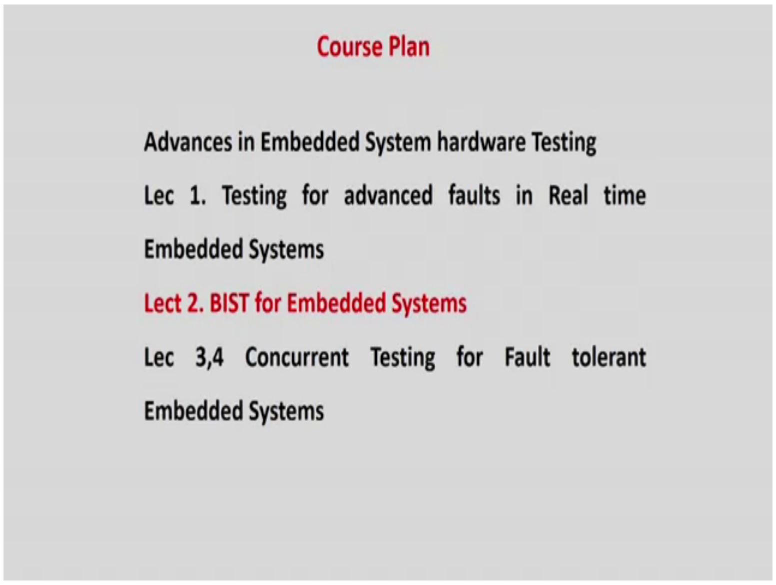
Embedded Systems - Design Verification and Test

Dr. Santosh Biswas
Prof. Jatindra Kumar Deka
Dr. Arnab Sarkar
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture – 33
BIST for Embedded Systems
Part-3: Embedded System Testing

(Refer Slide Time: 00:40)



Course Plan

Advances in Embedded System hardware Testing

Lec 1. Testing for advanced faults in Real time Embedded Systems

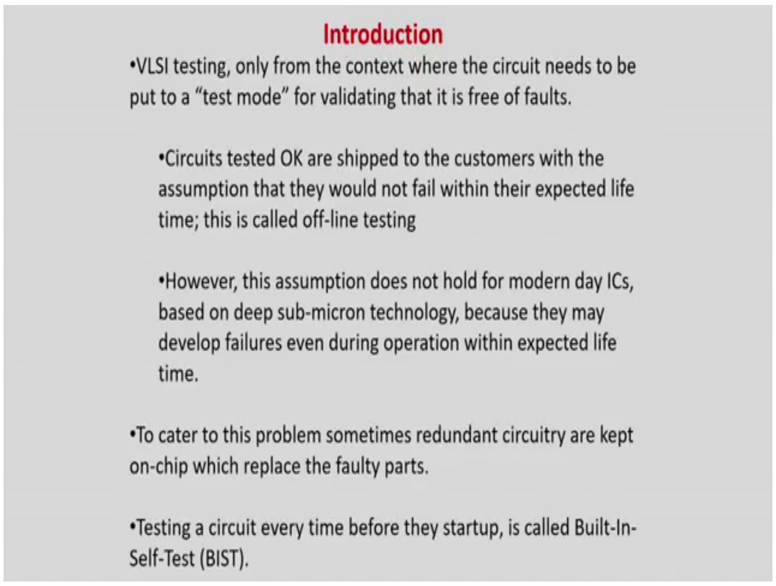
Lect 2. BIST for Embedded Systems

Lec 3,4 Concurrent Testing for Fault tolerant Embedded Systems

Hello everybody. Welcome to the testing part on the course on Embedded System-Design Verification and Test. So, as you know that now we are actually talking about the Advances of an Embedded System Testing. And now in fact in this module, we already discussed that we look at testing for advanced fault models, and then we look for built-in-self-test, and then fault tolerance.

So, already in the last lectures on this part, we have already seen basically how real time that is the one of the main the (Refer Time: 00:56) of embedded system nowadays is to meet the real time requirements. And how advanced fault models like that is delay fault models are required for such cases, and how they are tested basically. In this lecture, we are going to sees that what is something called build-in-self-test of embedded systems and in the next two lectures; we will go for fault tolerance.

(Refer Slide Time: 01:14)



Introduction

•VLSI testing, only from the context where the circuit needs to be put to a "test mode" for validating that it is free of faults.

- •Circuits tested OK are shipped to the customers with the assumption that they would not fail within their expected life time; this is called off-line testing

- •However, this assumption does not hold for modern day ICs, based on deep sub-micron technology, because they may develop failures even during operation within expected life time.

•To cater to this problem sometimes redundant circuitry are kept on-chip which replace the faulty parts.

•Testing a circuit every time before they startup, is called Built-In-Self-Test (BIST).

