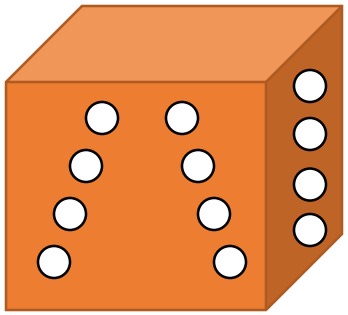
# System Verilog

## RANDOMIZATION

# Why Randomize?

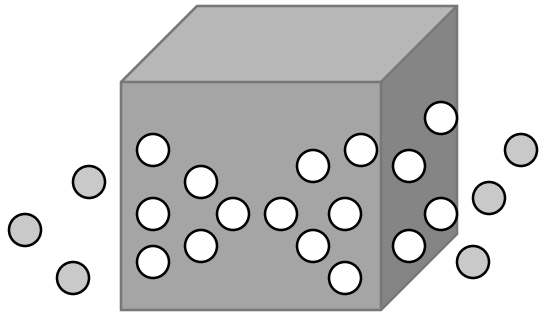
- As designs grow it becomes more difficult to verify their functionally through directed test cases.
- Directed test cases checks specific features of a design and can only detect anticipated bugs.
- Verifying your design using this approach is a time consuming process.
- Randomization helps us detecting bugs that we do not expect in our design.

# Comparison



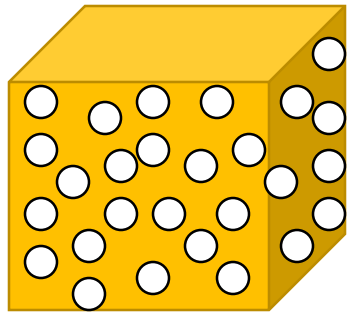
Directed

- Verifies specific scenarios.
- Time Consuming.
- Linear progress.



Random

- Broader Coverage.
- TB's are easy to write.
- Tests are redundant.
- Takes longer time to achieve functionality.



Constrained Random

- Broad and Deep
- Tests are more productive
- Finds corner cases
- Constrained to achieve functionality

# What to Randomize?

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data
- Protocol exceptions
- Errors and violations
- Delays
- Test order
- Seed for the random test

# Verilog Constrained Randomization

# Random in range

```
module test;
integer a, b, c;

initial
repeat(20) begin
a=$random % 10;                  //-9 to 9 (Random range)
b={$random} % 20;                //0 to 19 (Random range)
c=$unsigned($random)%15;          //0 to 14 (Random range)
#2; end
endmodule
```

# Random in range

```
module test;
integer a, b, c;

initial
repeat(20) begin
a= 10 +  {$random} % 6;          //10 to 15   (positive range)
b= -5 - {$random} % 6;           //-5 to -10   (negative range)
c = -5 + {$random} % 16;         //-5 to 10    (mix range)
#2; end
endmodule
```

# Algorithms

Positive Range:

result= min + {$random} % (max – min + 1);

Negative Range:

result= -min - {$random} % (max – min + 1);

Mix Range:

result= -min + {$random} % (max + min + 1);

//min is the magnitude of minimum number

//max is the magnitude of maximum number

# Random between ranges

```
module test;
integer a;

initial
repeat(20)
if ({$random} % 2)
#2 a=10 +  {$random} % 6;          //10 to 15
else
#2 a= 3 + {$random} % 5;          // 3 to 7
endmodule
```

# Weighted Random numbers

```
module test;
integer a, count=0;

always if(count< 10) #2 count=count+1; else #2 count=0;

initial repeat(20)
if (count<3)
#2 a=1 +  {$random} % 9;          //1 to 9
else
#2 a=11 + {$random} % 8;          // 11 to 18 Higher weight
endmodule
```

# Real random numbers

```
module test;
reg sign; reg [7:0] exp;
reg [22:0] mantisa; real a;

initial repeat(20) begin
sign=$random;
exp=$random;
mantisa=$random;
a=$bitstoshortreal({sign, exp, mantisa});
#2; end
endmodule
```

