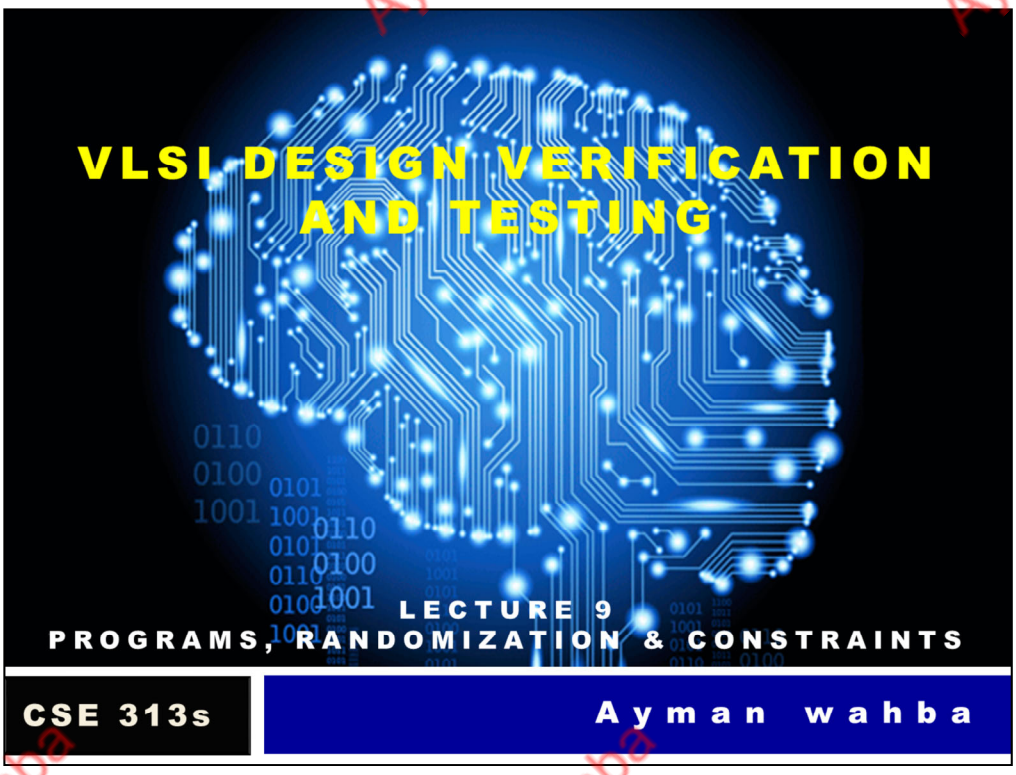
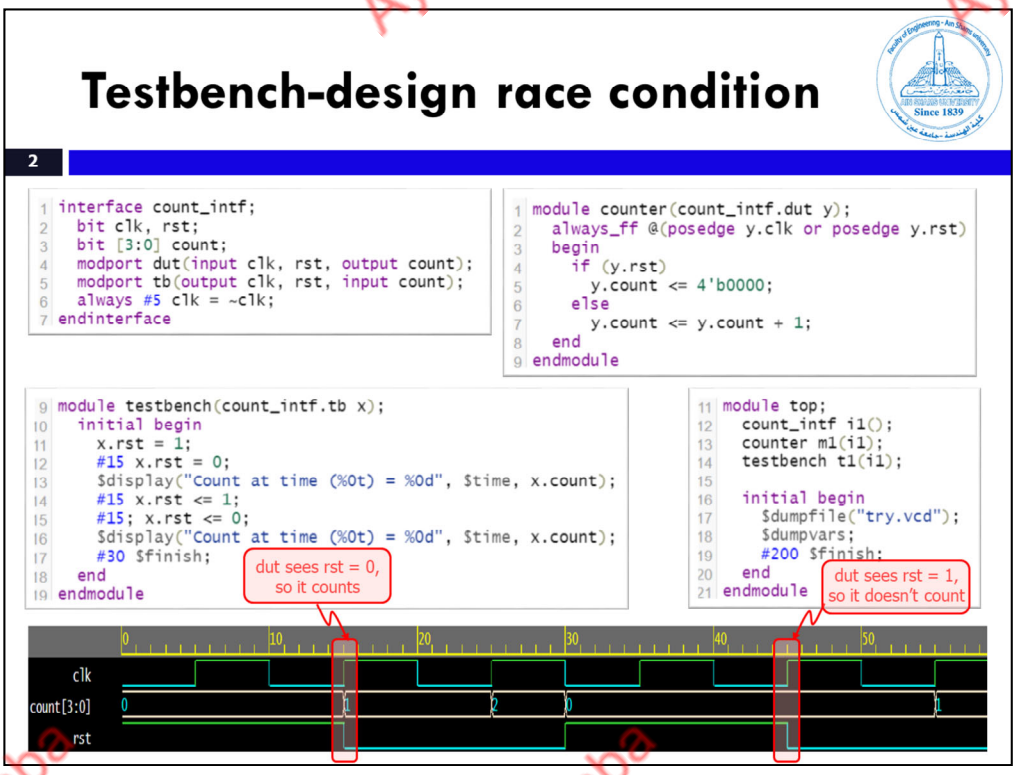
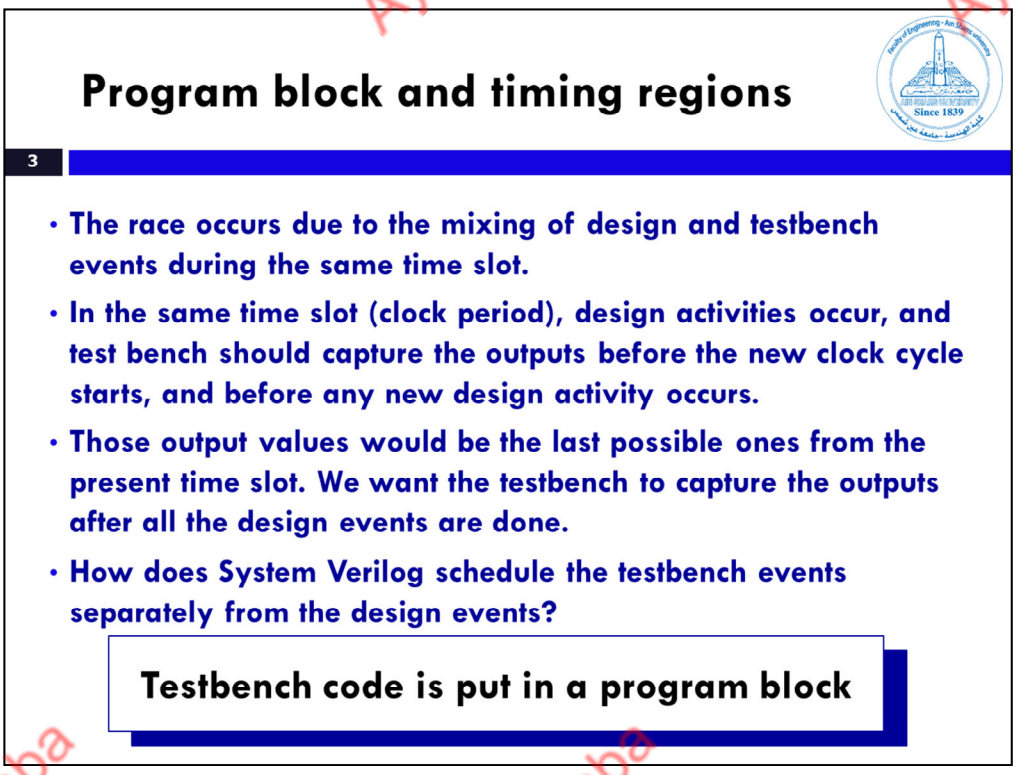

