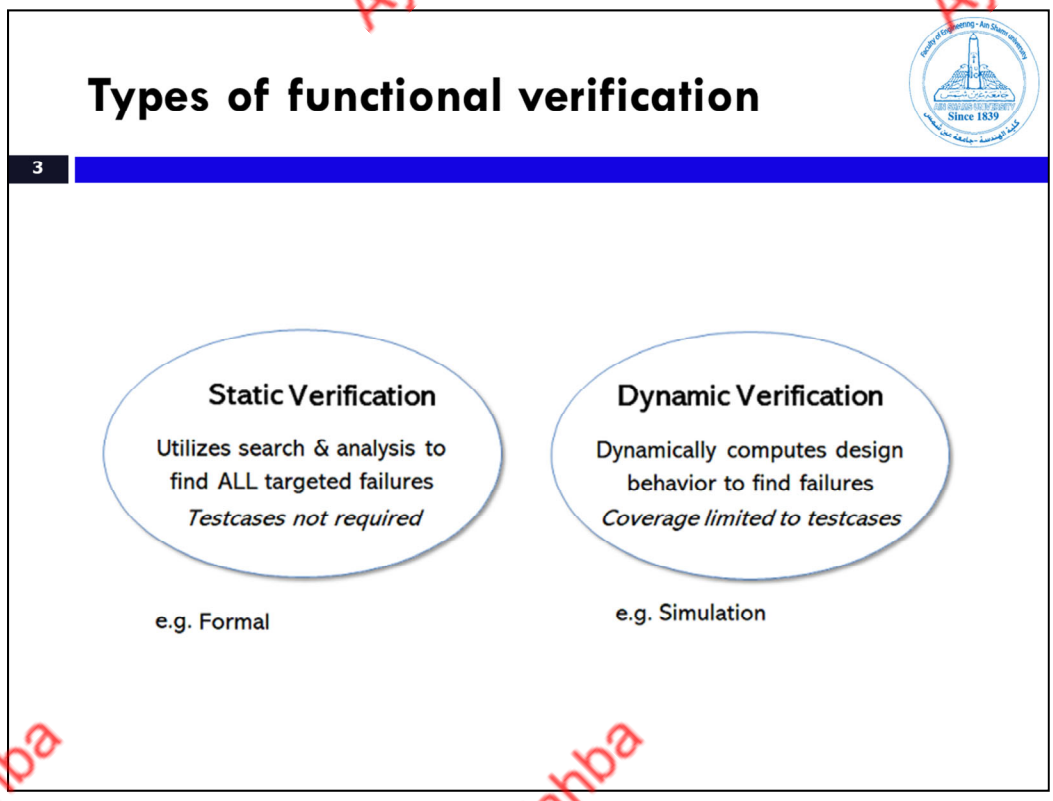

DIGITAL DESIGN
VERIFICATION & TESTING
LECTURE 2
FUNCTIONAL VERIFICATION
CSE 313s
Ayman Wahba

What is functional verification?

2

- Functional verification is concerned with the correctness of the design from the functional point of view.
- Functional verification ensures that the functionality of the implementation conforms to the specification.
- The design is exercised with different scenarios, to see if it responds in the correct way. This process is an act of verification.
- When a problem is found, we move into a debug phase where we get to understand what caused the problem and to decide how to fix the design.
- Verification is an *art rather than a science*.

- As its name implies, functional verification is concerned with the functionality of the design. It is not concerned with the performance, nor power consumption, … etc. Functional verification ensures that the functionality of the implementation conforms to the specification.
- While it was possible in the very early days of chip design to get it right the first time, design size and complexity soon grew to the point that it was very likely an engineer would make some mistakes.
- We have to exercise the design and see if it responds in the correct way. This process is an act of verification.
- When a problem is found, we move into a debug phase where we get to understand what caused the problem and to decide how to fix the design. Then we go back to verification until we gain enough confidence that no more bugs are left.
- Verification is an attempt to maximize the quality of the design in the most effective manner, and the ways in which this is done will be different for each company and sometimes for each individual product. For example, the quality level for a cheap child's toy is lower compared with an implantable medical device. This means there is no one right way to do verification, and many people have called it an *art rather than a science.*



- Fundamentally, verification is the comparison of two models.
- There are two primary ways in which the models are compared, and these are generally called *dynamic* and *static* verification methods.

Dynamic verification

4

- Mainly based on sending inputs (*stimulus*) to the design and checking the outputs (*response*) to see how it behaved.
- If errors are found that means:
  - The design is incorrect,
  - The reference we compare with is incorrect, or
  - There is a problem with the specification.
- The stimulus can be:
  - *directed* test, or
  - *random* test, or
  - *constrained random* test.

- **Dynamic verification** is most common. This method exercises the model by sending sample data into the model and checking the outputs to see what the model did. If we send in enough input data, then confidence grows that the model will always do the right thing.
- The input data stream - usually called *stimulus* - is constructed in a way that it will take the model into various areas of operation, and we can compare the output data - usually called *response* - between the two models.
- When a difference is found, it means:
  - The design model is incorrect,
  - The verification model is incorrect, or
  - There is a problem with the specification.
- Each set of data that is sent into the models is called a *test*. The comparison between the two models is done by a checker.
- The stimulus can be constructed:
  - to take the design into a specific area of functionality, normally called a *directed test*, or
  - can be based on randomized data streams, called *random test*, or
  - use controlled randomization, usually referred to as *constrained random test*.

Static Verification

5

- Static verification is a mathematical proof that two models (design model and verification model) are identical under all conditions.
- No stimulus is necessary as all possible stimuli are implicitly considered.
- The second model (the verification model) is usually constructed in a different way using what are called *properties*.
- A property defines a behavior that must be exhibited by the design, or alternatively it may define something that must never happen.

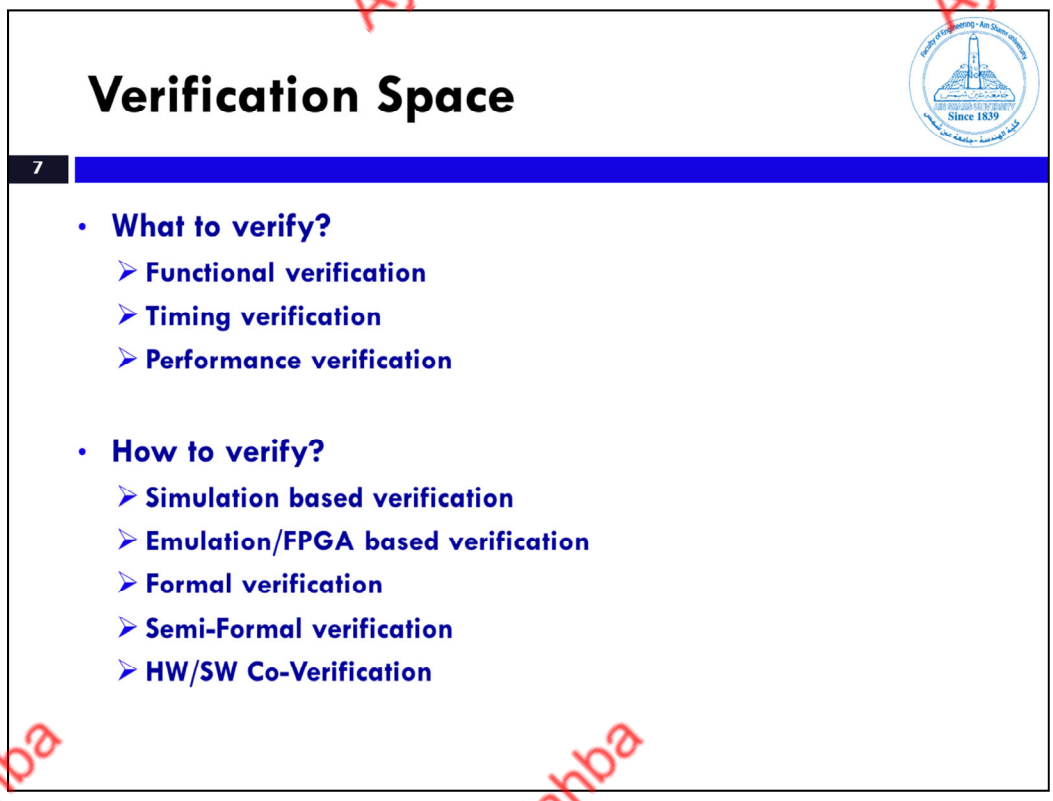
- **Static verification**, or often called formal verification, is a mathematical proof that the two models (design and verification model) are identical under all conditions.
- No stimulus is necessary as all possible stimuli are implicitly considered.
- The second model (the verification model) is usually constructed in a different way using what are called *properties*.
- A property defines a behavior that must be exhibited by the design, or alternatively it may define something that must never happen.

Start with a verification plan

6

- It is a specification document for the verification effort
- A mechanism to ensure all essential features are verified as needed
  - What to verify?
    - Features and under what conditions to verify them
  - How to verify?
    - What methodologies to use: Formal, Checking, Coverage … etc.
    - What should be: Stimulus, Checkers, Coverage … etc.
  - Priority for features to be verified
    - What methodologies

- So, what is a verification plan ? It is a specification document for the verification effort.
- It is a documentation that captures the plan you will follow to execute the verification, just like any other plan you do to get your tasks done efficiently.
- It is a mechanism to ensure all the features are verified as needed.
- The verification plan is built based on a design document being as reference.
- This document should capture ***what is to be really verified***: It capture all the features the design should support, and under what conditions this should be verified. Also, whether to verify the features independently or in a combined manner … etc.
- The document should capture also ***how to verify***. It should capture details about what are the methodologies to be used; will the simulation be good enough or there should be other areas where you can use formal verification, what are the checking methods to be used? shall we apply coverage or not? .. .etc.
- That plan should also capture what are the stimulus that should be applied to get the verification done, and what are the kind of checkers you will implement, and what levels of coverage you will implement.
- The plan should also have a priority for the features to be verified, especially when the designs become more and more complex, and the number of features and combination increase drastically. It is always important to capture the priority of which feature must be verified and which can be waived.
- The verification plan, should contain all of these details. This helps in executing the plan smoothly and efficiently.



