VLSI Design Flow: RTL to GDS

Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 8
Overview of VLSI Design Flow: V

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the seventh lecture, and in this lecture, we will be continuing with the overview of VLSI design flow. So, the topics we will cover in this lecture are verification and testing. In the earlier lectures, we have seen how to implement a design. Implementing a design means that we are making changes to a design. We started with an idea, then we carried out system-level design, and then we got an RTL, and then we went through RTL to GDS flow, and at the end of this, we got a GDS or the layout.

So, in this transformation from RTL to the layout, a design goes through various EDA tools. It also goes through various individuals or teams. So, during this transformation and this design phase, errors can be introduced in our design. And what could be the reason for the error? There can be many reasons.

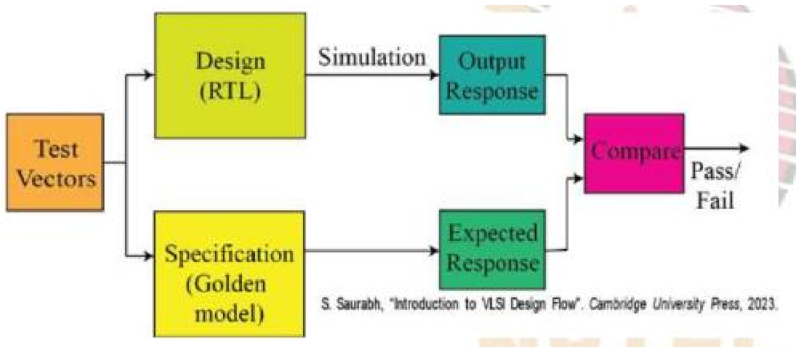
Some of them can be human error or miscommunication between two teams or two individuals within the same team, or it can be wrong usage of a tool, or there can be some unexpected or buggy behavior of a tool. So, these are a few examples of what can go wrong in this implementation flow. To check whether we have not made any mistakes or whether the design is still good, we need to carry out the verification step along with the implementation step, and this is what we will be covering in this lecture. So, what is verification? We perform verification to ensure that the design works as per the given functionality. So, the functionality is defined at the top-level, and we ensure that our design, as it is transformed in the VLSI design flow, should still be delivering the same functionality that we started with or that was given to us. And to ensure that we need to carry out verification multiple times throughout a VLSI design flow.

So, whenever the design goes through some non-trivial changes, we must verify that the change is correct and the design is still behaving correctly or delivering the required functionality. And we need to do it multiple times because we want to catch an error or if there is a problem in our design as soon as possible. Why do we want to do that? Because then it allows us to fix this problem, and the quicker or earlier we fix the problem in the design flow, the fix can be done with less design effort or less effort in terms of time. So, whenever there is a verification failure, we need to take remedial action, and if this remedial action is taken as soon as the error is introduced in the design, the cost of fixing that design bug is less, and that is why you want to carry out this verification step throughout our design flow and catch an error or verification problem as soon as possible. So, currently, a lot of effort goes into verification, and with the advancement in technology and the addition of more features in our design, the verification effort is increasing with time or with the increasing design complexity.

So, verification plays a very important role in VLSI design flow. In this lecture, we will look into a top-level view of some important verification methods that we use in VLSI design flow, and in later lectures, we will look into these verification tasks in more detail. So, here we are, just getting an overview of what are the verification steps that we need to carry out in a VLSI design flow. So, at the top level, when we are given a functionality, we write an RTL model, or we get an RTL model through, say, a behavior synthesis tool or through IP that we get from a vendor. So, now the RTL that we have got or that we have designed must match or deliver the functionality as it is given in the specification. So, to ensure this, what do we do? So, the most common method to ensure that the RTL is as per the given functionality or specification is simulation.

So, simulation is a technique for ensuring the functional correctness of an RTL using test vectors. Now, what are test vectors? So, test vectors are sequences of zeros and ones with the associated timing information, meaning that when the transition happened from 0 to 1 and vice versa. So, test vectors are the stimulus that we apply to the design, and using the test stimulus, we verify whether the functionality of our design is correct or matches the specification or not. So, there can be many different frameworks under which simulation can be performed.

