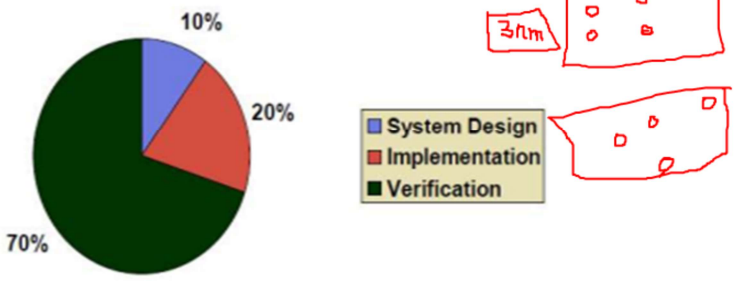
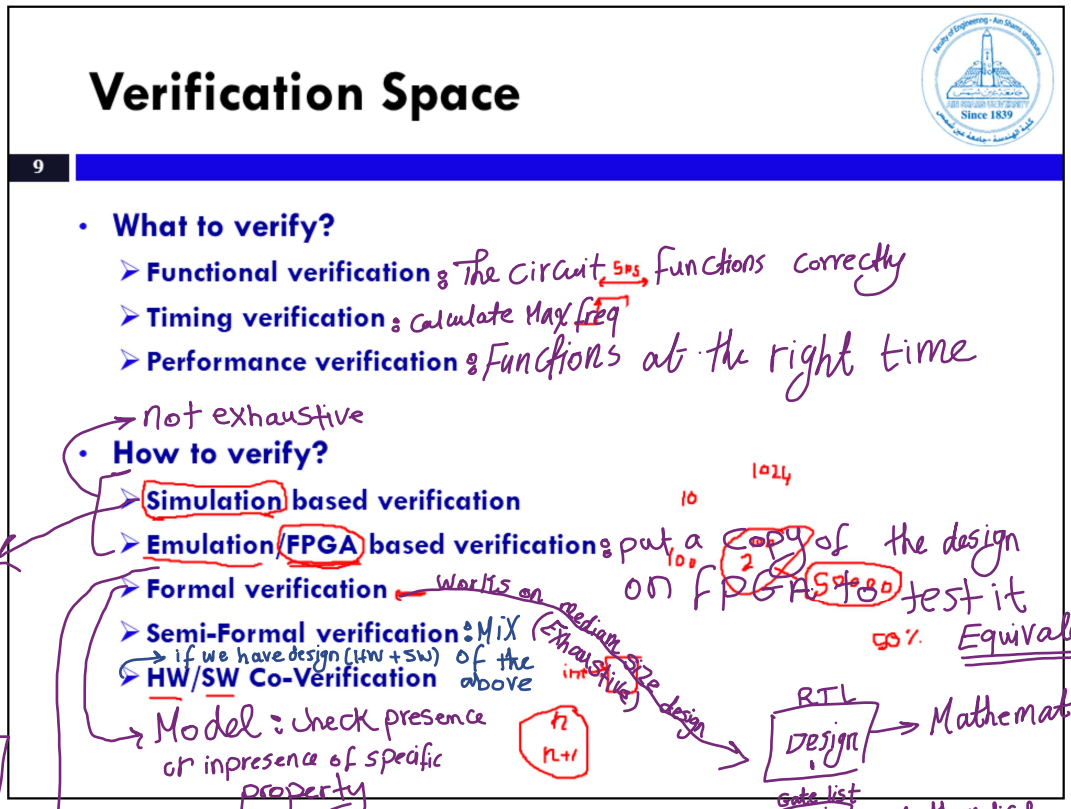
Importance of Verification

8

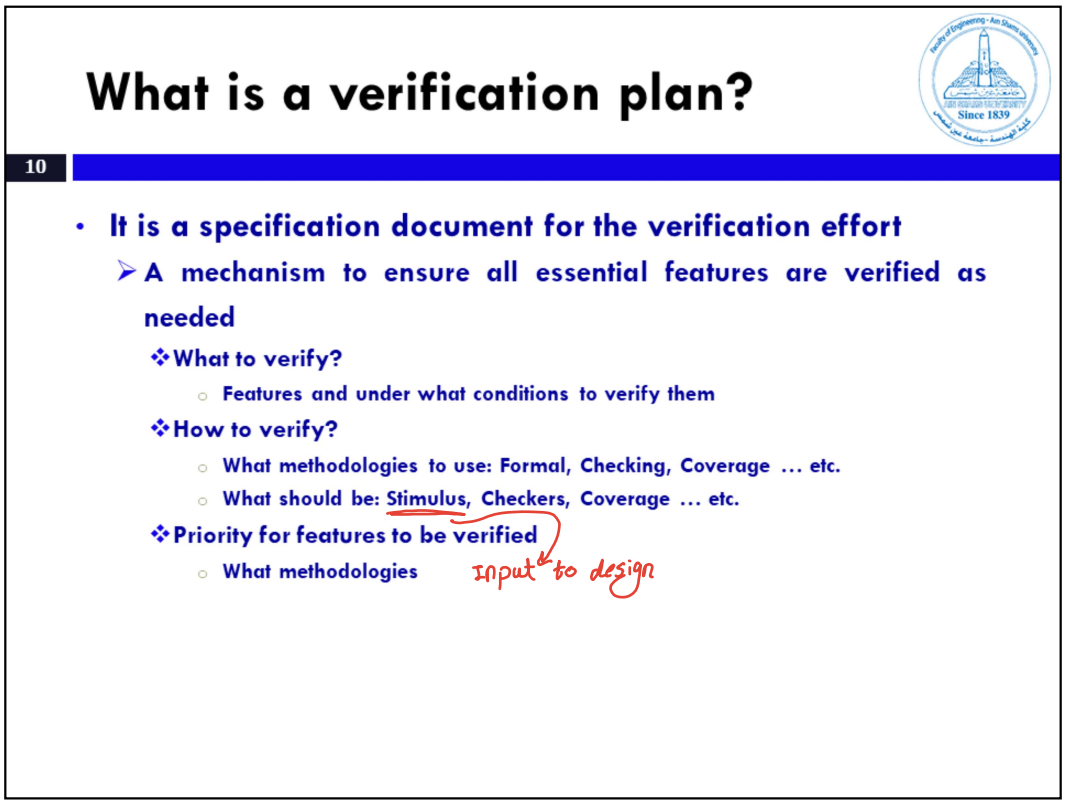
- Bug escapes to silicon can be costly including re-spin
- 70% od design cycles is spent on verifying design
- Ever increasing complexity of designs makes this harder
- Hence Verification is always on the critical path for any product design



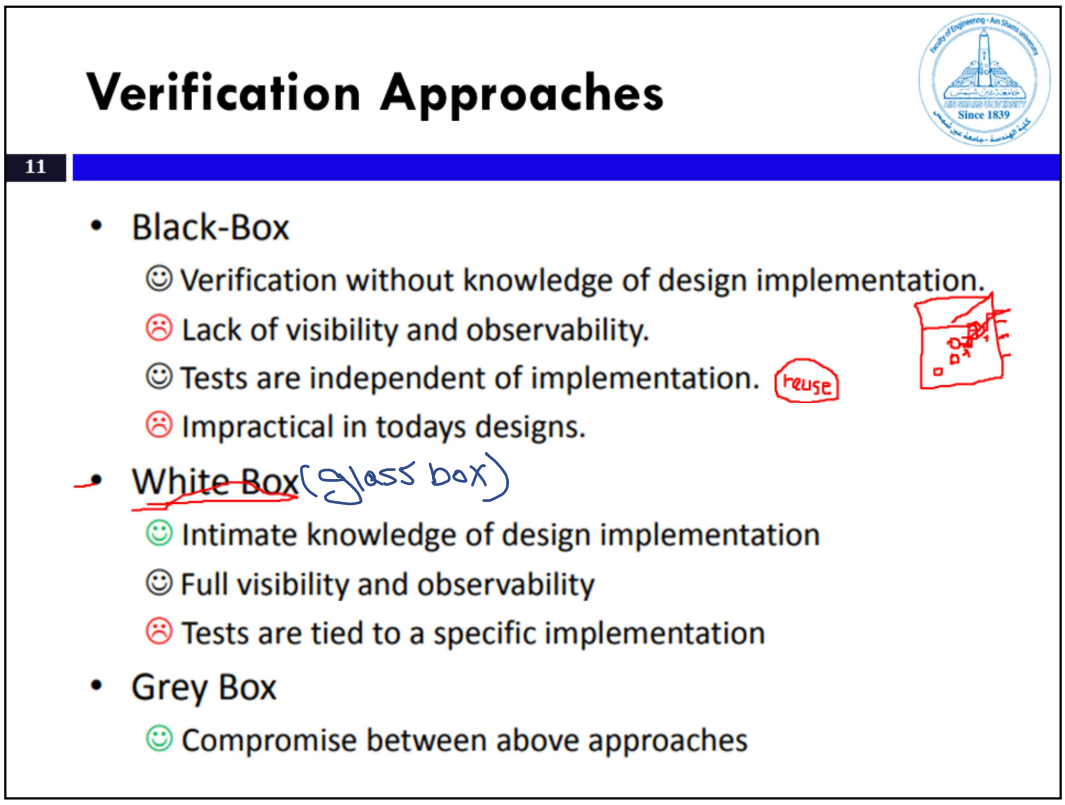
- It is important to understand that any bug that escapes the design process to the silicon can be very costly and can also result in a re-spin of the same chip. Unlike the software which can have a batch to solve any bugs even after production. Any bug that escapes into the silicon in a chip design can be very costly, that is why it is also important to understand the importance of verification overall the chip design process.
- Statistical data shows that around 70% of the project time is spent in verification.
- This is very important specially in the ever increasing complexity of different designs.
- In every design project Verification is on the critical path for the product design.
- Data collected from the industry shows that time spent in the system design represent only 10% of the project time. The implementation takes 20% of the project time, while 70% of the time is needed for the verification.



- Next question is what really to be verified (this is known as verification space)
- There are three aspects to the verification space:
  - Functional verification: a process focusing more on whether the functionality of the design is verified. This kind incudes 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 what is the maximum frequency on which the design can operate.
  - Performance verification: is 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.
  - The Emulation or FPGA based verification.
  - Formal verification
  - Semi-formal verification
  - HW/SW Co-verification
- We will learn about all these methods in future lectures.

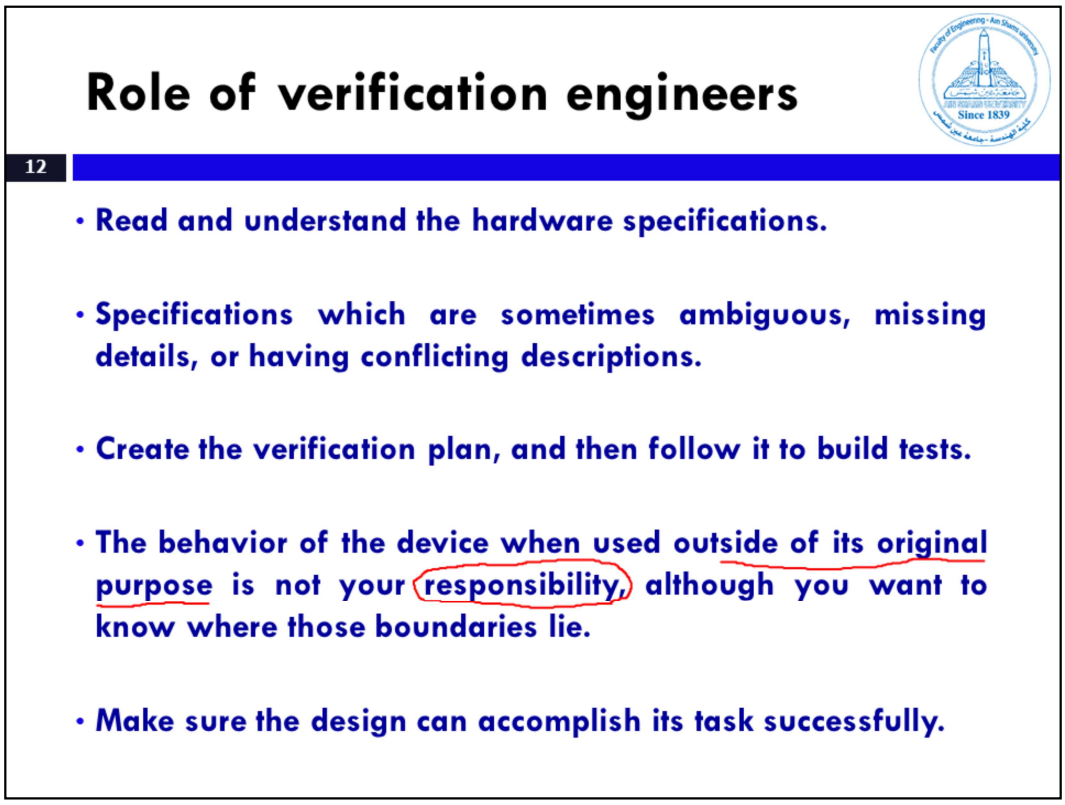


- So, what is a verification plan ? It is a specification document for the verification effort.
- It is a documentation that captures the plan you will follow to execute the verification, just like any other plan you do to get your tasks done efficiently.
- It is a mechanism to ensure all the features 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 will 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.