# Unique random numbers

Generate 10 unique random numbers

```
integer rec [0:9];
integer i, temp, num, index=0;

initial begin
$monitor("num=%0d", num);
```

```
while(index!=10) begin
temp=$random;
    begin: loop
    for(i=0; i<index; i=i+1)
    if(rec[i]==temp)
    disable loop;
rec[index]=temp;
index=index + 1; num=temp;
#2; end
end end
```

# Result

# num=303379748
# num=-1064739199
# num=-2071669239
# num=-1309649309
# num=112818957
# num=1189058957
# num=-1295874971
# num=-1992863214
# num=15983361
# num=114806029

# Unique random numbers

Generate 10 unique random numbers between 0 to 99

```
integer rec [0:9];
integer i, temp, rand, index=0;
while(index!=10) begin
temp={$random} % 100;
    begin: loop
    for(i=0; i<index; i=i+1)
    if(rec[i]==temp)
    disable loop;
rec[index]=temp;
index=index + 1; rand=temp;
#2; end
end
```

# Result

# num=48
# num=97
# num=57
# num=87
# num=57
# num=25
# num=82
# num=61
# num=29

# Other types

- Verilog also offers few more randomization system functions apart from $random. They can be categorized as following:
  - $dist_uniform (seed, start, end)
  - $dist_normal (seed, mean, standard_deviation)
  - $dist_exponential (seed, mean)
  - $dist_poisson (seed, mean)
  - $dist_chi_square (seed, degree_of_freedom)
  - $dist_t (seed, degree_of_freedom)
  - $dist_erlang (seed, k_stage, mean)

# $dist_uniform

```
module test;
integer num1, num2, seed;

initial
repeat(20) begin
num1=$dist_uniform (seed, 5, 15);        //5 to 15
num2=$dist_uniform (seed, -5, 10);      //-5 to 10
#2; end
endmodule
```

# SV Constrained Randomization

# $urandom

```
module test;
integer num1, num2, seed;

initial
repeat(20) begin
#2 num1=$urandom (seed);    //Unsigned 32-bit Random Number
num2=$urandom;
end
endmodule
```

# $urandom_range

```
module test;
integer num1, num2 , num3;

initial
repeat(20) begin
#2 num1=$urandom_range(35, 20);        //35:max to 20:min
num2=$urandom_range(9);                //9:max to 0:min
num3=$urandom_range(10,15);            //10:min to 15:max
end
endmodule
```

# Result

```
# num1=27 num2=8 num3=10
# num1=32 num2=0 num3=11
# num1=26 num2=0 num3=14
# num1=29 num2=0 num3=13
# num1=21 num2=6 num3=12
# num1=25 num2=4 num3=10
# num1=20 num2=7 num3=12
# num1=23 num2=2 num3=12
# num1=33 num2=2 num3=13
# num1=22 num2=1 num3=11
# num1=34 num2=8 num3=14
# num1=24 num2=2 num3=15
```

# Randomize function

- SV provides scope randomize function which is used to randomize variables present in current scope.
- randomize() function can accept any number of variables which have to be randomized as an arguments.
- This function returns true or ‘1’ if randomization was successful else false or ‘0’.
- User can also provide “inline” constraints to control range of random values.

# Randomize function

```
module test;
integer num1, num2;

initial
repeat(20) begin
if(randomize(num1, num2))        //Randomize num1 and num2
$display(“Randomization Successful”);
else $display(“Randomization Failed”);
#2 ; end
endmodule
```

# Randomize function with constraint

```
module test;
integer num;

initial
repeat(20) begin
if(randomize(num) with {num>10; num<20;} )
$display(“Randomization Successful”);
//num should be between 10 and 20 Inline Constraint
#2 ; end
endmodule
```

# Result

```
# num=19
# num=15
# num=11
# num=13
# num=15
# num=14
# num=16
# num=15
# num=17
# num=15
# num=11
# num=15
```

# Randomize Object Properties

- In SV properties (variables) inside a class can also be randomized.
- Variables declared with rand and randc are only considered for randomization.
- A class built-in randomize function is used to randomized rand and randc variables.
- User can also specify constraint blocks to constrain random value generation.