- Next question is what really to be verified (this is known as verification space)
- The most three important aspects of the verification space are:
  - Functional verification: a process focusing more on whether the functionality of the design is verified. This kind includes making sure what are the design protocols implemented, and the features supported by the design, etc.
  - Timing verification: This focuses more on making sure that the actual timing of the implemented design meets the specifications of the design. Some examples could be making sure that the design works well on the maximum frequency on which it is designed.
  - Performance verification: is the process in which we focus more on the actual performance goals of the design. Some examples could be, if the design is a microprocessor, then the design has to execute a number of instructions every clock cycle. Another example, if the design has a memory, then we should make sure that the design is capable of making a read or write operation every cycle.
- The next question is how to verify, or what are the different approaches that can be used for verifying a design:
  - The first one is simulation-based verification which is still the most commonly used in industry.
  - Formal verification
  - Semi-formal verification

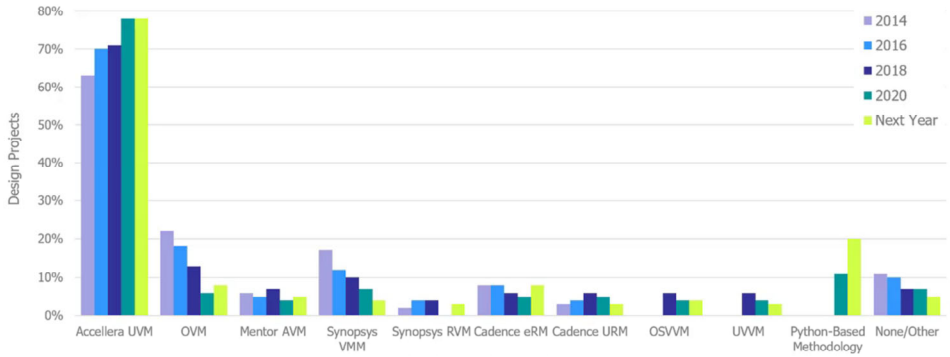
Verification Methodologies

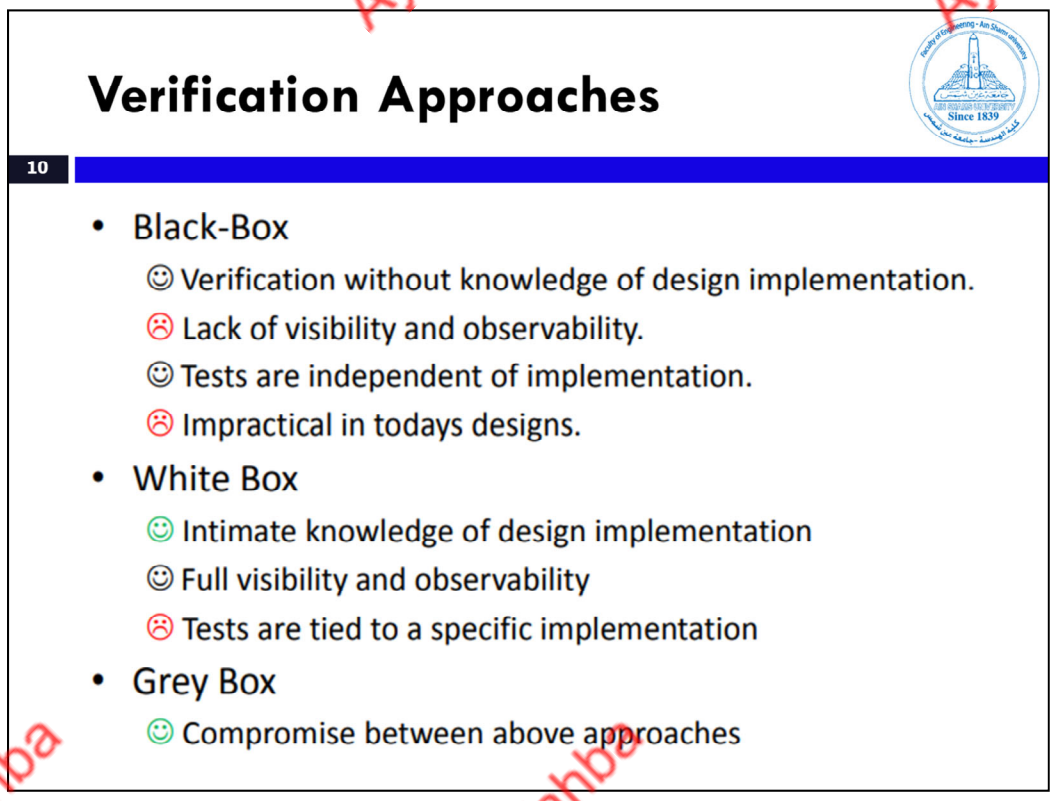
A standardized way to verify integrated circuit designs.

- **The main purpose of a methodology is to *optimize some aspect of the design* while minimizing the time spent on it.**
- **It also ensures consistency within a company such that *reuse becomes possible*.**
- **Over the years, companies have released numerous different methodologies using different languages.**

- A methodology defines the models that are created, how they are used and the ways in which tools are used to manipulate them.
- Models can define the design at several levels of abstraction, they can define the requirements of the design or they can define closure criteria.
- The main purpose of a methodology is to *optimize some aspect of the design* while minimizing the time spent on it.
- It also ensures consistency within a company such that *reuse becomes possible*.
- Over the years, companies have released numerous different methodologies using different languages.

Some Verification Methodologies



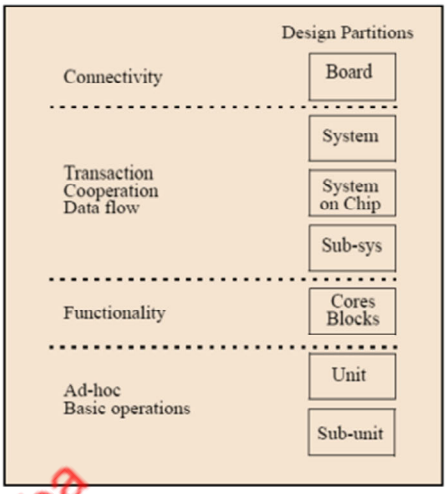


- Now let's see what are the kinds of approaches that we use to make the verification. There are basically three approached:
  - **Black Box Approach:** it is an approach where the verification engineers do not need to know everything about the design implementation. He just needs to know what are the features that the design supports. So, the design is treated as a black box, and you approach the verification in that way.
    - The major advantages of this approach since it is independent on the design implementation, is that you can reuse your test with different design implementations. Also, the verifier does not need to care about understanding all the details of the implementation.
    - The major disadvantage since you don't really care about understanding the implementation, is that you will have less visibility of the internals of the design. Also, with the design complexity that we deal with, in the current days this way is impractical.
  - **White Box Approach:** Here the verifier must have a good understanding of the details about the implementation before we start the verification.
    - The major advantage here is that you have full visibility of the design implementation, so you write the test stimulus to cover every detail of the implementation.
    - The disadvantage here is that, depending on the way you write the test, it could be tied to the design implementation, and can't be reused when the implementation changes.
  - **Grey Box Approach:** It is a compromise between both approaches. You don't really need to know everything about the design, and you can go and create test, but for certain features and scenarios you may need to know the details of some parts of the implementation.

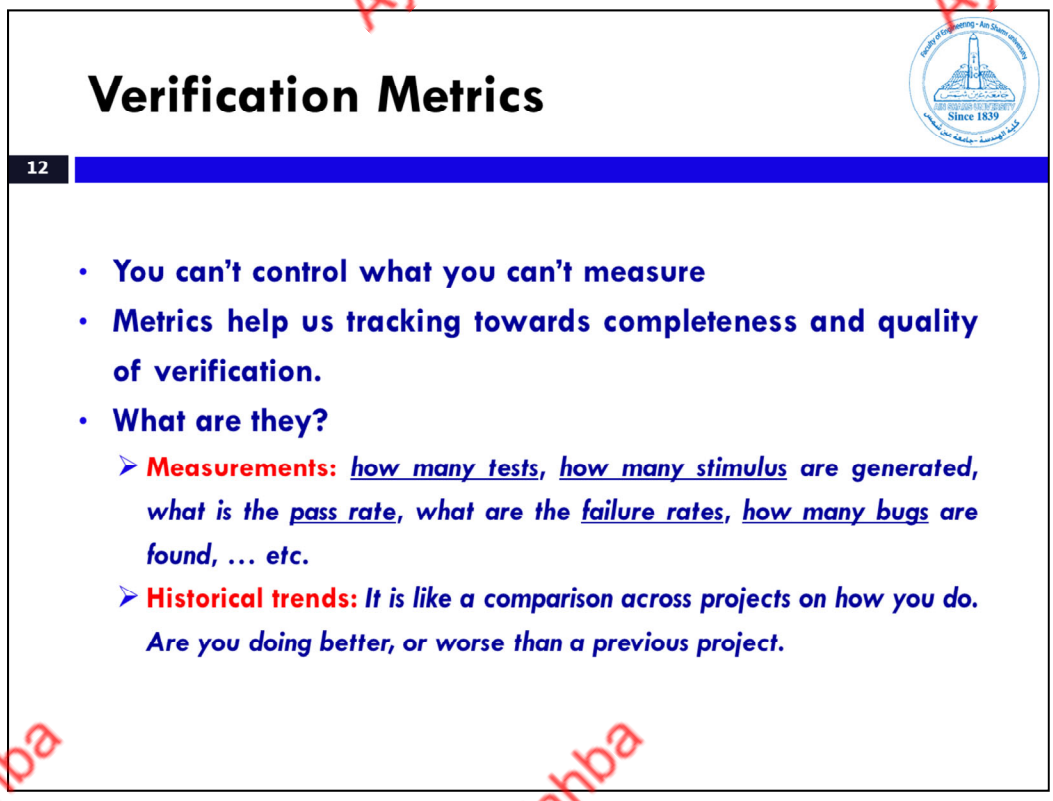
Levels of Verification

11

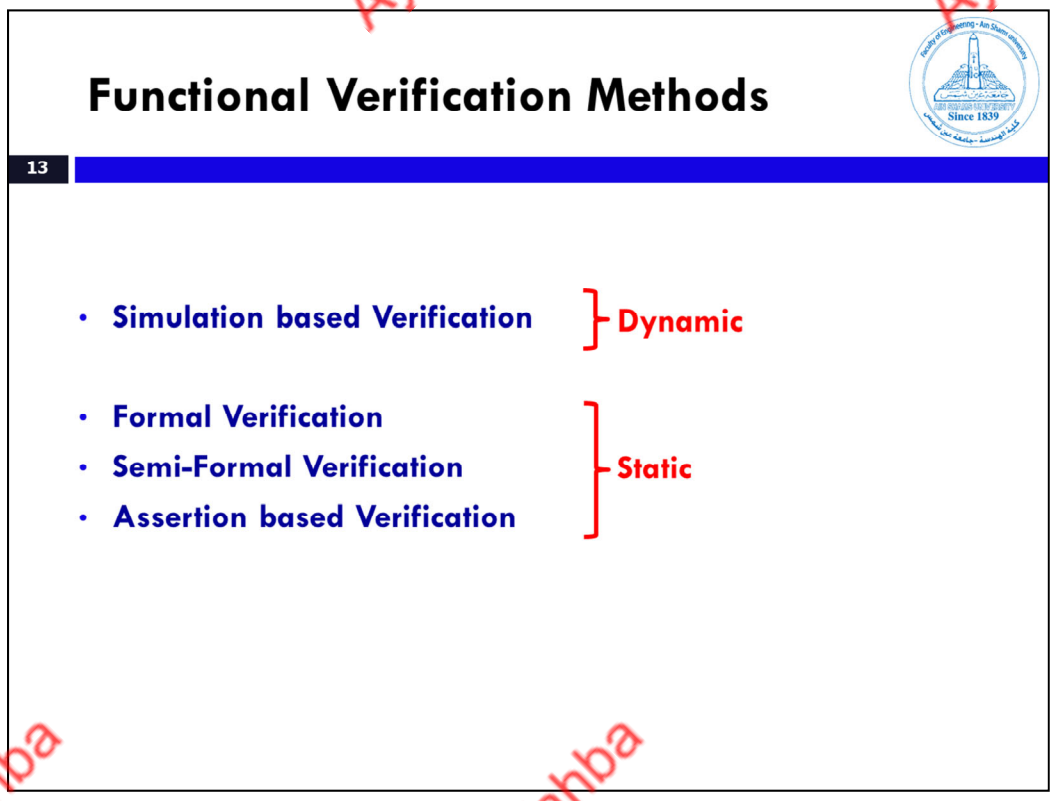
- Each level of Verification will be suited for a specific objective
  - Lower levels have better controllability and visibility
- Block level Verification helps designs to be verified independently and in parallel
- System Level Verification focuses more on interactions



- Every design is partitioned into systems, subsystems, core IP blocks, units, and sub-units.
- Then the design starts using a bottom-up approach. Most of the sub-units, and the units that are needed for the design to work are designed in parallel.
- Once the design units come in place, they form the core blocks. The blocks are put together to form a sub-system, and then SOC, and finally the chips are assembled on a board.
- Just as the design is partitioned, you can also have your verification partitioned into levels. Each level of verification will be suited for a specific objective.
- At the unit and sub-unit you don't really need to do a proper testbench in the verification environment. You can do ad-hoc basic operations like forcing certain input signals, and make sure that the design output behaves as the design specification.
- The more verification you can do at the lower levels, you can find most of the early bugs in parallel to the design activities, so that you can save your time.
- Once the design stabilizes, you can put multiple of those blocks into a sub-system and do more of system level verification, where the focus will be more on interactions across these units.
- These levels of verifications can go up all the way till product verification.