One of them is as shown in this figure. So, what we do is that we apply test vectors or stimuli to our design, which is in the RTL form, and then take it through a tool which is known as a simulator. So, the simulator gives us, or it basically computes mathematically, that given this test vector and the RTL model, what will be an output response, that will be computed by the simulator. For the same test vector, we also compute the expected output response from the given specification, which is modeled in C, C++, MATLAB, etc. So, we get this expected response just by running their binary or executables.


Test Vectors
Design (RTL)
Simulation
Output Response
Compare
Pass/ Fail
Specification (Golden model)
Expected Response
S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

So, we have a set of output responses that we are getting from the RTL and an expected response for the same test vector from our specification. We compare both of them. If, after comparing, we find that these two are the same, then we say that the verification has passed or our RTL model is meeting the specification. Now, this simulation-based verification is very fast, and it is also versatile. By versatile, I mean that this simulation method can be used at various levels of abstraction. So, here I have shown it is being used for the RTL model, but we can also, instead of an RTL model, we can take a gate-level netlist to do a simulation and see what the output response is.

Similarly, we can do this kind of simulation at the transistor level. So, simulation is a kind of a very versatile method or technique of verifying a design or a model. It is also very fast; it is the same as running an executable. When we write a computer program, after compiling it, we see whether that program is producing the correct answer by running it on a given input. So, simulation is a similar kind of method. For a given test vector, we are seeing whether our model is producing the expected response or not. But what is the downside of this functional verification using simulation? These are very simple technique, but the downside is that it is incomplete.

Incomplete in the sense that the test vectors that we give during simulation cannot be exhaustive. By exhaustive, I mean that if we are giving all possible test vectors to a simulator, then we say that it is exhaustive. But for industry scale design or a practical design, it is simply impossible to give all possible test vectors. Why is it impossible? Because the number of test vectors will be exponential in the number of inputs. And then there are many other possibilities that come because of the state elements or flip flops in our design.

And therefore, covering all possible scenarios is simply impossible. Therefore, in simulation, what we give as a test vector is typically in a region of operation of our model where we think that there could be a bug. So, there may be some corner cases that we may be simulating or an area that we think is a brittle area or where some errors can be there, or some area of our code or our model where lots of computation is done and which are more vulnerable to bugs. So, we typically choose the area where we want to simulate and do the simulation accordingly. So, in that sense, functional verification using simulation is incomplete.

So, there are alternative ways to do the functional verification. One of the alternatives is model checking. So, what is model checking? So, in model checking, we also check the functionality of our design and see whether it matches the specification or not. But the difference between model checking and simulation-based checking is that in model checking, we typically use formal methods. And what are these formal methods? So, formal methods basically establish the proof of a given property using formal mathematical tools such as deductions. Rather than relying on the test vectors, it is trying to prove something mathematically.

And so once we have proven a property mathematically, it is guaranteed to hold for all the stimuli. Why is it guaranteed to hold? Because it implicitly covers all the possible scenarios. Let us take an example and understand what is the difference. Suppose a property is given that a function, a Boolean function f(x)=g(x), and we need to verify it. So, in simulation, what will be done is that we will try for different values of x, and we will see what is the output of f and g and match the output of these two functions, and based on that, we say that the function f(x) and g(x) are equal.

But we cannot always cover this g(x). For example, if it is all the possible integers, then it is simply impossible to cover all the integers. On the other hand, what formal methods will do is that it will try to establish a mathematical proof. For example, it will try to prove that f(x) implies g(x) and g(x) implies f(x). If the tool is able to establish this property, then we can say that f(x) is equal to g(x) no matter what x is. So, implicitly, it is covering all possible scenarios.

So, the verification using the formal method, the merit is that it offers completeness, but the demerit is that it is computationally difficult. We cannot apply this model checking for all kinds of problems. There are computational difficulties in establishing these mathematical proofs. Now, in property checking or model checking, what do we do as a designer? So, as a designer, we define properties that must be satisfied for a given specification, and then the tool will check whether those properties are being satisfied in the implemented design RTL using the formal method.

Now, what could be these properties? Let us see a few examples. For example, if you are designing, say, a traffic controller, then we can define a property that not more than one light should be green. So, this is a property that should always be true for a traffic controller that is generating signals for various lights in a traffic light. Similarly, in a resource-sharing environment, there can be a property that a request for a resource should be granted eventually. So, this will ensure that some requester is not starved of resources.