- 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 and not like what was followed in most of the verifications.
  - **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 every thing 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.

UVM ⇒ universal verification methodology



- The goal of hardware design is to create a device that performs a particular task, such as a DVD player, network router, or radar signal processor, based on a design specification.
- Your purpose as a verification engineer is to make sure the device can accomplish that task successfully – that is, the design is an accurate representation of the specification.
- Bugs are what you get when there is a discrepancy.
- The behavior of the device when used outside of its original purpose is not your responsibility, although you want to know where those boundaries lie.
- The process of verification parallels the design creation process.
- There is always ambiguity in interpreting the specifications, perhaps because of ambiguities, missing details, or conflicting descriptions.
- As a verification engineer, you must read the hardware specification, create the verification plan, and then follow it to build tests showing the RTL code correctly implements the features.

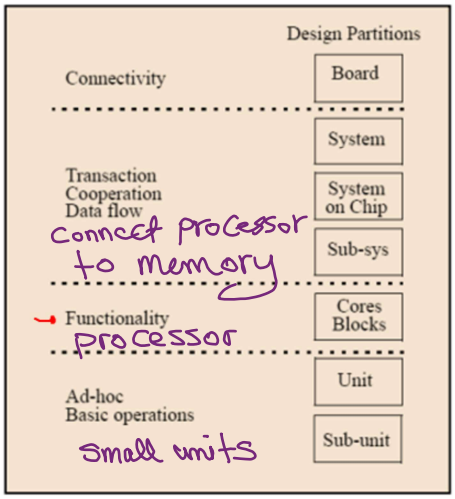
Levels of Verification

13

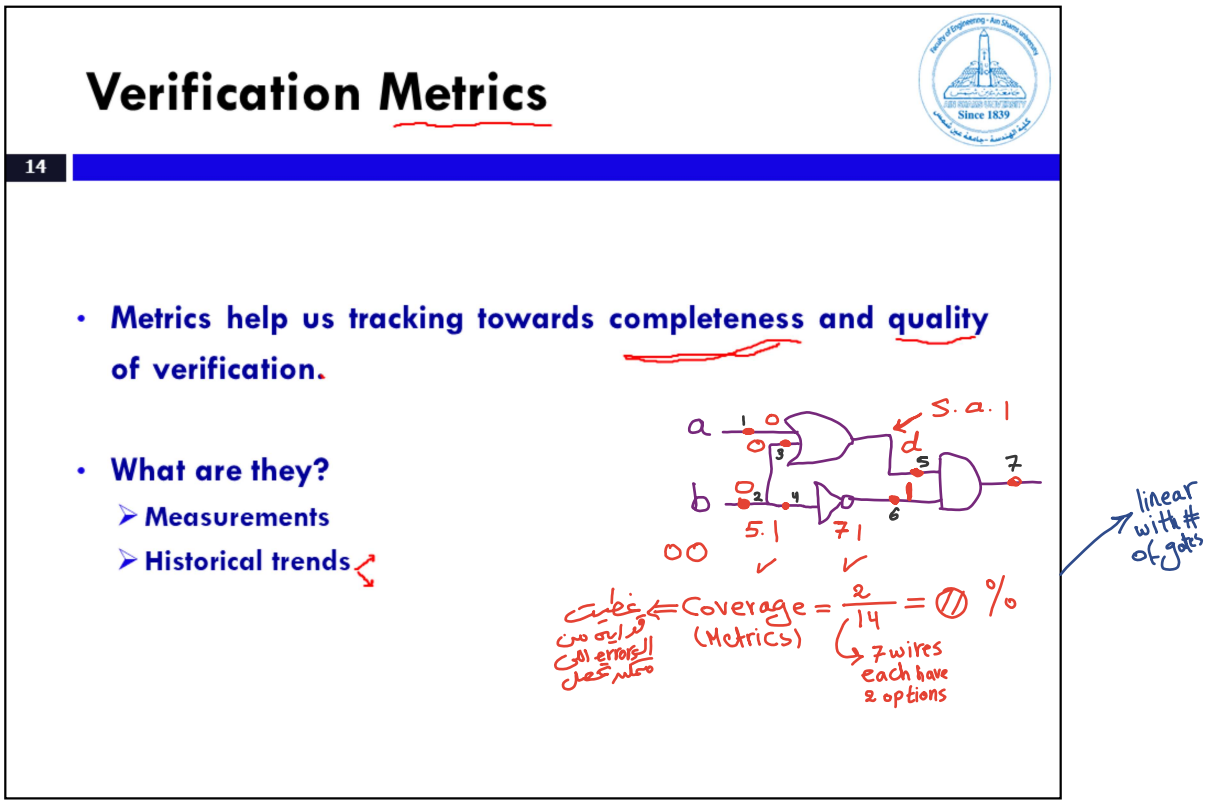
Bottom-up approach

- 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

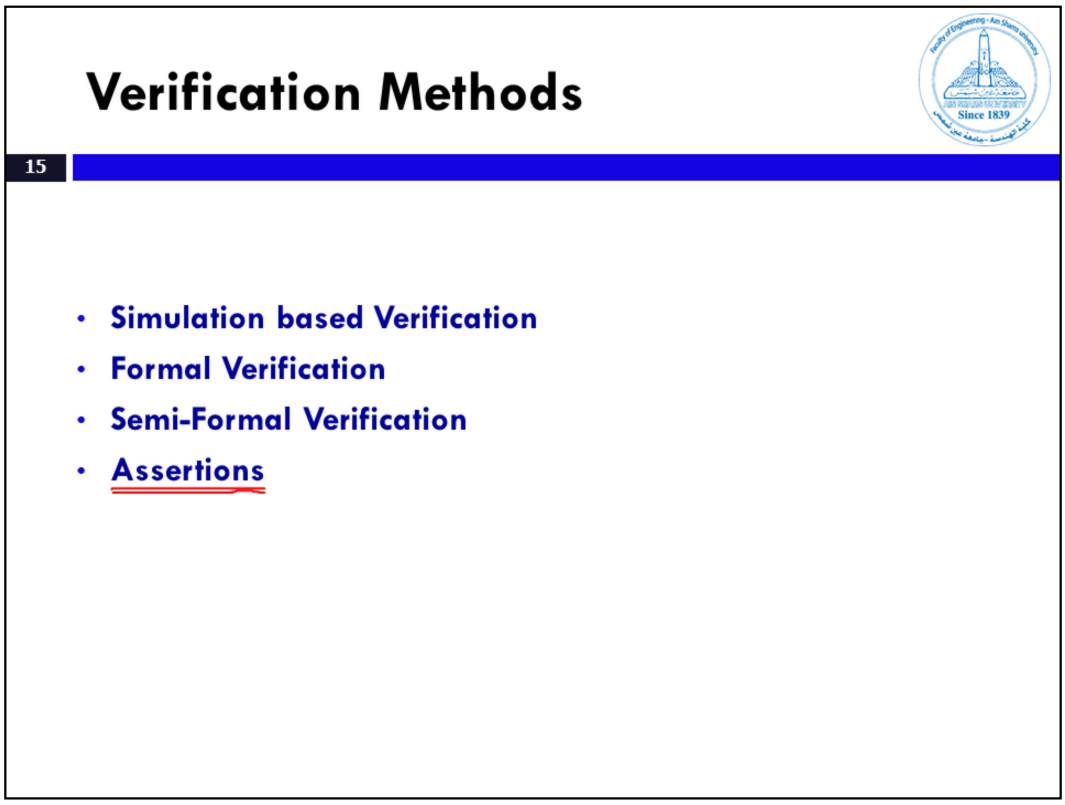
Test each level independently



- The design shows how the design is partitioned during the product design phase. 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 comes 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 have your verification partitioned into levels.
- Each level of verification will be suited for a specific objective.
- Lower levels when the design starts as a sub-unit, or unit, or core IP blocks, you can do a verification of that specific unit in a stand alone environment.
- 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.



- 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. So we have verification metrics which track how many tests are being returned 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.

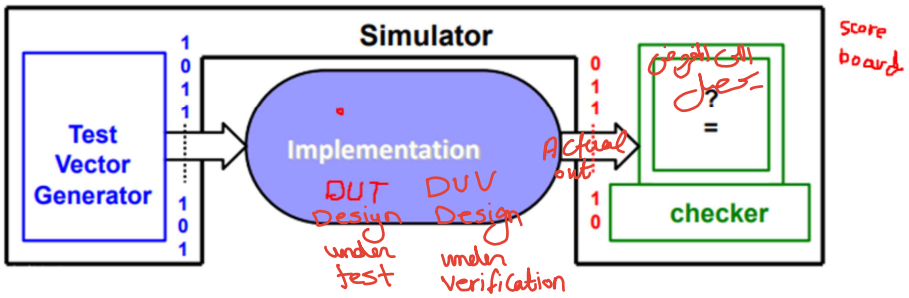


- We will learn some of verification methods that are most commonly used.
- We will learn about simulation based verification
- We will learn about formal verification
- We will learn about semi-formal verification
- We will also learn about assertions, and how they will be used in verification.

