Chapter 6, Randomization Topics

• Introduction to randomization
• What to randomize?
• A simple class with random variables
• Weighted distributions
• Conditional constraints
• Solution probabilities
• Controlling constraints
• Constraint tips and techniques
• Common Randomization Problems
• Iterative and Array Constraints
• Atomic Stimulus Gen vs Scenario Gen
• Random Device Configuration
Chapter 6: Randomization

• Directed testing:
  • Checks only anticipated bugs
  • Scales poorly as requirements change
  • Little upfront work
  • Linear progress

• Random testing:
  • Checks unanticipated bugs
  • Scales well as requirements change
  • More upfront work
  • Better than linear progress
6.1 Introduction

• A testbench based on randomization is a shotgun
• The features you are trying to test is the target

• How to cover untested areas?
  • More random testing with tighter constraints
  • Directed testing
• When is testing done?
  • Functional coverage
  • Code coverage

*Shotgun Verification or The Homer Simpson Guide to Verification*, Peet James
6.2 What to randomize?

Much more than data
1. Device configuration
2. Environment configuration
3. Primary input data
4. Encapsulated input data
5. Protocol exceptions
6. Errors and violations
7. Delays
8. Test order
9. Seed for the random test
6.3 Randomization in SystemVerilog

• Specified within a class along with constraints

• Variable declared with `rand` keyword distributes values uniformly

```
rand bit [1:0] y;
```

3, 2, 0, 0, 3, 1, .....  

• Variable declared with `randc` keyword distributes values cyclically

• No repeats within an iteration

```
randc bit [1:0] y;
```

```
initial permutation 0,3,2,1
next permutation 0,3,2,1
next permutation 2,0,1,3
```

• Constraints specified with `constraint` keyword

```
constraint y_c {y >= 1; y < 3;}
```

• New values selected when `randomize()` function called

• Returns 1 if constraints can be solved, 0 otherwise

```
<handle>.randomize();
```
6.3 Randomization in SystemVerilog

• Constraint syntax:

```
class constraint_name { constraint_expressions }
```
where:

– `constraint_name` is the name of the constraint block. This name can be used to enable or disable a constraint using the system task `$constraint_mode()`.

– `constraint_expressions` is a list of expression statements that restrict the range of a variable or define relations between variables. A constraint expression is any SystemVerilog expression, or one of the constraint-specific operators: `=>`, `inside` and `dist`.

• The declarative nature of constraints imposes the following restrictions on constraint expressions:

  – Calling tasks or functions is not allowed
  – Operators with side effects, such as `++` and `--` are not allowed.
  – `randc` variables cannot be specified in ordering constraints.
  – `dist` expressions cannot appear in other expressions (unlike `inside`); they can only be top level expressions.

Source: SystemVerilog 3.1 Random Constraints – Proposal Synopsys
### 6.3.1 Simple class with Random Variables

```verilog
class Packet;
    rand bit [31:0] src, dst, data[8];
    randc bit [7:0] kind;
    constraint c {src > 10; src < 15;}
endclass

Packet p;
initial begin
    p=new();
    if (!p.randomize()) // Returns a “0” if problem
        $finish;
    transmit(p);
end
```

**Constraint expressions**
6.3.2 Checking the result from randomization

• Always check the result of a call to `randomize()`
• `randomize()` will not be called if using an assertion to check result and assertions turned off

```plaintext
initial begin
  packet p = new();
  $assertoff;
  assert(p.randomize());
  p.display();
end
```

• `$rand_mode()` method can be used to enable or disable any random variable.
  • When a random variable is disabled, it behaves in exactly the same way as other non-random variables.
6.3.2 Checking the result from... (cont.)

Text uses a macro to check the results from randomization

```
define SV_RAND_CHECK(r) \
  do begin \
    if (!(r)) begin \
      $display("%s:%0d: Randomization failed \"%s\"", \ 
        `__FILE__, `__LINE__, `r`); \
      $finish; \
    end \
  end while (0)

SV_RAND_CHECK(p.randomize());
```

```
# test.sv:13: Randomization failed p.randomize()
```

`__FILE__` is a built-in macro that will display the name of the file the macro is called from

`__LINE__` is a built-in macro that will display the line of the file the macro is called from
6.3.3 The constraint solver

• Solves constraint expressions
• Same seed results in the same random values
• Use different seeds in each nightly regression run.
• Constraints may take a long time to solve if constraints are complex
• Solver is specific to each simulator vendor/release.
6.3.4 What can be randomized?

- 2-state variables
- 4-state variables, only generates 2 state values (no X/Z created)
- Integers
- Bit vectors
- Arrays
- Time
  - Real, string
  - Handle in constraint
Write the SystemVerilog code for the following items:
1) Create a class Exercise1 containing two variables, 8-bit data and 4-bit address. Create a constraint block that keeps address to 3 or 4.
2) In an initial block, construct an Exercise1 object and randomize it. Check the status from randomization.
Write the SystemVerilog code for the following items:

1) Create a class `Exercise1` containing two variables, 8-bit `data` and 4-bit `address`. Create a constraint block that keeps `address` to 3 or 4.