So these are some of the examples of these properties. Now, for a given design or specification, we can extract important properties that must be satisfied by that model, an RTL model, or our design, specify them, and give them to the model checking tool, and that model checking tool will try to prove it that whether the given property is true for all cases. So, again, the point to note is that it will not do a simulation and find out whether those properties are satisfied for all possibilities. It will use mathematical deductions or mathematical properties to arrive at an argument or a proof that establishes that the given property holds. So, during the design flow, a design goes through various steps of transformation.

Now, for example, when we do logic synthesis, our design is in terms of RTL, and at the end of logic synthesis, we get a netlist. Now, how do we make sure that functionally, this RTL and the netlist are delivering equivalent functions? To do this, we use a tool which is known as a combination equivalence checking tool. This tool basically establishes the functional equivalence of the two models, in this case, RTL and the netlist, using formal methods. So, combinational equivalence checking, or CEC is required, or we need to run these CEC tools whenever we make nontrivial changes to the design. Why? Because the transformation can introduce bugs in our design.

So, we need to carry out this kind of combinational equivalence checking throughout our design flow. So, to give you an illustration of how we use combinational equivalence checking, we can start with, say, an RTL and netlist. So, the first thing we discussed was that we had a functional specification. We established that this functional specification was being delivered by this RTL using simulation and also using model checking. Now, after we have got this RTL, we do logic synthesis, and then after doing logic synthesis, we do an equivalence checking between the RTL model and the netlist.

And if we establish the equivalence, we also establish that our netlist is meeting our functional specification. Why? Because our RTL was originally meeting our functional specification as verified by simulation and model checking. Similarly, as the design progresses through, say, floorplanning, placement, clock tree synthesis, or routing, if we do combinational equivalence checking between two consecutive steps throughout the flow, we can ensure that the final design or the layout that we are having that meets the functional specification that we started with. Why? Because if there is an error introduced by the tool or by a human anywhere, that will be caught in this combinational equivalence checking step. If we do combinational equivalence checking in subsequent steps, we are sure that the functionality we are getting is equivalent to the RTL and the RTL was originally meeting the functional specification.

So, all the transformations are correct in terms of functionality. Now, in addition to functionality, we also need to worry about the timing of our design. Now, in a synchronous design, how do we ensure that the data transfers between flip-flops? For example, if we have a flip-flop and then this data goes from this flip-flop through a combinational logic to a sequentially adjacent flip-flop, or in a synchronous circuit, the data that is launched at the launch flip-flop must be captured at the capture flip-flop in the next clock cycle that is what the synchronous behavior is. Now, the delay of the combinational logic (between the launch and capture flip-flop) can disturb that behavior, but we need to ensure that it is not disturbed. The synchronicity, that is, the data launched by a flip-flop gets captured in the sequentially adjacent flip-flop in the next clock cycle, is ensured by a static timing analysis tool or STA tool.

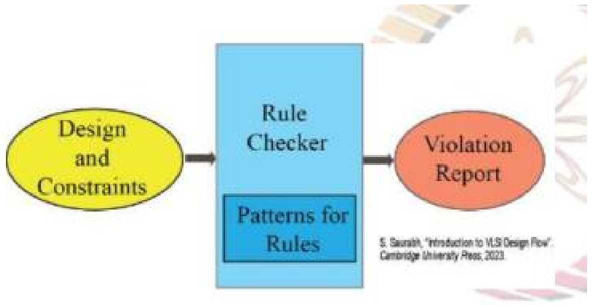
It ensures deterministic synchronous timing behavior in a circuit. It will also look into setup and hold constraints of the flip-flop, and if those are violated, it will be reported by the tool. Now, STA tools consider the worst-case behavior, which may be pessimistic, which is not realistic, while producing the result. Therefore, we can say that if a design passes STA, the design will always be safe on the side of pessimism, not on the side of optimism. That is why ensuring the timing behavior of our design and checking it with STA tools is very important, and in a design flow, this is carried out multiple times. So this is the timing analysis tool in which the design can come at various levels of abstraction, maybe at the netlist or floorplan design or place design or design that we get after clock tree synthesis or globally routed design or detail routed design or even after ECO. We can give design to an STA tool in various abstraction levels, and we also need to give constraints to the STA tool. What do these constraints contain? These constraints contain information about the expected timing behavior in terms of maximum operable frequency for a design, the characteristics of the signals that come into our chip or our design, and the expected behavior of the signal that leaves our design.