Simulation based Verification

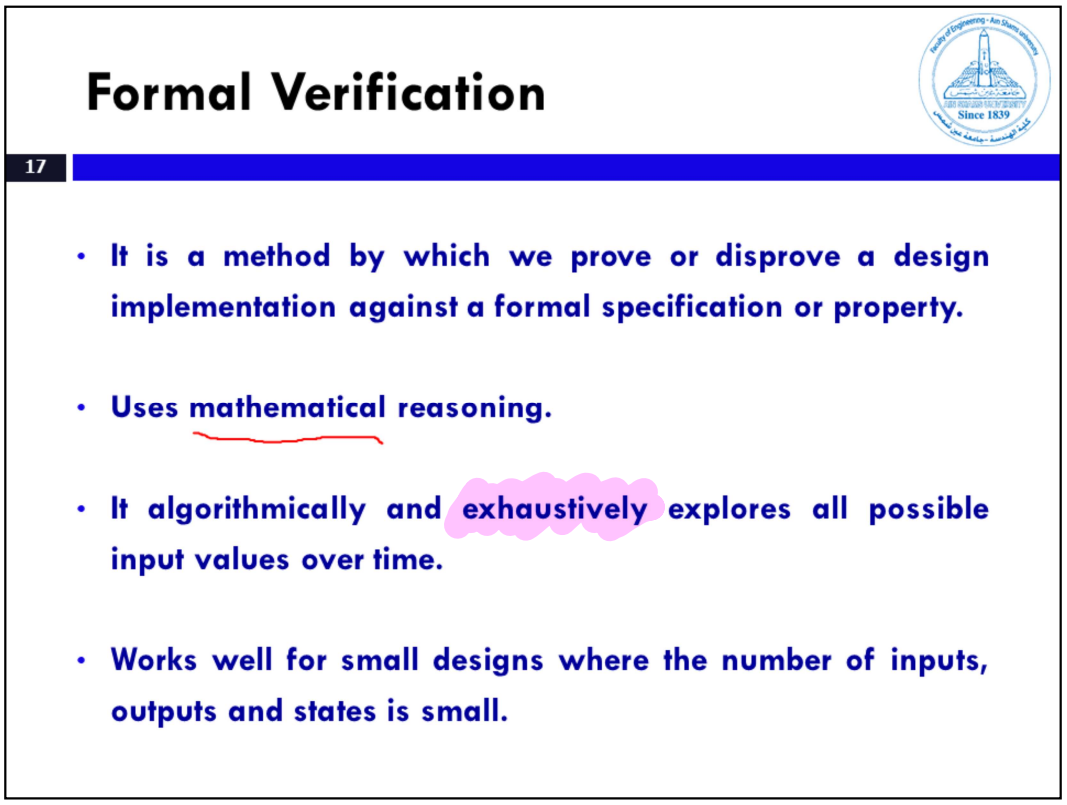
16

- Generate Input sequences [stimulus, test vector]
  - 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



*Coverage detects number of patterns generated

- 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 as a test vector generator to generate input stimulus to the design and 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 like random, or constrained random test generator.
- We also use several approaches like assertions and coverage metrics to make sure the verification is complete and also the quality it met.
- We will see most of the details about that in the following lectures.

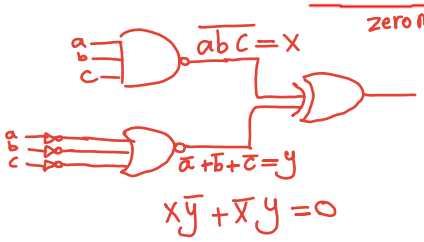


- The next most commonly used the verification method is formal verification.
- Formal verification is a method by which we prove or disprove a design implementation against a formal specification or a property.
- So in this method we mostly use mathematical reasoning, or a mathematical models and algorithms to prove the equivalence of a design against a specification.
- Formal verification exhaustively explores 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 – Equivalence Checking

18

- Equivalence checker checks the logical equivalence of the final version of the netlist with the RTL.
- Equivalence checker checks for functional equivalence and not functional correctness.



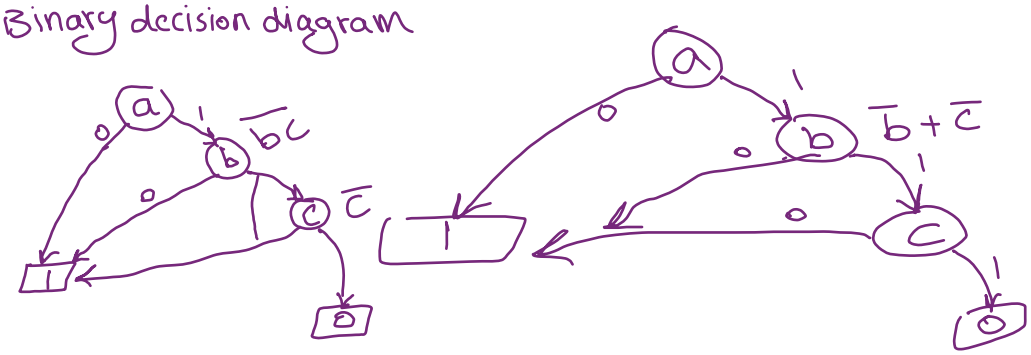
zero means no errors

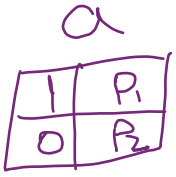
$\overline{abc}.\overline{(\bar{a}+\bar{b}+\bar{c})}$ DeMorgan
$+abc.(\bar{a}+\bar{b}+\bar{c})$ +
$=(\bar{a}+\bar{b}+\bar{c}).(abc)+0$ Distribution
$=0+0=0$

- Formal verification is used for 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.

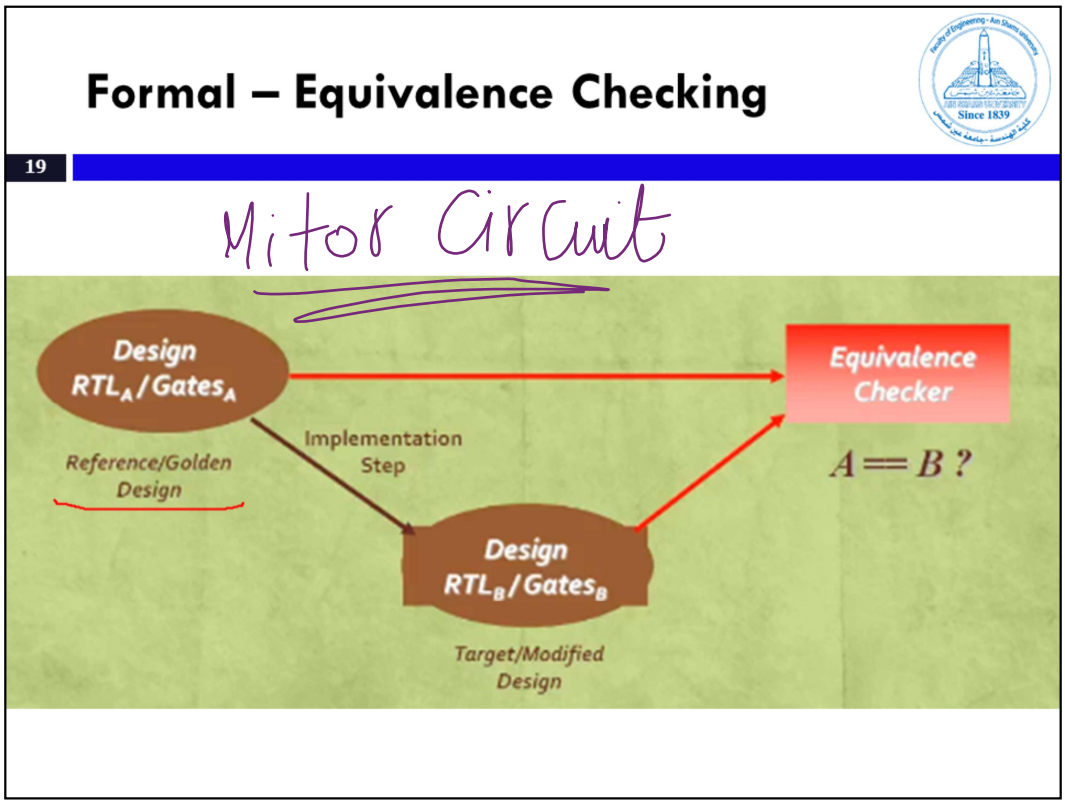
same order

BDD = Binary decision diagram

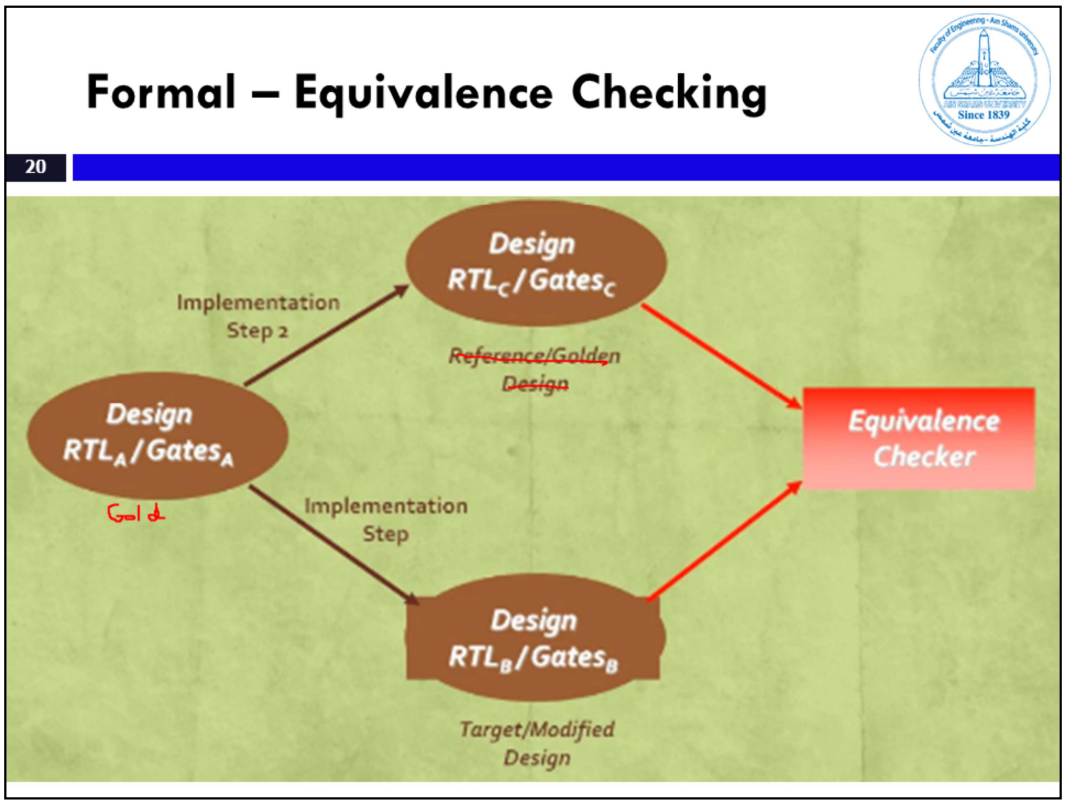




Compare pointers
Not values



- This diagram shows how the equivalence checking is done.
- $RTL_A/Gate_A$ is a kind of a reference or golden design.
- During the implementation, we implement another design $RTL_B/Gate_B$.
- So this can be a design just modified for certain library or it's modified for certain things.
- Then we do equivalence checking between A and B to make sure they are functionally equivalent.
- I doesn't guarantee both are correct, it just guarantees that they are equivalent.



- Similarly, we can have another kind of implementation for the same design form the reference model.
- We can also do an equivalent check between, say in this case, design C and design B.

