6. RANDOMIZATION

6.1 Constrained-Random Tests (CRT)

1. **Challenges**:
    - Hard to create complete stimuli for large designs.
    - Directed tests (manual, feature-specific test cases) can't cover all feature interactions.
    - Feature doubling increases complexity; bugs from feature interactions are missed.
2. **Limitations of Directed Tests**:
    - Only find anticipated bugs.
    - Impractical for all feature combinations.
3. **CRT Solution**:
    - Automatically generates random test cases.
    - Uses constraints to ensure tests are valid and relevant.
4. **Advantages**:
    - Wider coverage and exploration of scenarios.
    - Finds unanticipated bugs.
    - Reduces manual effort, increasing efficiency.

6.1.1 CRT Environment Setup

1. **Effort Comparison**:
    - CRT setup requires more effort than directed tests.
    - Directed test: Apply stimulus and manually check results using a golden log file(output file).
2. **CRT Requirements**:
    - Environment to predict results using a reference model, transfer function, or similar techniques.
    - Functional coverage to measure stimulus effectiveness.
3. **Advantages of CRT**:
    - Once set up, hundreds of tests can run automatically without manual result-checking.
    - Improves productivity by trading test-authoring time (manual effort) for CPU time (automated processing).

6.1.2 Components of CRT

1. **Parts of CRT**:
    - **Test Code**: Generates random input values for the DUT.
    - **Seed**: A starting value for the pseudo-random number generator (PRNG) that ensures the same sequence of random values can be reproduced for debugging or rerunning tests.

6.2 What to Randomize in CRT

1. **Focus of Randomization**:
    - Randomizing **data fields** (e.g., using `$random`) mainly uncovers data-path issues, but has limited effectiveness for complex bugs.
    - The more critical bugs are often in the **control logic**, so randomizing these areas is essential.
2. **Key Areas to Randomize**:
    - **Device Configuration**: Hardware settings and operational modes.
    - **Environment Configuration**: External conditions affecting the DUT.
    - **Primary Input Data**: Main data provided to the DUT.
    - **Encapsulated Input Data**: Data wrapped in structures or packages.
    - **Protocol Exceptions**: Edge cases in protocol behavior.
    - **Delays**: Timing variations between operations or signals.
    - **Transaction Status**: Progress or state of ongoing transactions.
    - **Errors and Violations**: Faults, incorrect operations, or rule violations.
3. **Goal of Randomization**:
    - **Increase path coverage**: By randomizing **decision points** (where control paths diverge), you ensure that the DUT explores a wide variety of execution paths, increasing the chances of finding hidden bugs.

6.2.1 Device Configuration in CRT

1. **Common Issue in RTL Testing**:
    - Bugs are often missed because not enough different configurations are tested.
    - Many tests use a fixed configuration (e.g., design in reset or with a fixed initialization vector).
2. **Real-World Example**:
    - A **multiplexor switch** with 600 input channels and 12 output channels was tested with a fixed configuration.
    - In real-world use, channels are allocated and deallocated randomly, making the configuration appear random.
3. **Traditional Testing Limitation**:
    - The engineer had to manually configure each channel using Tcl code, testing only a few configurations.
4. **CRT Approach**:
    - By randomizing the configuration of each channel and using a loop to configure the entire device, a broader range of configurations was tested.
    - This approach ensured that previously missed bugs were uncovered by simulating more realistic, varied configurations.

6.2.2 Environment Configuration in CRT

1. **Randomizing the Environment**:
    - The DUT operates in a system with other devices. Randomizing the environment (number of devices, their configuration, etc.) is essential for realistic testing.
2. **Example**:
    - **PCI Switch**: Randomized the number of PCI buses (1-4), number of devices on each bus (1-8), and device parameters (master/slave, CSR addresses).
    - This ensured that all possible configurations were tested, increasing coverage and bug detection.
3. **Goal**:
    - Randomizing the environment helps simulate real-world conditions and uncover bugs that may arise from interactions with other devices in the system.

6.2.3 Primary Input Data in CRT

1. **Randomizing Primary Input Data**:
    - The first thought is often to randomize transactions like bus writes or ATM cells with random values.
    - This is straightforward if the transaction classes are well-prepared in advance.
2. **Considerations**:
    - Anticipate **layered protocols** (e.g., communication protocols) to ensure valid input.
    - Include **error injection** to simulate real-world faults and edge cases.
3. **Goal**:
    - Ensure the random input data is valid, realistic, and comprehensive to test a wide range of scenarios.

6.2.4 Encapsulated Input Data in CRT

1. **Multiple Layers of Data**:
    - Devices often handle data in layers, like TCP inside IP packets, which are then inside Ethernet frames.
2. **Randomizing Control Fields**:
    - Each layer has its own settings (control fields) that can be randomized to test different combinations.
3. **Creating Valid Control Fields**:
    - You need to set rules (constraints) to make sure the control fields are correct, but also allow for errors to test how the system handles them.
4. **Goal**:
    - Randomize data and control fields across all layers, while ensuring the system is tested for both valid and faulty conditions.

6.2.5 Protocol Exceptions, Errors, and Violations in CRT

1. **Handling Errors**:
    - Errors will eventually occur, so the design must handle them without crashing or entering an illegal state.

    - It's important to test the system for all potential error cases, even those beyond the functional specification.
2. **Error Scenarios**:
    - What happens if a transfer is interrupted midway? The testbench should simulate such situations.
    - Test error detection and correction fields by trying all possible combinations.
3. **Random Error Injection**:
    - The testbench should be able to send correct inputs and, with a simple change, inject random errors at random points during the test.
4. **Goal**:
    - Ensure the design can handle all types of errors and exceptions gracefully, simulating real-world issues like interruptions or faulty transfers.