These things are described in this constraints file, and we also need to give the library, which contains the information of the standard cells that are used in this netlist. Based on the library information, the static timing analysis tool will do a delay calculation, and based on that, it will give us the timing report. Based on this timing report, we will see if there are violations, then we might need to fix our design or make changes in our design and then progress to the next steps.

Then, when we do physical design, we have to be careful about how we are laying out the components of our design, and we have to ensure that the layout does not have any manufacturing and connectivity issues and the yield that we will be getting out of this design will be acceptable. To ensure this, there are a set of checks that need to be done during physical design, and those are known as physical verification tasks. So, these tasks are carried out before sending a design to the foundry for fabrication. Some of the physical design verification tasks are design rule checks or DRC. So, in these DRC checks, we check for the rules that are coming from the foundry, and we ensure that those rules are taken care of, or those are followed by our design in the layout.

If there are any DRC violations, then we need to fix them before sending the layout to the foundry for fabrication. Then, we need to do an electrical rule check in the physical design. These electrical rule checks are basically rules that define and ensure the proper connectivity. For example, if there are two wires which are different wires, they shouldn't be short among themselves. If there is shorting because of some design error or layout error, then that should be found, that problem should be found and fixed.

So those kind of problems are caught in electrical rule check. We also need to carry out a layout versus schematics check, meaning that we need to ensure that the layout that we have created, the transistor level layout, meets the original functionality that we started with. This is done using a physical verification task, which is known as layout versus schematic check or LVS check. In addition to these checks that I have described, there are some rule checkings that need to be done in our VLSI design flow. Now, what is rule checking, or what are these rules? So, rules are some restrictions imposed on the design entity, such as RTL constraints, netlist, layout, etc., such that there are no issues in using that design entity downward in the VLSI design flow. So, what is the framework of rule checking? The framework is shown in this figure. So there is a rule checker, and in this rule checker, there are some patterns defined. Patterns defined that if this pattern exists in our design or in our constraints, then flag an error. So this rule checker will look for those patterns in our design and constraints and raise a flag if those patterns are found in a violation report, and then we have to look into those reports and perhaps make some fixes if required.


Design and Constraints
Rule Checker
Patterns for Rules
Violation Report
S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

These rules can be defined at various levels. For example, at the RTL level, there can be rules that ensure that the RTL constructs used in the design have no synthesis or simulation issues or will not create any synthesis or simulation issues down the flow. Now, these restrictions are more stringent than what is imposed by the language. So, the language may allow some constructs, but those constructs may be actually flagged by this RTL rule checker because those constructs can cause problems later down the flow. Maybe it may be creating, say, a simulation-synthesis mismatch or a similar kind of problem. So, these rule checkers are more restrictive than what is allowed by a given hardware description language.

Then, there can be rule checking on the constraints. We write these constraints in a format which is known as synopsis design constraint or SDC format. Now, we give this SDC file or constraint file to the tool. These SDC constraints or SDC files are typically manually written or manually edited. Now, while making these changes or writing these SDC files, we can make errors that can add conflicting constraints. So, if that happens, then it will be flagged by this constraints checker. It ensures that no conflicting constraints are applied or no constraint is missing in our SDC file. Then, there can be rule checking at the netlist level, which ensures that the connectivity of the instances does not cause any issues down the flow.

For example, there can be some connectivity in the netlist, which may later create problems in design for test for ensuring the testability of the design. So, those kinds of constructs can be flagged by these rule checkers. So, these are some of the verification tasks that we need to carry out during the VLSI design flow. Now, let us look into another aspect of VLSI design flow that is related to testing. So, till now, we have discussed the design flow, meaning that we are making changes to our design, and we have got the layout. And we have carried out the verification steps in this design part. And what verification is doing is that it is ensuring that the final layout that we are getting or the GDS representation of our design or circuit is correct such that the original specification is met. So whatever the original specification was there, that is made by the layout, that is what we mean by the verification task. Now, once we have done the verification, we have got the final layout, and then we give this layout to the foundry for fabrication. Now, after fabrication, when we get these chips, how do we ensure that the chips actually have the features that were defined in our design layout? There can be some problems in creating this chip, and how do we catch this? So we catch such manufacturing introduced defects or problems in our circuit in a step which is known as testing or manufacturing test. So, the testing ensures that the fabricated chip does not have any manufacturing defect.