VLSI DESIGN VERIFICATION AND TESTING
LECTURE 9
PROGRAMS, RANDOMIZATION & CONSTRAINTS
CSE 313s
Ayman wahba

Testbench-design race condition



- The shown example illustrates a race condition between a testbench and the device under test.
- At time t = 15, rst was deactivated, and at the same exact moment the a rising clock edge happens. The DUT reads rst = 0, so it increments its count to be 1.
- At time t = 45, the same exact scenario happens again. Amazingly, the DUT reads rst = 1, so it doesn't incremeny its count.
- This is a clear example of a race condition (race between the rising edge of the clock, and the change in rst. Will *rst* go to 0 before or after the clock edge?).
- Race condition leads to an unpredictable behavior.

- Tip: The main difference between *always* and *always_ff* blocks is the way that we can use blocking and non-blocking assignment.
  - ➢ When we model a circuit using the always_ff block, we can only use non-blocking signal assignment.
  - ➢ In contrast, we can use either blocking or non-blocking assignment in an always block.

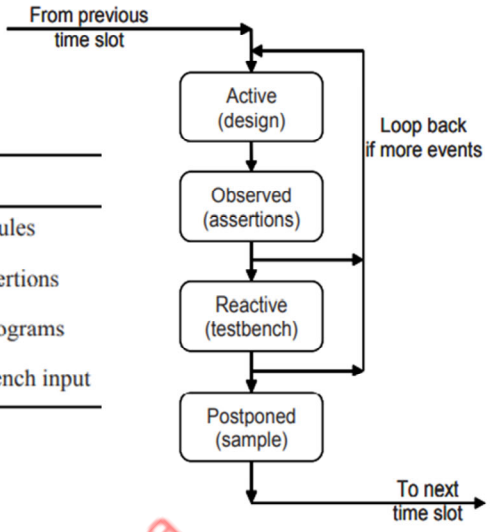


- The root of the problem is the mixing of *design* and *testbench* events during the same time slot. What if there were a way you could separate these events temporally, just as you separated the code?
- For example, assume the clock period is 10 ns, and at 100 ns your testbench should sample the design outputs before the new clock cycle starts, and before and design activity occurs.
- By definition. these values would be the last possible ones from the previous time slot (from 90 to 100 ns). We want the testbench to capture the outputs after all the design events are done.
- How does System Verilog schedule the testbench events separately from the design events?
- In System Verilog your testbench code is in a *program block,* which is similar to a module in that it can contain code and variables and be instantiated in other modules. However, a program cannot have any hierarchy such as instances of modules. interfaces. or other programs.

Timing regions of a program block

4

Name	Activity
Active	Simulation of design code in modules
Observed	Evaluation of SystemVerilog Assertions
Reactive	Execution of testbench code in programs
Postponed	Sampling design signals for testbench input



- A new division of the time slot is introduced in System Verilog as shown.
- *First* to execute during a time slot is the Active region, where design events run. These include your RTL and gate code plus the clock generator.
- *The second region* is the Observed region, where assertions are evaluated.
- Following, is the Reactive region where the testbench executes.
- Note that time does not strictly flow forwards – events in the Observed and Reactive regions can trigger further design events in the Active region in the current cycle.
- Lastly is the Postponed region, which samples signals at the end of the time slot, in the read-only period, after design activity has completed.

Program block and timing regions

5

- The Program construct provides a *race-free* interaction between the design and the testbench.
- All elements declared within the *program block* will get executed in the *reactive region*.
- Non-blocking assignments within the *module* are scheduled in the *active region*.
- An active region is scheduled before the reactive region this avoids the race condition between testbench and design.

- The Program construct provides a *race-free* interaction between the design and the testbench.
- All elements declared within the program block will get executed in the *reactive region*.
- Non-blocking assignments within the module are scheduled in the active region.
- This solves the problem of racing between design and the testbench, as the statements within the program block (scheduled in the reactive region) that are sensitive to changes in design signals declared in modules (scheduled in the active region), are executed at the end of the time slot.
- An active region is scheduled before the reactive region, and this avoids the race condition between testbench and design.