- You can't control what you can't measure.
- The verification metrics, like any other metrics help us to track the advancement of the verification towards the completion of the verification plan.
- So, what are these metrics ?
  - ➢ They are basically a kind of measurements we do on the verification plan. We capture what are the features to be verified, what are the stimuli … etc. Thus, we can track *how many tests* are run, and how *many stimulus are generated*. *What is the pass rate*, and *what are the failure rates*, *how many bugs are found* from these tests … etc. All of these kinds of measurements help us track towards the actual execution versus what we have planned for the verification.
  - ➢ The other set of metrics are like historical trends, especially if you are doing a project after project. The metrics are trends from your past that will help you determine how you are trending in the current project. That can be useful to compare your quality or the quality of verification for your current project versus how it was in a previous project. It is like a comparison across projects on how you do. Are you doing better, or worse than a previous project.

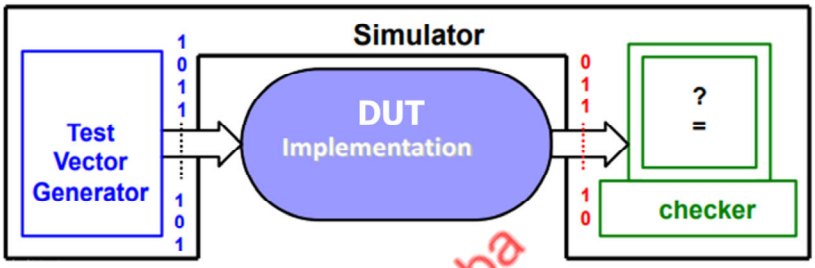


- We will learn some of the verification methods that are most commonly used.
- We will learn about:
  - Simulation based verification
  - Formal verification
  - Semi-formal verification
  - Assertions, and how they will be used in verification.

Simulation based Verification

14

- Generate Input sequences
  - Directed/Random/Constrained-Random sequences
- Generate expected output sequence (golden)
- Simulate the DUT with the generated input sequences
- Verify DUT output against golden output
- Use Coverage Metrics for ensuring completeness



- This diagram shows a design implementation and the simulator; typically software simulator surrounding it.
- The simulation based verification is the most commonly used verification method for most of the design verifications.
- In this approach we use a test vector generator to generate input stimulus to the design, and the design under test (DUT) or the design implementation is actually simulated using the simulator.
- We also use a golden reference model, which is some kind of a checker or a scoreboard to generate expected outputs which are sometimes called golden outputs.
- These are compared against the actual outputs coming from the design to ensure the correctness of the design.
- Now based on the complexity of the design, the test pattern generator can be very simple, or it can also be as complex as *directed*, *random*, or *constrained random* test generator.
- We also use several coverage metrics to make sure the verification is complete and the needed level of quality is met.
- We will see most of the details about that in the following lectures.

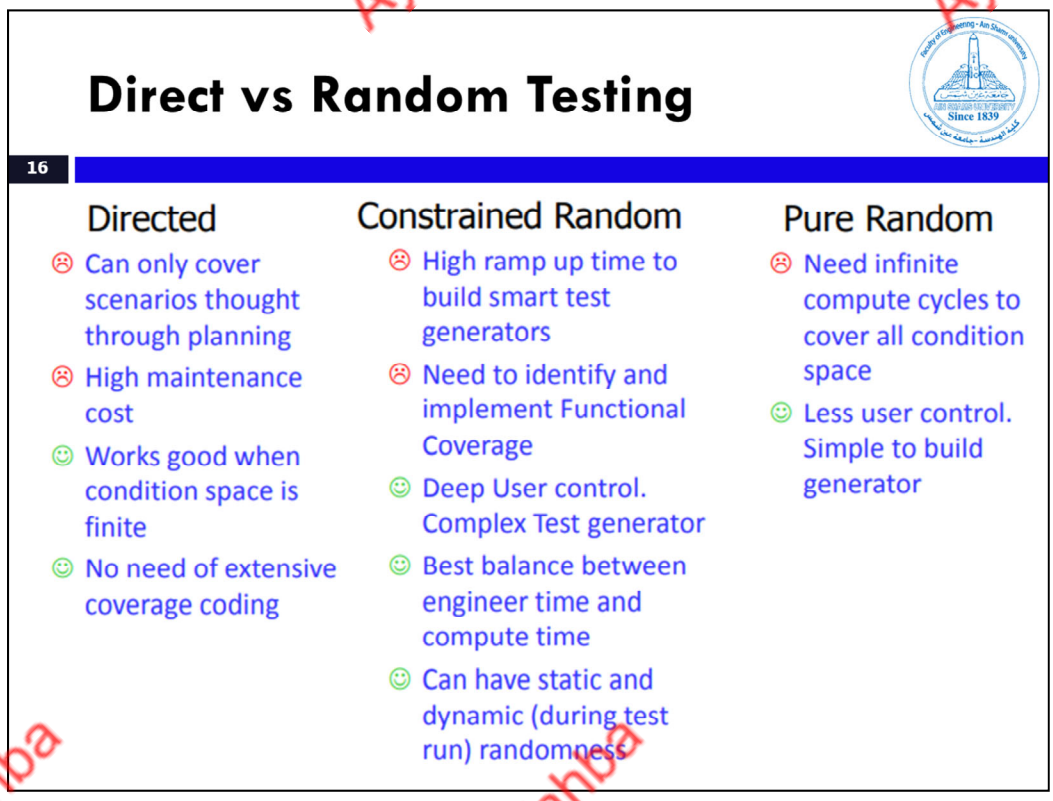
Direct vs Random Testing

15

- Directed Testing:
  - Directed testing involves designing specific test cases for each feature of the design.
- Random Testing:
  - Random testing, also known as fuzz testing, involves generating random inputs and feeding them into the hardware system to observe its behavior.
  - Test cases are created without specific knowledge of the system's internals or requirements.

- Directed testing and random testing are two different approaches to testing hardware systems
- Directed Testing:
  - Directed testing involves designing test cases based on specific requirements, functionality, or known issues in the hardware system.
  - Test cases are carefully crafted to target particular aspects of the system, such as input ranges, boundary conditions, or critical functionalities.
  - This method aims to systematically verify the correct behavior of the hardware by focusing on predetermined scenarios and edge cases.
  - Directed testing is typically more thorough and exhaustive but may miss unexpected bugs or behaviors that were not considered during test case design.
- Random Testing:
  - Random testing, also known as fuzz testing, involves generating random inputs and feeding them into the hardware system to observe its behavior.
  - Test cases are created without specific knowledge of the system's internals or requirements.
  - The goal is to uncover unexpected bugs or vulnerabilities by subjecting the system to a wide range of inputs that may not have been anticipated during design or testing.
  - Random testing can be useful for discovering issues that might not be found through directed testing alone, but it may also produce a large number of irrelevant or redundant test cases.
  - It's often used in conjunction with other testing methods to complement their coverage and identify weaknesses in the hardware system.

In summary, directed testing is a systematic approach focused on specific aspects of the hardware system, while random testing explores a broader range of inputs to uncover unexpected issues. Both methods have their strengths and weaknesses, and combining them can provide more comprehensive test coverage for hardware systems.



Directed testing:

- Can only cover scenarios that are thought in our planning. During the verification planning all scenarios or features that we can think about has to be tested.
- If we consider timing and cost we find that for every feature, we have to create a single test and based on the number of features and number of tests that might be used and these tests will have to be maintained throughout the project or across projects … etc.
- As I mentioned this works good when the condition space is finite. That's when we can exactly plan for all the scenarios.
- If the design space is huge, then it's not possible thinking through all the scenarios in a given project time frame.
- Other good point is that we don't need an extensive coverage coding since we know exactly what the test is covering. It's guaranteed the test is covering known scenarios. So there's no other effort needed to make sure whether those scenarios are covered.

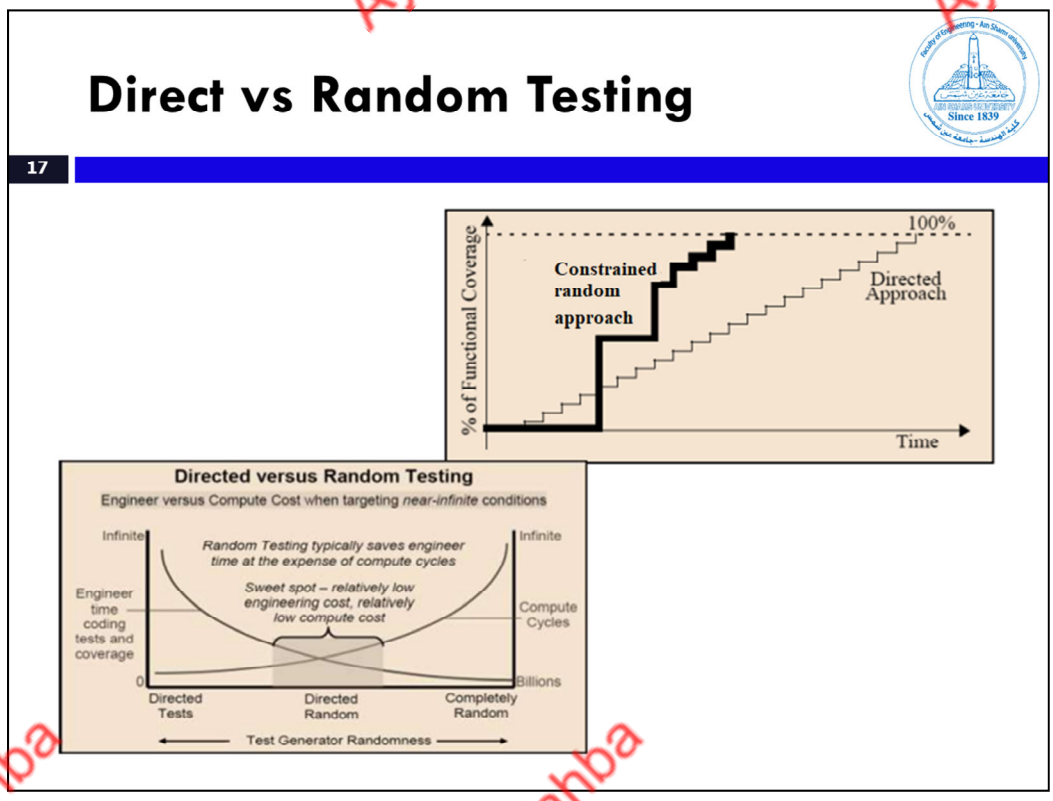
Random testing:

- In the pure random test approach we depend on the random test generated, that can randomly hit all the scenarios. So we need to run for an infinite number of states if we want to make sure all the conditions are covered
- In a pure random approach we depend on the generator to randomly create all the scenarios

Constrained random:

- In between the two approaches we have another approach called constrained random approach.
- We don't let the generator to be purely random, but we make the generator directed towards some interesting scenarios in the design.
- This approach definitely has a higher ramp up time since we have to understand which are the covered scenarios by the random test generator, and write a code to direct it for the uncovered scenarios.
- We also need to follow some kind of metrics to measure how good the constrained randomness is and also to measure the completeness.
- This approach will give you deep user control to building complex test generators. Test generators can be built with a lot of user control, in that way you can hit all the different state space in the design.
- I consider that the best balance between engineer time and compute time in the sense that you don't have to be running too many seeds and depend on the pure randomness to get everything. We constrained, a lot of randomness, so that a lot of scenarios can be hit much faster.
- We can have static and dynamic randomness, i.e. we can either create all the of test upfront, or it's also possible to build intelligence in the generator, or we can dynamically lead the randomness to hit more and more interesting scenarios.