So, basically what do mean by testing, which you have been learning in the first 3 lectures that a device is manufactured, then we put it into something called the automatic test equipment; we apply the patterns, test it, and sell it to the market. Basically, that is actually called chips are tested OK are shipped to the customers with the assumption that they would not fail in their expected life time; this is actually called off-line test.

So, what do you mean by off-line? Chip is manufactured; we know some of the very important output test patterns we have already generated by the ATPG algorithms. We put it in the tester, tested and ship it to the market.

But, whenever the embedded system hardware are becoming more complicated, complicated in the system you are putting more and more system on chip on a single code, making NOCs out of them or putting more and more transistors in a single die, and feature size is decreasing, because of more advanced or lower level of some micro design technology. These are actually already discussed that they are making the probability of occurrence of faults more higher in case of such devices.

So, now what is for such complex embedded systems or complex embedded hardware, you cannot assume that once the chip is manufactured, manufactured and tested properly, we will do so in its life time it is not guaranteed that way.

So, what is the next step we can do? The first step is called actually the off-line test. Because, in this case it is off-line verified tested basically, and then you are selling it to the mark customer. He is putting into the board, and he is testing, and he is using it. With assumption that as it is already tested, so he should not have any problem in this lifetime. This is expected life time actually.

But, nowadays it does not hold. So a fall can even occur, after it has been manufactured properly tested, and it is being deployed in the system. And after 3 or 4 rounds of rounds of usage, it may have a failure, which is month before its expected lifetime.

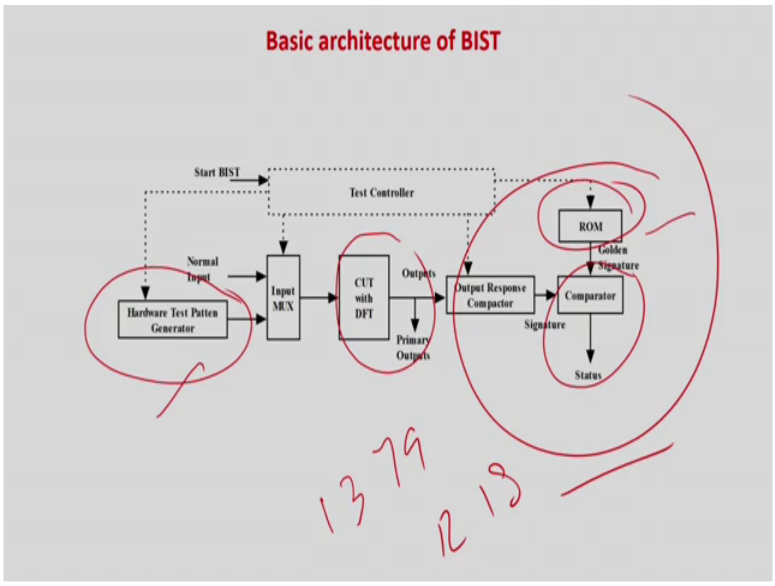
So, nowadays what happens, how we can handle it? So, basically what they do, they actually test this circuit every time it starts on. Now, it is very difficult, say that I have a laptop and some of the ICs are already embedded systems are already placed in that in that PDA or your mobile phone.

Now, before every start of your rebooting of your system or switching on your mobile, you should test the chip. How is it possible, can I bring the chip out of the mobile and put it in tester? No. But, every time you have put tested, because that only can ensure you that basically that every time before it starts up, it tests itself which actually that is called built-in-self-test. I will tell you, why it is called built it? Because, before you starts its operation it tests at itself, so that it gives you some kind of confidence and ok.

Now, I am going to start my operation, I am normal basically, so that that actually gives you more confidence that even if some failures, which may occur after the deployment of the system can also be detected in that manner. And if that fault is detected, then it can flag off an error or if they are redundant modules that can be switched on, and if there is no redundant module like in a home appliance like a mobile phone or a washing machine or a simple microwave controller, then it will give you in a error, and it will also tell you that this chip is not working, and some vendor will come and replace this chip.

So, what is built-in-self-test? Built-in-self-test is very similar to something like ATE base testing, but in that case we do not have any equipment, and you cannot take the chip out of it. So, you will have a on chip pattern generator, and on rep response analyzer. So, in whatever test patterns you have generated, assume that we will put it in a memory and put it into the chip. And same thing, we will also have a golden response comparator, which is happening in the ATE can also be put on chip. So, we are making a miniature ATE, and putting in non-chip that is actually called built-in-self-test mechanism.

(Refer Slide Time: 04:38)



Basic architecture of BIST

This is the pictorial diagram. So, basically if you look at the pictorial diagram, we will basically have the circuit under test, and we have a hardware pattern generator. Here we are going to put all the test patterns, which were generated in the ATPG algorithm. And then we will apply it to the ATE to the cut under test, and then we can have a golden signature that is the expected output, and we can do the comparison.