Program block in brief

6

- It can be instantiated, and ports can be connected the same as a module
- It can contain one or more initial blocks
- It cannot contain always blocks, modules, interfaces, or other programs
- In the program block, variables can only be assigned using blocking assignments. Using non-blocking assignments within the program shall be an error

Example

7

```
//design code
module design_ex(output bit [7:0] addr);
  initial begin
    addr <= 10;
  end
endmodule

//testbench
module testbench(input bit [7:0] addr);
  initial begin
    $display("\t Addr = %0d",addr);
  end
endmodule

//testbench top
module tbench_top;
  wire [7:0] addr;

  //design instance
  design_ex dut(addr);

  //testbench instance
  testbench test(addr);
endmodule
```

```
Addr = 0
```

```
//design code
module design_ex(output bit [7:0] addr);
  initial begin
    addr <= 10;
  end
endmodule

//testbench
program testbench(input bit [7:0] addr);
  initial begin
    $display("\t Addr = %0d",addr);
  end
endprogram

//testbench top
module tbench_top;
  wire [7:0] addr;

  //design instance
  design_ex dut(addr);

  //testbench instance
  testbench test(addr);
endmodule
```

```
Addr = 10
```

End of Simulation

8

- In Verilog, simulation continues while there are scheduled events, or until a $finish is executed.
- In SystemVerilog, A program block is treated as if it contains a test. Simulation ends when you complete the last statement in every initial-block, as this is considered the end of the test.
- If there are several program blocks, simulation ends when the last program completes.
- You can terminate any program block early by executing $exit. Of course you can still use $finish to end simulation

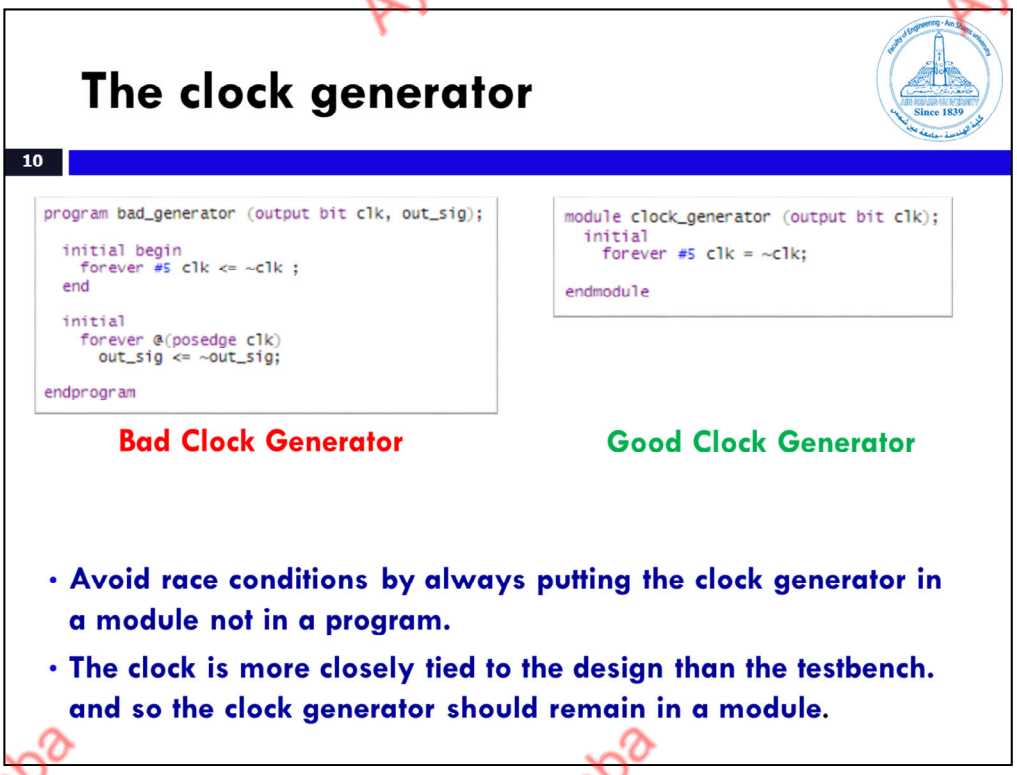
- In Verilog, simulation continues while there are scheduled events, or until a $finish is executed. SystemVerilog adds an additional way to end simulation.
- A program block is treated as if it contains a test. If there is only a single program block, simulation ends when you complete the last statement in every initial-block, as this is considered the end of the test. Simulation ends even if there are threads still running in the modules.
- As a result, you don't have to shut down every monitor and driver when a test is done.
- If there are several program blocks, simulation ends when the last program completes. This way simulation ends when the last test completes.
- You can terminate any program block early by executing $exit. Of course you can still use $finish to end simulation

Why are *always* Blocks Not Allowed in a Program?