6.2.6 Delays in CRT

1. **Randomizing Delays**:
    - Communication protocols specify delays (e.g., bus grant after 1-3 cycles, memory data valid after 4-10 cycles).
    - Directed tests often use the shortest delays for faster simulation, missing potential bugs.
    - The testbench should use **random legal delays** in every test to uncover bugs that may only appear with specific timing.
2. **Clock Jitter**:
    - Some designs are sensitive to **clock jitter** (small variations in the clock cycle).
    - By slightly shifting the clock edges, you can test if the design handles small timing changes correctly.
3. **Clock Generator**:
    - The clock generator should be a separate module outside the testbench, creating events like other design events.
    - It should have configurable parameters (e.g., frequency, offset) that the testbench can adjust during setup.
4. **Focus on Functional Errors**:
    - This methodology focuses on **functional errors**, not **timing errors**.
    - Timing errors like setup and hold violations should be checked using **timing analysis tools**, not through random delays in the testbench.

6.3 Randomization in SystemVerilog

1. **Randomization with OOP**:
    - SystemVerilog's random stimulus generation is most effective when used with **Object-Oriented Programming (OOP)**.
    - You create a **class** to hold related random variables, and the **random-solver** fills these variables with random values.
2. **Constraints**:
    - **Constraints** can be applied to ensure the random values are **legal** or to test specific features.
3. **Transaction-Level Randomization**:

- Randomizing individual variables one at a time is less useful.
- True **constrained-random stimuli** is created at the **transaction level**, where multiple related values are randomized together to form a complete test case.

6.3.1 Sample Class with Random Variables (Sample 6.1)

1. **Class Overview**:
   - The `Packet` class contains four random variables:
     - `src`, `dst`, and `data` use the `rand` modifier (values may repeat).
     - `kind` uses the `randc` modifier (random cyclic, ensures all possible values are used before repeating).
2. **Constraints**:
   - Constraints are rules for randomization.
   - Example: `src` must be between 10 and 15 (`src > 10; src < 15;`).
3. **Randomization Behavior**:
   - `rand`: Generates new values for every randomization (like rolling dice).
   - `randc`: Ensures all possible values are assigned before repeating (like shuffling a deck of cards).
4. **Randomization Process**:
   - Use the `randomize()` function to generate random values.
   - It returns `0` if constraints cannot be satisfied, stopping the simulation with `$finish`.
5. **Why Not Randomize in Constructor?**
   - The constructor initializes variables but does not randomize them.
   - Randomization may need additional constraints or changes later in the testbench.
6. **Best Practices**:
   - Keep random variables **public** for flexibility in tests.
   - Avoid randomizing configuration variables like weights or limits, as they are fixed at simulation start.
   - Use constraints wisely to control random values and ensure valid scenarios.
7. **Key Notes on `randc`**:
   - Simulators must support up to 8-bit `randc` variables (256 values), though many handle larger ranges.
   - A `randc` array creates independent patterns for each element, like separate shuffled decks.
8. **Why Use Constraints?**
   - Constraints ensure valid and targeted randomization for meaningful test scenarios.

Example Use Case:

The `Packet` class allows controlled randomization of test data (e.g., packet fields) while ensuring constraints are respected, making it ideal for creating diverse and valid stimuli for testing.

```
// Sample 6.1  Simple random class

class Packet;
        // The random variables
rand bit [31:0] src;
rand bit [31:0] dst;
rand bit [31:0] data[8];
randc bit [7:0] kind;

        // Limit the values for src

constraint c { src > 10;
               src < 15;}

endclass : Packet
module sample_6_1;
        Packet p;
        initial
        begin
        p = new();              // Create a packet
        if (!p.randomize())
                $finish;
        transmit (p);
        end
endmodule : sample_6_1
```

6.3.2 Randomization and Error Checking in SystemVerilog

1. Purpose of Randomization Check

- **Randomization**: In SystemVerilog, the `randomize()` function is used to assign random values to variables labeled with `rand` or `randc`. This function also ensures that all constraints applied to these variables are respected.
- **Error Checking**: It's important to check if the randomization was successful. If there are conflicting constraints or any issues during randomization, the randomization process can fail, leading to unexpected values in the variables, which can cause simulation failures.

2. Macro for Randomization Check: `SV_RAND_CHECK`

- **Definition**: The macro `SV_RAND_CHECK(r)` is used to verify the success of randomization. It checks the result of the `randomize()` function, and if it fails, an error message is displayed and the simulation is stopped.
- **Macro Structure**:
    - `do begin`: Starts a block that will execute the check.

- `if ((!r))`: This checks if the result of `randomize()` (denoted as `r`) is false. If the randomization failed, it enters the block.
- `$display`: Displays an error message, including the file name (`__FILE__`), line number (`__LINE__`), and the name of the randomization (`"r"`).
- `$finish`: Stops the simulation if randomization fails.
- `end` **and** `end while(0)`: The `do...while(0)` loop ensures that the macro can be used like a normal statement with a semicolon.

3. Packet Class with Random Variables

- **Class Definition**:
  - The `Packet` class contains four random variables:
    - `src`, `dst`, `data[8]`: These are random variables, meaning they will be assigned random values when `randomize()` is called.
    - `kind`: This is a cyclic random variable (`randc`), which ensures that each value is used exactly once before repeating.
- **Constraints**:
  - The `src` variable has constraints applied to it, limiting its value to between 10 and 15. This ensures that only values within this range are selected during randomization.