# rand vs randc

- Variables defined with rand keyword, distribute values uniformly.

```
rand bit [1:0] num1;
num1:  3, 2 , 0, 3, 0, 1, 2, 1, 3
```

- Variables defined with randc keyword, distribute values in a cyclic fashion without any repetition within an iteration.

```
randc bit [1:0] num2;
num2:   3, 2, 0, 1
        0, 2, 1, 3
        1, 3, 0, 2
```

**rand & randc can not be used inside module

# Example1

```
class sample;
rand int num1;
int num2;
endclass

program test;
sample sm;
initial begin
sm=new;
repeat(20)
assert(sm.randomize())      //assert checks randomization status
$display("num1=%0d  num2=%0d", sm.num1, sm.num2);
end
endprogram          //num1 is randomized num2 remains untouched
```

# Result

```
# num1=-1884196597  num2=0
# num1=-326718039   num2=0
# num1=1452745934   num2=0
# num1=-2130312236  num2=0
# num1=1572468983   num2=0
# num1=131041957    num2=0
# num1=1115460554   num2=0
# num1=-818992270   num2=0
# num1=2000525113   num2=0
# num1=1547354947   num2=0
# num1=1196942489   num2=0
# num1=736230661    num2=0
```

# Example2

```
class sample;
rand bit[1:0] num;
endclass

class main;
rand sample sm; //rand is must to
                //randomize num
function new;
sm=new;
endfunction
endclass
```

```
program test;
main m;
initial begin
m=new;
repeat(20)
assert(m.randomize())
$display(m.sm.num);
end
endprogram
```

# Example3

```
class sample;
typedef struct { randc int a;
                 bit [3:0] b;
               } st_t;
rand st_t st;
//rand is must to randomize
//int present inside structure
endclass
```

```
program test;
sample sm;
initial begin
sm=new;
repeat(20)
assert(sm.randomize())
$display(sm.st.a);
end
endprogram
```

# Example4

```
class sample;
rand bit[3:0] num;
endclass

class main;
rand sample sm1;
sample sm2;
function new;
sm1=new;  sm2=new;
endfunction
endclass
```

```
program test;
main m;
initial begin
m=new;
repeat(20) begin
assert(m.randomize()) ;
$display(m.sm1.num);
$display(m.sm2.num);
end
end
endprogram
```

# Result

# 14 # 0
# 4 # 0
# 9 # 0
# 6 # 0
# 5 # 0
# 15 # 0
# 4 # 0
# 13 # 0
# 1 # 0
# 8 # 0
# 9 # 0
# 14 # 0

# Specifying Constraints

```
class sample1;
rand int num;
constraint c { num>10;  num<100;}
endclass

class sample2;
randc bit [7:0] num;
constraint c1 { num>10; }
constraint c2 { num<100; }
endclass
```

```
class sample3;
randc int num;
int Max, Min;
constraint c1 { num>Min; }
constraint c2 { num<Max; }
endclass
```

# Example1

```
class packet;
rand bit [7:0] data;
int Max=50, Min=10;
constraint c1 { data>Min;data<Max; }
endclass

program test;
packet pkt;
initial begin
pkt=new;
repeat(10)
assert(pkt.randomize())
$display(pkt.data);
pkt.Min=30;
pkt.Max=100;
repeat(10)
assert(pkt.randomize())
$display(pkt.data);
end
endprogram
```

# Result

First randomization

# 22
# 22
# 29
# 27
# 46
# 43
# 33
# 43
# 46
# 36

Second randomization

# 72
# 53
# 66
# 79
# 68
# 69
# 78
# 95
# 65
# 34

# Example2

```
class packet;
rand bit [7:0] data;
constraint c2 { data>50;
                data<10; }
endclass
```

```
program test;
packet pkt;
initial begin
pkt=new;
repeat(10)
if(pkt.randomize())
$display("Randomization Success");
else
$display("Randomization Fails"); end
endprogram
```

# Result

# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails
# Randomization Fails

# pre_randomize and post_randomize

- Every class contains pre_randomize and post_randomize functions which are evoked every time randomize function is called.
- When randomize function is called, it first evokes pre_randomize and then randomization is done.
- post_randomize function is only called if randomization was successful.
- pre_randomize and post_randomize functions can be written in a class to offer user defined functionality before and after randomization.