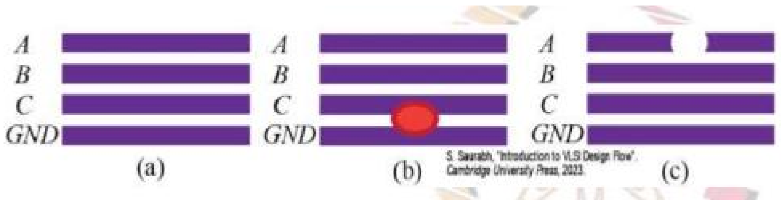
So, we should understand the difference between the verification task, which is carried out during the design step, and the testing or manufacturing test task, which is carried out after fabrication to ensure that the fabricated chip meets the original layout or original design that we started with in the fabrication. So, the purpose of both verification and testing are different. But then why should we consider testing during designing also? So we need to consider testing during the design step also because during the design step, that is, from the RTL to GDS step, we can introduce features in our design that allows easy testing or easy finding problem in our fabricated chip. So, testing is very important during the design phase because we need to add features in our design that will later allow the manufactured chip to be tested.

If we do not carry out those kinds of modifications or add features to our design, then testing will become difficult, and therefore, the testing needs to be considered during the design step also. RTL to GDS flow ensures that if some defect is found, then the problem can be easily diagnosed and debugged, and if required, we can fix some problems. Therefore, testing is very important even during the design step. However, testing is actually done after we have manufactured the chip, and we want to do this kind of testing so that we recognize or identify the chip that has defects and that has not been manufactured as per the given layout. We can identify that and disallow those chips to reach the end customer or end user. So what is a defect, and how do these defects originate? So defects are physical imperfections in a fabricated chip, and they are of a permanent nature.

So, how are these defects introduced in the first place? We actually carry out this fabrication in a very sophisticated environment, which is known as a clean room. So, in the clean room, the particulate impurities are very well controlled. Also, the static electricity and even vibrations are very well controlled inside this clean room. Then, despite the manufacturing being a very sophisticated task, how do we get defects in our chip? So, the defects originate because of many reasons. Some of them are statistical deviations in the material properties. Maybe some of the materials that we use in fabrication or maybe the chemicals have some statistical deviations.

For example, the concentration may not be exact but have some deviations of, say, the acid that we are using in our manufacturing, or there can be finite tolerances in the process parameters. For example, if we say that the furnace temperature is 800 degrees Celsius, it may not be exactly 800 degrees Celsius. It may vary between, say, 800 degrees Celsius plus minus 1 degree, just an example. There can be airborne particles and undesired chemicals. We try to reduce them but cannot permanently eliminate them. Those airborne particles and undesired chemicals can lead to defects in our chip, and also, there can be deviations in the mask features, which can lead to defects. Now, there are two kinds of defects. The first one is the large area defect that can be created by, say, wafer mishandling or by, say, misalignment of various masks, etc.

These are basically simpler to eliminate, and in advanced processes or in matured processes, these large-area defects are easily eliminated. But there are spot defects or small area defects, which have a random nature, and those inevitably appear in the chip during fabrication. So, the spot defects increase with the increase in the area, and these spot defects are the primary concern for test. Now, how are these manufacturing defects manifested? So, manufacturing defects can be manifested in many forms. One of them can be, most commonly, a short circuit of the lines or an open circuit of the lines. For example, suppose there were three lines and the ground line running together, and a conducting particle came and sat between line C and the ground. As a result, C and ground will be short.