4. Testbench Module: `sample_6_2`

- **Module Purpose**:
  - The module creates an instance of the `Packet` class, randomizes its values, and checks if the randomization was successful using the `SV_RAND_CHECK` macro.
- **Process**:
  - `p = new();`: Creates a new instance of the `Packet` class.
  - `p.randomize();`: Randomizes the packet's variables, assigning random values to them.
  - `SV_RAND_CHECK(p.randomize());`: Calls the `SV_RAND_CHECK` macro to verify if the randomization was successful. If it fails, the simulation is stopped.

5. Key Concepts

- **Randomization**:
  - In SystemVerilog, `rand` and `randc` are used to define random variables. `rand` generates random values each time, while `randc` generates values cyclically, ensuring all possible values are used before repeating.
- **Constraints**:
  - Constraints restrict the possible values that a random variable can take. In this case, the `src` variable is constrained to values between 10 and 15.
- **Error Handling**:

- The `SV_RAND_CHECK` macro ensures that the randomization is successful. If not, it provides an error message with the file and line number and stops the simulation.

6. Importance of Checking Randomization

- Always check the result of randomization to ensure that the constraints are respected and that no conflicts or errors occurred during the process.
- Failure to check can lead to unexpected values in the variables, which could cause incorrect behavior or failure of the simulation.

//Sample 6.2 Randomization check macro and example

```
// Macro definition to check randomization result
`define SV_RAND_CHECK(r) \
        do begin \
        // If randomization fails (r is false), display error and stop simulation
        if((!r)) begin \
        // Display the file name, line number, and the failed randomization statement
        $display("%s: %0d: Randomization failed \"%s\"", \
                `__FILE__, `__LINE__, `"r`"); \
        // Stop the simulation
        $finish; \
        end \
        end while (0)  // End of do-while loop

// Class definition for Packet
class Packet;
        // Declare random variables for the packet
        rand bit [31:0] src;  // 32-bit source address
        rand bit [31:0] dst;  // 32-bit destination address
        rand bit [31:0] data[8];  // Array of 8 32-bit data elements
        randc bit [7:0] kind;  // 8-bit kind field, with cyclic randomization (randc)

        // Constraint to limit the values for the src field
        constraint c {
        src > 10;  // src must be greater than 10
        src < 15;  // src must be less than 15
        }
endclass : Packet

// Testbench module
module sample_6_2;
        Packet p;  // Declare a Packet object
        initial begin
        // Create a new instance of the Packet class
```

```
        p = new();
        // Randomize the Packet object and check if the randomization was successful
        `SV_RAND_CHECK(p.randomize());  // Use the macro to check if randomization
succeeded
        end
endmodule : sample_6_2
```

6.3.3 The Constraint Solver in SystemVerilog

The **constraint solver** in SystemVerilog is responsible for finding values for random variables that satisfy all the constraints defined in the testbench. It works by selecting values from the **pseudo-random number generator (PRNG)**, which is initialized with a seed. The solver ensures that the random values it chooses obey the constraints set by the user.

Key Points:

1. **Constraint Solver's Role**:
    - The solver evaluates all the constraints defined in the SystemVerilog class and chooses random values that satisfy them.
    - It uses a **PRNG** to generate random values for variables. The seed of the PRNG controls the randomness.
2. **Reproducibility**:
    - If the same **seed** and **testbench** are provided, the solver will always produce the same set of random values, making the results reproducible.
    - This is important for debugging, as you can re-run the test with the same seed and obtain the same results.
3. **Tool and Version Dependency**:
    - The results of a constrained-random test may not be the same across different **simulators** or even different **versions of the same simulator**. This is because the constraint solver is specific to the simulation vendor.
    - Although the **SystemVerilog standard** specifies the meaning of constraint expressions and the legal values, it does not mandate the precise order or method the solver should use to resolve constraints.
4. **Seed Control**:
    - The seed used to initialize the PRNG plays a crucial role in the randomization process. If you need to specify the seed, you can do so to ensure consistent results.
    - Different tool versions or settings (like debug level) can change the results even with the same seed.

Summary:

The **constraint solver** in SystemVerilog is essential for generating valid random values that meet all the constraints. However, the exact behavior of the solver may vary depending on the simulator or its version. Understanding the importance of seeds and reproducibility is crucial for ensuring reliable and consistent test results.

6.3.4 What Can Be Randomized in SystemVerilog?

SystemVerilog allows randomization of **integral variables**, which are variables that hold a set of bits. These include:

1. **2-state types** (e.g., `bit`, `logic`, `reg`):
    - These types can hold two possible values: 0 or 1.
2. **4-state types** (e.g., `bit [n:0]`, `logic [n:0]`):
    - These types can hold four possible values: 0, 1, `x` (unknown), and `z` (high impedance).
    - Randomization only generates 2-state values (0 or 1), even for 4-state types.
3. **Integers**:
    - These are variables that can hold integer values, and they can be randomized.
4. **Bit Vectors**:
    - Bit vectors (arrays of bits) can also be randomized, allowing for more complex randomization scenarios.

What Cannot Be Randomized?

1. **Strings**:
    - SystemVerilog does not support randomization of string variables.
2. **Handles**:
    - You cannot randomize a reference to an object (i.e., handles to classes).
3. **Real Variables**:
    - Randomization of real (floating-point) variables is not defined in the **Language Reference Manual (LRM)**, so it's not supported.

Summary:

In SystemVerilog, you can randomize variables that are integral types (i.e., those that represent bit patterns). However, string variables, handles, and real variables cannot be randomized directly.

6.4 Constraints and Randomization in SystemVerilog

In SystemVerilog, constraints are used to define relationships between variables, ensuring that the generated random values meet certain conditions. These relationships help avoid generating illegal or undesirable stimulus values. Constraints are particularly useful when you want to control how variables interact with each other during randomization.

Key Points:

1. **Constraints Define Relationships**:
    - Constraints express relationships between variables, such as ranges or dependencies. The SystemVerilog constraint solver ensures that random values satisfy these constraints.