# Example1

```
class packet;
rand bit [7:0] data;

function void pre_randomize;
$display(“Pre-Randomize”);
endfunction

function void post_randomize;
$display(“Post-Randomize”);
endfunction

endclass
```

```
program test;
packet pkt;

initial begin
pkt=new;
repeat(5) begin
void'(pkt.randomize);
$display(pkt.data);
end
end
endprogram
```

# Result

```
# Pre-Randomize
# Post-Randomize   #  33
# Pre-Randomize
# Post-Randomize   #  25
# Pre-Randomize
# Post-Randomize   # 202
# Pre-Randomize
# Post-Randomize  # 138
# Pre-Randomize
# Post-Randomize   #  15
```

# Example2

```
class A;

function void pre_randomize;
$display("A: Pre-Randomize");
endfunction

function void post_randomize;
$display("A: Post-Randomize");
endfunction

endclass
```

```
class B extends A;

function void pre_randomize;
$display("B: Pre-Randomize");
endfunction

function void post_randomize;
$display("B: Post-Randomize");
endfunction

endclass
```

# Example2

```
program test;
B b1;

initial begin
b1=new;
repeat(2)
void'(b1.randomize);
end
endprogram
```

Result

```
# B: Pre-Randomize
# B: Post-Randomize
# B: Pre-Randomize
# B: Post-Randomize
```

Pre-Randomize and Post-Randomize of parent class are overridden

# Controlling Randomization

- Randomization nature of rand and randc variables can be turned on/off dynamically.
- rand_mode method is used to change randomization status of rand and randc variable.
- When used as a task, the argument determines the state of rand and randc variables.
- When argument is 0 then randomization is disabled(turned-off), when argument is 1 then randomization is enabled(turned-on).

# Controlling Randomization

- When used as a function, rand_mode returns the current status of rand and randc variables.
- It returns 1 if randomization is on else it returns 0.
- Hierarchal reference of variables in an object can also be given to disable/enable specific rand and randc variables.
- Randomization is enabled by default.

# Example2

```
class packet;
rand bit [7:0] data1;
rand int data2;
endclass

program test;
packet pkt;
initial begin
pkt=new;
repeat(10) begin
void'(pkt.randomize);
if(pkt.rand_mode()) //Check current Status
$display("Randomization on");
else $display("Randomization off");
end
pkt.rand_mode(0);
void'(pkt.randomize);
if(pkt.rand_mode())
$display("Randomization on");
else $display("Randomization off");  end
endprogram
```

# Example1

```
class packet;
rand bit [7:0] data;

endclass


program test;
packet pkt;

initial begin
pkt=new;
repeat(4) begin
void'(pkt.randomize);
$display(pkt.data); end
pkt.rand_mode(0);
//Disabling Randomization
repeat(3) begin
void'(pkt.randomize)
$display(pkt.data);
end end
endprogram
```

# Result

```
#  33
#  25
# 202
# 138
# 138
# 138
# 138
```

# Example3

```
class packet;
rand bit [7:0] data1;
rand byte data2;

endclass


program test;
packet pkt;

initial begin
pkt=new;
repeat(10)  if(pkt.randomize)
$display(pkt.data1, pkt.data2);
pkt.data2.rand_mode(0);
//turn off for data2
repeat(10)  if(pkt.randomize)
$display(pkt.data1, pkt.data2);
pkt.data2.rand_mode(1);
repeat(10)  if(pkt.randomize)
$display(pkt.data1, pkt.data2); end
endprogram
```

# Result

# 238  94
#  85  48
# 202  -92
#  29  38
# 155  48
# 225  -91
#  81  -66
# 232  -82
#  85  -112
# 141  -34
# 244  -34
#  32  -34
#   9  -34