2) In an `initial` block, construct an `Exercise1` object and randomize it. Check the status from randomization.

```systemverilog
class Exercise1;
    rand bit [7:0] data;
    rand bit [3:0] address;
constraint address_c {
    address > 2;
    address < 5;
    // or
    ((address==3) || (address==4));
    // or
    address inside {[3:4]};
} endclass

initial begin
    Exercise1 MyExercise1;
    MyExercise1 = new;
    `SV_RAND_CHECK(MyExercise1.randomize());
    `SV_RAND_CHECK(MyExercise1.randomize());
    $display("data = %0d, address = %0d", MyExercise1.data,  MyExercise1.address);
    `SV_RAND_CHECK(MyExercise1.randomize());
    $display("data = %0d, address = %0d", MyExercise1.data,  MyExercise1.address);
    `SV_RAND_CHECK(MyExercise1.randomize());
    $display("data = %0d, address = %0d", MyExercise1.data,  MyExercise1.address);
    `SV_RAND_CHECK(MyExercise1.randomize());
    $display("data = %0d, address = %0d", MyExercise1.data,  MyExercise1.address);
end
```

# data = 136, address = 3
# data = 59, address = 3
# data = 80, address = 3
# data = 125, address = 4
6.4.2 Simple Expressions

• Each constraint expression should contain only 1 relational operator
  • <, <=, ==, >, =>

```
class Order_bad;
    rand bit [7:0] lo, med, hi;
    constraint bad {lo < med < hi;}
endclass
```

• Constraint bad is broken down into multiple binary relational expressions from left to right.
• lo < med is evaluated. Results in 0 or 1
• 0 or 1 > hi is evaluated.

```
constraint good{lo < med;
    med < hi;}
```
6.4.3 Equivalence Expressions

• Suppose you want to constrain a value to be equal to an expression.

```verilog
class order;
    rand bit [7:0] addr_mode, size, len;
    constraint order_c {len == addr_mode*4 + size;}
endclass
```

• `len` must be declared as random.
• Using `=` is a syntax error.
6.4.4 Weighted Distributions

• Weighted distributions cause a non-uniform distribution
• Weights do not have to add up to 100% and can be variables
• Cannot be used with randc
• What would this be used for?
  • For a CPU want less or more of a particular opcode
  • For a datapath want max neg of 0, and max pos more often

constraint <constraint name> {<variable name> dist {<distribution>}};

constraint c_dist {
    src dist {0:=40, [1:3]:=60};
    dst dist {0:/40, [1:3]:/60};
}

:= operator indicates the weight is the same for all values
:/ operator indicates the weight is distributed across all values
6.4.4 Weighted Distributions := operator

:= operator indicates the weight is the same for all values

```
constraint src_dist { src dist {0:=40, [1:3]:=60} ;}
```

\[40 + 60 + 60 + 60 = 220\]

src = 0, weight = \(\frac{40}{220} = 18\%\)
src = 1, weight = \(\frac{60}{220} = 27\%\)
src = 2, weight = \(\frac{60}{220} = 27\%\)
src = 3, weight = \(\frac{60}{220} = 27\%\)
6.4.4 Weighted Distributions :/ operator

The :/ operator indicates the weight is distributed across all values.

\[
\text{constraint dst_dist} \{ \text{dst dist} \{ [0:40], [1:3]:/60 \} ; \}
\]

\[
dst = 0, \text{ weight } = 40/100 = 40% \\
dst = 1, \text{ weight } = 20/100 = 20% \\
dst = 2, \text{ weight } = 20/100 = 20% \\
dst = 3, \text{ weight } = 20/100 = 20%
\]
6.4.4 Weighted Distributions (cont.)

• Weights can be constants, ranges, or variables.
• Using variables allows the weights to be adjusted on the fly

```java
module sample6p9();

class BusOp;

    typedef enum {BYTE, WORD, LWRD} length_e;
    rand length_e len;
    bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

    constraint c_len {
        len dist {BYTE := w_byte,
                   WORD := w_word,
                   LWRD := w_lwrd};
    }

endclass

BusOp myBusOp;

initial begin
    myBusOp = new();
    $display("len = %0b", myBusOp.len);
    void'(myBusOp.randomize());
    $display("len = %0b", myBusOp.len);
    void'(myBusOp.randomize());
    $display("len = %0b", myBusOp.len);
    void'(myBusOp.randomize());
    $display("len = %0b", myBusOp.len);
    void'(myBusOp.randomize());
    $display("len = %0b", myBusOp.len);
    void'(myBusOp.randomize());
    $display("len = %0b", myBusOp.len);
end
```

# len = 0
# len = 1
# len = 2
# len = 2
# len = 0
Constraint Exercise 2

Modify the solution for Exercise 1 to create a new class Exercise2 so that:

1. data is always equal to 5
2. Probability of address = 4’d0 is 10%
3. Probability of address being between [1:14] is 80%
4. Probability of address = 4’d15 is 10%