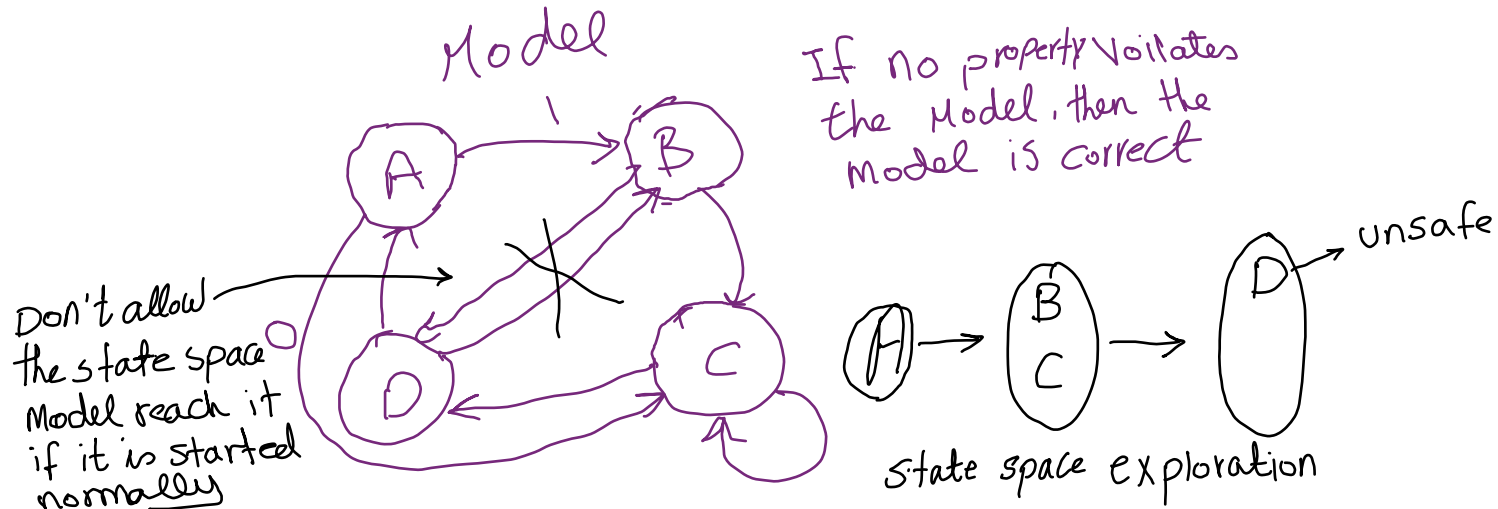
Formal – Model Checking

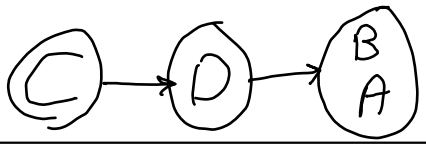
21

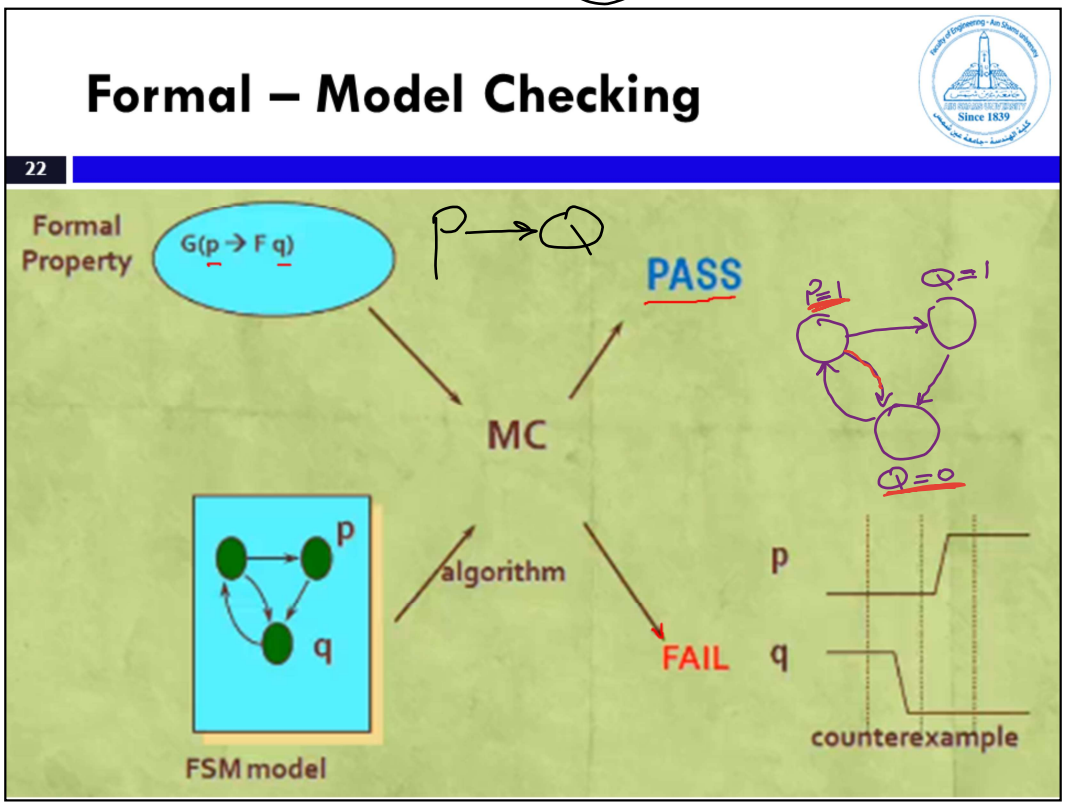
- Exhaustive search of FSM for state/property violations
- Two inputs to the algorithm

- A finite state transition system (FSM) representing the implementation.
- A formal property representing the specification
- The algorithm checks whether the FSM "models" the property correctly.

- The next most commonly used application for formal verification is model checking.
- Model checking is a way where we exhaustively search for finite state machines for state or property violations.
- So this approach is used mostly for designs which are more based on finite state machines.
- In this approach what we do is that we take two inputs to the system. One is the finite state transitions system or an FSM representation of the design, and the other is a formal property representing the specification.
- So these two inputs are taken to the algorithm, and the algorithm exhaustively looks for the input value combinations which lead to violating the property in the finite state machine model. If it doesn't find any input value combination that violate the property we can say the finite state machine is formally verified.







- This diagram explains the concept of model checking.
- Here we have a finite state machine based design.
- We also implement the formal property specification of the same design.
- These two inputs are taken to the model checking algorithms which are typically implemented in most of the formal verification tools.
- The tool will algorithmically explore all the possible input values combinations that can exercise all the different state space in the design, and then it can come up with counter-examples or combinations of input which can fail the design. If it can't find any combination that fails the design, then we can say that the design is formally verified.

Advantages of Formal Verification

23

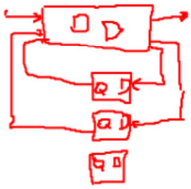
- Covers exhaustive state space, something difficult to achieve by simulation.
- Does not require generation of input stimulus, since exhaustive stimulus can be generated automatically by tool.
- No need to generate expected output sequences.
- Correctness of design guaranteed mathematically.

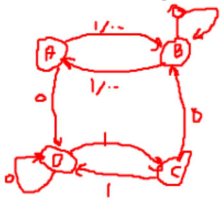
- So what are the advantages of formal verification ?
- It explores exhaustively all the state space. This is something that is very difficult to do by simulation as we need to manually create tests to hit all the combinations.
- So formal verification has the advantage that we don't have to generate any input stimulus, since the tool does the generation automatically.
- We also don't need to generate any expected output sequence, unlike simulation based where we create all the checker's scoreboard reference golden outputs.
- Also correctness of a design is guaranteed mathematically, that is also an advantage.

Disadvantages of Formal Verification

24

- **Scalability**
  - Each added flip/flop doubles the state space.
  - Limited to designs with small state spaces like interfaces, and small FSMs.
- **What properties to verify**
  - The specifications need to be translated into properties to verify that these properties hold under all inputs and all states.





- One of the disadvantages of formal verification is that it is not scalable for larger designs. Each additional flip/flop doubles the sate space. The formal verification tool has to exhaustively cover more combinations.
- So, formal verification is limited to designs with small state spaces like interfaces, whereas for larger designs, this doesn't really scale well.

- Also, for every specification we need to translate the specifications into some kind of properties, in some kind of property specification language. That also become harder if the state space keeps increasing.
- So because of that, formal verification approach works well for smaller designs.

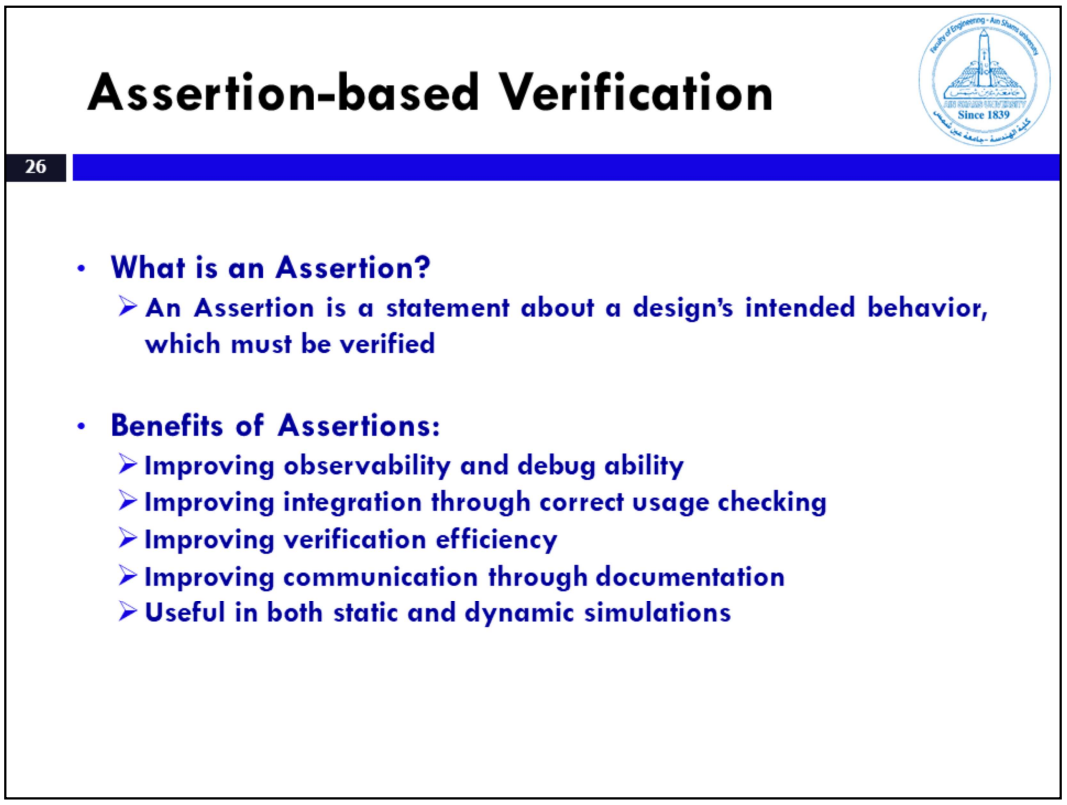
Semi-Formal Verification

25

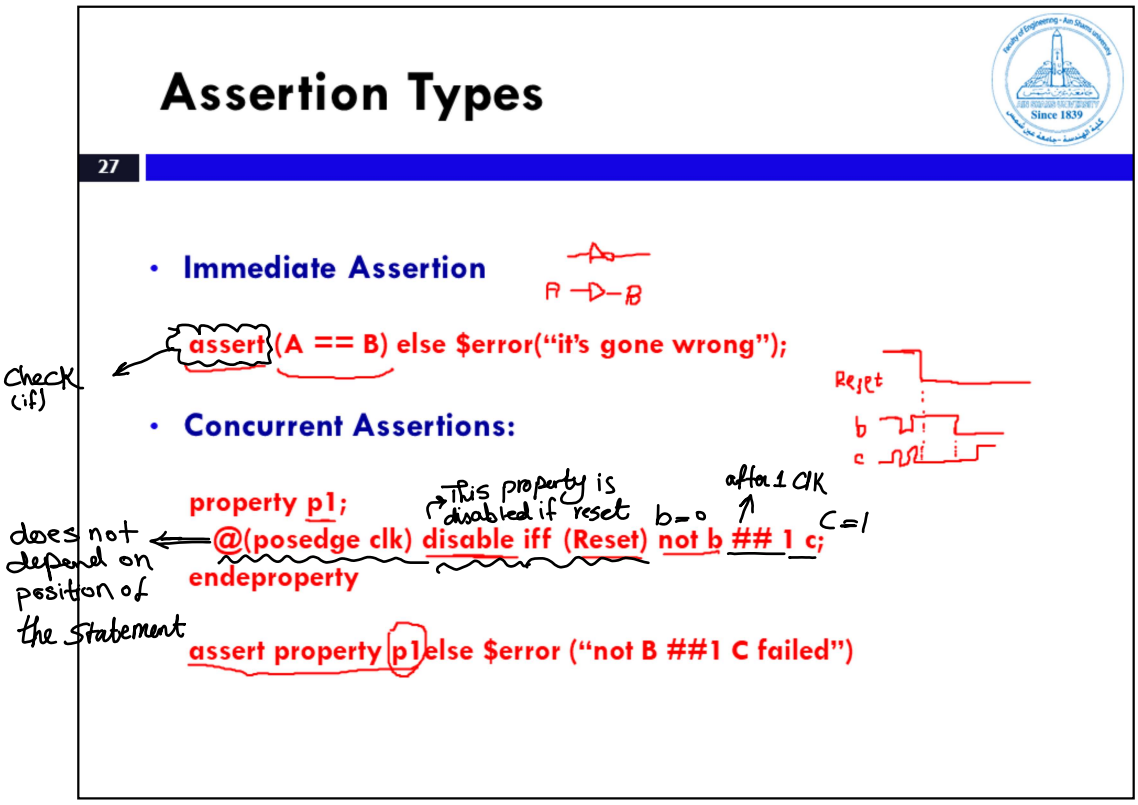
- Best of both words (Simulation & Formal)
- Use simulation to reach interesting states in the design
- Then fork off formal to do exhaustive analysis around the interesting state.
- Finds bugs sitting deep in the design
- Useful for bug hunting

Simulation first to reach a specific state then use formal verification.