2. **Random Variables in Constraints**:
    - At least one variable in each constraint expression must be marked as **random** (`rand` or `randc`). If no random variables are present, the constraint cannot generate random values but will only check if the existing values meet the constraint conditions.

Example and Explanation (Sample 6.3)

```
// Sample 6.3: Constraint without random variables

class Child;
        bit [7:0] age;  // Error - should be rand or randc
        constraint c_teenager {
        age > 12;
        age < 20;
        }
endclass : Child
```

- **Problem**: In this class, `age` is not marked as a random variable (`rand` or `randc`). This leads to an issue when calling the `randomize()` function. The randomization function will try to assign random values to the variables and check if the constraints are satisfied. However, since `age` is not random, the randomization process fails unless `age` is already within the specified range of 13 to 19.
- **Why It Fails**: Since `age` is not random, the `randomize()` function only checks if the value of `age` falls within the range specified by the constraint `c_teenager`. If `age` is not in the range of 13 to 19, the randomization will fail. However, this kind of check is better suited for an `assert` or `if` statement instead of a constraint.

Best Practice:

- Use **asserts** or **if-statements** to check the validity of non-random variables (like `age` in this case). This approach is more straightforward and easier to debug than relying on the randomization process to check constraints on non-random variables.

Summary:

- Constraints define relationships between variables and guide the randomization process.
- At least one random variable (`rand` or `randc`) must be involved in each constraint expression.
- If a variable is not random, use asserts or procedural checks instead of constraints to validate its value.

6.4.1 Constraint Introduction in SystemVerilog

In SystemVerilog, constraints are used to restrict the possible values that random variables can take. Constraints allow you to define relationships between variables and enforce rules

that must be satisfied during randomization. They help in creating more meaningful and realistic stimulus for the design under test (DUT).

Key Concepts:

1. **Constraint Block**:
   - A constraint block is used to group multiple constraint expressions together. These expressions define the legal values for the random variables.
   - The block is enclosed within curly braces `{}`.
   - The `begin...end` keywords are used for procedural code, but not in constraint blocks.
2. **Random Variables**:
   - Random variables are defined with the `rand` or `randc` keyword. These variables are subject to randomization during simulation.
   - `randc` is used for cyclic randomization, ensuring that all possible values are used before any value is repeated.

Example (Sample 6.4)

```
// Sample 6.4: Constrained-random class

class Stim;
        // Constant value for congestion address
        const bit [31:0] CONGEST_ADDR = 42;

        // Enum for different types of stimulus
        typedef enum {READ, WRITE, CONTROL} stim_e;

        // Random variables
        randc stim_e kind;              // Enumerated random variable
        rand bit [31:0] len, src, dst;  // 32-bit random variables
        rand bit congestion_test;       // Random bit for congestion test

        // Constraint block
        constraint c_stim {
        len < 1000;             // Length must be less than 1000
        len > 0;                // Length must be greater than 0

        // Conditional constraint based on congestion_test
        if (congestion_test) {
        // If congestion_test is true, restrict dst and src
        dst inside {[CONGEST_ADDR - 10 : CONGEST_ADDR + 10]};  // dst within a specific
range
        src == CONGEST_ADDR;  // src must equal the congestion address
        }
        else {
        // If congestion_test is false, src can take values in specified ranges
```

```
        src inside {0, [2:10], [100:107]};  // src can take values from these ranges
        }
        }
endclass : Stim
```

Explanation:

- **Constant Declaration**:
  - `const bit [31:0] CONGEST_ADDR = 42;` defines a constant value `CONGEST_ADDR` that is used later in the constraints.
- **Enumerated Type**:
  - `typedef enum {READ, WRITE, CONTROL} stim_e;` defines an enumerated type `stim_e` with three possible values: `READ`, `WRITE`, and `CONTROL`.
- **Random Variables**:
  - `randc stim_e kind;` defines a random cyclic enumerated variable `kind`.
  - `rand bit [31:0] len, src, dst;` defines three random 32-bit variables: `len`, `src`, and `dst`.
  - `rand bit congestion_test;` defines a random bit variable `congestion_test` that determines the behavior of the constraints.
- **Constraint Block**:
  - The constraint block `c_stim` defines the conditions that must be satisfied for randomization.
  - `len` is constrained to be between 0 and 1000.
  - The `if (congestion_test)` condition ensures that when `congestion_test` is true, the `dst` address is within a specific range of `CONGEST_ADDR`, and `src` must be equal to `CONGEST_ADDR`.
  - If `congestion_test` is false, `src` is allowed to take values from the specified ranges: `0`, `[2:10]`, and `[100:107]`.

Summary:

- **Constraint blocks** group multiple constraint expressions that define the legal values for random variables.
- **Conditional constraints** allow you to create different rules based on the value of other variables (e.g., `congestion_test` in the example).
- **Randomization** is controlled by these constraints, ensuring that only valid and meaningful stimulus is generated.

6.4.2 Simple Expressions in Constraints

In SystemVerilog, constraints allow you to define relationships between random variables using expressions. However, there are some important rules to follow when writing these expressions to avoid unexpected behavior.

Key Points:

- **Single Operator Per Expression**:
  - Each constraint expression should contain only one relational operator (e.g., `<`, `<=`, `==`, `>=`, `>`).
  - Multiple operators in a single expression can cause unexpected results, as shown in the "bad ordering" example.
- **Expression Order Matters**:
  - Constraints are evaluated from left to right, so the order of the variables and operators can affect the results.