Demonstrate its usage by generating 20 new data and address values and check for error.
package my_package;

class Exercise2;
    rand bit [7:0] data;
    rand bit [3:0] address;
    constraint data_c {data == 5;}
    constraint address_dist {
        address dist {0:=10, [1:14]:=80, 15:=10};
    }

    function void print_all;
        $display("data = %d, address = %d", data, address);
    endfunction

endclass // Exercise2
endpackage

program automatic test;
    import my_package::*;

    initial begin
        Exercise2 MyExercise2;
        repeat (20) begin
            MyExercise2 = new;
            `SV_RAND_CHECK(MyExercise2.randomize());
            MyExercise2.print_all();
        end
    end
endprogram
6.4.5 Set membership and the inside op

- Alternative to `{var>value1 ; var<value2}` is the inside keyword

```
constraint address_c {address > 2; address < 5;}
constraint address_range{address inside{[3:4]};}  
```

- Using the ! operator can exclude ranges

```
constraint c_range {
   !(c inside{[lo:hi]});
}
```
6.4.6 Using an array in a set

• Suppose you want to create multiple equivalence constraints
• For example: f can only equal 1, 2, 3, 5, 8

```plaintext
rand bit [7:0] f;
constraint c_fibonacci {
    (f == 1) || (f==2) || (f==3) || (f==5) || (f==8));}
```

• Alternate solution is to store the values in an array

```plaintext
rand bit [7:0] f;
bit [31:0] vals[] = '{1,2,3,5,8};
constraint c_fibonacci {f inside {vals;}}
```

• Can specify that values in the array are NOT to be chosen

```plaintext
rand bit [7:0] notf;
bit [31:0] vals[] = '{1,2,3,5,8};
constraint c_fibonacci {!(notf inside vals);}
6.4.6 Using an array in a set

```
rand bit [7:0] f;
bit [31:0] vals[] = '{1,2,3,5,8};
constraint c_fibonacci {f inside {vals;}}
```

```
# count[1] = 3943
# count[2] = 4054
# count[3] = 3922
# count[5] = 4094
# count[8] = 3987
```

- Values listed multiple times in array still occur with same as value listed once in array:

```
rand bit [7:0] f;
bit [31:0] vals[] = '{1,8,8,8,8};
constraint c_fibonacci {f inside {vals;}}
```

```
# count[1] = 10075
# count[8] = 9925
```

Not in book but needed for Questasim to compile
6.4.6 Using an array in a set

class Days;
    typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} days_e;
    days_e choices[];
    rand days_e choice;
    constraint cday {choice inside {choices};};
endclass // Days

initial begin
    Days days;
    days = new;
    // Nothing in queue yet
    $display("choices queue = %p", days.choices);
    days.choices = { Days:: SUN, Days:: SAT};
    // Now SUN and SAT in choices queue
    $display("choices queue = %p", days.choices);
    `SV_RAND_CHECK(days.randomize());
    $display("Random weekend day %s\n", days.choice.name());
    days.choices = {Days::MON, Days::TUE, Days::WED, Days::THU, Days::FRI};
    $display("choices queue = %p", days.choices);
    `SV_RAND_CHECK(days.randomize());
    $display("Random week day %s\n", days.choice.name());
end
6.4.7 Bidirectional Constraints

- Constraint blocks are not procedural but declarative.
- All constraints are active at the same time.

```
rand bit [15:0] r, s, t;
constraint c_bidir {
    r < t;
    s == r;
    t < 10;
    s > 5;
}
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>r</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
6.4.8 Implication Constraints

Suppose you want to impose different constraints depending on a variable. Remember that the constraint is bidirectional.

Solution 1:

```c
constraint mode_c {
    if (mode == small)
        len < 10;
    else if (mode == large)
        len > 100;
}
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>a (mode==small)</th>
<th>b (mode==large)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>10 &gt;= len &gt;= 100</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>len &lt; 10 or len &gt; 100</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>mode == small or mode == large</td>
</tr>
</tbody>
</table>
6.4.8 Implication Constraints

Suppose you want to impose different constraints depending on a variable.

Solution 2:
- Syntax is expression -> constraint set
  - If expression is true
    - Constraint solver should satisfy the constraint set
  - If expression is false
    - The random numbers generated are unconstrained by the constraint set.
- Boolean equivalent is:
  
\[ \neg a \lor b \]

```java
constraint mode_c {
  (mode == small) -> len < 10;
  (mode == large) -> len > 100;
}
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
6.4.9 Equivalence operator

- The equivalence operator \( <\rightarrow > \) is bidirectional.
- Relationship is “if and only if”.
- \( A <\rightarrow > B \) is defined as \((A \rightarrow B) \&\& (B \rightarrow A)\)

\[
\text{rand bit } d, e; \\
\text{constraint } c \{ \text{d==1 } <\rightarrow > \text{ e==1; } \}
\]

- when \( d == 1 \), the variable \( e \) must be 1
- when \( e == 1 \), \( d \) can be 0 or 1