Very simple story, then you can ask me, then why I should have a full lecture on built-in-self-test. The story is not very simple. ATE is a physical equipment, it is a very huge memory. And it test chips, you can erase the memory a new set of chips comes, and you can keep on repeating it. And it the device is very big.

Can I do something in the chip? If you are going to do it, the size of the test pattern generator, and the size of the ramp, where the expected responses are stored and the compactor would be so large that would be much much greater than the cut itself.

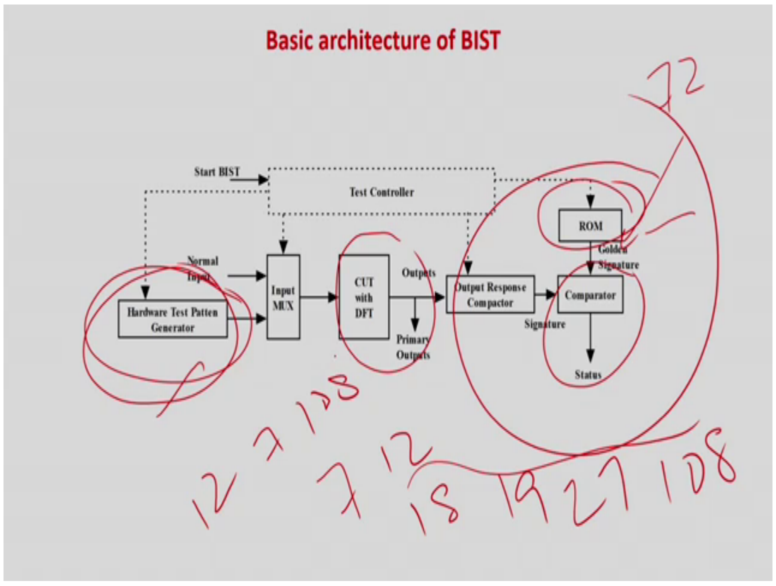
Soon will be like killing a mosquito with a cannon, you are not allowed to do that. You will be able to do a test provided that this hardware pattern generator, and this whole output response analyzer circuit very very small compared to the cut, so that is what is the challenge. Principle is same apply patterns, analyze response as in the off-line test.

But, in built-in-self-test and everything is on chip there also some kind of hardware which will do it, so you have to keep it very very small and limited in size.

How can you make it smaller? First of all you have to take care of very few limited test patterns, which are really required for doing the test that is first thing. Secondly, still you cannot have a full rigid digital circuit to do it. Like for example, I can say you that I have I have to apply patterns, say 1, 3, 7, 9, 12, 18 and so forth, a pattern (Refer Time: 06:09).

We I can very easily generate a pattern generator circuit, which we have already done in second year digital design course. Like we implement next (Refer Time: 06:15) present (Refer Time: 06:17) we can have (Refer Time: 06:18) then you go for a queen necklace your (Refer Time: 06:19) based optimization, and we can simply design a digital circuit to generate the pattern.

(Refer Slide Time: 06:32)



But, assume that if is a 512 bit input circuit, the pattern generator etcetera will be actually become a very large circuit. Similarly, if I have say 72 patterns for which I want to test this circuit, because the 72 test patterns so very important for some perspective. Then the ram should have the expected response from for all the 72 input patterns, this will actually make the things very very large.

So, the challenge of BIST is how I can make it very small in size, but still I have to generate the pattern sequence. Like as I have told you like maybe 7, 12, 18, 19, 27 say

108 these are the patterns mean, I have written in decimal. These are the patterns, which have to be applied to this circuit for doing the test.

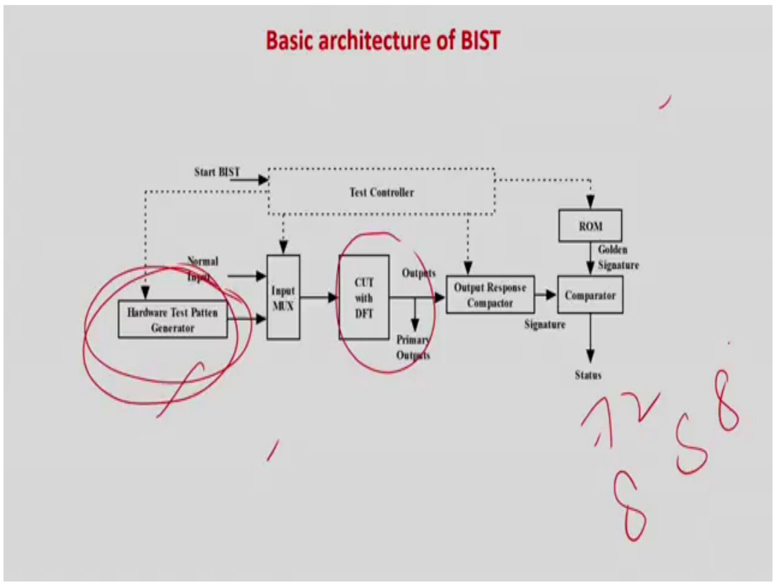
As we generate the same patterns maybe I can do it in a different order, because testing of first stuck-at fault of the second stuck-at fault and or vice versa does not have any impact. I can test the input fault first, I can test the output fault next or I can interchange the position. So, if this is the sequence of patterns required to test a circuit for all faults, I can also swipe, I can make it 12, 7, 108. So, I can easily change out the pattern sequence.