Example Breakdown:

Sample 6.5: Bad Ordering Constraint

```
// Sample 6.5: Bad ordering constraint

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

- **Explanation**:
  - The constraint `lo < med < hi` is incorrect because SystemVerilog interprets this as:
    - `(lo < med) < hi`
  - First, `lo < med` is evaluated, resulting in either `0` or `1`. Then, `hi` is constrained to be greater than `0` or `1`, which is not what was intended.
  - This can lead to unexpected values for `lo`, `med`, and `hi`.

Sample 6.6: Result from Incorrect Ordering Constraint

```
// Sample 6.6: Result from incorrect ordering constraint

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

- **Explanation**:
  - The results show that the values of `lo`, `med`, and `hi` do not follow the intended order. The constraint `lo < med < hi` was interpreted incorrectly, causing inconsistent results.

Sample 6.7: Correct Constraint for Fixed Orde

```
// Sample 6.7: Correct constraint for fixed order
```

```
class Order_good;
        rand bit [7:0] lo, med, hi;
        constraint good {
        lo < med;  // Only use binary constraints
        med < hi;
        }
endclass : Order_good
```

- **Explanation**:
    - In this example, the constraint is split into two separate binary expressions:
        - `lo < med`
        - `med < hi`
    - This ensures that `lo` is less than `med`, and `med` is less than `hi`, preserving the intended order of the variables.

Summary:

- **Constraints with Multiple Operators**: Avoid using multiple operators in a single expression like `lo < med < hi`. Instead, split the constraint into multiple expressions.
- **Correct Expression Order**: Ensure that constraints are written in a way that maintains the correct order of variables. Use separate binary relational expressions to define the relationships between variables.

6.4.4 Weighted Distributions in SystemVerilog

When generating random stimulus for testing, certain corner cases might take a long time to be generated. To accelerate the discovery of bugs or ensure that specific cases are tested more frequently, you can apply **weighted distributions**. This allows you to skew the stimulus towards certain values or ranges, improving the efficiency of your testing.

Key Concepts:

- **dist Operator**: The `dist` operator is used to apply weighted distributions to random variables. It assigns weights to different values or ranges, and values with higher weights are more likely to be chosen.
- **Weighting**: The weight is not a percentage, and the sum of the weights does not need to add up to 100. The weight determines the likelihood of a value being selected relative to other values.

Operators for Weighted Distribution:

- `:=` (Equal Weight): Specifies that all values in a range have the same weight.
- `:/` (Divided Weight): Specifies that the weight is equally divided among all values in the range.

Example 1: Simple Weighted Distribution

```
// Sample 6.8: Weighted random distribution with (dist)

class Transaction;

        rand bit [1:0] src, dst;
        constraint c_dist {
        src dist {0:=40, [1:3]:=60};
        // src = 0, weight = 40/220
        // 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 : Transaction
```

- **Explanation**:
  - **`src`**: The values `0`, `1`, `2`, and `3` are chosen with the following weights:
    - `src = 0` has a weight of 40.
    - `src = 1`, `src = 2`, and `src = 3` each have a weight of 60.
    - Total weight = 220, so the probability of selecting `0` is `40/220`, and the probability of selecting `1`, `2`, or `3` is `60/220` each.
  - **`dst`**: The values `0`, `1`, `2`, and `3` are chosen with the following weights:
    - `dst = 0` has a weight of 40.
    - `dst = 1`, `dst = 2`, and `dst = 3` each share a total weight of 60.
    - Total weight = 100, so the probability of selecting `0` is `40/100`, and the probability of selecting `1`, `2`, or `3` is `20/100` each.

Example 2: Dynamically Changing Distribution Weights

```
// Sample 6.9: Dynamically changing distribution weights

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 : BusOp
```

- **Explanation**:
    - The `len` variable is of type `length_e` (an enumerated type with values `BYTE`, `WORD`, and `LWRD`).
    - The weights for each value are defined as `w_byte = 1`, `w_word = 3`, and `w_lwrd = 5`.
    - This means that the `LWRD` value is more likely to be selected because it has the highest weight, followed by `WORD`, and `BYTE` is the least likely to be chosen.
    - The weights can be dynamically adjusted during simulation to change the distribution of the values, allowing for more flexible testing.

Summary:

- **Weighted Distributions**: Use the `dist` operator to skew the randomness towards specific values or ranges.
- **Dynamic Weighting**: You can change the weights during simulation to modify the distribution on the fly.
- **Benefits**: Weighted distributions help ensure that certain corner cases are tested more frequently, which can accelerate bug discovery and improve test coverage.

6.4.5 Set Membership and the `inside` Operator in SystemVerilog

The `inside` operator in SystemVerilog is used to constrain a random variable to belong to a specific set of values. This set can include individual values, ranges, or combinations of both. It provides an elegant way to define the allowable values for a random variable.

Key Features:

1. **Equal Probability**: By default, the solver chooses values from the set with equal probability unless additional constraints are applied.
2. **Dynamic Constraints**: The boundaries of the set can depend on non-random variables, allowing flexibility in testbench design.
3. **Inverted Constraints**: Use the `!` operator to specify values outside a given set.

Examples:

1. Random Set of Values

The following example demonstrates how to constrain a random variable `c` to lie within a range `[lo:hi]`.

```
// Sample 6.10: Random sets of values

class Ranges;
        rand bit [31:0] c;        // Random variable
        bit [31:0] lo, hi;// Non-random variables used as limits

        constraint c_range {
        c inside {[lo:hi]};       // lo <= c && c <= hi
        }
endclass : Ranges
```

- **Explanation**:
    - `c` is a random variable.
    - `lo` and `hi` are non-random variables that define the bounds of the range.
    - The constraint ensures that `c` takes a value within `[lo:hi]` (inclusive).