# Example4

```
class packet;
rand bit [7:0] data1;
byte data2;

endclass

program test;
packet pkt;

initial begin
pkt=new;
repeat(6)
if(pkt.randomize)
$display(pkt.data1, pkt.data2);
repeat(4)
if(pkt.randomize(data2))
//will only randomize data2
$display(pkt.data1, pkt.data2);
end
endprogram
```

# Result

```
# 238    0
#  85    0
# 202    0
#  29    0
# 155    0
# 225    0
# 225   75
# 225  115
# 225  -24
# 225  111
# 225 -119
```

# Example5

```
class packet;
rand int data;
int Max, Min;
constraint c1{ data> Min; data<Max; }
constraint c2 { Max> Min; }
task set(int Min, Max);
this.Min=Min;
this.Max=Max;
endtask
endclass
```

# Example5

```
initial begin
packet p1=new;
p1.set(5, 25);
repeat(5)  if(p1.randomize)
$display("Random value=%0d", p1.data);
p1.set(35, 20);
repeat(5) if(p1.randomize)
$display("Random value=%0d", p1.data);
else $display("Randomization Failed");
end
```

# Result

```
# Random value=14
# Random value=18
# Random value=15
# Random value=16
# Random value=16
# Randomization Failed
# Randomization Failed
# Randomization Failed
# Randomization Failed
# Randomization Failed
```

# Random Stability

```
module test;
class A;
rand bit [3:0] data;
endclass
A a1, a2;

initial begin
a1=new;
//Random seed initialized
a2=new;
//Random seed initialized with next seed value
repeat(5)
if(a1.randomize)
$display("a1.data=%0d",a1.data);
repeat(5)
if(a2.randomize)
$display("a2.data=%0d",a2.data);
end
endmodule
```

# Result

# a1.data=12
# a1.data=7
# a1.data=15
# a1.data=6
# a1.data=9
# a2.data=13
# a2.data=13
# a2.data=6
# a2.data=2
# a2.data=15

# Random Stability

```
module test;
class A;
rand bit [3:0] data;
endclass
A a1, a2;

initial begin
a1=new;
//Random seed initialized
a2=new;
//Random seed initialized with next seed value
repeat(5)
if(a2.randomize)
$display("a2.data=%0d",a2.data);
repeat(5)
if(a1.randomize)
$display("a1.data=%0d",a1.data);
end
endmodule
```

# Result

# a2.data=13
# a2.data=13
# a2.data=6
# a2.data=2
# a2.data=15
# a1.data=12
# a1.data=7
# a1.data=15
# a1.data=6
# a1.data=9

# Random Stability

```
module test;
class A;
rand bit [3:0] data;
function new(int seed);
srandom(seed);
//set a particular seed
endfunction
endclass

A a1, a2;

initial begin
a1=new(3);  a2=new(3);
repeat(5)
if(a1.randomize)
$display("a1.data=%0d",a1.data);
repeat(5)
if(a2.randomize)
$display("a2.data=%0d",a2.data);
end
endmodule
```

# Result

# a1.data=5
# a1.data=7
# a1.data=12
# a1.data=13
# a1.data=5
# a2.data=5
# a2.data=7
# a2.data=12
# a2.data=13
# a2.data=5

# Relation in Constraints

- Each constraint expression should only contain 1 relation operator.

< <= == > >= -> <-> || ! &&

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

```
low=20,   med=224,  hi=164
low=114, med=39,    hi=189
low=186, med=148,  hi=161
low=214, med=223,  hi=201
```

- low < med is evaluated. Results in 0 or 1
- hi > (0 or 1) is evaluated.

