

# DESIGN OF DIGITAL CIRCUITS AND SYSTEMS

## Testing: Randomization

**Instructor:** Justin Hsia

**Teaching Assistants:**

Colton Carroll

Hemil Patel

Rasya Fawwaz

Grace Zhou

Quinlyn Donohue

Rose Maresh

# Relevant Course Information

- ❖ Quiz 4 (Algorithms to Hardware) starts at 11:40 am
- ❖ Quiz 5 (STA, Pipelining, CDC) is *next* Thursday
  
- ❖ Lab 6 proposal due next week (5/27)
  - (1) Description of major project features
  - (2) Top-level block diagram
  - (3) Images/sketches of VGA output
  - “Proposal Workshop” in lecture on 5/26
  
- ❖ Homework 6 (Advanced Testing) released, due on 6/1

# Assertions Review Questions (1/3)

## ❖ Reminders:

- $| \rightarrow$  is overlapped implication
- $| \Rightarrow$  is non-overlapped implication
- $\#\#N$  delays next (RHS) sequence by  $N$  cycles
- $[*N]$  means  $N$  consecutive repetitions of the LHS
- $[=N]$  means  $N$  non-consecutive repetitions of the LHS
  - Any  $N$  above can be replaced by the inclusive range  $A : B$

## ❖ Write out the concurrent assertion that uses:

- Start  $| \Rightarrow$  a  $\#\#1$  b  $[*1:3]$   $\#\#1$  c

# Assertions Review Questions (2/3)

## ❖ Reminders:

- $| \rightarrow$  is overlapped implication
- $| \Rightarrow$  is non-overlapped implication
- $\#\#N$  delays next (RHS) sequence by  $N$  cycles
- $[*N]$  means  $N$  consecutive repetitions of the LHS
- $[=N]$  means  $N$  non-consecutive repetitions of the LHS
  - Any  $N$  above can be replaced by the inclusive range  $A : B$

## ❖ What does this property check for?

- Start  $| \Rightarrow$  a  $\#\#1$  b  $[*1:3]$   $\#\#1$  c

# Assertions Review Questions (3/3)

## ❖ Reminders:

- $| \rightarrow$  is overlapped implication
- $| \Rightarrow$  is non-overlapped implication
- $##N$  delays next (RHS) sequence by  $N$  cycles
- $[*N]$  means  $N$  consecutive repetitions of the LHS
- $[=N]$  means  $N$  non-consecutive repetitions of the LHS
  - Any  $N$  above can be replaced by the inclusive range  $A : B$

## ❖ Test the concurrent assertion yourself!

- Start  $| \Rightarrow a ##1 b [*1:3] ##1 c$
- <https://www.edaplayground.com/x/QUpq>

# Blocks Revisited

- ❖ So far, we have gotten away with using `module` for everything
  - A module is intended to describe hardware
- ❖ A `program` block provides an entry point to the execution of testbenches
  - Similar definition and instantiation syntax to a `module`, but cannot contain an `always` block
  - Can be defined within a module
  - Not strictly necessary – addresses some minor data race interactions between the dut and testbench

# Why Randomize?

## ❖ Directed testing

- Only checks for anticipated bugs, so you only find bugs that you think might be there to begin with
- Scales poorly as requirements increase and change
- Relatively little upfront work – manually or automatically checker

## ❖ Random Testing

- Can check for unanticipated bugs
- Scales relatively well as requirements increase and change
- More upfront work – create randomization environment and model/scoreboard to compute expected values

# What to Randomize?

- ❖ Different goals of testing
  - **Correctness:** Does the system do the “right” thing on expected states & inputs
  - **Robustness:** Does the system do something “reasonable” on unexpected states & inputs
  
- ❖ Much more should be randomized than your input:
  - Inputs: primary input data, encapsulated input data, delays, test order
  - Configurations: device, environment configuration
  - Erroneous state: protocol exceptions, errors, violations
  - Seeding: random test seed