2. Inverted Random Set Constraint

To specify that a random variable must **not** belong to a certain range, use the `!` (NOT) operator.

```
// Sample 6.11: Inverted random set constraint

class Ranges;
        rand bit [31:0] c;        // Random variable
        bit [31:0] lo, hi;// Non-random variables used as limits

        constraint c_range {
        !(c inside {[lo:hi]}); // c < lo or c > hi
        }
endclass : Ranges
```

- **Explanation**:
    - The `!` operator negates the constraint, so `c` must take a value **outside** the range `[lo:hi]`.

Key Notes:

1. **Empty Sets**:
    - If `lo > hi`, the set `[lo:hi]` becomes empty, and the constraint fails.

    - Ensure that `lo` and `hi` are assigned valid values before randomization.
2. **Dynamic Behavior**:
    - Since `lo` and `hi` are non-random, they can be updated dynamically during the simulation. This allows the testbench to modify the range of values for `c` without rewriting the constraint.
3. **Flexible Sets**:
    - The `inside` operator can handle complex sets, such as a combination of ranges and individual values:

      constraint c_complex {c inside {[10:20], 30, [40:50]};}

    - Here, `c` can take any value between 10–20, the value 30, or any value between 40–50.
2. **Weighted Constraints**:
    - Combine the `inside` operator with the `dist` operator to control the probability of selecting certain values or ranges.

Benefits:

- **Simplifies Constraints**: The `inside` operator makes it easier to define valid value ranges for random variables.
- **Dynamic Flexibility**: Non-random variables like `lo` and `hi` allow you to adjust constraints dynamically without modifying the code.
- **Debugging Aid**: Clearly defined sets make constraints more readable and easier to debug.

By using the `inside` operator effectively, you can create powerful and flexible constrained random stimulus for robust verification.

6.4.6 Array in Constraints

1. Randomizing Arrays

- Arrays can be randomized using `rand` or `randc` keywords.
- Constraints can be applied to individual elements or the entire array.

2. Applying Constraints to Arrays

- Use the `foreach` construct to apply constraints to each element in the array.
- Constraints can ensure properties like range, ordering, or specific values for elements.

3. Constraining Array Size

- Dynamic arrays can have their size constrained.
- Example use: Ensuring a dynamic array always contains a fixed number of elements.

4. Constraining Specific Elements

- Constraints can target specific indices in the array.
- Useful for defining values or ranges for particular positions in the array.

5. Array Slices in Constraints

- You can constrain a range of indices (slices) within an array.
- Allows for grouped constraints on subsets of the array.

6. Using Arrays in `inside` Constraints

- Arrays can act as sets, and the `inside` operator ensures values belong to these sets.
- Enables flexible and reusable constraints by referencing array elements.

7. Dynamic Constraints with Arrays

- Array values can be dynamically updated during simulation to adjust constraints.
- Helps in creating adaptive test scenarios based on simulation conditions.

8. Performance Considerations

- Using arrays in constraints can impact performance if the array is large.
- For better performance, consider alternatives like `randc` for choosing indices or values.

9. Histogram Analysis

- Arrays can be used to count occurrences of values during simulation.
- Helps validate the randomness and coverage of constraints by analyzing distribution.

10. Practical Applications

- Commonly used for scenarios like:
    - Randomizing packet headers or data streams.
    - Ensuring unique or ordered values in test cases.
    - Dynamic testbench adjustments during simulation.

6.4.7 Bidirectional Constraints

1. Declarative Nature of Constraints

- Constraints in SystemVerilog are **declarative**, not procedural.
- They are **active simultaneously**, meaning all constraints are considered together.

2. Bidirectional Solving

- SystemVerilog solves constraints **bidirectionally**, ensuring all random variables satisfy the constraints concurrently.
- Changes to one constraint affect the solution space of all related variables.

3. Example of Bidirectional Constraints

- If a variable `r` is constrained to be less than `t`, and `t` is further constrained to be less than 10, the solver adjusts `r` accordingly.
- If `s` is constrained to be equal to `r`, any change in `r` affects `s`.

4. Interdependent Variables

- Variables in a constraint block are **interdependent**.
- Adding or modifying a constraint on one variable impacts the solution for others connected through constraints.

5. Solving Complex Relationships

- The solver handles complex relationships between variables.
- For example:
  - If `r < t`, `t < 10`, `s == r`, and `s > 5`, the solver finds values that satisfy all conditions.
  - This reduces the solution space for `t` even though it has no explicit lower limit.

6. Benefits of Bidirectional Constraints

- Simplifies the creation of complex test scenarios.
- Automatically adjusts related variables, reducing manual effort.
- Ensures all constraints are respected without procedural sequencing.

7. Example Outcome

- For constraints like:
  - `r < t`
  - `s == r`
  - `t < 10`
  - `s > 5`
- Possible values:
  - `r` and `s` can be between 6 and 9.
  - `t` can range from 7 to 10, depending on `r`.

8. Practical Applications

- Useful in scenarios where multiple variables are interrelated, such as:
  - Packet generation with dependent fields.
  - Randomized configurations with hierarchical constraints.
  - Test cases with interdependent properties.

9. Debugging Bidirectional Constraints

- When constraints fail, review interdependencies.

- Use functional coverage or debug tools to verify if constraints are logically consistent.

10. Key Takeaway

- Bidirectional constraints are powerful for solving interrelated random variables, ensuring consistent and valid randomized values across complex scenarios.

6.5 Controlling Multiple Constraint Blocks

1. Overview

- A class can have **multiple constraint blocks**, each serving a specific purpose.
- Use cases for multiple constraint blocks:
  - Valid transaction generation.
  - Error scenario testing.
  - Generating different transaction patterns for specific tests (e.g., small or large packets).