# Relation in Constraints

```
constraint good{ low < med;
                 med < hi; }
```

low=20, med=40, hi=100
low=10, med=25, hi=90

- User can use == to constraint random value to a particular expression. Using = will give compilation error.

```
class packet;
rand int length, data, address;
constraint len { length==address * 5};
endclass
```

# Set Membership

- User can use inside operator to set membership in a constraint block.
- Example: To limit address in range from 1 to 5, 7 to 11 and to a set of values 15, 18, 25.

```
class packet;
rand int address;
constraint limit {address inside { [1:5],  [7:11], 15, 18, 25 };}
endclass
```

# Set Membership

- A ! Operator can be used to exclude set of values

```
class packet;
rand int address;
constraint limit { !(address inside { 6, [12:14]} ) ;}
endclass
```

- Using arrays to set membership.

```
class packet;
int arr [ ]= `{ 5, 7, 11, 13, 19};
rand int address;
constraint limit { address inside { arr }; }
endclass
```

# Set Membership

```
class packet;
rand int data;
constraint limit { ( (data==5) || (data==7) || (data==9) );}
endclass
```

There is a better way of providing such constraints:

```
class packet;
rand int data;
constraint limit { data inside { 5, 7, 9}; }
endclass
```

# Weighted Distribution

- User can provide weights for random numbers to obtain non-uniform distribution.
- := operator is used to assign same weight to all the values.
- :/ operator is used to distribute weight among all the values.
- dist operator is used to specify distribution.
- Weighted distribution does not work on randc variables.
- Example: constraint con { src dist { 0:=40, [1:3] :=60 };
  dst dist { 0:/40 , [1:3] :/60 }; }

# Example1

```
class packet;
rand int data;
constraint con { data dist { 0:=40, [1:4] :=60, [6:7]:=20 }; }
endclass
//Total weight= 40 + 60 + 60 + 60 + 60 + 20 + 20=320
```

data=0 weight=40/320=12.5%

data=1 weight=60/320=18.75%

data=2 weight=60/320=18.75%

data=3 weight=60/320=18.75%

data=4 weight=60/320=18.75%

data=6 weight=20/320=6.25%

data=7 weight=20/320=6.25%

# Example2

```
class packet;
rand int data;
constraint con { data dist { 0:/20, [1:3] :/60, [6:7]:/20 }; }
endclass
//Total weight= 20 + 60 + 20=100
```

data=0 weight=20/100=20%

data=1 weight=20/100=20%

data=2 weight=20/100=20%

data=3 weight=20/100=20%

data=6 weight=10/100=10%

data=7 weight=10/100=10%

# Example3

```
typedef enum {Red, Green, Blue} color;
class temp;
rand color col;
int redw=5, greenw=3, bluew=4;
constraint weight { col dist { Red:=redw,
                               Green:=greenw,
                               Blue:=bluew}; }
endclass
```

# Implication Constraints

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

Is equivalent to

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

- If mode is small that implies length should be less than 10.
- If mode is large that implies length should be more than 100.
- Implication helps in creating case like blocks.

# Efficient Constraints

```
rand bit [31:0] addr;
constraint slow { addr % 4096 inside { [0:20], [4075:4095] };}
```

```
rand bit [31:0] addr;
constraint fast { addr [11:0] inside { [0:20], [4075:4095] };}
```

- In slow, first addr is evaluated and then % is performed and then constraints are applied.
- In fast, constraints are directly applied on selected bits hence faster and achieves the same result.

# Bidirectional Constraints

- Constraints are not procedural but declarative.
- All constraints should be active at same time.

```
rand bit [15:0] a, b, c;
constraint cp { a < c;
                b == a;
                c < 10;
                b > 5;
               }
```

| Solution | a | b | c |
|---|---|---|---|
| S1 | 6 | 6 | 7 |
| S2 | 6 | 6 | 8 |
| S3 | 6 | 6 | 9 |
| S4 | 7 | 7 | 8 |
| S5 | 7 | 7 | 9 |
| S6 | 8 | 8 | 9 |

- Even though there is no direct constraint on lower value of c, constraint on b restricts choices.