So, as well discuss later that if I can randomize or change the pattern sequence, which does not affect testing, then this circuit for hardware test pattern generator will be very very very very small compared to a deterministic pattern sequence generator like if I want to have absolute like 7, 8, 9, 18, 20, if I can (Refer Time: 07:42) reorder them, then I will show you like you can have a very very small test pattern generator circuit for that purpose compared to the traditional digital design, which we all know for our second year course.

Secondly, this is how I can make the input response test pattern generator circuit lower in size. Secondly, if you look at the output response. So, if I have say 72 input patterns, so if I store the response for all the 72 test patterns, the wrong size also will be very very high, which is required to your golden signal matching. So, there is something called compaction.

So, you have not going to store the expected response for all the 72 patterns. In fact, you will clump many of the output patterns together, and then make a signature out of it. So, I will tell you what do you mean by signature, signature mean is the compaction. So, this is a lossy compaction. So, what do you mean by lossy? If my if I have not compacted that, then all faults should have been detected, but if I make a compaction is a lossy compaction.

(Refer Slide Time: 08:34)



So, may be out of 72 patterns, we can make a batch of 8 8 patterns, I am just taking an example. So, 8 response, I will compact into 1 response; again 8 response, I will compact into 1 response; then I will make a golden signature out of it, and I will do a match. But, they will find that there are aliasing that if I have not the compaction was not there, then all faults should have been detected, but if I make a compaction, some faults will be missed. But, still will see the area overhead of the raw means very very very very low.

So, what it gives me? Obviously, I cannot have as rigorous testing as in the ATE, but still I will have a test, which is basically much much lower in quality and quantity compared to ATE based testing, but still I will get something in based. Instead of something is better than nothing, there is the philosophy.

If I do not have a BIST, I cannot take my circuit before startup. But, if I have a build circuit, even if is not very small part of the original test, which you are doing in off-line ATE, but still that I can apply with a very lower hardware area overhead for the test pattern generator and the response analyzer with compaction. But, still some level of confidence can be obtained if I have a base, so more reliability can be given.

And research have shown that even very easily a very good compaction came, so that the aliasing is less. There is very very low area over a test pattern generation by something called linear feedback shift registers. And we will discuss in details today, which can give the random order patterns, random order in the sense that the way the control of the patterns may not be as you desire, but all the patterns will be generated, but in a different sequence.

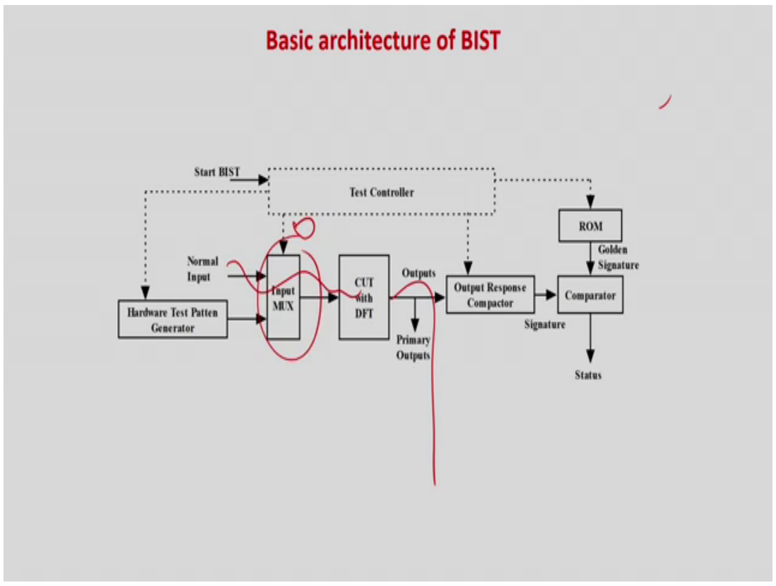
For given sequence we if we change the order, we will show that very load very very low area over hardware test patterns can be generator can be designed, so that you can have a built-in-self-test for maybe around 50 percent level of confidence compared to the ATE, but you can do the test.

But, researcher says solve that we need such low area over it for the compaction based output, and the LFSR base test pattern generation, still you can have a very smaller circuits will be there, but still you can have very good quality pattern generators, very good quality signatures can be generated even with compaction, which will still maintain the level of confidence of testing around 70 percent to 80 percent.