<table>
<thead>
<tr>
<th>((d==1) \rightarrow (e==1))</th>
<th>((e==1) =\text{false})</th>
<th>((e==1) =\text{true})</th>
</tr>
</thead>
<tbody>
<tr>
<td>((d==1) =\text{false})</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>((d==1) =\text{true})</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
class Stim;
    const bit [31:0] CONGEST_ADDR = 42;
    typedef enum {READ, WRITE, CONTROL} stim_e;
    randc stim_e kind;
    rand bit [31:0] len, src, dst;
    bit congestion_test;

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

What are the constraints on len, dst, and src for this code?
6.4.1 Constraint Introduction - Exercise

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

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

- len must be between 1 and 999 inclusive
- if congestion_test == 1, dst must be inside 42-10 = 32 to 42+10=52 and src=42
- else src can take values of 0, 2 to 10, and 100 to 107. dst is unconstrained
It’s important to understand how constraints affect the probability of the solution.

**Unconstrained:**

```
class Unconstrained;
    rand bit x;
    rand bit [1:0] y;
endclass
```

As there are no constraints, each has the same probability.
6.5.2 Implication

class Impl1;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
    }
endclass

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>1/8</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>1/8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(x==0) -&gt; (y==0)</th>
<th>(y==0)=false</th>
<th>(y==0)=true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x==0)=false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(x==0)=true</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
6.5.3 Implication and bidirectional constraints

class Imp2;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        y>0;
        (x==0)->y==0;
    }
endclass

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/3</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>1/3</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>1/3</td>
</tr>
</tbody>
</table>
6.5.4 Guiding Distribution with solve/before

- **Solve before** tells the solver to solve for 1 variable before another.
- The possible solutions does not change, just the probability.
- When \( x = 0 \), \( y \) must be 0, when \( x = 1 \), \( y \) can be 0, 1, 2, or 3.

```plaintext
class SolveBefore;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        (x==0) -> y==0;
        solve x before y;
    }
endclass
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/8</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>1/8</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>1/8</td>
</tr>
</tbody>
</table>
solve y before x;

• When y = 1, 2, or 3, X must be 1. When y = 0, x can be 0 or 1.

class Impl;
  rand bit x;
  rand bit [1:0] y;
  constraint c_xy {
    (x==0)->y==0;
    solve y before x;
  }
endclass

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>1/8</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1/4</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>1/4</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>1/4</td>
</tr>
</tbody>
</table>
Solution Probabilities Exercise

Complete the table for the following constraints

class MemTrans;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        y inside{[x:3]};
        solve x before y;
    }
endclass

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Solution Probabilities Exercise

Complete the table for the following constraints

```
class MemTrans;
    rand bit x;
    rand bit [1:0] y;
    constraint c_xy {
        y inside{[x:3]};
        solve x before y;
    }
endclass
```

<table>
<thead>
<tr>
<th>Solution</th>
<th>x</th>
<th>y</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>$\frac{1}{8}$</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>$\frac{1}{8}$</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>$\frac{1}{8}$</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
<td>$\frac{1}{8}$</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>$\frac{1}{6}$</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>2</td>
<td>$\frac{1}{6}$</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>3</td>
<td>$\frac{1}{6}$</td>
</tr>
</tbody>
</table>

Probability of $x = 0$ is $\frac{1}{2}$
Probability of $y = 0, 1, 2, 3$ is $\frac{1}{4}$
Total = $\frac{1}{2} \times \frac{1}{4} = \frac{1}{8}$

Probability of $x = 1$ is $\frac{1}{2}$
Probability of $y = 0$ as not inside 1:3
Probability of $y = 1, 2, 3$ is $\frac{1}{3}$
Total = $\frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$
6.6 Controlling Multiple Constraint Blocks

Use the `constraint_mode()` function to turn constraints on/off:

```
<handle>.constraint_mode(<0/1>);
<handle>..<constraint>..constraint_mode(<0/1>);
```

```verbatim

class Packet

    rand bit [31:0] length;
    constraint c_short {length inside {[1:32]};}
    constraint c_long {length inside {[1000:1023]};}

endclass

Packet p:

initial begin
    p=new();
    p.c_short.constraint_mode(0);
    `SV_RAND_CHECK(p.randomize());
    transmit(p);
    ...
    p.constraint_mode(0);
    p.c_short.constraint_mode(1);
    `SV_RAND_CHECK(p.randomize());
    transmit(p);
    ....
    end // initial
```
6.7 Valid Constraints

A suggested technique to creating valid stimulus is to create valid constraints.

Turn the constraint off to test the system’s response to invalid stimulus.

For example, suppose a read-modify-write command is only valid if the length is a long word.

class Transaction;
    typedef enum {BYTE, WORD, LWRD, QWRD} length_e;
    typedef enum {READ, WRITE, RMW, INTR} access_e;
    rand length_e length;
    rand access_e access;

    constraint valid_RMW_LWRD {
        (access == RMW) -> (length == LWRD);
    }
endclass
6.8 In-line Constraints

• In-line constraints create constraints outside of the class.
• Add to existing constraints if they are enabled.
• For example, a single test needs to be written with tighter than usual address constraints

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

initial begin
    Transaction t;
    t=new();
    `SV_RAND_CHECK(t.randomize() with {addr >=50; addr <=1500;
        data <10;});
    driveBus(t);
    `SV_RAND_CHECK(t.randomize() with {addr ==2000; data >10;});
    driveBus(t);
end
```
Valid and In-line Constraint Exercise