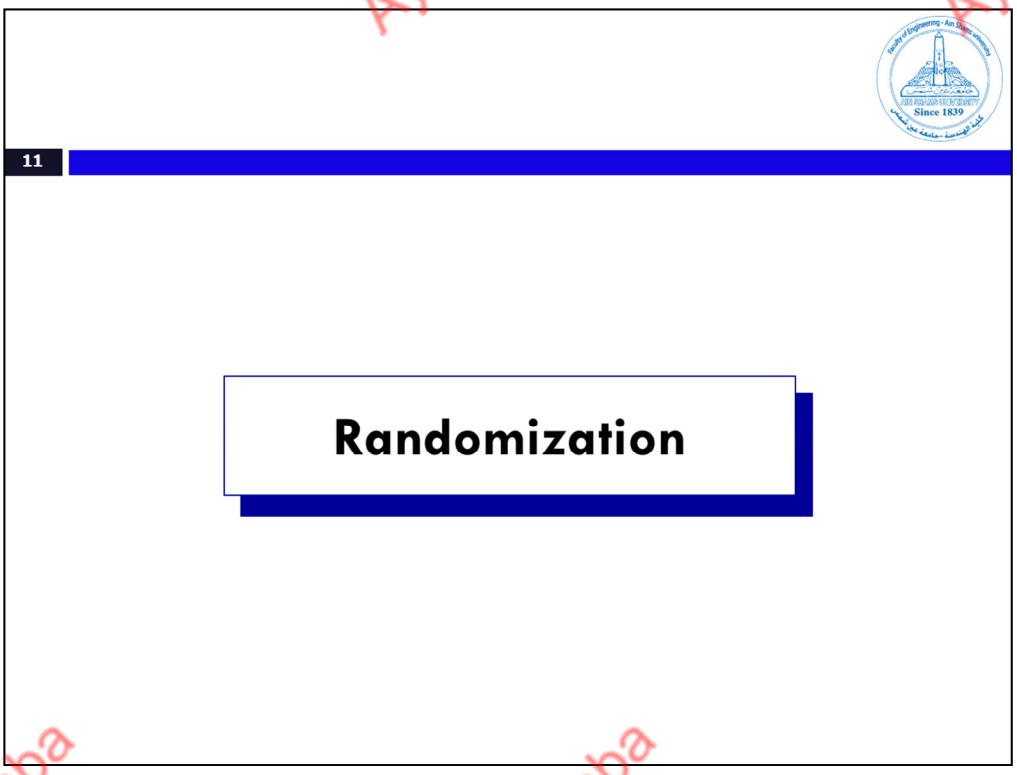
9

- In System Verilog, you can put *initial* blocks in a program, but not *always* blocks.
- An always block might trigger on every positive edge of a clock from the start of simulation.
- In contrast, a testbench has the steps of *initialization*, *stimulate* and *respond* to the design, and then wrap up simulation.
- An always block that runs continuously would not work.
- Simulation ends when the last initial block in the program completes, as if you had executed $finish.
- If you had an always block, it would never stop, so it is not allowed in a program.

- In System Verilog, you can put *initial blocks* in a program, but not *always blocks*.
- This may seem odd, but there are several reasons.
- In a design, an always block might trigger on every positive edge of a clock from the start of simulation.
- In contrast, a testbench has the steps of *initialization*, *stimulate* and *respond* to the design, and then wrap up simulation.
- An always block that runs continuously would not work.
- When the last initial block completes in the program, simulation implicitly ends just as if you had executed $finish. If you had an always block, it would never stop, and so you would have to explicitly call $exit to signal that the program block completed.
- But don't despair. If you really need an always block, you can use *initial forever* to accomplish the same thing



- Now that you have seen the program block, you may wonder if the clock generator should be in a module.
- The clock is more closely tied to the design than the testbench. and so the clock generator should remain in a module.
- As you refine the design, you create clock trees, and you have to carefully control the skews as the clocks enter the system and propagate through the blocks.
- The testbench is much less picky. It just wants a clock edge to know when to drive and sample signals. Functional verification is concerned with providing the right values at the right cycle, not with fractional nanosecond delays and relative clock skews.
- The program block is not the place to put a clock generator. The sample on the left tries to put the generator in a program block but just causes a race condition. The *clk* and *out_sig* signals both propagate from the Reactive region to the design in the Active region and could cause a race condition depending on which one arrived first.
- The sample on the right shows a good clock generator in a module. It deliberately avoids an edge at time 0 to avoid race conditions.



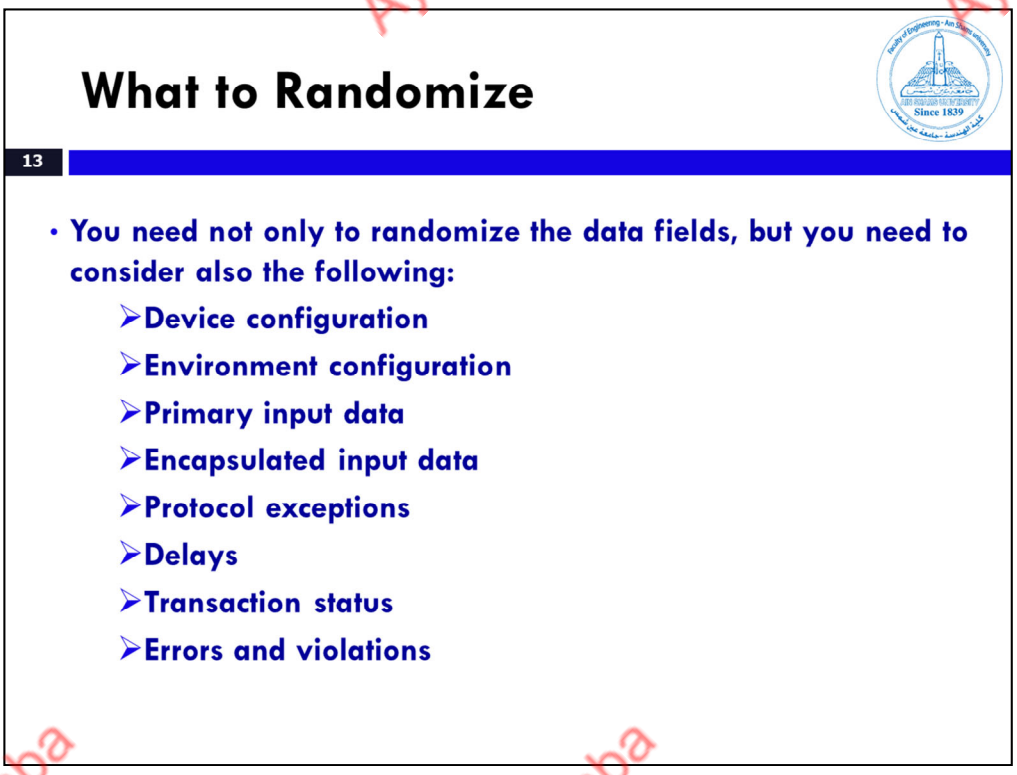
- Ver

Randomization

12

- It is difficult to create a complete set of stimuli.
- Directed tests are used to check a certain set of features, but not all of them for big designs.
- We create test cases automatically using constrained-random tests (CRT).
- A directed test finds the bugs you expect to be there, but a CRT finds bugs you never thought about, by using random stimulus.
- To know when you have covered all aspects of the design, we need to measure verification progress by using functional coverage.