A
B
C
GND
(a)
A
B
C
GND
(b)
A
B
C
GND
(c)
S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

So, this is a short circuit defect. Similarly, if there is a non-conducting particle, it can break this wire or the path/electrical path in this wire, and signal A is an open circuit. There can be other defects that can change the circuit parameters, and as a result, delay can increase or decrease, and this can lead to problems in the circuit. A point to note is that there can be distortions in our fabricated chip, and these distortions can be due to optical effects such as diffraction, etc. These diffraction effects, while we are carrying out photolithography, can distort the image that we form on our substrate, and therefore, the features will be slightly distorted. It may not be perfectly rectangular in shape, for example. So, these kinds of distortions need to be handled in some way because these are inevitable, and we cannot avoid them. But the point to note is that testing does not involve detecting these kinds of distortions.

So, the focus of testing is not looking into these distortions of the features. These distortions are inevitable and need to be handled using some techniques that we will be looking at in the next lecture. And there can be inconsequential flaws also. For example, suppose the size of this dust particle was very small. So, even if this dust particle or conducting particle was there in our chip, it would not lead to any observable change in our chip behavior.

And these things, these inconsequential flaws in our chip, are also not the focus area of testing. So, the testing does not detect these inconsequential flaws. So, if the testing does not detect distortions and inconsequential flaws, then what does testing detect? So, testing detects faults. So, testing focuses on finding defects that can be problematic for a circuit's behavior.

These defects are modeled as faults in our circuit. For example, we can model this defect of conducting particle sitting between line C and ground by a fault, which is known as stuck at fault. We can say that line C is stuck to 0 by this fault or by this defect. So, we model defects using faults, and the purpose of testing is to detect those faults. Now, let us also look into the effects of having defects in our chip.

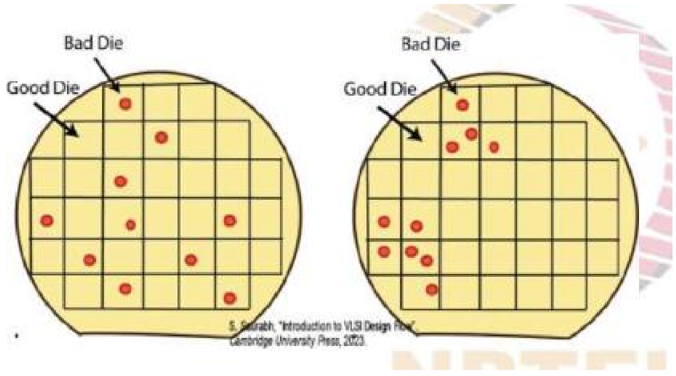
So, the first effect is that there will be yield loss. Now, what is yield? Yield is the fraction of dies on a chip that are good or without any fatal manufacturing defect. For example, if we are manufacturing 400 dies and out of them, say, 300 dies are good, and 100 have some defects. Then we say that the yield of this process is

Yield = 300 / 400 *100 = 75%

So, this is the fraction of dies on the wafer that are good expressed in percentage. Now, on what factors yield depend? So, it is a function of the complexity and maturity of the process.

If the manufacturing process is mature, then typically, the yield will be more than 90%, and for a recently developed process, the yield may be lower than even 50%. So, depending on the maturity of the process, the yield may be lower or higher. There are other factors that affect the yield, and one of the most important factors of a chip is the die area. When the chip area or the die area increases, then the probability of having a defect on the chip increases, and therefore, the yield comes down. Similarly, the defect density is a function of the process technology and how mature it is.

So, the yield will depend on the defect density. It is defined as the average number of defects per unit area of chip. And the third important factor that impacts yield is the cluster. Now, what is clustering? Clustering is defined as how the defects are distributed over a wafer. So, let us understand how clustering can impact our yield. Consider that we have a wafer in which we are fabricating, say, 34 dies. So, these are 34 dies we are manufacturing, and in one case, the defects are distributed all over the wafer.