So that is the comparison between the three approaches, and it is clear that the constrained random approach seems to be the best balance especially for complex designs, here we will not be able to create a test for every scenario, and we cannot be depending on a pure random generator to get everything out of luck.



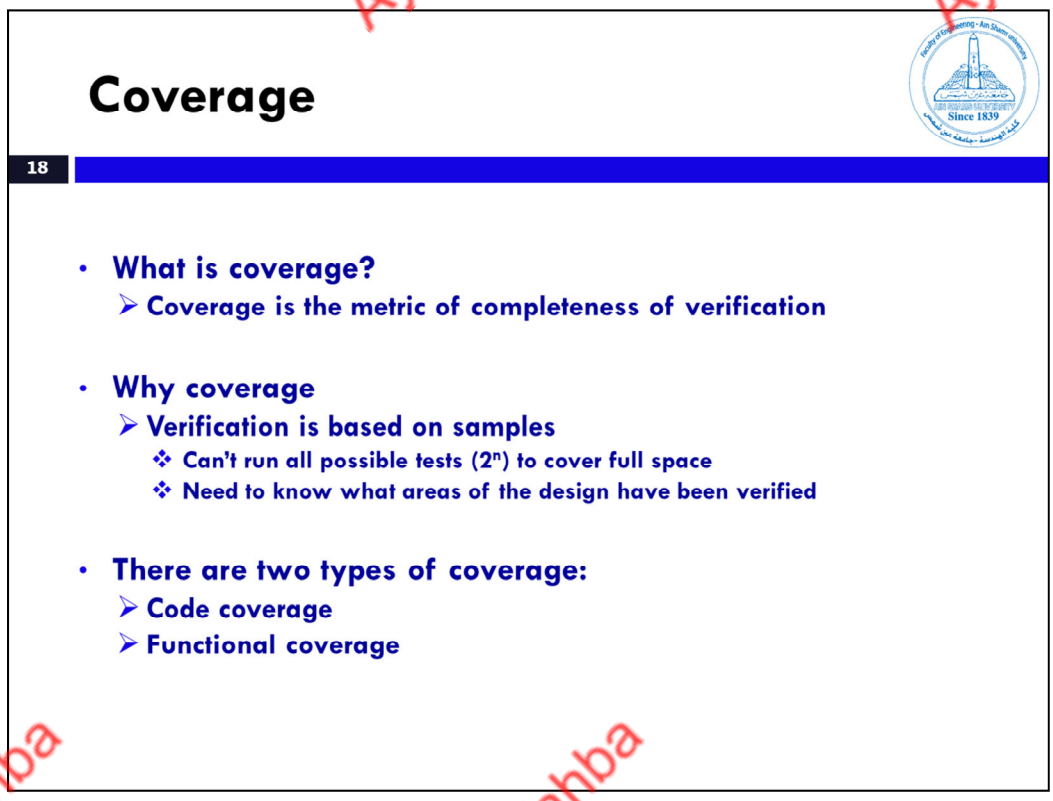
These diagrams again explain directed versus random test coverage.

First diagram:

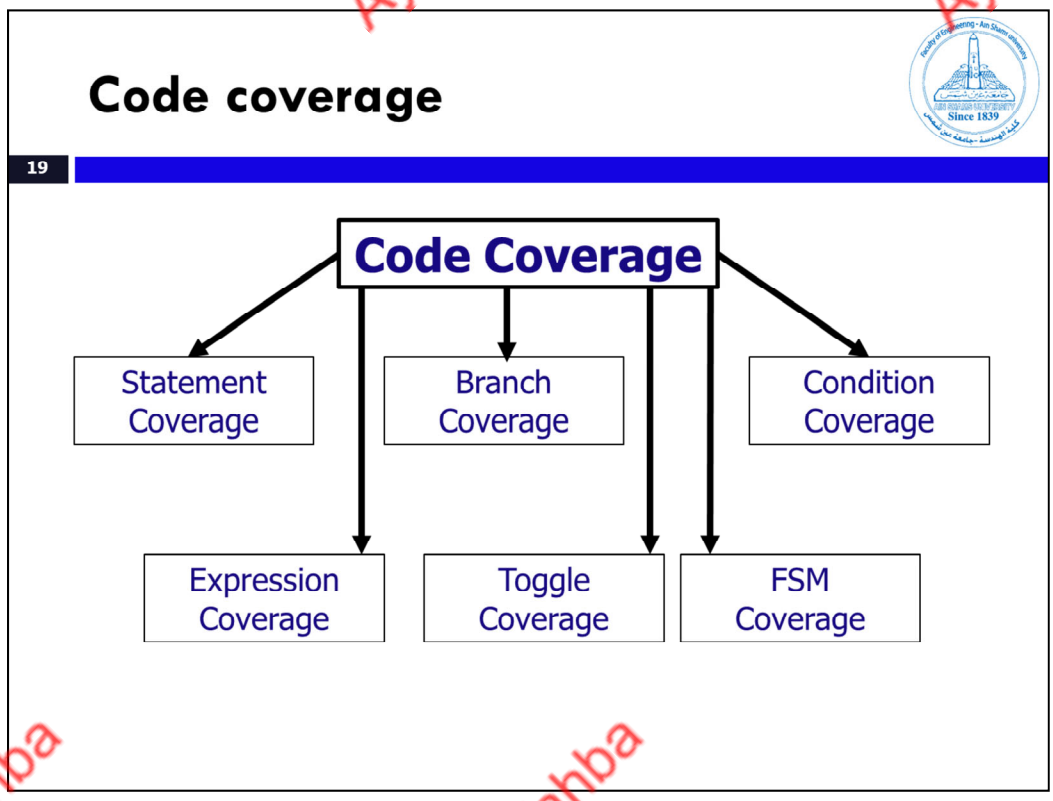
- The first diagram shows the coverage versus time
- So in the directed test approach, with every generated test we hit a certain percentage of the functional coverage, so it's like step improvement throughout the project and takes let's say so much time to get 100% coverage to be hit.
- In the constrained random approach, initially a lot of effort has to be put to build the generator. During this time, you will not be able to test anything
- Once the generator is up you can get a quick jump, and then you can see a gain in coverage improvement, so you can go and tune the generator, and we will again get a big jump.
- We can hit the 100% coverage easier than the directed approach.

Second diagram:

- The second graph also shows a comparison between the engineer time spent versus the compute cycles time spent in all the three approaches.
- So as you can see in the directed test, engineers build a directed test for every scenario, but for completely random engineers put a small effort.
- On the contrary, compute cycles are low with directed testing, and very big for completely random test.
- The sweet spot, is in between, while using directed random test, where we have reasonable engineer time, and compute cycles.



- What is coverage and how it is used in the verification?
- Coverage is defined as a metric for completeness a verification.
- Why do we do coverage ?: As explained in the constrained random verification which is most commonly used for complex designs, we don't do an extensive verification of the complete space, we do a verification based on samples, as we can not run all the 2^n combinations to cover the entire verification space. So we follow the constrained random approach to create a test which only tests the interesting scenarios. So, we need to know by some means what are the areas of the design that have been verified.
- Functional coverage and code coverage are two types of coverage that will help us identify what areas of design have been verified well.



- By measuring code coverage, hardware designers and verification engineers can assess the thoroughness of their testing efforts and identify areas of the design that have not been adequately exercised. Increasing code coverage helps improve the reliability and quality of the hardware design by identifying potential bugs or issues that may arise during operation.
- Code coverage comes in different flavors. It is a metric of the code covered during simulation.
- **Statement coverage:** it is a straightforward metric which tells you how much of the statements in the source code are executed during your test.
- **Branch coverage:** Branch coverage evaluates the percentage of decision points (branches) in the code that have been taken during testing. It ensures that both the true and false branches of conditional statements have been exercised. Examples of branches are the *if statement*, *while statement*, … etc. You should make sure that all those branching conditions are covered by your tests or not.
- **Condition coverage:** it gives you a measure of whether every Boolean subexpression is evaluated to true or false.

Code coverage

20

- **Statement**
  - Has each statement of the source code been executed?
- **Branch**
  - Percentage of decision points (branches) in the code that have been taken during testing. Has each control structure been evaluated to both true and false?
  - Examples: if, while, repeat, forever, for, loop
- **Condition**
  - Has each Boolean sub-expression evaluated both to true and false?

- By measuring code coverage, hardware designers and verification engineers can assess the thoroughness of their testing efforts and identify areas of the design that have not been adequately exercised. Increasing code coverage helps improve the reliability and quality of the hardware design by identifying potential bugs or issues that may arise during operation.
- Code coverage comes in different flavors. It is a metric of the code covered during simulation.
- **Statement coverage:** it is a straightforward metric which tells you how much of the statements in the source code are executed during your test.
- **Branch coverage:** Branch coverage evaluates the percentage of decision points (branches) in the code that have been taken during testing. It ensures that both the true and false branches of conditional statements have been exercised. Examples of branches are the *if statement*, *while statement*, … etc. You should make sure that all those branching conditions are covered by your tests or not.
- **Condition coverage:** it gives you a measure of whether every Boolean subexpression is evaluated to true or false.