2. Enabling and Disabling Constraints

- Constraints can be toggled **on or off** using the `constraint_mode()` function.
- This allows dynamic control of constraints during simulation:
  - Turn off constraints to test error scenarios.
  - Enable specific constraints to focus on a particular behavior.

3. Methods to Control Constraints

1. **Control a Single Constraint:**

   handle.constraint_name.constraint_mode(arg);
   `arg = 0`: Disable the constraint.
   `arg = 1`: Enable the constraint.

2. **Control All Constraints in an Object:**

   handle.constraint_mode(arg);

Disables or enables all constraints in the object.

4. Example Code: Controlling Constraints

```
class Packet;

  rand bit [31:0] length;

  // Constraint for short packets
  constraint c_short {
  length inside {[1:32]};
  }
```

```
    // Constraint for long packets
    constraint c_long {
    length inside {[1000:1023]};
    }
endclass : Packet

module sample_6_32;

    Packet p;

    initial begin
    p = new();

    // Generate a long packet by disabling the short constraint
    p.c_short.constraint_mode(0); // Disable short packet constraint
    `SV_RAND_CHECK(p.randomize());
    $display("Generated long packet with length: %0d", p.length);

    // Generate a short packet by disabling all constraints and enabling only the short
constraint
    p.constraint_mode(0);           // Disable all constraints
    p.c_short.constraint_mode(1); // Enable short packet constraint
    `SV_RAND_CHECK(p.randomize());
    $display("Generated short packet with length: %0d", p.length);
    end
endmodule : sample_6_32
```

5. Explanation of the Code

- **Constraints in the `Packet` class:**
    1. `c_short`: Restricts `length` to values between 1 and 32.
    2. `c_long`: Restricts `length` to values between 1000 and 1023.
- **Simulation Steps:**
    1. Disable `c_short` to allow the solver to generate long packets.
    2. Disable all constraints, then enable `c_short` to focus on generating short packets.

6. Advantages

- **Flexibility:** You can dynamically adjust constraints during runtime.
- **Reuse:** Reuse the same class with different constraint configurations for multiple test scenarios.
- **Efficiency:** Quickly switch between valid and error scenarios without rewriting constraints.

7. Challenges

- **Complexity:** Managing multiple constraints can become cumbersome if there are too many.
- **Dependencies:** Turning off constraints that validate data may inadvertently allow invalid values.

8. Key Takeaway

- The `constraint_mode()` function is a powerful tool to control constraint behavior dynamically.
- Use it to tailor randomization to specific test scenarios, ensuring thorough DUT validation.

6.6 Valid Constraints

1. Overview

- **Valid constraints** are used to ensure that random stimulus meets certain correctness rules.
- These constraints enforce specific relationships or conditions between variables to model valid scenarios for the DUT.

2. Purpose of Valid Constraints

- Ensure that random transactions adhere to **protocol or design rules**.
- Prevent invalid scenarios during normal operation.
- Provide a foundation for generating **valid and realistic test cases**.

3. Example Code: Valid Constraints

```
class Transaction;

    typedef enum {BYTE, WORD, LWRD, QWRD} length_e; // Data length types

    typedef enum {READ, WRITE, RMW, INTR} access_e; // Access types


    rand length_e length; // Randomized length

    rand access_e access; // Randomized access type


    // Valid constraint: RMW access requires LWRD length

    constraint valid_RMW_LWRD {

    (access == RMW) -> (length == LWRD);

    }

endclass : Transaction
```

4. Explanation of the Code

- **Enumerations:**
  - `length_e`: Represents the data length (BYTE, WORD, LWRD, QWRD).
  - `access_e`: Represents the type of access (READ, WRITE, RMW, INTR).
- **Constraint:**
  - The `valid_RMW_LWRD` constraint ensures that if the `access` type is `RMW` (Read-Modify-Write), the `length` must be `LWRD` (Longword).

5. Turning Off Valid Constraints

- Use the `constraint_mode` function to disable valid constraints when generating **error scenarios**.
- Example:

```
Transaction t = new();

t.valid_RMW_LWRD.constraint_mode(0); // Disable valid constraint
```

6.7 In-Line Constraints in SystemVerilog

In SystemVerilog, **in-line constraints** allow you to add extra constraints during randomization without modifying the class's original constraints. This feature is useful when you need to apply additional constraints temporarily or when working with a large number of constraints that might conflict with each other. The in-line constraint is added using the `randomize with` statement.

Key Points:

1. **In-Line Constraints**:
   - The `randomize with` statement allows you to add extra constraints to an object during randomization.
   - These constraints are added to the existing constraints of the object, meaning they work alongside any previously defined constraints.
2. **Syntax**:
   - The `with` block must use curly braces `{}`, not parentheses `()`. This is because curly braces are used for constraint blocks in SystemVerilog.
   - Inside the `with` block, you can specify additional conditions for the randomization of variables.
3. **Example Usage**:
   - If a class has constraints already defined, you can add new constraints on the fly during randomization using the `randomize with` statement.
   - You can also use `constraint_mode` to disable conflicting constraints if needed.

Example:

In Sample 6.34, the `Transaction` class has a constraint `c1` that limits `addr` to specific ranges. Then, additional constraints are applied using `randomize with` to further limit the values of `addr` and `data`.

Code Breakdown:

1. **Base Class Constraints**:
   - `c1` restricts `addr` to values within the ranges `[0:100]` and `[1000:2000]`.
2. **Randomization with Extra Constraints**:
   - The first `randomize with` statement applies extra constraints to `addr` (between 50 and 1500) and `data` (less than 10).
   - The second `randomize with` statement forces `addr` to be 2000 and `data` to be greater than 10.
3. **Important Considerations**:
   - Constraints inside the `with {}` block are evaluated along with the class-level constraints.
   - The scope inside the `with` block refers to the variables in the class directly, so you do not need to prefix them with `t.` (the instance name).
4. **Avoiding Mistakes**:
   - A common mistake is using parentheses `()` instead of curly braces `{}` in the `randomize with` block. Always use curly braces for in-line constraints.