- 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.
- So 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,
- And also helps us to find bugs sitting deep inside the design because we do exhaustive formal verification around those interesting states.

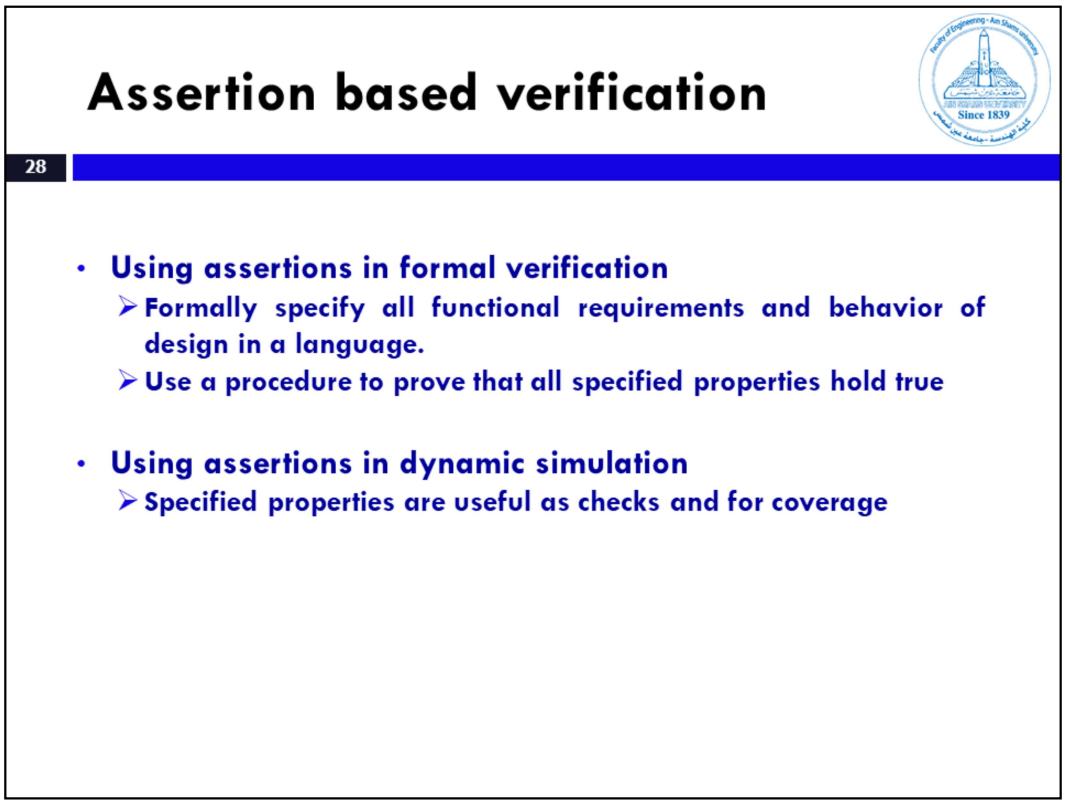


- Now let's see what is assertion based verification.
- An Assertion is a statement about a design's intended behavior, which must be verified, or in other words, it is a way of capturing the design intention or part of design specification in the form of a property.
- This property can be used along with your normal dynamic simulation or along the formal verification to make sure that specific intention is being met or not met.
- The benefits of assertions:
  - It improves the observability and the debug ability of the verification process, because now the design intention is captured in the form of a property you know whether exactly that specific property has happened or not during your debug.
  - It improves the integration through correct usage checking. This benefit is useful when there are multiple units designed by multiple people, and those units are then being integrated in a top level design. Each of the interface assertions for these specific units will help to make sure that after integration the design still works properly.
  - All of these advantages will improve the verification efficiency because a lot of the time that is spent in debugging the system can be saved.
  - It does improve the communication, as capturing the design intentions as assertions, does add to your documentation.
  - And as I mentioned it's useful for both static verification and dynamic simulation.



- Let's see what they defend that assertion. There are basically two types of assertions.
- **Immediate assertion:** it is something that will be checked at all times. So in this example we see assert (A == B) otherwise there is error "it's gone wrong". So this is an immediate assertion, because at any given point of time if A and B are not equal then that's kind of a violation of the design intention and then it can give you an error message.
- **Concurrent assertion:** it is basically where you can create an assertion which is checked at certain points in time. So in this case you are writing a property p1 which tells that at the +ve edge of clock, disable if and only if Reset otherwise not(b) followed by one cycle c.
- So what it means is that whenever Reset is not asserted, then when B is not 1, then after one cycle c has to be 1. If this doesn't happen then there will be a failure of this property.

So these are two types of assertions



- Assertion base verification is a method where assertion are used.
- The method can be used for formal verification as well as in dynamic simulation.
- Basically all the formal properties can be written using assertions and use this procedure to prove that all the specified properties hold true. So that is what formal verification does.
- Assertions can be used in dynamic simulations.
- These properties or assertions become useful as checkers during your simulation, , and can also as coverage to make sure that those conditions are being good.

Summary

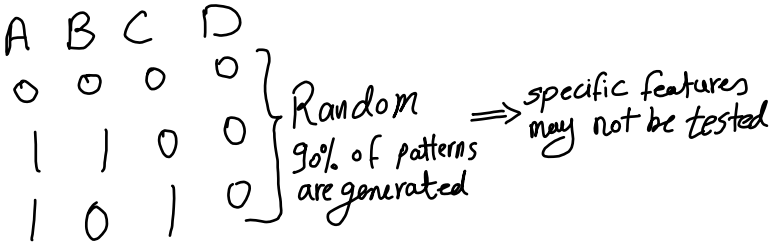
29

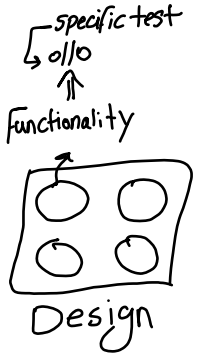
- Simulation based verification is used to find most of bugs
- Formal verification is useful in selected areas
- Assertions are of a great added values.

small problems only

we know which input caused the error

- 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.

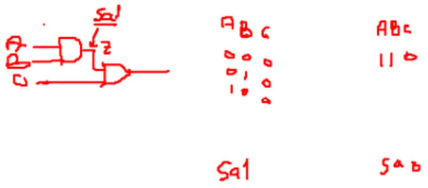




Direct vs Random Testing

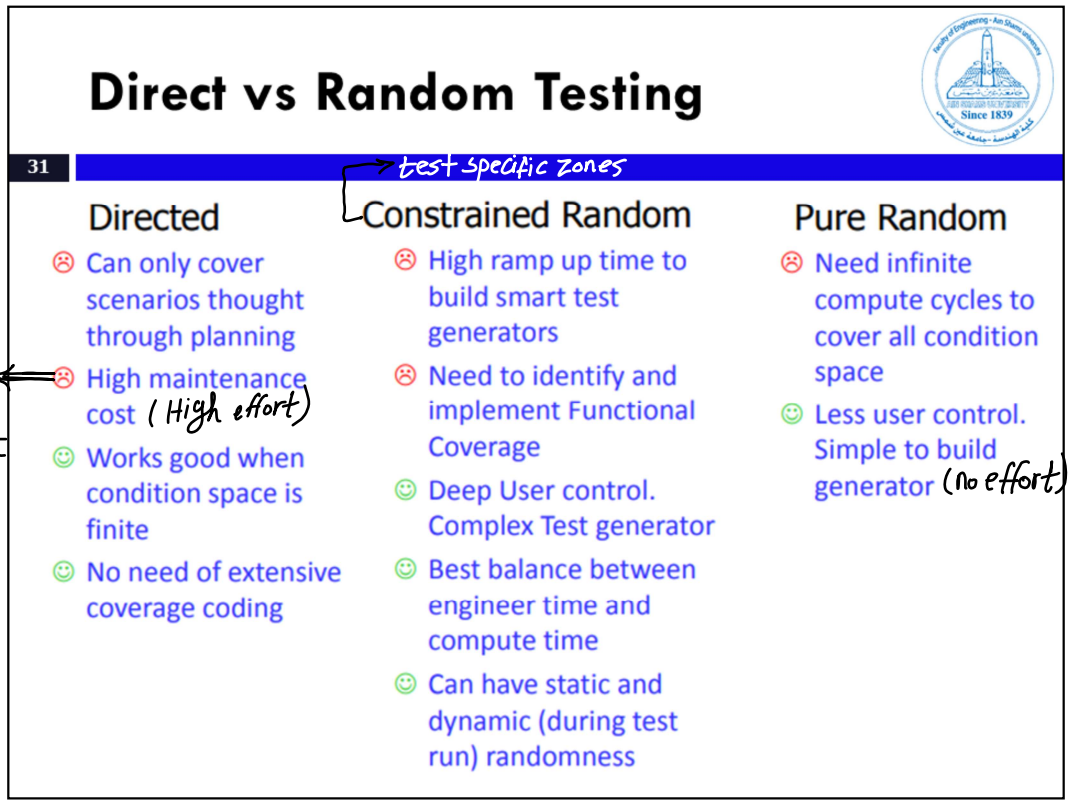
30

- What is Directed Testing?
- What is Radom Testing



- In this lecture will understand more about directed testing versus random testing.
- We will understand what is directed and what is random testing and we'll do a comparison between both approaches.
- So what does directed testing? directed testing is an approach where we create a test that is a directed for every feature in the design.
- If there are multiple features that has to operate in different conditions you might create a test for each of those features to be working in different conditions.

- On the other hand random testing is an approach where you don't create a test for every single feature.
- Rather we depend on a random test or a test generator that can create multiple scenarios in a random manner.
- Each of the directed test or random test has its own benefits and disadvantages. We'll see some of that in the following slide.
- But in General directed testing works well especially when the design is in the early stages of development, when the design is less mature, and also when that design scope is limited we can think through all the scenarios and all the features of the design.
- Whereas on the other side, random testing will be more beneficial if the design complexity is high and the design is also matured in the later stages of the design development where each features might work. So in that case random testing might be used.
- In the following slide we will make a comparison between directed and random testing.



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.