# Random Testing in SystemVerilog

- ❖ We can create **Constrained Random Tests (CRTs)**
  - Test code uses a stream of constrained random values to create input to the DUT
    - Random variables and constraints must be defined within a class
  - The inputs and behavior will change based on the seed of its **pseudo-random number generator (PRNG)**
- ❖ **Random stability**
  - Random number generation needs to be reproducible, otherwise testing failures are not easily reproducible
  - [extra] *Thread locality* means each thread has independent PRNGs; *hierarchical seeding* means that different parts of your code within the thread inherit seeding

# Random Number Functions

- ❖ Functions that return a random number from within a specified distribution, *e.g.*,
  - `$random` – flat distribution of *signed* 32-bit numbers
  - `$urandom` – flat distribution of *unsigned* 32-bit numbers
  - `$urandom_range` – flat distribution of specified range
  - `$dist_normal` – bell-shaped distribution
- ❖ Not particularly useful by themselves
  - Pseudo-randomness between subsequent calls but stability of each call for same seed
  - More efficient to use directed testing on known edge cases or more effective to test all input combinations
    - *Unconstrained* functions

# Changing Seed

- ❖ Method `srandom()` sets the random seed for a particular part of the hierarchy

- ```
// change seed on object
MyClass obj;
obj = new();
obj.srandom(<number>);
```

- ```
// change seed on thread
process pt;
pt = process::self();
pt.srandom(<number>);
```

- ❖ Warnings [extras]
  - This gets more complicated the more simultaneous PRNGs you have to deal with
  - Careful with multiple calls to `srandom` on same component
  - Ordering of random function calls and randomization matters

SHORT TECH

BREAK

# Constraints

- ❖ Unconstrained randomization
  - Search/sample space quickly becomes unwieldy
  - Good portion of search/sample space may be completely nonsensical
- ❖ Can introduce conditions/**constraints** for random variables within a class object
  - Restricts the search/sample space
  - A way to express the relationships between variables
  - Your simulator's **constraint solver** will attempt to solve all of the given constraints *simultaneously*
    - This can fail and its behavior is implementation-dependent

# Random Variables

- ❖ Keywords `rand`/`randc` make *randomizable* variable
  - Can only apply to integral datatypes (includes `enums`)
  - `rand` variable values distributed *uniformly*
  - `randc` variable values distributed *cyclically*
  - *e.g.*,

```
rand bit [1:0] x;  
randc bit [1:0] y;
```
- ❖ Call `randomize` method to assign random values
  - Can be called multiple times
  - Can also pass random variable names as arguments to `randomize` only a subset
  - *e.g.*,

```
MyClass obj = new();  
obj.randomize(); // equiv: obj.randomize(x,y);
```

# Defining Constraints

- ❖ Constraints named and specified with the `constraint` keyword and curly braces
  - Each **constraint expression** (separated by semi-colons) should contain at least one random variable and generally one comparison operator (*e.g.*, `<`, `<=`, `==`, `>=`, `>`)
  - Can have multiple expressions within a constraint and multiple constraints within a class, *e.g.*,

```
class Child;  
    rand bit [7:0] age;  
    constraint c_teen {age > 12; age < 20;}  
endclass
```

- ❖ If all constraints cannot be met simultaneously, `randomize` will return 0  
– **always check for this!**

# Bidirectional Constraints

- ❖ All constraint expressions are active *simultaneously*
- ❖ Exercise: what are all possible outcomes from `randomize()`?

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

Valid solutions:

r	s	t

# Example Constrained Testbench

```
program testbench;

  class Packet;
    rand bit [31:0] src, dst, data[8];
    randc bit [7:0] kind;
    constraint c_src {
      src > 10;
      src < 15;
    }
    constraint c_dst {dst < 32;}
  endclass

  Packet p;
  initial begin
    p = new();
    if (!p.randomize())
      $finish;
  end

endprogram
```

# Constraint Exercise #1

- ❖ Write out a SystemVerilog program that:
  - Defines a class called MemRead that contains an 8-bit random variable data and a 4-bit random variable addr
  - Constrain addr to 3, 4, or 5
  - Construct a MemRead object and randomize it, making sure to check if the randomization succeeded