(op.randomize() with {mem_op == read;});
31         $display("op -> %s",op.convert2string);
32     end
33 end
34 endmodule // top
    
```

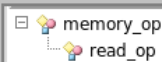
```

VSIM 1> run -all
# op -> addr aba6 data 89 op read
# op -> addr 0377 data 75 op read
# op -> addr 85a5 data 81 op read
# op -> addr 31c0 data 9c op read
# op -> addr 1924 data dc op read
    
```



37

Add Constraints by Extending Classes



```

24 class read_op extends memory_op;
25     constraint read_only_c {mem_op == read;};
26 endclass // read_op
    
```

```

28 module top;
29     read_op op;
30
31     initial begin
32         op = new();
33         repeat (5) begin
34             assert(op.randomize());
35             $display("op -> %s",op.convert2string);
36         end
37     end
    
```

```

VSIM 1> run -all
# op -> addr aba6 data 89 op read
# op -> addr 0377 data 75 op read
# op -> addr 85a5 data 81 op read
# op -> addr 31c0 data 9c op read
# op -> _addr 1924 data dc op read
    
```



38