So, if I say that 100 percent is for ATE, at least 70 to 80 percent confidence can be even still obtained with such lower head circuit. So, for that a lot of researchers has been done on how to compact the output, what are the different type of input sequences that can be generated, what are the very peculiar very important test patterns that can be stored (Refer Time: 10:51) do the testing. So, all this will discuss basically.

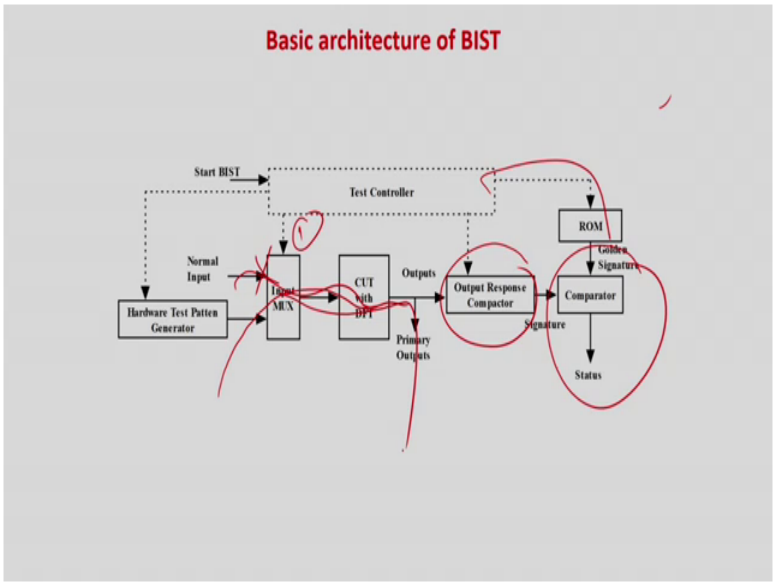
So, the motivation is that build-in-self-test will can give you a certain more of (Refer Time: 10:58) obviously more level of confidence compared may means before or for reliability, because the circuit is tested every time it starts up. But, of course the testing cannot be as rigorous as off-line test.

(Refer Slide Time: 11:13)


Basic architecture of BIST

Now, this architecture tells you that as is an on chip hardware, so there is a always a multiplexer, which will be put in the front of the device. So, if the circuit is normal, so I mean if the circuit is doing is normal operation that is test mode is not, then the circuit is running. Then the test controller will make this line as 0, so the normal inputs will be fed to the circuit, and the output will be going to the primary output.

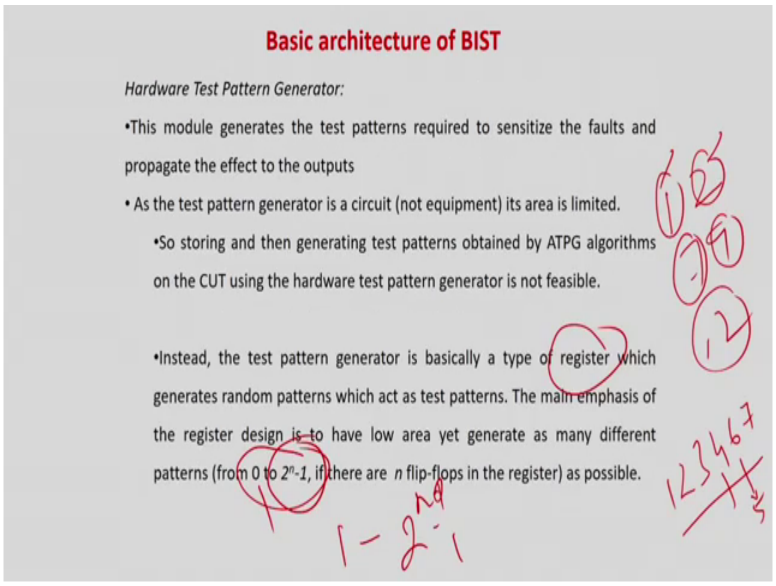
(Refer Slide Time: 11:37)


Basic architecture of BIST

Now, if the test mode is 1, so test mode is 1 means the circuit is just about to start up, so you have put test it devices. So, basically make the mask as 1, so normal inputs will not go it too into it, it will this one will we (Refer Time: 11:41) I mean that depict a test patterns, which are generated by this LFSR or the hardware test pattern generator will be fed to the cut, and basically the wrong will be activated, and the output responses will be compacted, and there is a comparator, which will tell you whether the circuit is properly operating or not.

If it is operating properly, then basically the test mode will become zero, normal inputs will be coming, and the circuit will be doing its normal operation. But, if it is error has been found out, then it will stop the chip, you have to replace the IC that is what is the scheme of built-in-self-test. Now, this architecture is very very simple. Now, what we will concentrate on is how we can go for designing a very low area (Refer Time: 12:17) hardware test pattern generation. And how responses can be compacted, so that we gave have a very low area over it.