- As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality.
- You can write a directed test case to check a certain set of features, but you cannot write enough directed test cases when the number of features keeps growing on each project.
- Worse yet, the interactions between all these features are the source for the most difficult bugs that are the least likely to be caught.
- The solution is to create test cases automatically using Constrained-Random Tests (CRT). A directed test finds the bugs you think about, but a CRT finds bugs you never thought about, by using random stimulus.
- You restrict the test scenarios to those that are both *valid* and *of interest* by using *constraints*.
- Creating a CRT environment takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result.
- A CRT environment needs not only to create the stimulus but also to predict the result, using a reference model. Once this environment is in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity.
- A CRT is made of two parts:
  - The test code that uses a stream of random values to create input to the DUT,
  - A seed to the pseudo-random number generator (PRNG). You can make a CRT behave differently just by using a new seed.
- How do you know when you have covered all aspects of the design? The stimulus space is too large to generate every possible input. Later you will learn how to measure verification progress by using functional coverage.
- There are many ways to use randomization. We highlight the most useful techniques, but you should choose what works best for you.



- When you think of randomizing the stimulus to a design, the first thing you may think of are the data fields. These are the easiest to create - just call $random.
- The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed.
- *The challenging bugs are in the control logic*. As a result, you need to randomize all decision points in your DUT.
- Wherever control paths diverge, randomization increases the probability that you'll take a different path in each test case. You need to think broadly about all design input such as the following:
  - ➢ Device configuration
  - ➢ Environment configuration
  - ➢ Primary input data
  - ➢ Encapsulated input data
  - ➢ Protocol exceptions
  - ➢ Delays
  - ➢ Transaction status
  - ➢ Errors and violations

What to Randomize

(Device Configuration)

14

- What is the most common reason why bugs are missed during testing of the RTL design?
  - Not enough different configurations have been tried!
- Most tests just use the design as it comes out of reset or apply a fixed set of initialization vectors to put it into a known state.
- This is like testing a PC's operating system right after it has been installed, and without any applications; of course the performance is fine, and there are no crashes.

- What is the most common reason why bugs are missed during testing of the RTL design? Not enough different configurations have been tried!
- Most tests just use the design as it comes out of reset, or apply a fixed set of initialization vectors to put it into a known state.
- This is like testing a PC's operating system right after it has been installed, and without any applications; of course the performance is fine, and there are no crashes. Over time, in a real world environment, the DUT's configuration becomes more and more random.

What to Randomize

(Environment Configuration)

15

- The device that you are designing operates in an environment containing other devices.
- When you are verifying the DUT, it is connected to a testbench that mimics this environment.
- You should randomize the entire environment, including the number of objects and how they are configured.
- For example, to test I/O switch chip that connects multiple PCI buses to an internal memory, you should for example randomize the number of PCI buses (1-4), the number of devices on each bus (1-8), and the parameters for each device (master or slave, … etc.).

- The device that you are designing operates in an environment containing other devices.
- When you are verifying the DUT, it is connected to a testbench that mimics this environment.
- You should randomize the entire environment, including the number of objects and how they are configured.
- For example, a company was creating an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation the customer used randomization to choose the number of PCI buses (1-4), the number of devices on each bus (1-8), and the parameters for each device (master or slave, … etc).

What to Randomize

(Primary input data)

16

- This is what you probably thought of first when you read about random stimulus.
- Take a transaction such as a bus write and fill it with some random values.
- This is fairly straightforward as long as you carefully prepare your transaction.

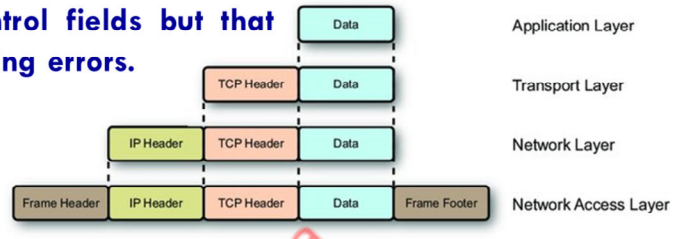
- This is what you probably thought at first when you read about random stimulus: take a transaction such as a bus write for example, and fill it with some random values.
- How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction.

What to Randomize

(Encapsulated input data)

17

- Many devices process multiple layers of stimulus.
- For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets.
- So you are randomizing the data and the layers that surround it.
- You need to write constraints that create valid control fields but that also allow injecting errors.



- Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets.
- Each level has its own control fields that can be randomized to try new combinations.
- So you are randomizing the data and the layers that surround it. You need to write constraints that create valid control fields but that also allow injecting errors.

What to Randomize

(Protocol exceptions, Errors and Violations)

18

- The most challenging part of design and verification is how to handle errors in the system.
- A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond (we call them corner cases).
- For example, When two devices communicate, what happens if the transfer stops halfway through? Can your testbench simulate these breaks?
- The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

- Anything that can go wrong, will, eventually, go wrong.
- The most challenging part of design and verification is how to handle errors in the system.
- You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state.
- A good verification engineer tests the behavior of the design to the *edge* of the functional specification and sometimes even beyond (*corner cases*)
- When two devices communicate, what happens if the transfer stops halfway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried.
- The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

What to Randomize

(Delays)

19

- Many communication protocols specify ranges of delays.
  - The bus grant comes one to three cycles after request.
  - Data from the memory is valid in the fourth to tenth bus cycle.
- Your testbench should always use random, legal delays during every test to try to find combinations that reveals a design bug.
- Below the cycle level, some designs are sensitive to clock jitter.
  - By sliding the clock edges back and forth by small amounts, you can make sure your design is not sensitive to small clock timings.
- The generator should have parameters such as frequency and offset that can be set by the testbench.
- You are looking for functional errors, not timing errors. Your testbench should not try to violate setup and hold requirements.

- Many communication protocols specify ranges of delays. For example:
  - The bus grant comes one to three cycles after request.
  - Data from the memory is valid in the fourth to tenth bus cycle.
- However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test, that only tries various delays.
- Your testbench should always use random, legal delays during every test to try to hopefully find one combination that exposes a design bug.
- Below the cycle level, some designs are sensitive to clock jitter.
  - By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle.
- The clock generator should be in a module outside the testbench. The generator should have parameters such as frequency and offset that can be set by the testbench.
- Note that you are looking for functional errors, not timing errors. Your testbench should not try to violate setup and hold requirements. These are better validated using timing analysis tools.

Randomization in System Verilog

20

- **Random stimulus generation in SystemVerilog is most useful when used with OOP.**
- **You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values.**
- **You can create constraints to limit the random values to legal values, or to test-specific features.**
- **Note that you can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.**

- Random stimulus generation in SystemVerilog is most useful when used with OOP.
- You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values.
- You can create constraints to limit the random values to legal values, or to test-specific features.
- Note that you can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.