Bad Die
Good Die
Bad Die
Good Die
S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

Now, in this figure, these red dots represent a defect. Now, there are ten such dots, meaning that ten dies have defects, and the rest are defect-free. So, in this case, what will be the yield? So, in this case, the yield will be

Yield = Defective dies/ Total dies * 100% = (34-10)/34*100 % = 71%

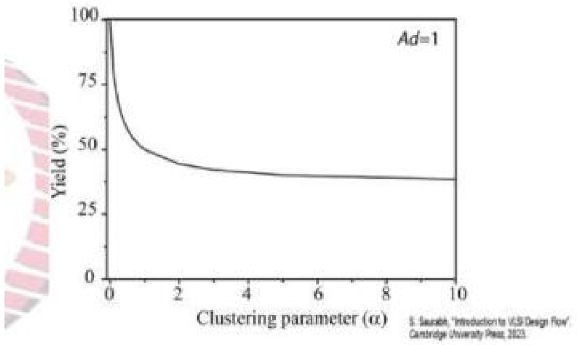
Now, let us look into another fabrication process in which we have the same number of red dots, meaning that we have the same number of defects, but those defects are clustered.

So, defects are lying in a smaller region. So, the number of defects is the same, but it is distributed over a smaller area in this case. Now, in this case, what is the yield? So, there are 26 dies without any red dots, meaning that they are defect-free, and therefore, the yield is:

Yield = 26/34*100% = 76%

So, in this case, the yield is higher, meaning that if the defects are clustered on the wafer, then the yield will be higher because whenever a defect is on a die, adding more defects on the same die does not lead to extra yield loss. So, if the defects are clustered in a technology, then that technology will have more yield.

So, fortunately, what happens is that in the fabrication of the chip, these defects are found to be clustered in groups over the wafer. Now, when we start a chip designing venture, we need to decide or understand the profitability of our venture, and in this equation or in this analysis of profitability, yield is a very important criterion because if the yield is low, a lesser number of wafers or lesser number of dies can be fabricated for a given wafer area and therefore, the effective cost of the chip will go high, and the profitability will go down. Therefore, before starting a venture, we need to understand or have some estimate of what the yield of that chip will be, and to do that, we need to understand the yield models. We need to have some understanding of yield models. In literature, there are many yield models with varying degrees of accuracy and computational requirements. One of the popular yield models is based on assuming that the probability of having a defect in a given area increases linearly with the number of defects, and this is because of the cluster.



If we assume this, then we can arrive at a yield model, which is shown here. So, yield

$Y = (1 + Ad/\alpha)^{-\alpha} \times 100\%$

Here, A is the area of the die, d is the defect density, and α is the clustering parameter. Now, let us understand how this yield changes with clustering. So, to do that, we assume that Ad=1, just for the sake of understanding this equation. If we put Ad is equal to 1, then what happens is that this equation boils down to $(1+1/\alpha)^{-\alpha}$, and if we plot this equation, we will get this curve. So, whenever the clustering parameter alpha has a low value, it means that the defects get clustered in a very narrow region, and as a result, a very small number of chips or dies are being impacted, and the yield is very high, and if it is distributed all over the area meaning that when alpha is very large, then there is a high yield loss.

Now, how can we use this formula or the yield model to arrive at the yield value for a given chip? Now, based on historical data for a given technology, we can arrive at this number d, defect density, and the clustering parameter by curve fitting. Once we have these parameters d and alpha, we can get an estimate of yield using the chip area and the past estimates or past data of yield, and using curve fitting, we can get the parameters d and alpha. So, this is how we can use the yield model for computing yield and maybe the profitability of our venture. Now, how is testing done? So, testing is done using automatic test equipment.

This is the most important equipment in testing. So, this testing technique that I have shown is a testing technique that is based on automatic testing equipment, and this is one of the testing paradigms. There are other testing paradigms that we will be seeing later in this course. So, in this testing paradigm, what we do is that we apply test vectors or test patterns to the fabricated die with the help of automatic test equipment. Now, this automatic test equipment has a test head, probe cards, and probe needles. So, what happens is that there are needles which are there in automatic test equipment, and those needles can make contact with the wafer, and this contact makes an electrical path. Through this electrical path, a test vector or a 0 or 1 can be applied to the die, and through those needles, the values produced can be read out from the die.

So, now there are test programs in this automatic test equipment that control all these operations. What does this test program do? It will load the test patterns, and these test patterns will be applied to the die through the test needles. Then, it will observe the response through the probe needles, and the actual responses will be compared with the expected response. If they match, then we say that it has passed the test; if they do not match, then we say that it has failed the test. Now, these test patterns are designed or are arrived at by considering our design functionality such that if we apply this test pattern for a given fault, the good circuit and the bad circuit with a defect will produce different output responses, and by observing the response and if those responses are different we can say that there exists a fault in our fabricated die. So, this is the basic technique of test pattern-based testing or manufacturing test.