(Refer Slide Time: 12:28)



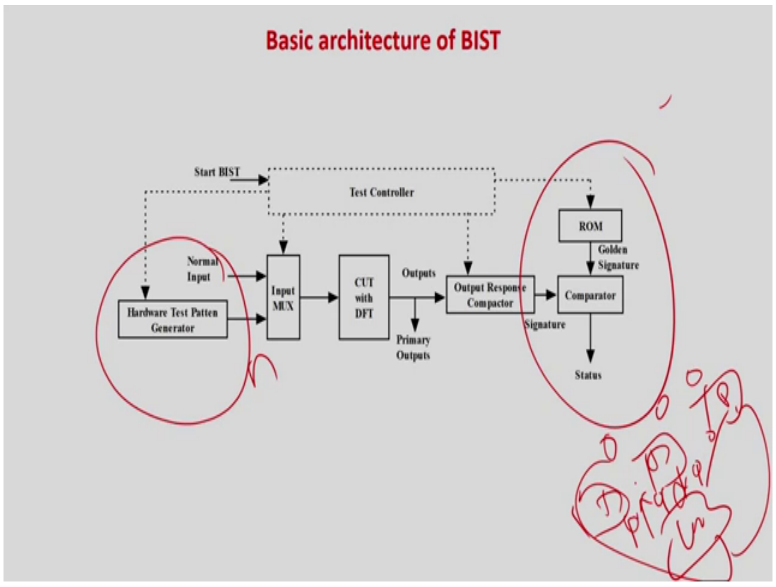
Basic architecture of BIST

Hardware Test Pattern Generator:

- This module generates the test patterns required to sensitize the faults and propagate the effect to the outputs
- As the test pattern generator is a circuit (not equipment) its area is limited.
  - So storing and then generating test patterns obtained by ATPG algorithms on the CUT using the hardware test pattern generator is not feasible.
  - Instead, the test pattern generator is basically a type of register which generates random patterns which act as test patterns. The main emphasis of the register design is to have low area yet generate as many different patterns (from 0 to 2^n-1, if there are *n* flip-flops in the register) as possible.

So, what just as I told you, what are the different components again they are written in this circuit. So, hardware test pattern generator basically what it does, it is simple means it is a register.

(Refer Slide Time: 12:41)

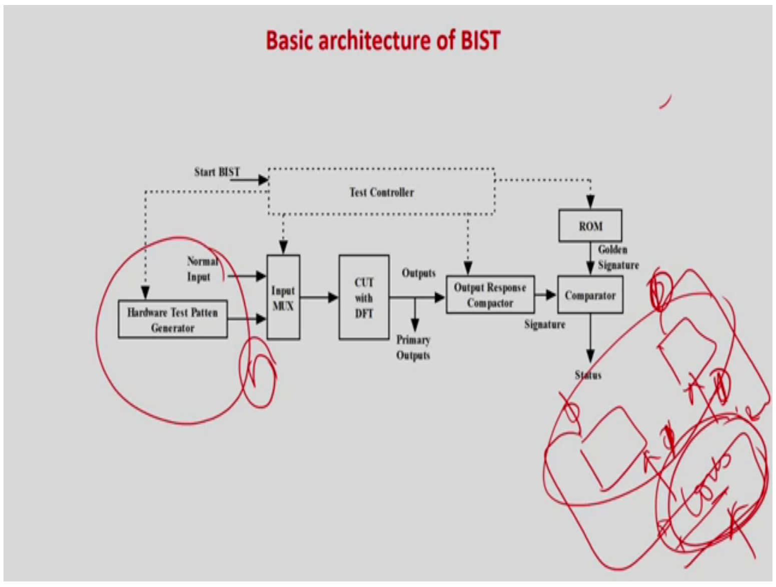


Because, if you have generate patterns, and there is of course it is just some kind of a counter, so n number of if the n bit if this a n bit input, so a n bit register will be there, but so these are the n bit registers.

But, these are some of the combinational, which actually feed these registers and also take the inputs back to this. This is a simple algorithm for the digital counter up or a pattern generator. So, these are the D-flops maybe, so the inputs to D-flops come from a combinational circuit, the outputs of the D-flop again go to the combinational circuit.

So, basically it is a counter, so if it is 0 0 0 0 present state is sorry the present state is 0 0 0 0, the counter the next state will be becoming 1 1 0 0 1. Let me just look here slightly clear it up, and explain you.