Simple class with Random Variables

21

```
program test;
class Packet;
  // The random variables
  rand bit [31:0] src, dst, data[8];
  randc bit [7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new();// Create a packet
  assert (p.randomize())
  else
    $fatal(0,"Packet::randomize failed");
  $display(p) ;
end
endprogram
```

```
src =         11
dst = 3910850489
data = '{'ha1098c6c, 'ha13af8a7, 'hb08870a3, 'h7afa219c, 'hcf54c234, 'h4d777439, 'h69cf3483, 'h7288d514}
kind = 252
$finish at simulation time                0
           V C S   S i m u l a t i o n   R e p o r t
```

- This class has four random variables.
- The first three use the *rand* modifier, so that every time you randomize the class, the variables are assigned a value. Think of it as rolling dice: each roll could be a new value or a repeated one.
- The *kind* variable is *randc*, which means random cyclic. so that the random solver does not repeat a random value until every possible value has been assigned. Think of it as selecting a card from a group of cards, and then putting it aside. If you take a new card, its value will never be as the previously selected values, as they are already put aside.
- A *constraint* is just a set of relational expressions that must be true for the chosen value of the variables. In this example. the *src* variable must be greater than 10 and less than 15.
- Note that the constraint expression is grouped using curly braces:{}.
- The randomize () function assigns random values to any variable in the class that has been labeled as *rand* or *randc*, and also makes sure that all active constraints are obeyed. Randomization can fail if your code has conflicting constraints (e.g. src > 15 and src < 10), and so you should always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.
- The randomize () function returns 0 if a problem is found with the constraints, that is what we check with the above assertion.
- Randomization only works with 2-state values. You can have integers, bit vectors, etc. You cannot have a random string for example.

Constraints:
Introduction

22

```
class Stim;
  const bit [31:0] CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_e;
  randc stim_e kind;      // Enumerated var
  rand bit [31:0] len, src, dst;
  randc bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    if (congestion_test) {
      dst inside {[CONGEST_ADDR+50:CONGEST_ADDR+100]};
      src == CONGEST_ADDR;
      }
      else
        src inside {0, [2:10], [100:107]};
    }
endclass

program test;
  Stim s;
  initial begin
    s = new();
    for (int i = 1; i<=5; i++) begin
      assert (s.randomize())
        else $fatal("Problem");
      $display(s);
    end
  end
endprogram
```

```
'{CONGEST_ADDR:'h2a, kind:CONTROL, len:'hd2, src:'h4, dst:'heaba8d58, congestion_test:'h0}
'{CONGEST_ADDR:'h2a, kind:READ, len:'hf2, src:'h2a, dst:'h84, congestion_test:'h1}
'{CONGEST_ADDR:'h2a, kind:WRITE, len:'h268, src:'ha, dst:'h663fb6af, congestion_test:'h0}
'{CONGEST_ADDR:'h2a, kind:CONTROL, len:'h2fa, src:'h2a, dst:'h8e, congestion_test:'h1}
'{CONGEST_ADDR:'h2a, kind:WRITE, len:'hdc, src:'h2a, dst:'h73, congestion_test:'h1}
```

- This code shows a simple class with random variables and constraints.
- Note that you can use an *if* statement to change the constraints according to some conditions.
- Inside the constraint, we use *{}*, rather than *begin ... end*
- You can use *randomize()* standalone but it is a good practice to put it in an assertion, to detect any conflicting constraints.
- If one of the *if statement* branches has a problems in the constraints, the constraint solver will never select random values that make you enter this branch.
- Note that, there can be a maximum of only one relational operator <, <=, = =, >=, or > in an expression. For the first two lines of the above constraint, you can't write *0 < len < 1000*.
- Constraint block can only contain expressions. The most common mistake with constraints is trying to make an assignment in a constraint block. Instead, use the equivalence operator to set a random variable to a value, e.g., len==42.
- You can use $ as a shortcut for the minimum and maximum values for a range. This is helpful when you are building constraints for variables with different ranges.

```
rand bit [6: 0] b; // 0 <= b <= 127
rand bit [5: 0] e; // 0 <= e <= 63
constraint c _range {
  b inside {[$:4], [20:$]}; // 0 <= b <= 4 or 20 <= b <= 127
  e inside {[$:4], [20:$]}; // 0 <= e <= 4 II 20 <= e <= 63
}
```

Constraints:
Weighted Distribution

23

```
class weighted;
  rand int src, dst;
  constraint c_dist {
    src dist {0:=40, [1:3] :=60};
    // src 0, weight 40/220 [(40+60+60+60)=220]
    // src 1, weight 60/220
    // src 2, weight 60/220
    // src 3, weight 60/220

    dst dist {0:/40, [1:3]:/60};
    // dst 0, weight 40/100
    // dst 1, weight 20/100
    // dst 2, weight 20/100
    // dst 3, weight 20/100
  }
endclass

program test;
  weighted s;
  initial begin
    s = new();
    for (int i = 1; i<=10; i++) begin
      assert (s.randomize())
        else $fatal(0,"Problem");
      $display(s);
    end
  end
endprogram
```

```
'{src:1, dst:3}
'{src:1, dst:1}
'{src:2, dst:3}
'{src:3, dst:2}
'{src:1, dst:2}
'{src:1, dst:0}
'{src:3, dst:0}
'{src:0, dst:0}
'{src:2, dst:0}
'{src:1, dst:2}
```

- The *dist* operator allows you to create weighted distributions so that some values are chosen more often than others.
- The *dist* operator takes a list of values and weights, separated by the ":=" or the ":/" operator.
- The values and weights can be constants or variables.
- The values can be a single value or a range such as [low : high].
- The weights are not percentages and do not have to add up to 100.
- The := operator specifies that the weight is the same for every specified value in the range.
- The :/ operator specifies that the weight is to be equally divided between all the values of the range.
- Once again, the values and weights can be constants or variables. You can use variable weights to change distributions on the fly *or even to eliminate choices by setting the weight to zero*.