For the following class create:
1. A constraint that limits read transactions addresses to the range 0 to 7, inclusive
2. Write behavioral code to turn off the above constraint. Construct and randomize a MemTrans object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

```verilog
class MemTrans;
    rand bit rw; // read if rw = 0, write if rw = 1
    rand bit [7:0] data_in;
    rand bit [3:0] address;
endclass
```
Valid and In-line Constraint Exercise (cont.)

For the following class create:

1. A constraint that limits read transactions addresses to the range 0 to 7, inclusive
2. Write behavioral code to turn off the above constraint. Construct and randomize a MemTrans object with an in-line constraint that limits read transaction addresses to the range 0 to 8, inclusive. Test that the in-line constraint is working.

```verbatim
class MemTrans;
  rand bit rw; // read if rw = 0, write if rw = 1
  rand bit [7:0] data_in;
  rand bit [3:0] address;
  _constraint valid_rw_addr {rw == 0)->(address inside {[0:7]});}  
endclass // MemTrans

MemTrans MyMemTrans;
initial begin
  MyMemTrans = new();
  MyMemTrans.valid_rw_address.constraint_mode(0);
  `SV_RAND_CHECK(MyMemTrans.randomize() with {(rw == 0)->(address inside {[0:8]});}); 
end
```
6.9 pre_randomize/post_randomize

- Implicitly called before/after every call to `randomize()`
- `void` function
  - Cannot consume time.
  - Can only call other functions.
  - Does not return a value
- Overload to add your functionality
- `post_randomize()` is good for cleaning up

- `$\text{dist_exponential} \ (\text{seed_expression}, \ \text{mean_expression})$`
  - `mean_expression` parameter causes the average value of the return value to approach the mean. The `mean_expression` must be an integer greater than zero (0).

- Other distributions available:
  - Integer and real
module sample6p35_tb();

class Bathtub;
int value;
int WIDTH = 50, DEPTH = 6, seed = 1;

function void pre_randomize();
value = $dist_exponential(seed, DEPTH);
if (value > WIDTH) value = WIDTH;
if ($urandom_range(1))
    value = WIDTH - value;
$display("value = %0d", value);
endfunction // pre_randomize
endclass // Bathtub

Bathtub bathtub;
initial begin
bathtub = new();
repeat(250)
    bathtub.pre_randomize;
end
endmodule // sample6p35_tb
6.11 Constraint Tips and Techniques

• Instead of hardcoding constraints, use variables with defaults
  • Allows the constraint to be modified without modifying the class
  • Allows invalid stimulus to be generated

```verbatim
class Packet;
    rand bit [31:0] length;
    constraint c_length {
        length inside {{1:100}};
    }
endclass

initial begin
    Packet p1 = new();
    p1.max_length = 200;
    p1.randomize();
end
```

```verbatim
class Packet;
    rand bit [31:0] length;
    int max_length= 100;
    constraint c_length {
        length inside {{1:max_length}};
    }
endclass
```
6.11.2 Using Nonrandom Values

- `constraint_mode()` turns on/off constraints
- `rand_mode()` makes a variable or every variable in an object non-random

```verilog
class Packet;
    rand bit [7:0] length;
    constraint c_length{length > 0;}
    .... // Other constraints depending on length
endclass

initial begin
    Packet p = new();
    `SV_RAND_CHECK(p.randomize());
    p.length.rand_mode(0);
    p.length = 0;
    `SV_RAND_CHECK(p.randomize());
    p.rand_mode(0);
end
```

- Make length nonrandom
- Create an invalid length
- Value for length will be included in constraint solution
6.11.3 Checking Values using Constraints

• If you change the value of random variables how do you know all your random variables are still valid?
• Use a call to `<handle>.randomize(null)` to check.