So, if we get a failure in a chip, then we may like to diagnose it, find the root cause, and fix the problem, and then if corrective measures are required, then we can take those to reduce the effect of the defect in the subsequent fabrication. Anyway, the failed dies will not go to the next part of the design process and will not reach the end customer or end user. Now, in this automatic test equipment-based testing, we apply test vectors to our manufactured chip. Now, how do we measure that the test vectors that we are applying are of good quality? We measure it using a parameter which is known as fault coverage. Now, what is fault coverage? It measures the ability of the set of test patterns to detect the class of fault.

So, fault coverage is equal to the number of faults detectable by a given set of test vectors divided by all possible faults in the circuit. So, fault coverage is the measure of the quality of testing. Ideally, we want the fault coverage to be 100 percent, but that is very difficult to attain, and practically, we try to get it typically more than 99 percent. What it means is that if we leave 1 percent of fault undetected, then it may happen that some chips that are defective can pass the test and still go to the customer. Why? Because those chips contained defects that were not tested, those 1 percent were left untested.

So, the perceived quality of a chip based on the customer will depend on how much fault coverage we are doing. If we are 100 percent, then no defective chip will escape testing. It will always be caught, and it also depends on yield, meaning that if the yield is 100 percent, then all the chips will be error-free. So, if the yield is high, then the chip will not have a defect, and the perceived quality of the chip will be very good. So, the quality of the chip is measured by a quantity which is known as defect level. So, what is a defect level? It is the ratio of chips that are bad or faulty among the chips that have passed the test.

So, typically, it is measured in parts per million. So, let me give you an illustration: suppose we are manufacturing 100 chips. Now, let us assume that the yield of the process is 90 percent, and therefore, out of this, 10 percent, i.e., 10 chips will be bad, and 90 chips will be good. Now, let us assume that fault coverage for the sake of argument is just 50 percent. Ideally, it should be 100 percent. Suppose it is 50 percent. Then, out of these defective 10 chips, 50 percent will be caught because those will contain those 50 percent defects, which are being tested by the test patterns, and the rest, 50 percent, will escape the testing. Those will have the fault but will not be captured by the test pattern, and therefore, they will go undetected.

So, 5 bad chips will go undetected, and 5 bad chips will be caught, and those will be eliminated, or those will not reach the end customer. But 5 bad chips will be considered good because the fault coverage is 50 percent. So, the number of chips that will reach the end customer will be 90 plus 5, that is 95, and out of this, 5 will be actually defective. So, the defect level will be:

DL = 5/95 * 1,000,000

as we are measuring defect level in terms of parts per million, we multiply by 1 million, and this is what the defect level will be for this process. Now, defect level is the measure of the effectiveness of the test. If the testing is fully effective, then the defect level will be 0, meaning that those 5 bad chips will also be caught during testing, and only these 90 chips will go to the customer; and therefore, the defect level will be 0.

For commercial chips, typically, the defect level is less than 500 parts per million. So, now, in this, we see that the testing is actually done after doing the manufacturing, but why should we be concerned about testing during the design phase? Because we need to carry out a lot of design steps that are targeted for test, and those design steps are known as design for test or DFT. So, what are those tasks? The first task is to insert a logic structure inside our fabricated die such that testing becomes easy. The second part is to get good test patterns that will actually be able to detect faults or differentiate between a faulty and a fault-free circuit, and the third is what should be the expected response from a good circuit. We need to get that information during the design phase. So, these are some of the tasks that need to be carried out during the design steps, and these are intended for manufacturing tests.

We will discuss this task in detail later in this course. So, this is the reference that can help you go deeper into this topic, and so let me summarize what we have discussed in this lecture. So, in this lecture, we have looked into two important parts of the design flow, that is, verification and testing, and so we have now covered our design flow right from the idea up to the layout. We have covered the implementation of the design, we have covered the verification of the design, we have also covered the testing of the design, and we have taken an overview of all these processes. In the next lecture, we will be looking briefly at the fabrication step, meaning that once we get a layout, what is done in the foundry, and how do we get the final chips? So, this is the end of the lecture. Thank you very much.