Constraints:

Using Variable Distribution Weights

24

```
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD} length_e;
  rand length_e len;

  // Weights for dist constraint
  bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

  constraint c_len {
    len dist {BYTE := w_byte,     // Choose a random
              WORD := w_word,     // length using
              LWRD := w_lwrd};    // variable weights
    }
endclass
//-----------------------------
program test;
  BusOp s;
  initial begin
    s = new();
    for (int j = 1; j<=10; j++) begin
      $display(" ");
      for (int i = 1; i<=10; i++) begin
        assert (s.randomize())
          else $fatal(0,"Problem");
        $write(s.len.name,", ");
      end
    end
  end
endprogram
```

```
LWRD, WORD, LWRD, WORD, BYTE, WORD, BYTE, BYTE, WORD, LWRD,
LWRD, BYTE, BYTE, LWRD, WORD, WORD, WORD, WORD, LWRD, WORD,
LWRD, LWRD, LWRD, BYTE, LWRD, LWRD, WORD, LWRD, LWRD, LWRD,
LWRD, WORD, LWRD, LWRD, WORD, LWRD, LWRD, WORD, WORD, LWRD,
WORD, BYTE, WORD, WORD, LWRD, LWRD, WORD, WORD, LWRD, LWRD,
LWRD, LWRD, WORD, WORD, LWRD, LWRD, LWRD, BYTE, LWRD, WORD,
WORD, LWRD, LWRD, WORD, WORD, WORD, LWRD, BYTE, WORD, WORD,
LWRD, WORD, WORD, LWRD, LWRD, WORD, WORD, BYTE, WORD, LWRD,
BYTE, LWRD, LWRD, LWRD, WORD, BYTE, BYTE, LWRD, LWRD, LWRD,
LWRD, WORD, WORD, LWRD, LWRD, LWRD, WORD, BYTE, LWRD, WORD,

LWRD: happened 48 times
WORD: happened 38 times
BYTE: happened 14 times
```

- The len enumerated variable has three values (BYTE, WORD, and LWRD).
- With the default weighting values, longword lengths are chosen more often, as w_lwrd has the largest value.
- Don't worry, you can change the weights on the fly during simulation to get a different distribution.

Constraints:
Conditional

25

```
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD} length_e;
  enum {READ,WRITE} op;
  rand length_e len;

  // Weights for dist constraint
  bit [31:0] w_byte=1, w_word=10, w_lwrd=50;
  constraint c_len {
    len dist {BYTE:=w_byte,
              WORD:=w_word,
              LWRD:=w_lwrd};
  }
  constraint c_len_rw {
    if (op == READ){
      len inside {[BYTE:LWRD]};}
    else len == LWRD;
  }
endclass

//-----------------------------
program test;
  BusOp s;
  initial begin
    s = new();
    s.op = 0;
    for (int i = 1; i<=10; i++) begin
      assert (s.randomize())
        else $fatal(0,"Problem");
      $write(s.op.name, ": "); $display(s.len.name);
    end
  end
endprogram
```

```
READ: LWRD
READ: WORD
READ: LWRD
READ: LWRD
READ: LWRD
READ: WORD
READ: LWRD
READ: BYTE
READ: LWRD
READ: LWRD
```

```
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
WRITE: LWRD
```

```
constraint c_len_rw {
  (op == WRITE)-> len == LWRD;
}
```

You can write the constraint this way also

- SystemVerilog supports two implication operators, "->" and "if-else" in constraint definition.

Bidirectional Constraints

26

```
class random;
  rand logic [15:0] r, s, t;
  constraint c_bidir {
    r < t;
    s == r;
    t < 30;
    s > 25;
  }
endclass

program test;
  random y;
  initial begin
    y = new();
    for (int i = 1; i<=10; i++) begin
      assert (y.randomize())
        else $fatal(0,"Problem in constraits");
      $display("r = %0d,  s = %0d,  t = %0d",y.r,y.s,y.t);
    end
  end
endprogram
```

```
r = 26,  s = 26,  t = 27
r = 27,  s = 27,  t = 28
r = 26,  s = 26,  t = 28
r = 26,  s = 26,  t = 28
r = 27,  s = 27,  t = 28
r = 27,  s = 27,  t = 28
r = 26,  s = 26,  t = 27
r = 26,  s = 26,  t = 27
r = 26,  s = 26,  t = 28
r = 28,  s = 28,  t = 29
```

- Now you may have realized that constraint blocks are not procedural code, executing from top to bottom. They are declarative code, all active at the same time.
- If you constrain a variable with the inside operator with the set [10:50] and have another expression that constrains the variable to be greater than 20. System Verilog solves both constraints simultaneously and only chooses values between 21 and 50.
- System Verilog constraints are bidirectional, which means that the constraints on all random variables are solved concurrently. Adding or removing a constraint on any one variable affects the value chosen for all variables that are related directly or indirectly.
- The SystemVerilog solver looks at all four constraints simultaneously. The variable *r* has to be less than *t,* which has to be less than 30. However. *r* is also constrained to be equal to *s*. which is greater than 25. Even though there is no direct constraint on the lower value of *t*. the constraint on *s* restricts the choices, as shown in the shown results.
- *t* can be any value in the range [0:29]
- *r* is less than t, so r can take any value from [0:28]
- *s* is greater than 25, (i.e. 26, 27, 28, 29, 30, …). But at the same time *s* is equal to *r*, so *s* can only take the values (26, 27, 28).

Controlling Multiple Constraint Blocks

27

```
class Packet;
  rand int length;
  constraint c_short {length inside {[1:32]};}
  constraint c_long {length inside {[1000:1023]};}
endclass

program test;
  Packet p;
  initial begin
    p = new();
    //Create a long packet by disabling short constraint
    p.c_short.constraint_mode(0);
    assert (p.randomize());
    $display ("Length = %0d", p.length);

    // Create a short packet by disabling all constraints
    // then enabling only the short constraint
    p.constraint_mode(0);
    p.c_short.constraint_mode(1);
    assert (p.randomize());
    $display ("Length = %0d", p.length);
  end
endprogram
```

Disable a single constraint

Disable all constraints

Enable a single constraint

```
Length = 1014
Length = 25
```

- A class can contain multiple constraint blocks.
- Sometimes you may need to disable some constraint blocks.
- At run-time. you can use the built-in *constraint_mode()* routine to turn constraints on and off.
- You can control a single constraint with *handle.constraint.constraint_mode()*.
- To control all constraints use *handle.constraint_mode()*.
- The above constraints are contradicting, so you must disable one of them.