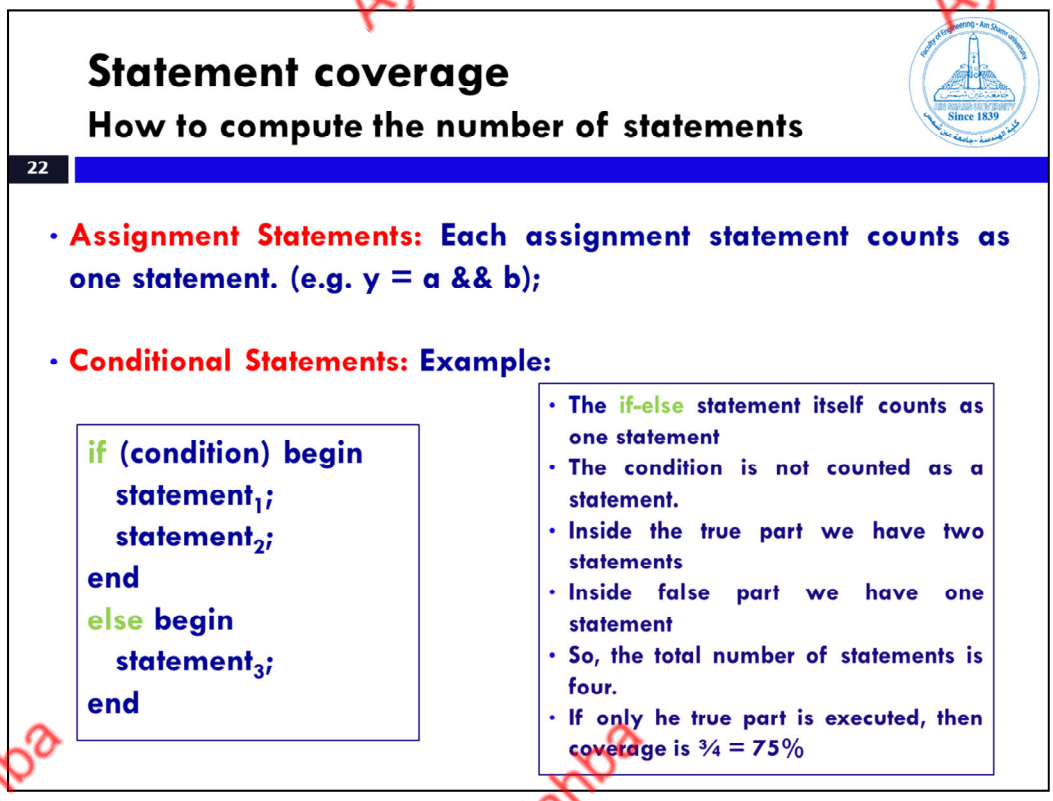
Code coverage

21

- **Expression**
  - Covers the RHS of an assignment statement
  - Example: x <= a xor (not b);
- **Toggle**
  - Tracks the percentage of flip-flops or registers that have changed state during simulation.
  - Standard – covers 0 -> 1 and 1 -> 0 transitions
  - Extended – covers 0 <-> 1, z <-> 1 and z <-> 0 transitions
- **FSM**
  - State coverage and FSM Arc coverage

- **Expression coverage:** it considers the RHS of expressions. In this example the coverage will give you whether all the combinations of ***xor*** and ***not*** are covered.
- **Toggle coverage:** Toggle coverage tracks the percentage of flip-flops or registers that have changed state during simulation, indicating whether these elements have been exercised.

  There are two forms of toggle coverage: standard coverage, and extended coverage.
  - Standard: makes sure that all nodes experienced transitions from 1 to 0, and from 0 to 1.
  - Extended: makes sure that all nodes experienced transitions from between all 4 logic states (0,1,z,x).
- **FSM coverage:** gives a coverage about the state transitions, and the different arc coverages.



- **How to compute the number of statements?**

➢ Assignment statements: each assignement statement is counted as one statement.

➢ Conditional statement:
  - The if-else statement itself counts as one statement
  - The condition is not counted as a statement.
  - Inside the true part we have two statements
  - Inside false part we have one statement
  - So, the total number of statements is four.
  - If only he true part is executed, then coverage is ¾ = 75%

Statement coverage

How to compute the number of statements

23

- **Loop Statements**

```
for (int I = 0; I < N; i++)
begin
   statement1;
   statement2;
   statement3;
end
```

- The for-loop statement itself counts as one statement
- The loop initialization, condition, and increment parts (int i = 0; i < N; i++) are not counted as separate statements.
- Inside the loop body we have three statements
- So, the total number of statements is four.

- **Module Instantiation: Each module instantiation counts as one statement.**

Ayman Wahba

➢ Loop statements:
- The for-loop statement itself counts as one statement
- The loop *initialization*, *condition*, and *increment* parts (int i = 0; i < N; i++) are not counted as separate statements.
- Inside the loop body we have three statements
- So, the total number of statements is four.

➢ Module instantiation: Each module instantiation counts as one statement.

Statement coverage

How to compute the number of statements

24

- **Procedural blocks: (always, initial, fork)**

```
always @(posedge clk)
begin
  statement1;
  statement2;
  statement3;
end
```

- The always block itself counts as one statement
- The sensitivity list (posedge clk) are not counted as separate statements, as they are part of the always block.
- Inside the block body we have three statements
- So, the total number of statements is four.

- **Function and Task calls: Each function or task call counts as one statement.**
- **Control Flow Statements: Each control flow statement (break, continue, return) counts as one statement.**

- Procedural blocks (always, initial, fork, etc.): In the shown example:
  - The always block itself counts as one statement
  - The sensitivity list (posedge clk) is not counted as a separate statement, as it is part of the always block.
  - Inside the blcok body we have three statements
  - So, the total number of statements is four.
- Function and Task calls: each function or task call counts as one statement
- Control Flow Statements: (break, continue, return). Eacj contrl flow statement counts as one statement.

Branch coverage
Example

25

```
module simple_module (
    input logic a,
    input logic b,
    output logic y
);

    always_comb begin
        if (a && b)
            y = 1;
        else
            y = 0;
    end

endmodule
```

- **When writing the testbench you need to explore both branches, so you need two test cases:**
  - **Test case 1: a = 1, b = 1**
  - **Test case 2: a = 1, b = 0**

- Branch coverage in SystemVerilog involves ensuring that every possible branch in your code is executed at least once during testing. Let's demonstrate branch coverage with the shown example using an if-else statement.
- To achieve branch coverage for this module, we need to ensure that both branches of the if-else statement are taken during testing.
- When writing you testbech you need to explore both branches, so you need two test cases:
  - Test case 1: a = 1, b = 1
  - Test case 2: a = 1, b = 0

  By running these test cases, we ensure that both branches of the if-else statement are taken during testing, achieving branch coverage for the module.

Condition coverage

Example

26

```
module condition_module (
    input logic a,
    input logic b,
    output logic y
);

    always_comb begin
        if (a && b) begin
            y = 1;
        end else if (a && !b) begin
            y = 2;
        end else if (!a && b) begin
            y = 3;
        end else begin
            y = 4;
        end
    end

endmodule
```

- When In the shown example we have 3 conditions:
  - C1: (a && b)
  - C2: (a && !b)
  - C3: (!a && b)
- To achieve condition coverage, the test cases are:
  - Test case 1: (a = 1, b = 1) → C1=true, C2=false, C3=false
  - C2: (a && !b): (a = 1, b = 0) → C1=false, C2=true, C3=false
  - C3: (!a && b): (a = 0, b = 1) → C1=false, C2=false, C3=true

- Condition coverage in involves ensuring that every Boolean condition in your code evaluates to both true and false at least once during testing.
- In the shown example we have 3 conditions:
  - C1: (a && b)
  - C2: (a && !b)
  - C3: (!a && b)

- To achieve condition coverage, the test cases are:
  - Test case 1: (a = 1, b = 1) → C1=true, C2=false, C3=false
  - C2: (a && !b): (a = 1, b = 0) → C1=false, C2=true, C3=false
  - C3: (!a && b): (a = 0, b = 1) → C1=false, C2=false, C3=true

- Note that these test cases achieve 100% condition coverage, but only 75% branch coverage as the branch (y = 4) is not exercised.

Expression coverage
Example

27

- **Assume we have an expression like:**

  Result = 3*(a+b);
- **Ensure that various combinations of operands a and b are evaluated with different values to exercise different paths and scenarios within the expression.**
  - **Create a set of test cases that cover a wide range of possible values for a and b.**
  - **This includes *positive*, *negative*, *zero*, *maximum*, and *minimum* values (edge cases and boundary conditions).**
- **Apply Constraints: you may apply constraints to guide the generation of test stimuli to specific values.**

- Assume we have an expression like:

  Result = 3*(a+b)

  To achieve expression coverage for this expression, you would need to ensure that various combinations of operands a and b are evaluated with different values to exercise different paths and scenarios within the expression.
- Here's how you can approach achieving expression coverage:
  - Create a set of test cases that cover a wide range of possible values for a and b.
  - This includes *positive*, *negative*, *zero*, *maximum*, and *minimum* values (*edge cases* and *boundary conditions*).
  - Apply Constraints: you may apply constraints to guide the generation of test stimuli to specific values. Constraints can help ensure that the test cases cover specific ranges or conditions of a and b that are critical for expression coverage.

Toggle coverage
Example

28

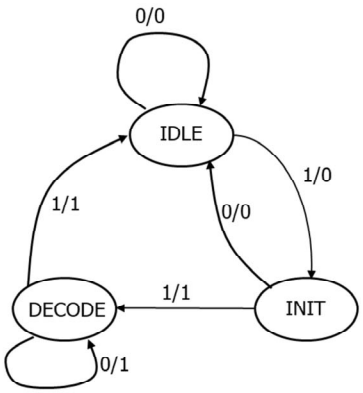
```
module ToggleCoverageExample;
  logic signal1;
  logic signal2;

  // Generate stimuli for simulation
  initial begin
    signal1 = 0;
    signal2 = 0;
    #10 signal1 = 1;
    #10 signal2 = 1;
    #10 signal1 = 0;
    #10 signal2 = 0;
    $finish;
  end
endmodule
```

- **Here we define two signals: signal1 and signal2.**
- **In the initial block, we generate stimuli to toggle both signals between low and high values.**
- **Note that we made a transition from 1 to 0, and another transition from 0 to 1.**

- Here's an example demonstrating toggle coverage of a signal in SystemVerilog
- In the shown example, we define two signals named signal1 and signal2.
- In the initial block, we generate stimuli for both signals by toggling them between low and high values.
- Note that we made a transition from 1 to 0, and another transition from 0 to 1.
- This example demonstrates how to set up toggle coverage in SystemVerilog to monitor the activity of signals.

FSM coverage
Example

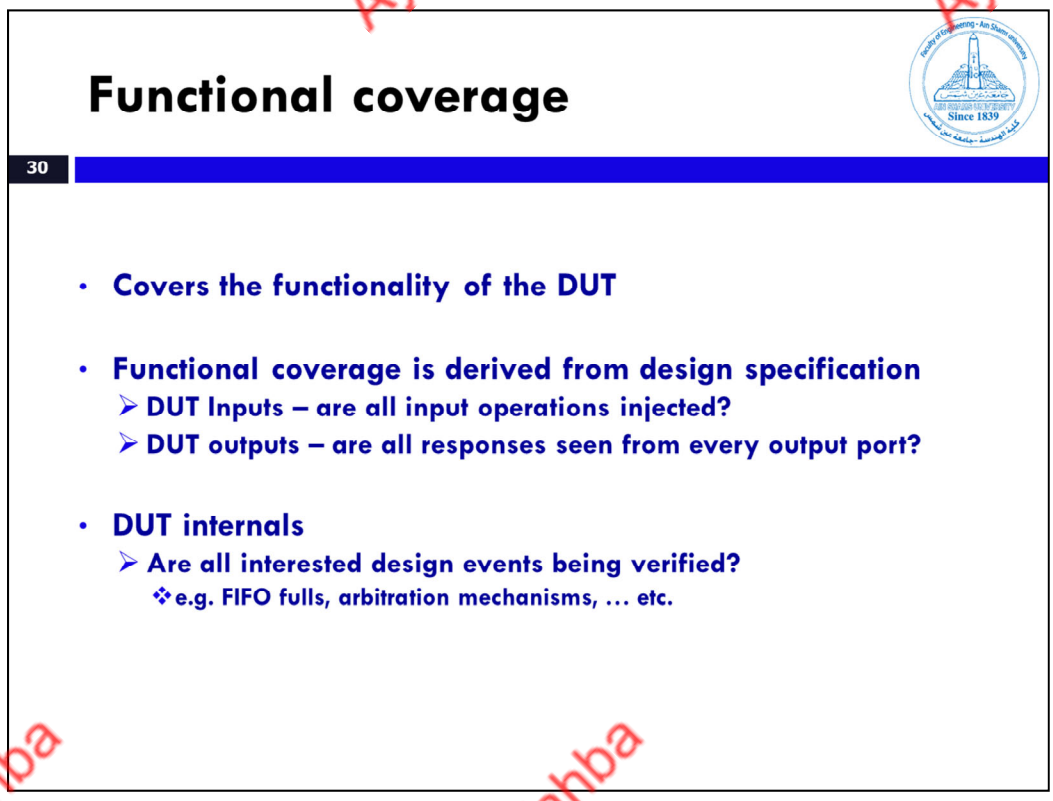
29



- Here we have a finite state machine, with 3 states and 6 transitions.
- We have to generate test stimuli, to make sure that you go to each one of the states; *IDLE*, *INIT*, and *DECODE*.
- Also you should go through all the transitions:

IDLE → IDLE, IDLE → INIT,
INIT → IDLE, INIT → DECODE,
DECODE → DECODE, DECODE → IDLE

- FSM coverage, in the context of hardware verification, aims assess the effectiveness of testing of finite state machines (FSMs).
- FSMs are crucial components in digital hardware design, governing the behavior of sequential logic circuits.
- FSM coverage metrics help ensure that the design is adequately tested to verify its functionality and correctness.
- It involves tracking various aspects such as:
  - State Coverage: Ensures that every state of the FSM is reached during simulation
  - Transition Coverage: Verifies that each transition between states is exercised.
- By analyzing FSM coverage metrics, engineers can identify untested or under-tested areas of the design and improve their verification strategies to achieve higher confidence in the correctness of the hardware.