Example:

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

module sample_6_34;
        Transaction t;

        initial begin
        t = new();

        // Apply additional constraints using randomize with
        // addr is between 50-100, 1000-1500, and data < 10
        `SV_RAND_CHECK(t.randomize() with { addr >= 50; addr <= 1500; data < 10; });

        driveBus(t);

        // Force addr to a specific value (2000), and data > 10
        `SV_RAND_CHECK(t.randomize() with { addr == 2000; data > 10; });
```

```
        driveBus(t);
        end
endmodule : sample_6_34
```

6.8 The pre_randomize and post_randomize Functions

The `pre_randomize` and `post_randomize` Functions in SystemVerilog

In SystemVerilog, the `pre_randomize` and `post_randomize` functions are special functions that allow you to perform actions immediately before or after the randomization of class variables. These functions are automatically created for any class that contains random variables.

Key Points:

1. **`pre_randomize` Function**:
   - This function is executed **before** the randomization of the class variables.
   - It is commonly used to set or adjust non-random variables, perform calculations, or prepare data before the randomization starts.
   - For example, you might use this function to set limits, calculate error correction bits, or prepare the data for randomization.
2. **`post_randomize` Function**:
   - This function is executed **after** the randomization of the class variables.
   - It is useful when you need to perform some actions or adjustments after the randomization process is completed.
   - For example, you might use this function to validate the randomization results or adjust the random variables based on certain conditions.
3. **Automatic Creation**:
   - These functions are automatically available in any class that contains random variables. You don't need to explicitly declare them unless you want to customize the behavior.

Example: Building a Bathtub Distribution (Nonlinear Distribution)

In some applications, you may need a **bathtub distribution**, where values are more likely to be at the extremes (low or high) and less likely to be in the middle. This is a **nonlinear distribution**, and while SystemVerilog provides functions like `$dist_exponential` for exponential distributions, it doesn't directly support a bathtub distribution.

In Sample 6.35, the `pre_randomize` function is used to generate values that follow a bathtub distribution by combining two exponential curves. The process is as follows:

1. **Calculate an Exponential Value**:
   - The `$dist_exponential` function is used to generate a value following an exponential distribution with a given `seed` and `DEPTH`.
2. **Limit the Value**:
   - The value is limited to a maximum of `WIDTH` to ensure it doesn't exceed a certain range.

3. **Randomly Choose Curve**:
    - A random number (`$urandom_range(1)`) is used to decide whether the value should be placed on the **left curve** or the **right curve** of the bathtub distribution. This creates a nonlinear distribution where values are more likely to be at the extremes and less likely in the middle.

Code Example:

```
class Bathtub;
        int value;                  // Random variable with bathtub distribution
        int WIDTH = 50, DEPTH = 6, seed = 1;

        function void pre_randomize();
        // Calculate an exponential curve
        value = $dist_exponential(seed, DEPTH);
        if (value > WIDTH) value = WIDTH;

        // Randomly place this point on the left or right curve
        if ($urandom_range(1)) // Randomly choose 0 or 1
        value = WIDTH - value;  // Place on the left curve
        endfunction : pre_randomize
endclass : Bathtub
```

Explanation of the Code:

- **`value`**: This is the random variable that will follow the bathtub distribution.
- **`WIDTH`** and **`DEPTH`**: These variables control the shape and range of the distribution.
- **`pre_randomize` function**:
    - First, it calculates a value using `$dist_exponential` based on the given `seed` and `DEPTH`.
    - If the generated value exceeds `WIDTH`, it is capped to `WIDTH`.
    - Then, it randomly decides whether to place the value on the **left curve** or the **right curve** of the bathtub distribution using `$urandom_range(1)`.

Conclusion:

- **`pre_randomize`** is useful for setting up or adjusting values before randomization, especially when you need to apply custom logic or calculations.
- The **bathtub distribution** example shows how to generate a nonlinear distribution by combining two exponential curves, which can be useful for testing certain edge cases, like buffer overflow scenarios.

Void Functions

In SystemVerilog, the `pre_randomize` and `post_randomize` functions are special functions that can only call **functions**, not **tasks**. This restriction is important because tasks

in SystemVerilog may involve operations that consume time, such as delays or event synchronization, which cannot be executed during the randomization process.

Key Points:

1. **Void Functions**:
   - `pre_randomize` and `post_randomize` are **void functions**, meaning they do not return a value.
   - They are executed automatically before and after randomization, respectively.
   - They are intended to perform setup or post-processing operations without introducing delays or time-consuming operations.
2. **No Delays Allowed**:
   - You **cannot** have delays (`#` or `@` delays) or other time-consuming operations inside `pre_randomize` and `post_randomize`. These functions must execute quickly and synchronously.
   - This is because randomization must be a quick process, and adding delays or waiting for events could disrupt the randomization flow.
3. **Debugging**:
   - If you want to add debugging information during randomization, you can use **void functions** that print or display information. For example, you could call display routines that do not involve delays.
   - This is useful when debugging randomization issues, as you can print the state of the variables or the constraints before or after randomization without affecting the randomization process itself.

6.9 Random Number Functions

$random — Flat distribution, returning signed 32-bit random

$urandom — Flat distribution, returning unsigned 32-bit random

$urandom_range — Flat distribution over a range

$dist_exponential — Exponential decay

$dist_normal — Bell-shaped distribution

$dist_poisson — Bell-shaped distribution

$dist_uniform — Flat distribution