Constraining a Constraint

28

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100], [1000:2000]};}
endclass

program test;
  Transaction t;
  initial begin
    t = new();
      for (int i=1; i <=10; i++) begin
      // addr is 50-100, 1000-1500, data < 10
      assert (t.randomize() with
              {addr >= 50;addr <= 1500;data < 10;});
      $display("Addr = %0d, Data = %0d:", t.addr, t.data);
      // force addr to a specific value, data> 10
        assert (t.randomize () with
                {addr == 2000; data> 10; data < 100;});
      $display("Addr = %0d, Data = %0d:", t.addr, t.data);
        $display("----------------");
    end
  end
endprogram
```

```
Addr = 1099, Data = 9:
Addr = 2000, Data = 13:
----------------
Addr = 1056, Data = 5:
Addr = 2000, Data = 54:
----------------
Addr = 1033, Data = 4:
Addr = 2000, Data = 64:
----------------
Addr = 1042, Data = 7:
Addr = 2000, Data = 23:
----------------
Addr = 1400, Data = 6:
Addr = 2000, Data = 55:
----------------
Addr = 1500, Data = 8:
Addr = 2000, Data = 13:
----------------
Addr = 1167, Data = 0:
Addr = 2000, Data = 14:
----------------
Addr = 1005, Data = 2:
Addr = 2000, Data = 42:
----------------
Addr = 1341, Data = 8:
Addr = 2000, Data = 45:
----------------
Addr = 1262, Data = 5:
Addr = 2000, Data = 98:
```

- As you write more tests, you can end up with many constraints. They can interact with each other in unexpected ways, and the extra code to enable and disable them adds to the test complexity.
- Additionally, constantly adding and editing constraints to a class could cause problems in a team environment.
- Many tests only randomize objects at one place in the code. SystemVerilog allows you to add an extra constraint using *randomize () with*.
- This is equivalent to adding an extra constraint to any existing ones in effect.
- The extra constraints are added to the existing ones in effect.
- Use constraint_mode if you need to disable a conflicting constraint.
- Note that inside the *with{}* statement, SystemVerilog uses the scope of the class. That is why we used just *addr*, not *t.addr*.

Random Number Functions

29

```
program test;
  int a;
  int seed=655;

  initial begin
    //Flat distribution, returning signed 32-bit random
    a = $random(); $display("random a = %0d",a);

    //Flat distribution, returning unsigned 32-bit random
    a = $urandom(); $display("urandom a = %0d",a);

    //Flat distribution over a range (min, max)
    a = $urandom_range(50,100); $display("urandom_range a = %0d",a);

    // Exponential decay (seed, mean)
    a = $dist_exponential(seed,30); $display("exponential a = %0d",a);

    // Bell-shaped distribution (seed, mean, deviation)
    a = $dist_normal(seed,30,2); $display("normal a = %0d",a);

    // Bell-shaped distribution (seed, mean)
    a = $dist_poisson(seed,30); $display("poisson a = %0d",a);

    // Flat-shaped distribution (seed, start, end)
    a = $dist_uniform(seed,10,50); $display("uniform a = %0d",a);

  end
endprogram
```

```
random a = 303379748
urandom a = 98710838
urandom_range a = 70
exponential a = 137
normal a = 31
poisson a = 42
uniform a = 12
```

- Some of the useful random functions include the following:
- $random returns a random 32-bit signed integer.
- Seed controls the numbers that $random returns. The seed must be a *reg*, *integer* or *time* variable. Everytime you run the code it will generate the same random numbers as long as the seed is the same. So a good practice is to make the seed equals the computer time.

- The $dist_functions returns pseudo-random values whose characteristics are described by the function name.
  - $random [(seed)]; // the seed is optional
  - $urandom [(seed)]; // the seed is optional
  - $urandom_range (min, max);
  - $dist_exponential(seed, mean);
  - $dist_normal(seed, mean, deviation);
  - $dist_poisson(seed, mean);
  - $dist_uniform(seed, start, end);

Constraining Array and Queue Elements

30

```
class good;
  rand int len[];
  constraint c_len {foreach (len[i])
                      len[i] inside {[1:255]};
                    len.sum < 1024;
                    len.size() inside {[1:8]};}
endclass

program test;
  good a;

  initial begin
    a = new();
    for(int i = 1; i <=10; i++) begin
      assert(a.randomize());
      $display("Sum = %0d, Size = %0d, Array = ",
               a.len.sum, a.len.size, a.len);
    end
  end
endprogram
```

```
Sum = 170, Size = 2, Array = '{150, 20}
Sum = 255, Size = 2, Array = '{120, 135}
Sum = 387, Size = 5, Array = '{149, 27, 20, 116, 75}
Sum = 413, Size = 7, Array = '{7, 144, 4, 102, 35, 12, 109}
Sum = 296, Size = 2, Array = '{141, 155}
Sum = 272, Size = 2, Array = '{142, 130}
Sum = 988, Size = 8, Array = '{210, 205, 220, 111, 101, 79, 59, 3}
Sum = 367, Size = 2, Array = '{221, 146}
Sum = 829, Size = 5, Array = '{215, 122, 137, 124, 231}
Sum = 465, Size = 3, Array = '{153, 255, 57}
```

- SystemVerilog lets you constrain individual elements of an array using *foreach*.
- While you might be able to write constraints for a fixed-size array by listing every element, the *foreach* style is more compact.
- The only practical way to constrain a dynamic array or queue is with *foreach*.

- The constraint solver in SystemVerilog acts like a magic. It can handle very complex constraints.
- How complex can these constraints become? Constraints have been written to solve Einstein's problem (a logic puzzle with five people, each with five separate attributes), the Eight Queens problem (place eight queens on a chess board so that none can capture each other). and even Sudoku.