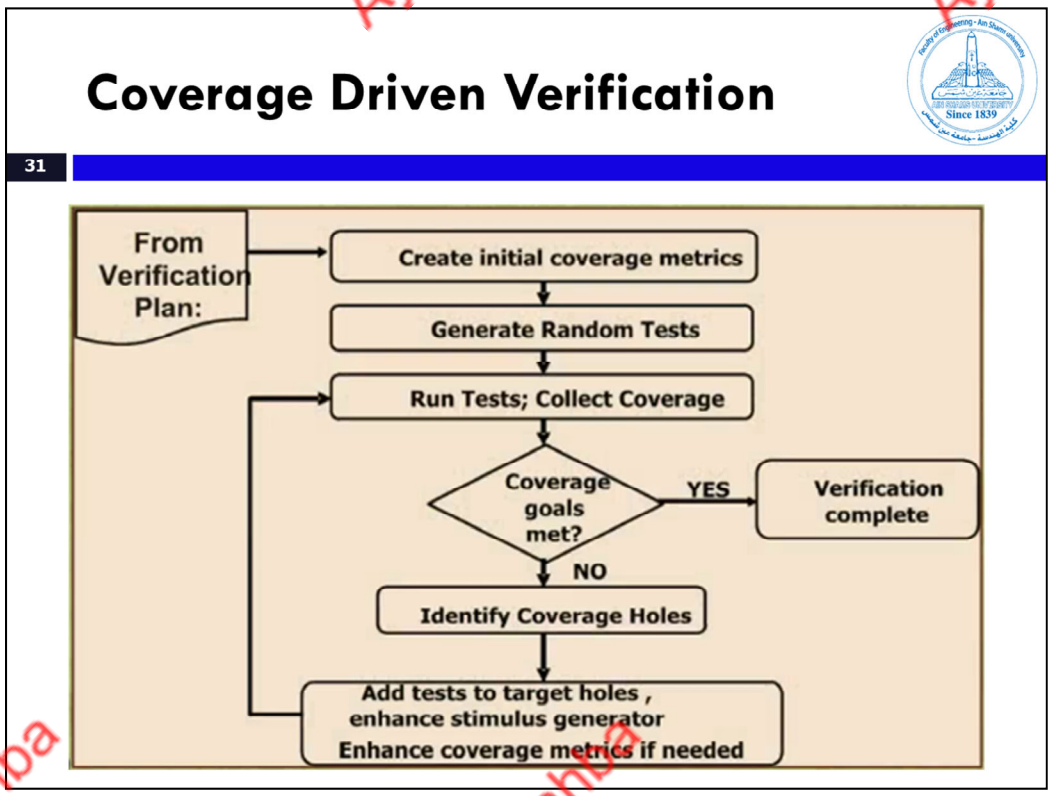
- **Functional coverage:** it is different from code coverage. It covers the functionality of the design. It is derived from the design specification. Tools can't not generate an automatic functional coverage unlike code coverage.

  In this case, based on the design specification, we need to create functional coverage monitors, which a tool uses to extract functional coverage during the simulation.

  For example, we should check whether all the DUT inputs of an operation are correctly injected, and whether all the possible responses are seen at every output port.

  Also doing internal coverage, like whether every interested design events are covered, like FIFO fulls, arbitration mechanisms are being covered.

Coverage Driven Verification



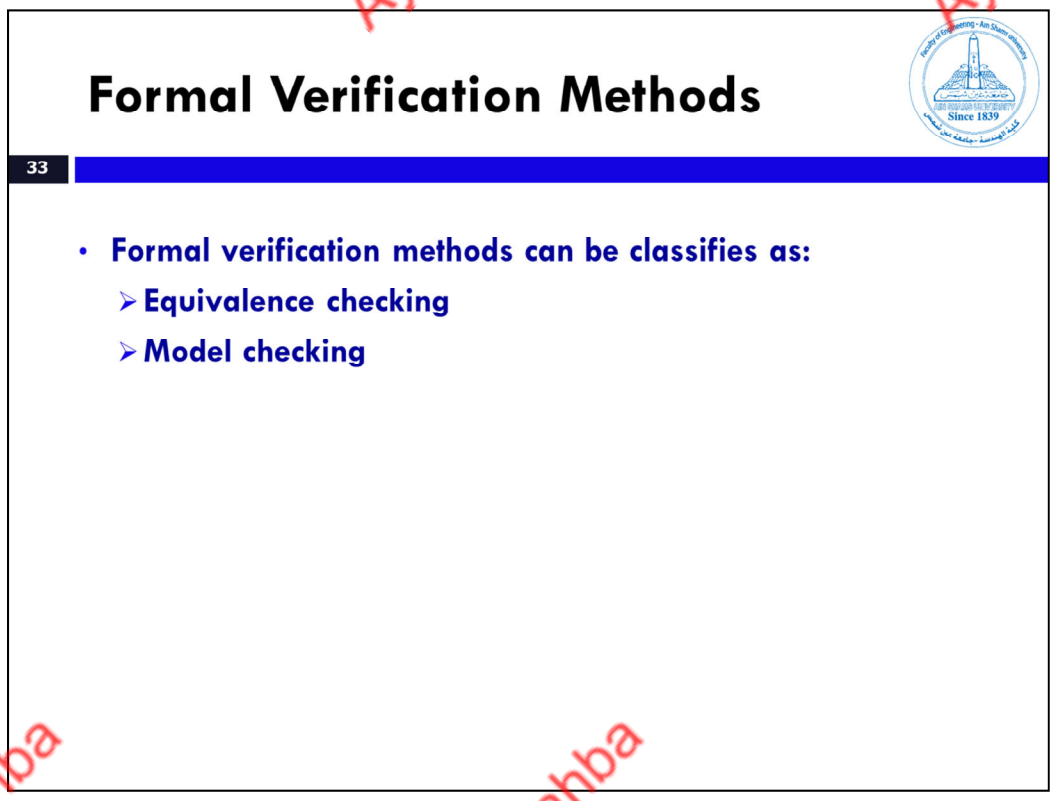
- We will use both code coverage, and functional coverage, especially in constrained random based verification to measure the quality of verification.
- This diagram shows a process when we are making a coverage driven verification. So what we do is to start from a verification plan, and create a set of coverage metrics, that tells you what exactly has to be covered.
- Then we create random tests using test generators
- Then we run the tests and collect the coverage report.
- Then we see whether this coverage meets the goals or not, if so then the verification is complete.
- Otherwise we define the holes (the parts which are not covered), and in order to cover those parts we need to create new tests, or we enhance our test generators to get all the scenarios.
- During this whole process we can enhance the coverage metrics based on what we find, and repeat this process until we get confidence that the DUT has been thoroughly verified and is free of defects before it is manufacrured.

Formal Verification

32

- It is a method by which we prove or disprove a design implementation against a formal specification or property.
- Uses mathematical reasoning.
- It algorithmically and exhaustively explores all possible input values over time.
- Works well for small designs where the number of inputs, outputs and states is small.

- The next most commonly used verification method is formal verification.
- Formal verification is a method by which we prove or disprove a design implementation against a formal specification or a property.
- So, in this method we mostly use mathematical reasoning, or mathematical models and algorithms to prove the equivalence of a design against a specification.
- Formal verification exhaustively explores (using mathematical models) all the possible input values combinations across time, which can exercise all the various state spaces in your design.
- This approach works well for small designs when the number of inputs, outputs and states is small.
- As the design state space starts increasing, formal verification may not be that effective.
- Formal verification methods can be classifies as:
  - Equivalence checking
  - Model checking



- Formal verification methods can be classifies as:
  - Equivalence checking
  - Model checking

Formal – Equivalence Checking

34

- Equivalence checkers can prove/disprove the logical equivalence between the final version of the netlist and the initial RTL.
- Equivalence checking is crucial in the design flow of digital circuits to ensure that optimizations, transformations, or synthesis steps haven't introduced errors.
- Equivalence checker checks for functional equivalence and not functional correctness.

- Formal verification is used within two types of applications. The first is *equivalence checking*.
- Equivalence checking is truly not a functional correctness, but it is an equivalence checking between two kinds of implementations.
- So, equivalence check can check the logical equivalence, or the functional equivalence between say an RTL implementation versus a final version of the netlist. That is the most commonly used application of equivalence checking.
- Equivalence checking is crucial in the design flow of digital circuits to ensure that *optimizations*, *transformations*, or *synthesis steps* haven't introduced errors, or altered the intended behavior of the circuit.

Formal – Equivalence Checking (Example)

35

High-Level RTL description

```
module adder(
  input [3:0] A,
  input [3:0] B,
  output [3:0] sum
  );

  assign sum = A + B;
endmodule
```

?

Gate-level netlist representation

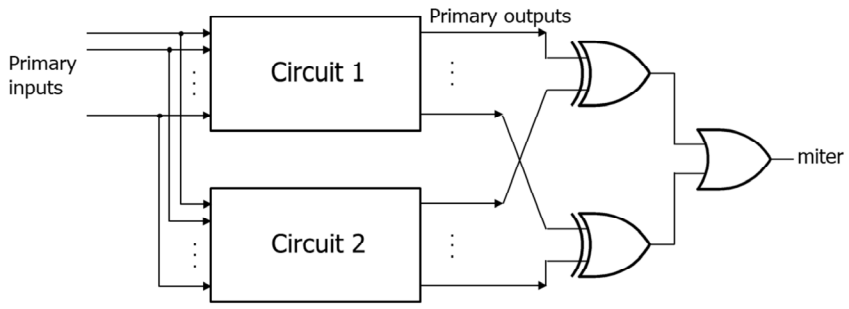
```
module adder_gate_level(
  input [3:0] A,
  input [3:0] B,
  output [3:0] sum,
  );

  wire [3:0] a_xor_b;
  wire [3:0] a_and_b;
  wire [3:0] carry;

  assign a_xor_b = A ^ B;
  assign a_and_b = A & B;
  // Generate carry for each bit
  assign carry = (A & B) << 1;
  assign m = a_xor_b ^ carry;
endmodule
```

We compare the outputs of these two representations for all possible combinations of inputs A and B.