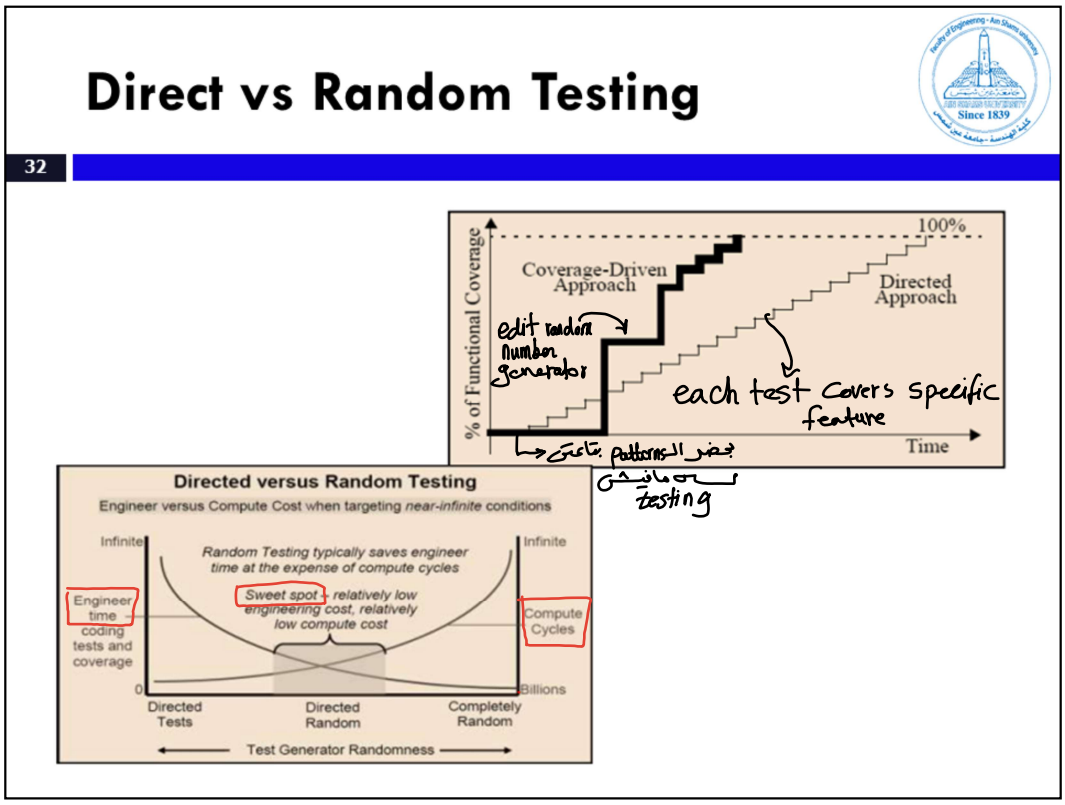
Random testing:

- 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.



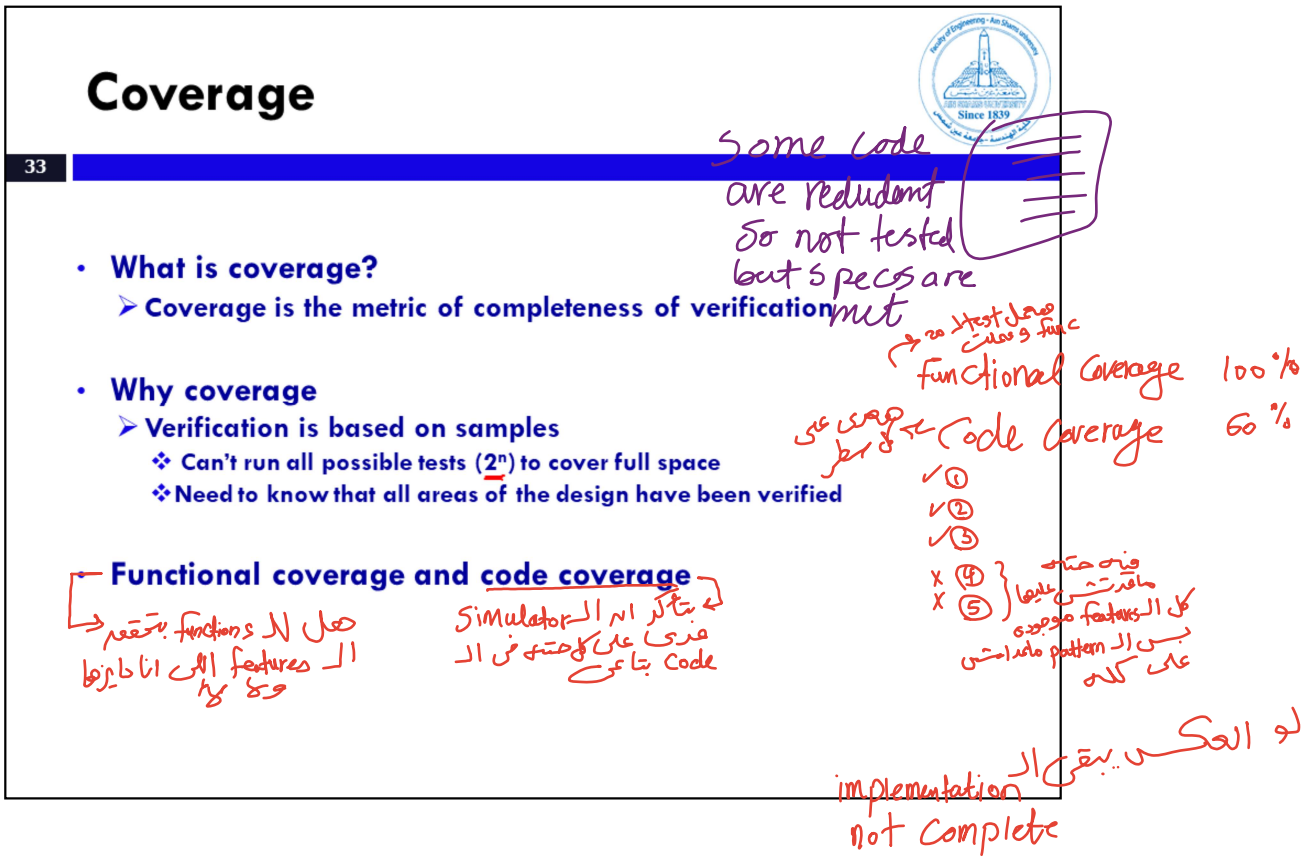
These diagrams again explain directed versus random test coverage.

First diagram:

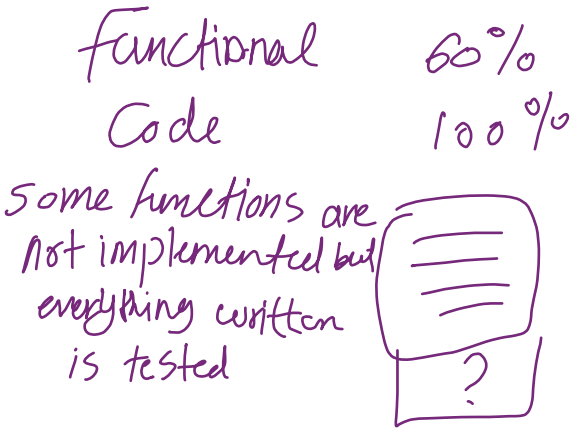
- The first diagram shows the coverage versus time
- So in the directed test approach, with every generated test we hit a certain percentage of the functional coverage, so it's like step improvement throughout the project and takes let's say so much time to get 100% coverage to be hit.
- In the constrained 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 coverage are two types of coverage that will help us identify what areas of design have been verified well.

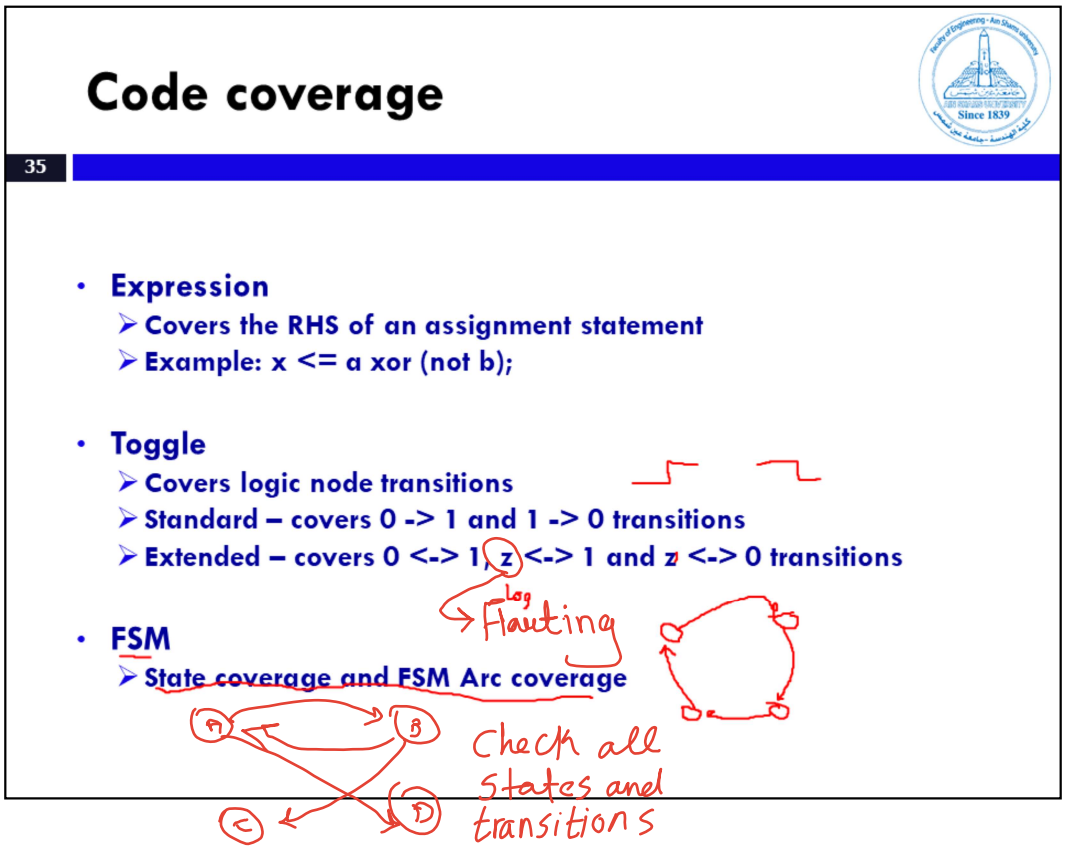


Code coverage

34

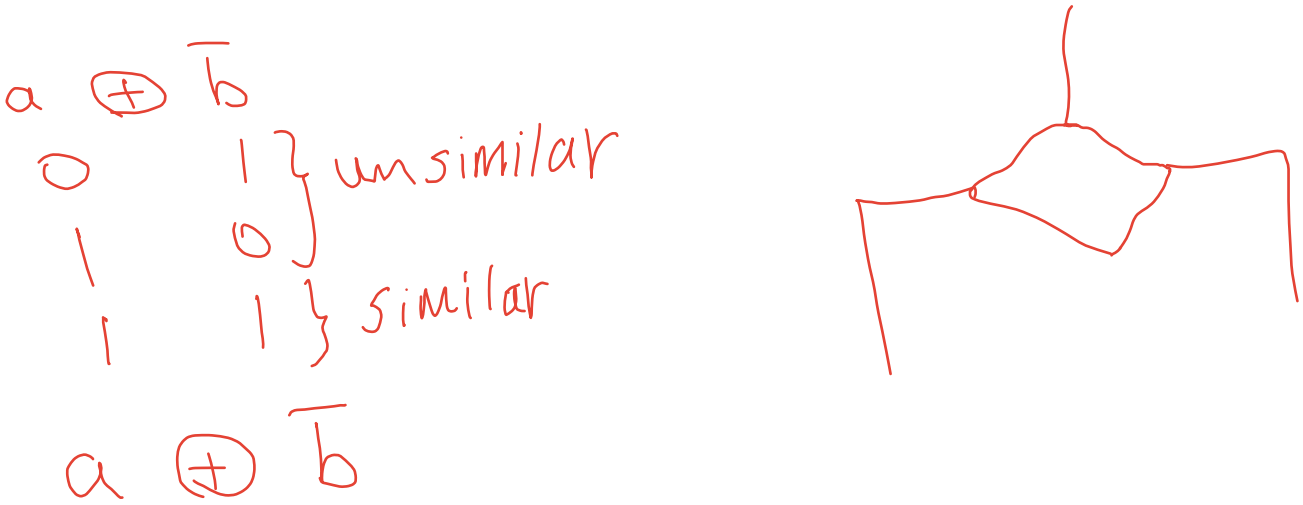
- **Statement**
  - Has each statement of the source code been executed?
- **Branch**
  - Has each control structure been evaluated to both true and false?
  - Examples: if, case, while, repeat, forever, for, loop
- **Condition**
  - Has each Boolean sub-expression evaluated both to true and false?