(Refer Slide Time: 13:21)



So, this pattern generator is says the n bit pattern generation, let us take n as 2. So, there will be two flip flops. So, there is the input to these D-flops. So, this is actually your combinational cloud, and the outputs of the flip flops are again feedback over here. So, maybe the start may be the present state is 0 0, so it is a simple up counter. So, the next this is the present state is 0 0, next state will be actually 0 1. So, the next state will become 0 1, and this would be the present state, and the next state will be actually 2, so it will be 1 0. And then it will become basically 1 0, the next pattern will be here 1 1, and it will keep on going.

Now, if it is a predefined sequence like 1, 7, 9, 15 etcetera, so this combinational circuit will start becoming larger. So, you cannot do a magic, you cannot relieve if the n bit counter or n bit sequence generator, the number of flip flops will be you cannot do any magic over there. What role it is played basically, how can we reduce this combinational circuit? If you look back your digital design fundamentals, you will find out that this size is not very less.

So, if it is a 512 bit counter is there, the size of this one will be extremely large. So, what role people place or the intelligent the hydroid design techniques play is to reduce this combinational circuit. Because, if it is a n bit input is there by no means, you can actually make this input the counter or this one have less than n bit flip flops that is not possible.

So, what is the hardware test pattern generator basically is nothing so it will have a state register of n bits. So, but as the test pattern generator circuit is a circuit not an equipment, its area is limited as I already told you. So, storing and then generating the test patterns obtained by ATPG algorithm on the CUT using the hardware test pair generator is not feasible. So, therefore you it cannot be a predefined counter. So, basically instead the test pattern generator is basically a type of register, which generates random patterns, so that pattern frequency have not containing.

As I told you, if the first test pattern is tested last sorry the first fault is tested last and the first and the last part fault is tested first, I have no problem fault should be tested. So, basically test pattern sequence I do not control, but the number of test patterns I control.

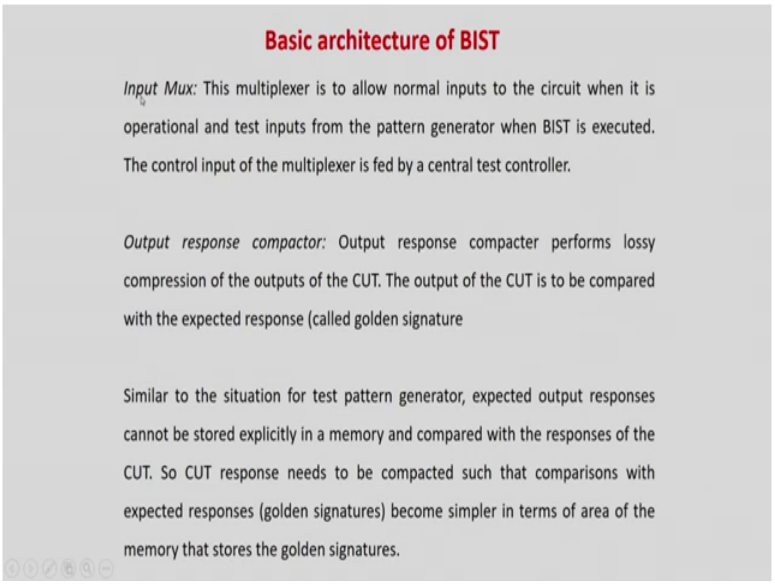
Like for example, as I told you 1, 2, 7, 9, 12 see are the test patterns ok, it may be detecting the 1st fault, the 2nd fault, 3rd fault may be I have number the faults in numbers. But, this I have no problem if I start with 7, then with 12, then with 9, then with 12, and the 9, I have no problem in that. So, as will show you there is something called linear feedback shift register based pattern generators, where if you randomized this order these size of this combinational circuit, this combinational circuit becomes very very less.

So, basically the main emphasis of the register design as to have low area yet generate as many possible patterns in the flip flops, so because I mean as I give here I given say example, like say 1, 2, 7, 9, 12. So, this test pattern the state space is bit (Refer Time: 15:56), but it may not be the case. It may be 1, 2, 3, 4 maybe one gap I will have 6 maybe 7. So, these are all the patterns, which is required for a 3 bit counter. So, maybe only one is remaining that is actually say 1, 2, this 5 is basically remaining, which I do not require as a test pattern.

So, basically my main job here will be to reduce this combinational circuit area, but still generate as many as patterns possible from 0 to 2 to the power n minus 1 full range. And in fact, luckily the theory has been established that there exist circuits with very very low area over it, which can generate patterns from 2 to the power n minus 1. In fact, I will show you zero is not possible to be generated.

So, we have patterns basically from 1 to 2 to the power n minus 1 almost full range patterns can be generated from one to do the pair, but you know very low area over it, but this pattern sequence cannot be controlled. Maybe 2 to the power n minus n minus 1s will come over here first, maybe after that 1, then 9, then 7 and so forth, so that is what is the hardware test pattern generator.