- To perform equivalence checking, we would compare the outputs of these two representations (RTL and gate-level netlist) for all possible combinations of inputs (A and B).
- If the outputs match for every input combination, then we can conclude that the two representations are functionally equivalent.
- If there's a mismatch for any input combination, it indicates a discrepancy that needs to be investigated further.
- For small designs we can do that by simulation.
- For big designs, exhaustive simulation becomes impractical, so we revert to other techniques like the use of *miter circuits*.

Formal – Equivalence Checking (Using a miter circuit)

36



If we can find in input pattern that generates '1' at the output of the miter circuit, then the two designs are not equivalent, otherwise they are equivalent.

- A "miter circuit" typically refers to a circuit used in formal verification processes, particularly in the context of equivalence checking.
- Miter circuits are created by connecting the outputs of two different designs or representations (such as RTL and gate-level netlist) to XOR gates for the sake of comparing their outputs. An XOR gates generate '1' when its two inputs are different.
- The same input values are connected to the inputs of the two designs to be compared.
- If we can find in input pattern that generates 1 at the output of the miter circuit, then there is a discrepancy between the two designs, indicating that they are not equivalent, otherwise the two designs are equivalent.
- To find such input values, we use what is called SAT [1] solvers and/or ATPG techniques.

[1] Kunmei Hu, Zhufei Chu, "An efficient circuit-based SAT solver and its application in logic equivalence checking", Microelectronics Journal, Volume 142, 2023,

Formal – Model Checking

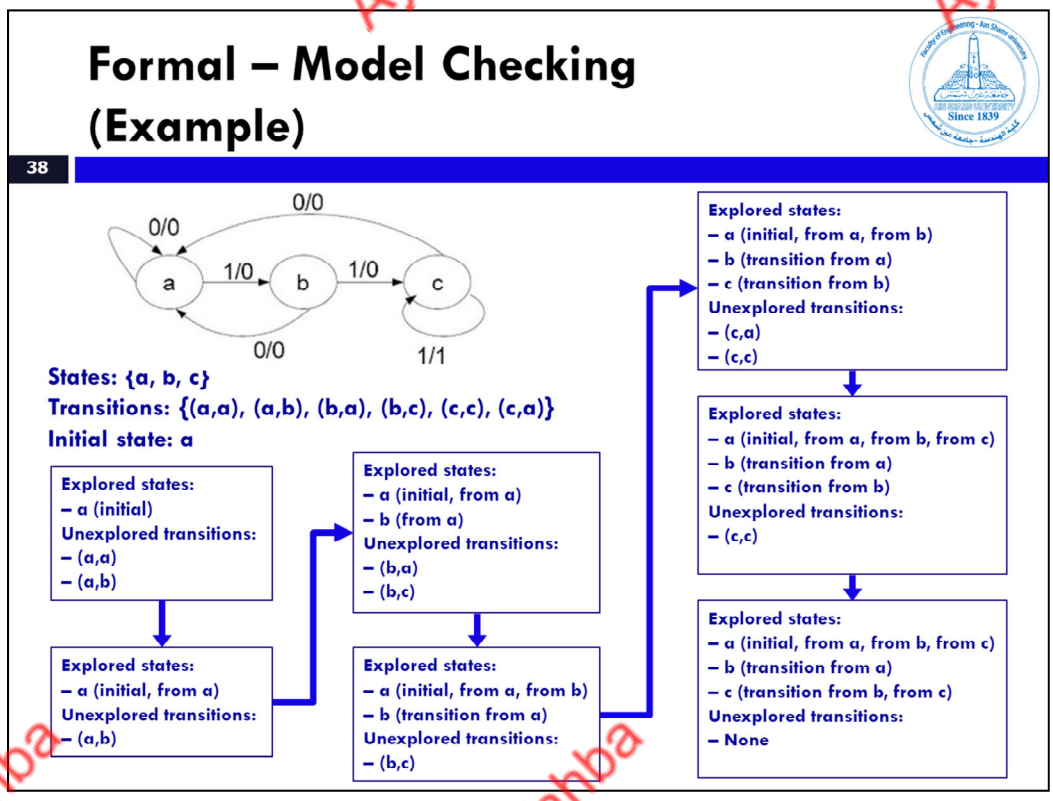
37

- **Model checking is a formal verification technique used to verify whether a system meets a certain property or specification.**
- **Properties are classified into categories such as: *safety*, *liveness*, and *fairness* properties.**
- **Model checking involves exploring all possible states of a finite-state model of the system to determine if there is a state that violates the property to be verified.**
- **If a state that violates the property is found, a *counterexample* is generated.**

- Model checking is a formal verification technique used to verify whether a system meets a certain property or specification.
- It does not check if the whole functionality of the systems. It only checks whether some properties are satisfied or violated.
- For example, one of the properties may be "the door of the list is never opened while the lift is moving". Another property may be "if some one calls the lift from a certain floor, the lift will eventually stop at that floor".
- Properties are classified into categories such as: *safety*, *liveness*, and *fairness* properties.
  - Liveness: it means that the system keeps making progress, responding to inputs, and producing output within a reasonable timeframe. It ensures that the system doesn't get stuck is a deadlock or an unresponsive state indefinitely. (*something good eventually happens*)
  - Safety: it means an undesirable state would never happen.(*something bad would never happen*)
  - Fairness: fairness properties are often applied to concurrent or distributed systems to ensure that all processes or components of the system are given a fair chance to progress or access shared resources.

- Model checking involves exhaustively exploring all possible states of a finite-state model of the system to determine if there is a state that violates the property to be verified.
- If a state that violates the property is found, a *counterexample* is generated. A counter example is an input sequence that shows how the system is put in the violating state. Counterexamples are very useful for debugging purposes.

- Model checking approach is used mostly for designs that are more based on finite state machines.
- The model checker takes two inputs: One is the *finite state machine* representation of the design, and the other input is a *formal representation of some properties* representing the specification (or the properties to be checked).



- Model let's illustrate the state exploration process:
  - Initialization: Start with the initial state.
  - Explore: From the initial state, follow each transition to explore new states.
  - Backtrack: If the new state has already been visited, backtrack and explore other unexplored transitions.
  - Repeat: Continue exploring and backtracking until no more transitions re found.
- This approach ensures that we systematically cover all possible sequences of actions or operations in the finance state machine, including cases where there are multiple possible transitions from the same state.
- If for example, accessing state C from state C violates a property, then the model checker would produce a counter example sequence: 111

Advantages of Formal Verification

39

- **Completeness:** explores all behaviors and corner cases.
- **Rigor:** based on mathematical principles providing rigorous proofs of correctness or violation of properties.
- **Automation:** tools automate the process of verification.
- **Formal Guarantees:** it provides a formal proof of correctness.
- **Coverage:** can achieve higher coverage of design space.
- **Debugging Support:** tools often provide detailed counterexamples or traces when a property is violated.
- **Regulatory Compliance:** In safety-critical industries formal verification is often required to comply with standards and certification processes.

Formal verification and simulation are both valuable techniques for verifying the correctness of hardware designs, but they have different strengths and weaknesses. Here are some reasons why formal verification can be considered better than simulation in certain contexts:

1. Completeness: Formal verification aims to exhaustively explore all possible behaviors and corner cases of a hardware design, providing stronger guarantees about correctness compared to simulation, which typically covers only a subset of possible scenarios.
2. Rigor: Formal verification techniques are based on mathematical principles and logic, providing rigorous proofs of correctness or violation of properties. Simulation, on the other hand, relies on empirical observation and may miss subtle bugs or edge cases.
3. Automation: Formal verification tools automate the process of checking design properties, making it more efficient when compared to simulation, which often requires manual test case generation and analysis.
4. Formal Guarantees: When a property is formally verified to hold for a hardware design, it provides a formal guarantee that the system will behave correctly under all possible scenarios specified by the property. Simulation, on the other hand, cannot offer the same level of formal assurance.
5. Coverage: Formal verification can achieve higher coverage of design space compared to simulation, Simulation may miss certain corner cases or rare events that are covered by formal methods.
6. Debugging Support: Formal verification tools often provide detailed counterexamples or traces when a property is violated, helping designers to understand the root cause of the issue and debug the design effectively. Simulation may provide limited insight into the cause of failures.
7. Regulatory Compliance: In safety-critical industries such as aerospace, automotive, and medical devices, formal verification is often required to comply with stringent regulatory standards and certification processes. Formal methods provide documented evidence of correctness, which is essential for regulatory approval.

Overall, formal verification enhances the reliability, safety, and quality of hardware systems by providing a rigorous and systematic approach to verifying their correctness.

Disadvantages of Formal Verification

40

- **Complexity:** requires expertise in formal methods, mathematical logic, and model checking algorithms.
- **Scalability:** becomes computationally expensive for large and complex hardware designs.
- **State Explosion:** the number of reachable states becomes prohibitively large, making verification infeasible.
- **Limited Expressiveness:** may have limitations in expressing certain types of properties.
- **Modeling Assumptions:** incorrect modeling assumptions can lead to misleading verification results.
- **Resource Intensive:** may require significant computational resources, including memory, CPU time, and specialized hardware accelerators.

While formal verification offers numerous advantages, it also has some limitations compared to simulation. Here are some disadvantages of formal verification:

1. Complexity: Formal verification techniques often require expertise in formal methods, mathematical logic, and model checking algorithms. Developing formal specifications and properties can be challenging, especially for complex hardware designs.
2. Scalability: Formal verification can become computationally expensive, especially for large and complex hardware designs. As the design size increases, the state space grows exponentially, leading to longer verification times and increased memory requirements.
3. State Explosion: For designs with a large number of states or complex control logic, formal verification may suffer from the "state explosion" problem, where the number of reachable states becomes prohibitively large, making verification infeasible within reasonable time and resource constraints.
4. Limited Expressiveness: Formal verification techniques may have limitations in expressing certain types of properties or reasoning about specific aspects of hardware designs. This can result in certain design features or behaviors being difficult to verify formally.
5. Modeling Assumptions: Formal verification relies on accurate and complete models of the hardware design and its environment. Incorrect or incomplete modeling assumptions can lead to verification results that do not accurately reflect the behavior of the actual hardware.
6. Tool Complexity: Formal verification tools can be complex to use and require significant setup and configuration. Users may need to understand the underlying algorithms and parameters used by the tool to achieve meaningful verification results.
7. Resource Intensive: Formal verification may require significant computational resources, including memory, CPU time, and specialized hardware accelerators. This can limit the scalability and accessibility of formal verification tools for certain applications or organizations.

Despite these disadvantages, formal verification remains a valuable technique for verifying the correctness of hardware designs, especially for safety-critical systems where rigorous assurance of correctness is essential. When used judiciously and in combination with other verification methods, formal verification can help identify design errors and improve the reliability and quality of hardware systems.

Semi-Formal Verification

41

- Best of both words (Simulation & Formal)
- Use simulation to reach interesting states in the design
- Then fork off formal to do exhaustive analysis around the interesting state.
- Finds bugs sitting deep in the design
- Useful for bug hunting

- Now let's see what is meant by semi-formal verification.
- That is supposed to be the best of both to simulation and formal words.
- Used especially for designs with large state space where formal verification can't be relied on.
- In this approach what we do is that we initially do simulations to cover most of the interesting states in the design and then around those interesting states of the design we do formal verification to exhaustively cover all the states around those interesting states.
- So, this approach will help us to cover the larger state space faster,
- This helps us to find bugs sitting deep inside the design because we do exhaustive formal verification around those interesting states.