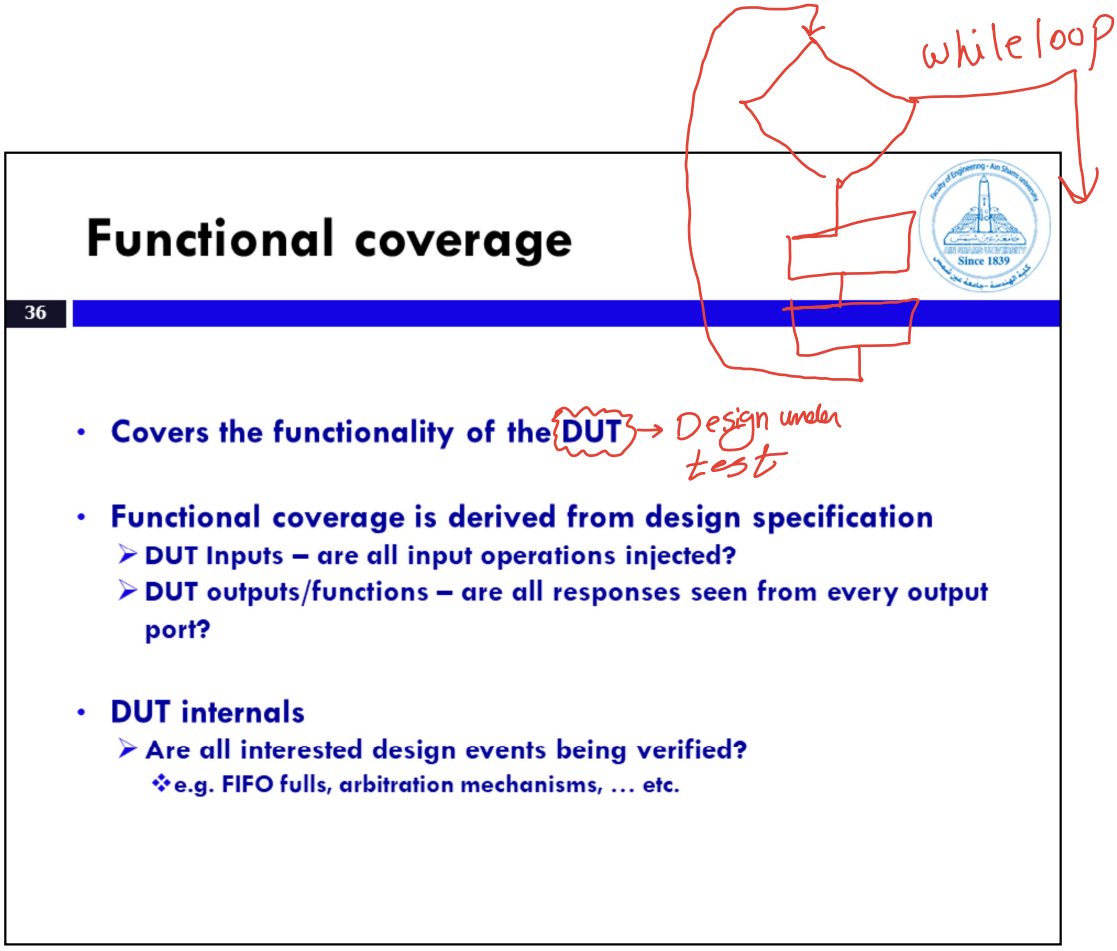
- Code coverage comes in different flavors. It is a metric of the code is covered during simulation.
- **Statement coverage:** it is a straight forward metric which tells you how much of the statements in the source code are executed during your test.
- **Branch coverage:** it is another similar one which tells you whether every control structure have been evaluated to both true and false condition. Which are your branch conditions. Examples of branches are the if statement, case 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.



- **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:** executes the toggle report whether all nodes covered all the possible transitions.

  Two forms of toggle coverage, are 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.
- **FSM coverage:** gives a coverage about the state transitions, and the different arc coverages.



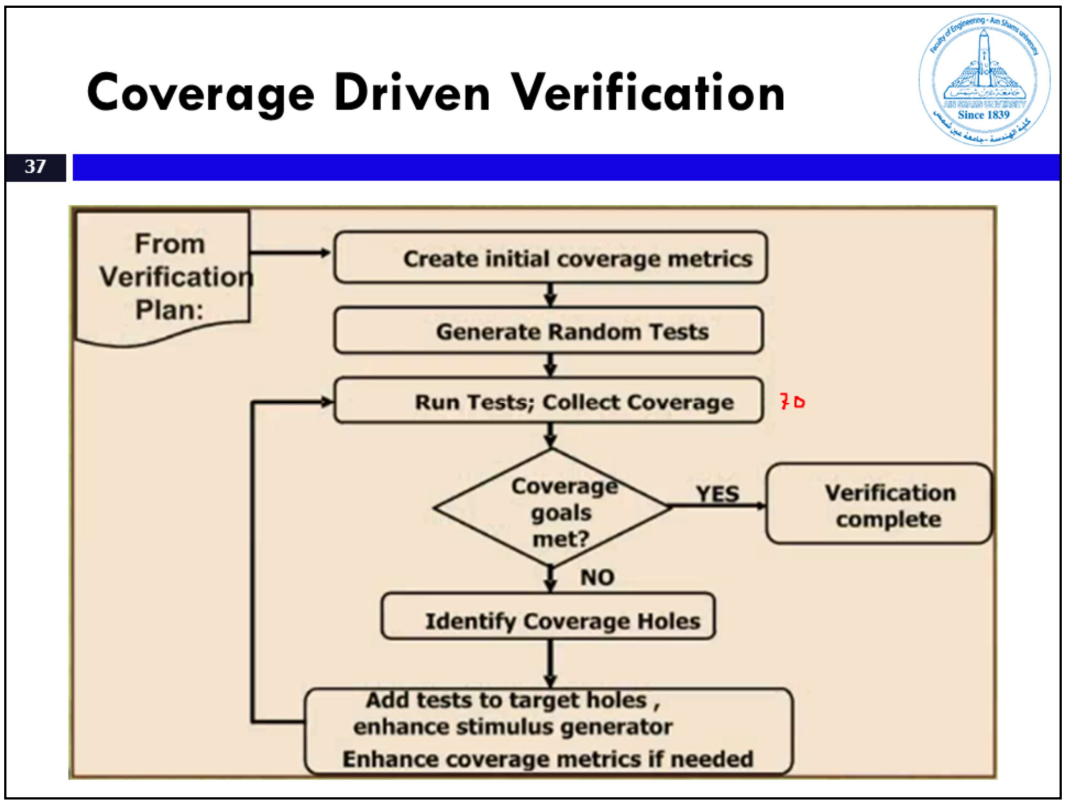


- **Functional coverage:** it different from code coverage. It covers the functionality of the design. It is derived from the design specification. Tools can't not generate an automatic functional coverage unlike code coverage.

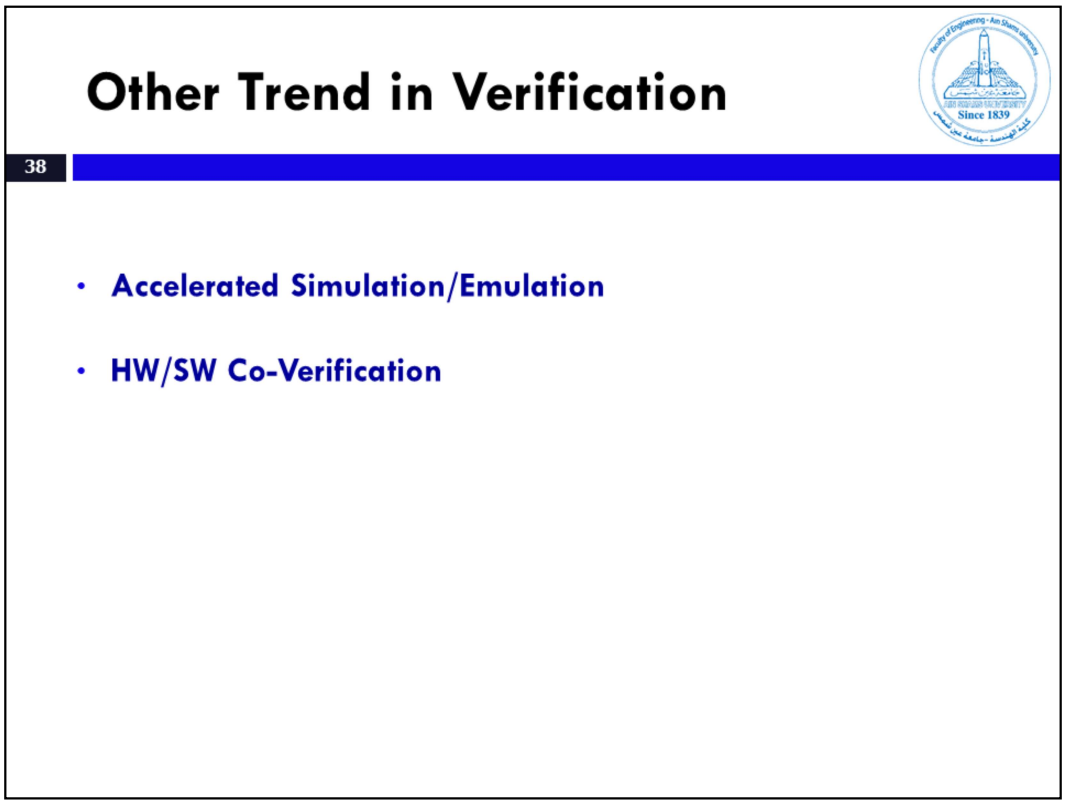
  In this case, based on the design specification, we need to create functional coverage monitors, which a tool uses to extract functional coverage during the simulation.

  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.

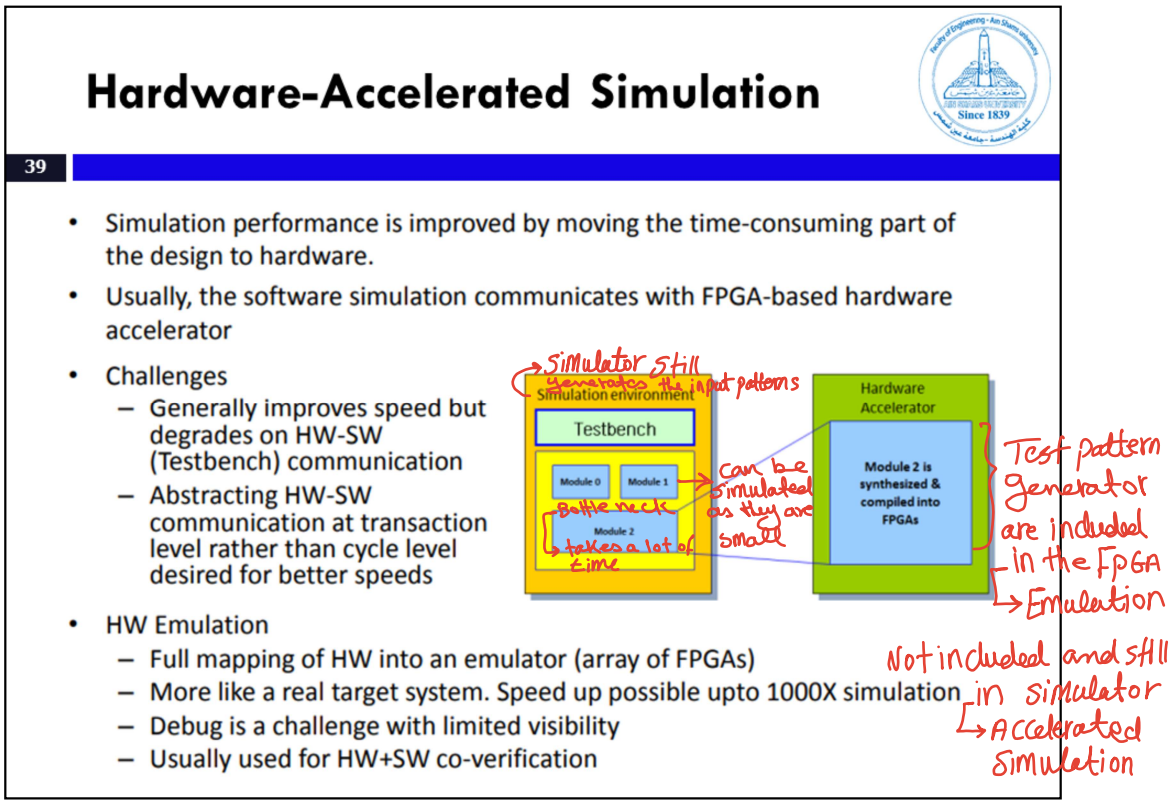
  Also doing internal coverage, like whether every interested design events are covered, like FIFO fulls, arbitration mechanisms are being covered.