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

Transaction t;
initial begin
    t=new();
    `SV_RAND_CHECK(t.randomize());
    t.addr = 200;
    `SV_RAND_CHECK(t.randomize(null));
end
```

...Randomization failed

```verilog
`define SV_RAND_CHECK(r) \
    do begin \
        if (!r) begin \
            $display("%s:%0d: Randomization failed ", \
                `__FILE__, `__LINE__, `"r"); \
            $finish; \
        end \
    end while (0)
```
6.11.4 Randomizing Individual Variables

Can pass variables to `randomize()` to randomize only a subset of variables

```verbatim
class Rising;
  bit [7:0] low;
  rand bit [7:0] med, hi;
  constraint up { low < med; med < hi; }
endclass

initial begin
  Rising r;
  r = new();
  `SV_RAND_CHECK(r.randomize());
  `SV_RAND_CHECK(r.randomize(med));
  `SV_RAND_CHECK(r.randomize(low));
  `SV_RAND_CHECK(r.randomize(low, med));
end
```
6.11.5 Turn Constraints Off and On

• Use many simple constraints instead of 1 complex constraint
• Turn on the constraints needed

```plaintext
class Instruction;
    typedef enum {NOP, HALT, CLR, NOT} opcode_e;
    rand opcode_e opcode;
    bit [1:0] n_operands;
    constraint c_operands{
        if (n_operands == 0)
            (opcode == NOP) || (opcode == HALT);
        else if (n_operands == 1)
            (opcode == CLR) || (opcode == NOT);
        .......
    }
endclass
```
6.11.5 Turn Constraints Off and On (cont.)

class Instruction;
typedef enum {NOP, HALT, CLR, NOT} opcode_e;
    rand opcode_e opcode;
    constraint c_no_operands{
        (opcode == NOP) || (opcode == HALT);}
    constraint c_one_operand{
        (opcode == CLR) || (opcode == NOT);}

    ....
}
endclass

initial begin
    Instruction instr = new();
    instr.constraint_mode(0);
    instr.c_no_operands.constraint_mode(1);
    `SV_RAND_CHECK(instr.randomize());
end
6.12 Common Randomization Problems

• Using a signed variable isn’t an issue if you control the values

```c
for (int i=0; i<=5; i++)
```

• However, a randomized signed variable will produce negative values

```c
class SignedVars;
    rand byte pkt1_len, pk2_len;
    constraint total_len {pkt1_len + pk2_len == 64;}
endclass
```

• Some valid solutions of \{pkt1_len, pkt2_len\} are:

```
(32,32)  
(2,62)   
(-63, 127)
```
6.12.1 Use Signed Values with care

• Might be tempted to declare `pkt1_len, pk2_len` as large unsigned

    
    class Vars32;
    
    rand bit [31:0] pkt1_len, pk2_len;
    
    constraint total_len {pkt1_len + pk2_len == 64;}  
    
    endclass

• A valid solution of `{pkt1_len, pkt2_len}` is

    
    (32'h8000_0040, 32'h8000_0000) = 33'h140 = 32'h40 == 32’d64

• One solution is to constrain the max values of `pkt1_len` and `pk2_len`

• Best solution is to only use values as wide as required

    
    class Vars8;
    
    rand bit [7:0] pkt1_len, pkt2_len;
    
    constraint total_len {pkt1_len + pkt2_len == 9’d64;}  
    
    endclass
6.12.2 Solver performance Tips

- Operators like \(/\), \(*\), \(\%\) will slow down the constraint solver
- Use left shifts for multiply by 2
- Use right shifts for divide by 2
- Use bit-wise AND and mask for modulo by 2

\[ x \% y \equiv (x \& (y - 1)) \]

<table>
<thead>
<tr>
<th>9%4 = 1001 &amp; 0011</th>
<th>8%4 = 1000 &amp; 0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0000</td>
</tr>
</tbody>
</table>

- Use $urandom$ or $urandom_range$ instead of the solver.
6.13 Iterative and Array Constraints

• Constraining the size of an array

```
class dyn_size;
    rand bit [31:0] d[];
    constraint d_size {d.size() inside {[1:10]};}
endclass
```

Constrains the array element contents between 1 and 255

• Constraining the elements of an array

```
class good_sum5;
    rand uint len[];
    constraint c_len {foreach (len[i])
        len[i] inside {[1:255]};
        len.sum < 1024;
        len.size() inside {[1:8]};
    }
endclass
```

Constrains the sum of the array contents to less than 1024
Constrains the # of array elements from 1 to 8

• The solver can handle hundreds of constraints.

<table>
<thead>
<tr>
<th>sum</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>83  249 197 187 152 95 40 8</td>
</tr>
<tr>
<td>1012</td>
<td>213 252 213 44 196 20 20 54</td>
</tr>
<tr>
<td>370</td>
<td>118 76 176</td>
</tr>
</tbody>
</table>

… see Sample 6.59/6.60
typedef int unsigned uint;

module slide55_tb();
test t1();
endmodule // slide55_tb

program automatic test;

class good_sum5;
    rand uint len[];
    constraint c_len {foreach (len[i])
        len[i] inside {[1:255]};
        len.sum < 1024;
        len.size() inside {[1:8]};}

    function void display();
        $write("sum=%4d, val=", len.sum());
        foreach(len[i]) $write("%4d ", len[i]); $display;
    endfunction

good_sum5 c;
initial begin
    c = new();
    repeat (20) begin
        `SV_RAND_CHECK(c.randomize());
        c.display();
    end
end
endprogram
6.13.6 Randomizing an array of handles

• Need to allocate all the elements before randomization as the random solver never constructs objects.

• For a dynamic array, allocate the maximum number of elements needed and then use a constraint to resize array.

• A dynamic array of handles can remain the same size or shrink during randomization but it can never increase in size.
6.13.6 Randomizing an array of handles