Semi-Formal Verification (Example - Processor)

42

Typical approach used to verify a microprocessor:

1. **Formal methods (e.g. model checking) are used to rigorously verify critical components, such as the *control logic* or *arithmetic units*.**
2. **Informal techniques (e.g. simulation) can be used to validate other aspects of the microprocessor, such as its *performance* under different conditions or its *compatibility with various software* programs.**

- **Combining formal with informal techniques ensures correctness while balancing the resources required for verification.**

- Assume the verification of a microprocessor.
- In semi-formal verification:

1. Formal methods like model checking can be employed to rigorously verify critical components of the microprocessor, such as the *control logic* or *arithmetic units*, ensuring that they adhere to specifications and do not contain any logical errors.
2. Informal techniques like simulation can be used to validate other aspects of the microprocessor, such as its performance under different conditions or its compatibility with various software programs.

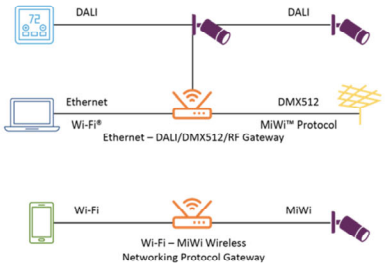
- By combining formal methods for critical components with informal techniques for less critical aspects, semi-formal verification helps ensure the reliability and correctness of the digital system while balancing the resources required for verification.

Semi-Formal Verification (Example – Communication Protocol)

43

Typical approach used to verify a communication protocol:

1. **Formal methods can be used to verify *key properties* of the protocol, such as its *correctness*, *safety*, and *liveness* properties.**
   - **This might involve formal specification languages, along with model checking or theorem proving techniques.**
2. **Informal methods, such as simulation, can be used to assess the protocol's under various network conditions, its *scalability*, and its *robustness* against common network issues like packet loss or delays.**



- Assume the verification of a communication Protocol.
- In semi-formal verification:

1. Formal methods can be used to verify *key properties* of the protocol, such as its *correctness*, *safety*, and *liveness* properties. This might involve formal specification languages, along with model checking or theorem proving techniques.
2. Informal methods, such as simulation, can be used to assess the protocol's *performance* under various network conditions, its *scalability*, and its *robustness* against common network issues like packet loss or delays.

Assertion-based Verification

44

- **What is an Assertion?**
  - An Assertion is a statement about a design's intended behavior, which must be verified
- **What is Assertion-Based Verification (ABV)?**
  - it involves using assertions to verify the correctness of the system under test.

- Now let's see what is assertion based verification.
- An Assertion is a statement about a design's intended behavior, which must be verified, or in other words, it is a way of capturing the design intention or part of design specification in the form of a *property*.
- This property can be used along with your normal dynamic simulation or along the formal verification to make sure that specific intention is being met or not met.

How ABV works?

45

- **Assertions:** Engineers embed assertions directly into the HDL code to specify properties that must hold true.
- **Verification:** During simulation, these assertions are continuously evaluated. If an assertion evaluates to false it indicates a violation of the specified property.
- **Debugging:** When an assertion fails, engineers trace back to identify the root cause of the failure.
- **Coverage:** Assertions can also be used to measure coverage, indicating how thoroughly the design has been tested against specified properties.

• Here's how it works:

1. Assertions: Engineers embed assertions directly into the hardware description language (HDL) code to specify properties that must hold true during simulation, synthesis, or formal verification.
2. Verification: During simulation, these assertions are continuously evaluated. If an assertion evaluates to false during simulation, it indicates a violation of the specified property, highlighting a potential bug or design flaw.
3. Debugging: When an assertion fails, engineers can trace back through the simulation to identify the root cause of the failure, aiding in debugging and refining the design.
4. Coverage: Assertions can also be used to measure coverage, indicating how thoroughly the design has been tested against specified properties.

Assertion-based verification offers several advantages, including early detection of design errors, improved debug efficiency, and enhanced verification completeness by explicitly specifying design properties. It's widely used in the verification of complex hardware designs to ensure their correctness and reliability.

Advantages of Assertions

46

- **Error Detection:** Assertions help in detecting errors or bugs in the design early in the verification process.
- **Debugging Aid:** When an assertion fails, it provides valuable about the design state at that moment, aiding in debugging.
- **Verification Closure:** By verifying that all assertions hold true, engineers can achieve verification closure, ensuring that the design meets its requirements.
- **Documentation:** Assertions serve as executable documentation of the design's intended behavior.

- The benefits of assertions:
  - Error Detection: Assertions help in detecting errors or bugs in the design early in the verification process. By defining properties that must hold true at specific points in the design, assertions can identify violations of these properties, indicating potential issues.
  - Debugging Aid: When an assertion fails during verification, it provides valuable information about the state of the design at that moment, aiding in debugging. This helps engineers quickly pinpoint the cause of the failure and fix it.
  - Verification Closure: Assertions provide a means to specify the intended behavior of the hardware design comprehensively. By verifying that all assertions hold true, engineers can achieve verification closure, ensuring that the design meets its requirements.
  - Formal Verification: Assertions are essential for formal verification techniques, where mathematical methods are used to prove or disprove the correctness of a design. Formal verification relies heavily on assertions to express properties that must be satisfied by the design.
  - Documentation: Assertions serve as executable documentation of the design's intended behavior. They provide insights into the design's requirements and constraints, making it easier for engineers to understand and maintain the design.

Overall, assertions play a crucial role in ensuring the correctness, reliability, and maintainability of hardware designs by facilitating early error detection, aiding in debugging, and enabling comprehensive verification.

Disadvantages of Assertions

47

- **Overhead:** Assertions require additional resources, including processing power and memory, to execute during verification.
- **Complexity:** Writing and maintaining assertions is complex, especially for designs with numerous states and behaviors.
- **False Positives/Negatives:** Assertions may produce false positives (incorrectly flagging an error) or false negatives (failing to detect an actual error).
- **Dependency on Testbench:** They rely on the testbench to provide stimuli and monitor the behavior of the design.
- **Limited Coverage:** Assertions may not cover all possible scenarios or corner cases in the design.
- **Learning Curve:** Engineers may require training and experience to effectively use assertions in hardware verification.

- Overhead: Assertions require additional resources, including processing power and memory, to execute during verification. This overhead can impact simulation performance, especially for designs with a large number of assertions.
- Complexity: Writing and maintaining assertions can be complex, especially for intricate designs with numerous states and behaviors. Ensuring that assertions accurately capture the intended properties of the design requires careful specification and validation.
- False Positives/Negatives: Assertions may produce false positives (incorrectly flagging an error) or false negatives (failing to detect an actual error). Achieving the right balance between sensitivity and specificity in assertion checks can be challenging and may require iterative refinement.
- Dependency on Testbench: Assertions rely on the testbench environment to provide stimuli and monitor the behavior of the design. Changes to the testbench or environment may necessitate corresponding modifications to assertions, increasing maintenance overhead.
- Limited Coverage: Assertions may not cover all possible scenarios or corner cases in the design. Achieving comprehensive assertion coverage requires careful analysis and may not always be feasible, leaving gaps in the verification process.
- Learning Curve: Engineers may require training and experience to effectively use assertions in hardware verification. Understanding assertion syntax, semantics, and best practices can take time, particularly for those new to formal verification techniques.

Despite these disadvantages, the benefits of using assertions generally outweigh the drawbacks, as they provide critical support for error detection, debugging, and verification closure in hardware design projects.

Assertion Types

48

- **Immediate Assertion**

```
assert (A == B) else $error("it's gone wrong");
```

- **Concurrent Assertions:**

```
property p1;
  @(posedge clk) disable iff (Reset) not b ## 1 c;
endeproperty

assert property p1else $error ("not B ##1 C failed")
```

- There are two main assertion types in SystemVerilog:

➢ Immediate Assertions: these assertions are evaluated continuously during simulation. Immediate assertions are typically used to verify conditions, such as checking the value of a signal or the relationship between multiple signals.

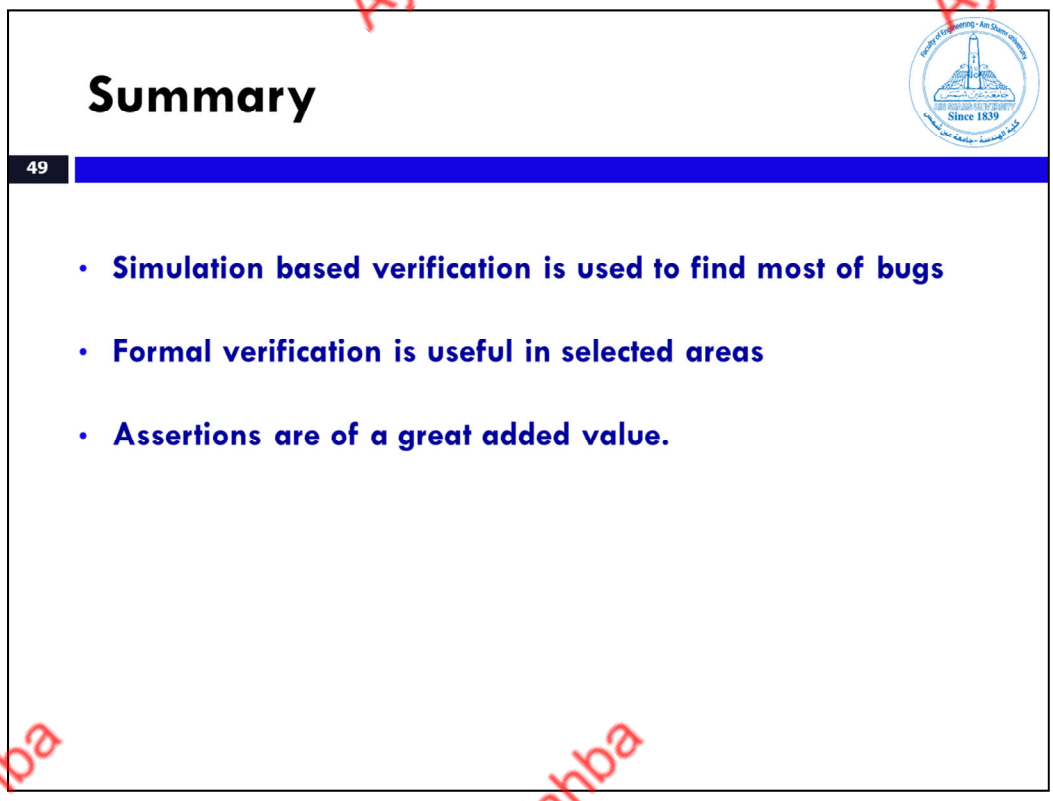
So in the shown example we see:

assert (A == B) else $error("it's gone wrong");

So this is an immediate assertion which means; if at any point of time A and B are not equal then that's kind of a violation of the design intention and an error message is issued.

➢ Concurrent Assertions: Concurrent assertions are used to specify temporal properties that must hold true at specific points of time.

So, in the shown example, you are writing a property p1 which is checked at every +ve edge of the clock, as long as Reset is not high (disabe iff Reset). The thing to be checked is that if b is not 1, then after one cycle c must be true. If this does not happen then the property fails, and an error message is displayed.



- In summary we have seen the following:
- Simulation based verification, which is still the most commonly used verification method to find most of the bugs.
- Formal verification is used in selected areas for finite state machine based designs and specially when the state space is small.
- Assertions are also of a great value, that can be used with both dynamic simulations and formal verification.