- 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 coverage metrics, that tell 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 this process until we get confidence



- We will see some of the latest trends in verification and will wrap up what we have learnt till now.
- We will learn about what is accelerated simulation, and emulation.
- We will also learn about what is NW/SW codesign

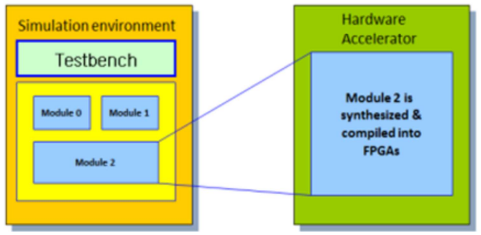


- Hardware accelerated simulation as an alternative method used especially at the higher levels of verification like a system or a full chip verification, primarily because the simulation-based verification tends to slow down as the size of design increases. That is when at full chip or SOC level since the design size is huge and the simulations run pretty slow and that is why we look for alternative methods.
- That is primarily because the simulator is a software program that runs on a host CPU that runs at every cycle or at every change in a signal.
- At a higher level as an alternative we try to move the time consuming part of the design to the hardware. In the shown diagram on the left we have a set of modules representing your design implementation, and the testbench module which comprises all the behavioral components like stimulus generators, checkers, … etc., used in the simulation environment. The most time consuming part of this simulation environment is the design implementation in terms of modules, because those are the ones that need to get evaluated at every cycle, or at any event of signal change.
- In this approach, we take those time consuming modules, and synthesize and compile them into real hardware FPGAs, so they can run on a real hardware much faster.
- We still keep the testbench components running on the host CPU, and then we use some means of communication between the testbench components existing on the software simulator talking to the hardware implementation on the FPGA.
- So we can still gain speeds of 10x – 20x of your simulation.

Hardware-Accelerated Simulation

40

- Simulation performance is improved by moving the time-consuming part of the design to hardware.
- Usually, the software simulation communicates with FPGA-based hardware accelerator
- Challenges
  - Generally improves speed but degrades on HW-SW (Testbench) communication
  - Abstracting HW-SW communication at transaction level rather than cycle level desired for better speeds



- HW Emulation
  - Full mapping of HW into an emulator (array of FPGAs)
  - More like a real target system. Speed up possible upto 1000X simulation
  - Debug is a challenge with limited visibility
  - Usually used for HW+SW co-verification

- This definitely come with some challenges. As I mentioned the speed can be improved using this approach as we move come design components into a real hardware system like FPGAs. But then base on how the HW & SW communicate, this can still degrade your speed. The communication happens every clock cycle, or at every signal change event, so you lose a lot of benefit in terms of speedup.
- So the approach that is normally used is to abstract this HW/SW communication to an transaction level rather than to a cycle level or signal change level. In that case you don't need to communicate between the HW and SW at every clock cycle. You will need only to communicate at the transaction boundary.
- This is accelerated HW simulation.

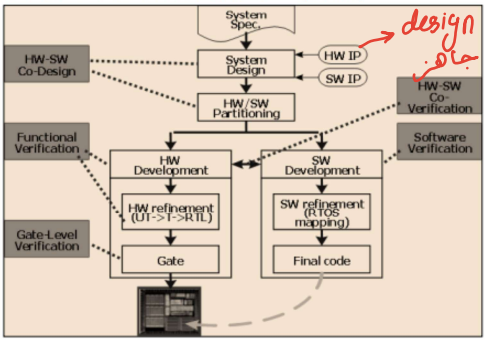
HW emulation:

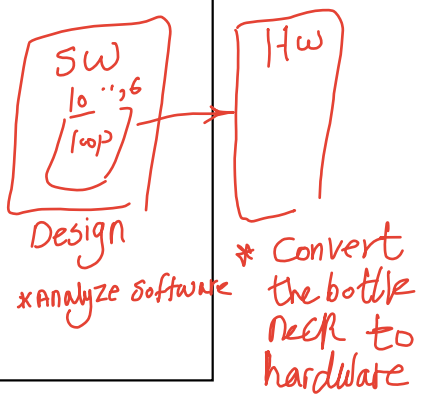
- To further improve the speeds, we can do a full mapping of the hardware to the emulator (in this case we have a array of FPGAs not a single one).
- The hardware emulation will look more like a real target system. Speedups can reach like 1000x.
- There can be several approaches for doing HW Emulation. We can either synthesize your testbench if the testbench is either done in a synthesizable fashion on the emulator, or we can set up configurations which look like a real hardware system and then use another real stimulus generator that might be used for real silicon testing.
- In either cases debug is still a challenge on hardware emulations since we may not have a full visibility of the internals of the design.
- Hardware emulation is typically used for co-verifying the hardware and the software together.

HW/SW Co-Verification

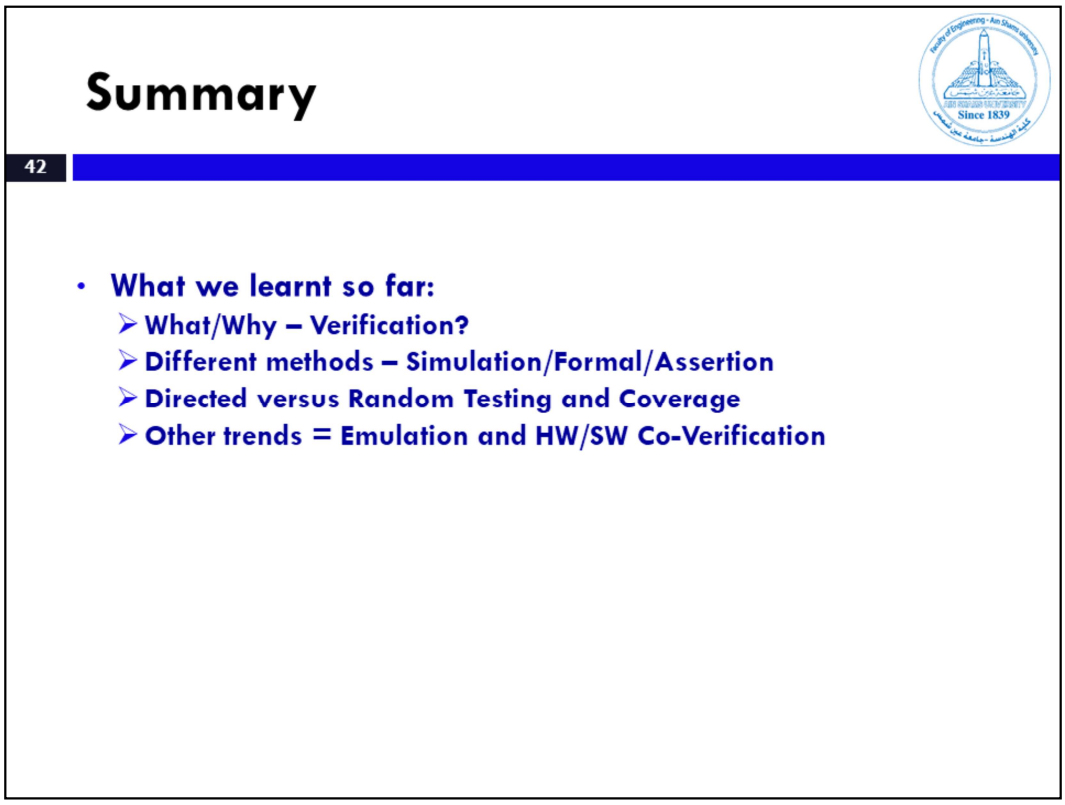
41

- SOC designs involve both HW and SW development
- HW and SW components are verified separately but needs to work together in real product
- Co-Verification will help projects to complete in short time and with higher confidence on both HW and SW quality





- Now let's see what is HW/SW Co-Verification
- What I mentioned in the very beginning that an SOC involves both a hardware development phase, as well as a SW development phase.
- This diagram shows that at a high level we come up with a system specification with what the system has to do.
- During the system design phase we partition those specifications into what has to be implemented in HW versus what has to be implemented in SW. and then the two developments go in parallel.
- On one side you continue with the HW development. We refine the HW specs and then implement the actual design using RTL and then synthesize to gates … etc.
- On the other side the SW development continues. We refine the software specs and then define what has to go to RTOS level and what has to go to the different layers of driver software, and then we finally develop the SW code.
- Then the HW and SW together has to work on a real system.
- In the hardware phase we do functional verification, as well as GL verification to make sure the HW is working standalone.
- On the side of SW development involves software verification to make sure the SW works standalone.
- Traditionally we never used to do verification of HW + SW before we actually go to the real silicon in the lab.
- But with the increased design complexity of the recent SOC designs, it is becoming more and more important to verify the HW and SW together before we get the actual silicon, because that can save a lot of surprises in terms of silicon bugs and avoiding re-spins etc.
- That is why in recent SOC designs HW/SW Co-Verification is becoming more and more important.
- Also, the Co=Verification will help projects to complete in shorter time with higher confidence since we do both HW and SW quality testing together before the actual tape out or before the actual silicon. That gives you much higher confidence in this approach.



- Let's summarize what we have learnt till now
- We started earning what is verification and why we do verification in SOC or chip design flows. So, verification is a process by which we ensure the functional correctness of design. If we don't ensure the design if functionally correct it can lead to bugs in the silicon which might mean re-spins and costing like millions of dollars. Also, verification is becoming the most time-consuming part of a chip design flow, that is why it is becoming more and more important.
- We learnt about different methods. We learnt about simulation-based verification, formal verification and assertion-based verification.
- We also did a comparison between directed testing approach versus random testing approach. We also learnt what is coverage and what is the importance of coverage and especially in random and in constrained random testing.
- We also saw some of the recent trends like the accelerated simulation or emulation, as well as HW/SW Co-Verification.
- I hope that have given you the basics of the different verification concepts.

Bugs are good

3

- Don't by shy of revealing a bug. You should be proud.
- Each bug found before tape-out is one fewer in the customer's hands.
- Finding bugs in earlier phases are easier and less expensive to fix.
- Don't let the designers steal all the glory – without your craft and cunning, the design might never work!

- The most important principle you can learn as a verification engineer is "Bugs are good." Don't by shy of finding a bug.
- The entire project team assumes there are bugs in the design, so that each bug found before tape-out is one fewer that ends up in the customer's hands.
- You need to be twisting and torturing the design to extract all possible bugs now, while they are still easy to fix.
- Don't let the designers steal all the glory – without your craft and cunning, the design might never work!

IP = design = circuit
↳ intellectual property

Respin ⇒ return to the factory and start all over again