```verilog
`define SV_RAND_CHECK(r) \
  do begin \
    if (!r) begin \
      $display("%s:%0d: Randomization failed \"%s\", \n        __FILE__, __LINE__, "r"); \n      $finish; \n    end \
  end while (0)

module sample6p67_tb();
  parameter MAX_SIZE = 25;
  class RandStuff;
    rand bit [31:0] value;
  endclass // RandStuff
  class RandArray;
    rand RandStuff array[];
    constraint c {array.size() inside {[1:MAX_SIZE]};}
    function new();
      array = new[MAX_SIZE];
      foreach (array[i])
        array[i] = new();
    endfunction; // new
  endclass // RandArray
  RandArray ra;
  initial begin
    ra = new();
    SV_RAND_CHECK(ra.randomize());
    foreach (ra.array[i])
      $display("ra.array[%2d]: ", i, ra.array[i].value);
  end
endmodule // sample6p67
```

```text
# ra.array[ 0]: 3507107891
# ra.array[ 1]: 4013053989
# ra.array[ 2]: 2773609279
# ra.array[ 3]: 528996684
# ra.array[ 4]: 3476166007
# ra.array[ 5]: 3634918746
# ra.array[ 6]: 1796308729
# ra.array[ 7]: 1730985904
# ra.array[ 8]: 1222782346
# ra.array[ 9]: 1298121202
# ra.array[10]: 1119090603
# ra.array[11]: 4220722116
# ra.array[12]: 1074911065
# ra.array[13]: 1269454440
# ra.array[14]: 2256144349
# ra.array[15]: 806154056
```

max decimal value for 32 bits: 4294967295
6.13.6 Randomizing an array of handles

• For the homework in Chap6 you might create a fixed array of handles

```verilog
localparam TESTS = 10;
Transaction Transaction_array[TESTS];
```

• Can create a random array of handles

```verilog
class TransactionArray;
    rand Transaction Transaction_array[];
    constraint c {Transaction_array.size() inside {[1:TESTS]}; }
    function new();
        Transaction_array = new[TESTS];
        foreach (Transaction_array_array[i])
            Transaction_array_array[i] = new();
    endfunction;
endclass
```
Array Constraint Exercise

Create a class for a graphics image that is 10x10 pixels. The value for each pixel can be randomized to black or white. Randomly generate an image that is, on average, 20% white.
package my_package;
    parameter int HEIGHT = 10;
    parameter int WIDTH = 10;
    parameter int PERCENT_WHITE = 20;
    typedef enum bit {BLACK, WHITE} colors_t;

class Screen;
    rand colors_t pixels [HEIGHT] [WIDTH];
    constraint colors_c {foreach (pixels[i][j]) pixels[i][j] dist { BLACK := 100 - PERCENT_WHITE,
    WHITE := PERCENT_WHITE};

    //    or
    // constraint colors_c {(pixels.sum() with (item.sum() with (int'(item))) == PERCENT_WHITE);}"

    // This function is not necessary for the in-class solution
    function void print_screen;
    int count = 0;
    foreach (pixels[i])
        $display("%p", pixels[i]);
    foreach (pixels[i][j])
        count += pixels[i][j];
    $display("Num of pixels=WHITE = %0d", count);
endfunction //
endclass
endpackage

`default_nettype none
program automatic test;
import my_package::*;
Screen my_screen;
initial begin
    my_screen=new();
    `SV_RAND_CHECK(my_screen.randomize());
    `SV_RAND_CHECK(my_screen.print_screen();
end
endprogram
6.14 Atomic Stimulus Gen vs Scenario Gen

• Up to this point we’ve covered creating atomic random transactions
• Real-world scenarios have long sequences of transactions
• Methods to create scenarios
  • An atomic generator with history
  • Randsequence
  • Random array of objects
6.14.1 An Atomic generator with History

Suppose we want to create a sequence of transactions where back to back writes are not allowed

```plaintext
package my_package;
    typedef enum {READ, WRITE} rw_t;
    class Transaction;
        rw_t old_rw;
        rand rw_t rw;
        rand bit [31:0] addr, data;
        constraint rw_c{if (old_rw == WRITE) rw != WRITE;};
    function void post_randomize;
        old_rw = rw;
    endfunction
    function void print_all;
        $display("addr = %d, data = %d, rw = %s", addr, data, rw);
    endfunction
endclass
endpackage
```
module slide63_tb();
    import my_package::*;

    Transaction t;

    initial begin
        for (int i=0; i<=50; i++) begin
            t = new();
            `SV_RAND_CHECK (t.randomize());
            t.print_all;
        end
    end

endmodule // slide63_tb

package my_package;
    typedef enum {READ, WRITE} rw_t;

    class Transaction;
        rw_t old_rw;
        rand rw_t rw;
        rand bit [31:0] addr, data;

        constraint rw_c{if (old_rw == WRITE) rw != WRITE;};
        function void post_randomize;
            old_rw = rw;
        endfunction //

        function void print_all;
            $display("addr = %d, data = %d, rw = %s", addr, data, rw);
        endfunction
    endclass
endpackage
6.14.1 Usage of atomic generator w/ history