# Solution Probabilities

```
class Unconstrained;
rand bit x;
// 0 or 1
rand bit [1:0] y;
// 0, 1, 2, or 3
endclass
```

| Solution | x | y | Probability |
|---|---|---|---|
| S1 | 0 | 0 | 1/8 |
| S2 | 0 | 1 | 1/8 |
| S3 | 0 | 2 | 1/8 |
| S4 | 0 | 3 | 1/8 |
| S5 | 1 | 0 | 1/8 |
| S6 | 1 | 1 | 1/8 |
| S7 | 1 | 2 | 1/8 |
| S8 | 1 | 3 | 1/8 |

# Solution Probabilities

```
class Implication1;
rand bit x;
// 0 or 1
rand bit [1:0] y;
// 0, 1, 2, or 3
constraint c {
(x==0) -> (y==0); }
endclass
```

| Solution | x | y | Probability |
|---|---|---|---|
| S1 | 0 | 0 | 1/2 |
| S2 | 0 | 1 | 0 |
| S3 | 0 | 2 | 0 |
| S4 | 0 | 3 | 0 |
| S5 | 1 | 0 | 1/8 |
| S6 | 1 | 1 | 1/8 |
| S7 | 1 | 2 | 1/8 |
| S8 | 1 | 3 | 1/8 |

# Solve before

- A solve before keyword can be used to specify order in which random variables would be solved.

```
class solvebefore;
rand bit x;
// 0 or 1
rand bit [1:0] y;
// 0, 1, 2, or 3
constraint c {
(x==0) -> (y==0);
solve x before y; }
endclass
```

| Solution | x | y | Probability |
|---|---|---|---|
| S1 | 0 | 0 | 1/2 |
| S2 | 0 | 1 | 0 |
| S3 | 0 | 2 | 0 |
| S4 | 0 | 3 | 0 |
| S5 | 1 | 0 | 1/8 |
| S6 | 1 | 1 | 1/8 |
| S7 | 1 | 2 | 1/8 |
| S8 | 1 | 3 | 1/8 |

# Solution Probabilities

```
class Implication2;
rand bit x;
// 0 or 1
rand bit [1:0] y;
// 0, 1, 2, or 3
constraint c {
y>0;
(x==0) -> (y==0); }
endclass
```

| Solution | x | y | Probability |
|---|---|---|---|
| S1 | 0 | 0 | 0 |
| S2 | 0 | 1 | 0 |
| S3 | 0 | 2 | 0 |
| S4 | 0 | 3 | 0 |
| S5 | 1 | 0 | 0 |
| S6 | 1 | 1 | 1/3 |
| S7 | 1 | 2 | 1/3 |
| S8 | 1 | 3 | 1/3 |

# Solve before

```
class solvebefore;
rand bit x;
// 0 or 1
rand bit [1:0] y;
// 0, 1, 2, or 3
constraint c {
(x==0) -> (y==0);
solve y before x; }
endclass
```

| Solution | x | y | Probability |
|---|---|---|---|
| S1 | 0 | 0 | 1/8 |
| S2 | 0 | 1 | 0 |
| S3 | 0 | 2 | 0 |
| S4 | 0 | 3 | 0 |
| S5 | 1 | 0 | 1/8 |
| S6 | 1 | 1 | 1/4 |
| S7 | 1 | 2 | 1/4 |
| S8 | 1 | 3 | 1/4 |

****randc  variable cannot be used for solve_before construct**

****randc is always evaluated first compare to rand**

**ASSIGNMENT:: randomization**

1. Declare a class of eth_pkt (Ethernet packet) with following fields
a. Count, da (destination address), len (length), payload, crc
b. Declare count as static
c. Declare da, len, payload as random
d. CRC is not random (why?)
e. Payload declared either as dynamic array or Queue (why?)

2. Instantiate eth_pkt inside module top
a. Write $display to display contents of eth pkt
b. List down disadvantages of $display to print pkt

- **3. Randomize pkt**
- **a. Try without assert**
- **b. Try with assert**
- **c. Try with if condition to confirm if randomize is passing or failing**
- **d. You should notice of above 3 and find out why assert is useful**

- **4. Declare a method print to display the contents of eth_pkt**
- **a. Notice in both prints above payload will print 0 elements**
  - **b. How to solve above problem?**
  - **i. Declare a constraint in eth_pkt as below**
  - **1. Constraint payload_c { payload.size() == 10; }**

  - **ii. Now print the pkt contents**
  - **1. Before you print we should randomize pkt again**
  - **2. Now we should see payload printing with 10 elements**