```
initial begin
    for (int i=0;i<=50;i++) begin
        t=new();
        `SV_RAND_CHECK (t.randomize());
        t.print_all;
    end
end
```

Back to back writes

```
# ... rw = READ
# ... rw = WRITE
# ... rw = READ
# ... rw = WRITE
# ... rw = WRITE
# ... rw = WRITE
```
6.14.2 Random Array of Objects

- Create as many transaction objects as needed
- Constrain the current object by previous objects

```verilog
class Transaction;
    rand rw_t rw;
    rand bit [31:0] addr, data;
endclass

class RandTransaction;
    rand Transaction trans_array[];
    constraint rw_c {foreach (trans_array[i])
        if ((i>0) && (trans_array[i-1].rw == WRITE))
            trans_array[i].rw != WRITE;}

    function new();
        trans_array = new[TESTS];
        foreach (trans_array[i])
            trans_array[i] = new();
    endfunction
endclass
```
module slide66_tb();
    
    parameter TESTS = 5;

typedef enum {READ, WRITE} rw_t;

class Transaction;
    rand rw_t rw;
    rand bit [31:0] addr, data;
endclass

class RandTransaction;
    rand Transaction trans_array[];
    constraint rw_c {foreach (trans_array[i])
        if ((i>0) && (trans_array[i-1].rw == WRITE))
            trans_array[i].rw != WRITE;}

function new();
    trans_array = new[TESTS];
    foreach (trans_array[i]) begin
        trans_array[i] = new();
    end
endfunction;
endclass

RandTransaction t;

initial begin
    t = new();
    foreach(t.trans_array[i])
        $display("i = %0d, rw = %p",i, t.trans_array[i]);
    `SV_RAND_CHECK(t.randomize());
    foreach(t.trans_array[i])
        $display("i = %0d, rw = %p",i, t.trans_array[i]);
end
endmodule
SystemVerilog introduces the `randsequence` construct
Uses a Backus-Naur Form (BNF) like syntax to describe the grammar of the transaction

```verilog
initial begin
  for (int i=0; i<15; i++) begin
    randsequence (stream)
      stream : cfg_read := 1 | weights
        io_read := 2 |
        mem_read := 5;
    endsequence
    cfg_read : { cfg_read_task; } | { cfg_read_task; } cfg_read;
    mem_read : { mem_read_task; } | { mem_read_task; } mem_read;
    io_read : { io_read_task; } | { io_read_task; } io_read;
  endsequence
end end
```
Scenario Generation Exercise

Expand the Transaction class in slide “6.14.1 An Atomic generator with History” or slide “6.14.3 Random Array of Objects” so that back to back transactions of the same type do not have the same address.
Scenario Generation Exercise

Expand the Transaction class in slide “6.14.1 An Atomic generator with History” or slide “6.14.3 Random Array of Objects” so that back to back transactions of the same type do not have the same address.

If you modified the An Atomic generator with History (6.14.1) add to the class:

```
bit [31:0] old_addr;
constraint addr_c{if (old_rw == READ)
    old_addr != addr;}
```

to the post_randomize function add old_addr = addr;
6.17 Random Device Configuration

- Randomizing data is not enough, randomize the environment
- Using an example of a 4-port Ethernet switch

```vhdl
class EthCfg;
    rand bit [3:0] in_use;       // Ports used in test: 3, 2, 1, 0
    rand bit [47:0] mac_addr[4]; // MAC addresses
    rand bit [3:0] is_100;       // 100mb mode for ports 3, 2, 1, 0
    rand uint run_for_n_frames;  // # frames in test

    // Force some addr bits when running in unicast mode
    constraint local_unicast {
        foreach (mac_addr[i])
            mac_addr[i][41:40] == 2'bx00;
    }

    constraint reasonable {
        // Limit test length
        run_for_n_frames inside {[1:100]};
    }
endclass : eth_cfg
```
class Environment;
    EthCfg cfg; EthGen gen[4]; EthMii drv[4];
    function new(); cfg = new(); endfunction

    // Use random config. to build the environment
    function void build();
        foreach (gen[i]) begin
            gen[i] = new(); drv[i] = new();
            if (cfg.is_100[i]) drv[i].set_speed(100); end
        endfunction

    function void gen_cfg;
        `SV_RAND_CHECK(cfg.randomize());
    endfunction

    task run();
        foreach (gen[i])
            if (cfg.in_use[i]) begin
                // Only start the testbench transactors that are in-use
                gen[i].run();
                ...
            end
        endtask
endclass : Environment
program test;
    Environment env;
    initial begin
        env = new();
        env.gen_cfg;
        env.build();
        env.run();
        env.wrap_up();
    end
endprogram

Turn on/off constraints here
Calls randomize here
Override generated values here