(Refer Slide Time: 16:51)



Basic architecture of BIST

Input Mux: This multiplexer is to allow normal inputs to the circuit when it is operational and test inputs from the pattern generator when BIST is executed. The control input of the multiplexer is fed by a central test controller.

Output response compactor: Output response compacter performs lossy compression of the outputs of the CUT. The output of the CUT is to be compared with the expected response (called golden signature

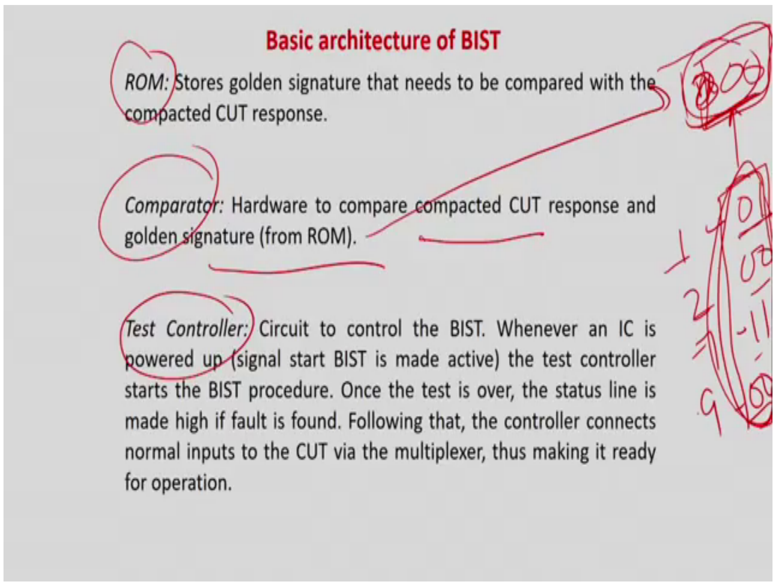
Similar to the situation for test pattern generator, expected output responses cannot be stored explicitly in a memory and compared with the responses of the CUT. So CUT response needs to be compacted such that comparisons with expected responses (golden signatures) become simpler in terms of area of the memory that stores the golden signatures.

Input Mux as I already told you, it tells that when the system is operating in a normal operation, PI should come in primary inputs would coming in case of test the patterns on the ATP that is hardware pattern generation should come in.

Output response compactor as I have already told you that for each individual ATP test patterns, if I store the expected responses in the ramp, so it will be very large, so basically we compact the response, and it is a lossy compaction, so in that means, out of say 100 responses would be making bunches of 10. So, every 10 10 10 10 or some type of algorithm, I can use which can compact the responses and make a 1 bit or 8 bit or some x number of bit representation for this 10 output responses.

And it will be represent by single bit or 2 bits whatever you can decide. But, that is a lossy compaction, but still we have to do it for is minimizing the area of it, so that is actually call output response compactor.

(Refer Slide Time: 17:40)



Basic architecture of BIST

ROM: Stores golden signature that needs to be compared with the compacted CUT response.

Comparator: Hardware to compare compacted CUT response and golden signature (from ROM).

Test Controller: Circuit to control the BIST. Whenever an IC is powered up (signal start BIST is made active) the test controller starts the BIST procedure. Once the test is over, the status line is made high if fault is found. Following that, the controller connects normal inputs to the CUT via the multiplexer, thus making it ready for operation.

Then ROM, which actually stores the respected x respected response of the compacted input sequence or the compaction means for all 10 output see first explicitly store the responses wrong size will be large, so I have make a compaction.

So, they say input pattern 1, 2, 7, 9, 12 I made a club, so this is the all the output responses may be 1, 0, 7, 0, 1, 9, 0, 1, 2, 3 these are outputs expected for these are this inputs. Like see input as I say 1, 2, 7, 9 may be these are the four inputs, and may be the output expected in this case is 0 1, 0, 0 say this is again 1 1 may be 0 0. So, this for these four for every four the input sequences the output sequence, I will make it as a single response as I am compacting.

So, these four responses I will compact, and maybe I will represent it by 2 bit or 1 bit some algorithms are there as will discuss, so but all the output explicitly, I will not store the expected responses. Basically, I will make a compaction and store it into the ROM mean row less bit on number of bits. Like here in this case 1, 2, 3, 4, 5, 6, 7, 8 eight bits are required to do this. But, if I compact, it maybe I will be representing by 2 bits may be 0 0. How I am getting 0? 0 from all this, we will discuss that algorithm later.

So, I will represent it in only 2 bits or may be 1 bit or even 3 bit also I can keep it, but will less than 8 bits, and I will put it into the wrong. So, basically now whenever I am getting all the responses for this pattern like 1, 2, 7, 8, this one will be compacted and compared with the and that value be compared with the expected compacted response, right for 1, 2, 9, 7 and 9 in as input vectors, the output respected at this and this.

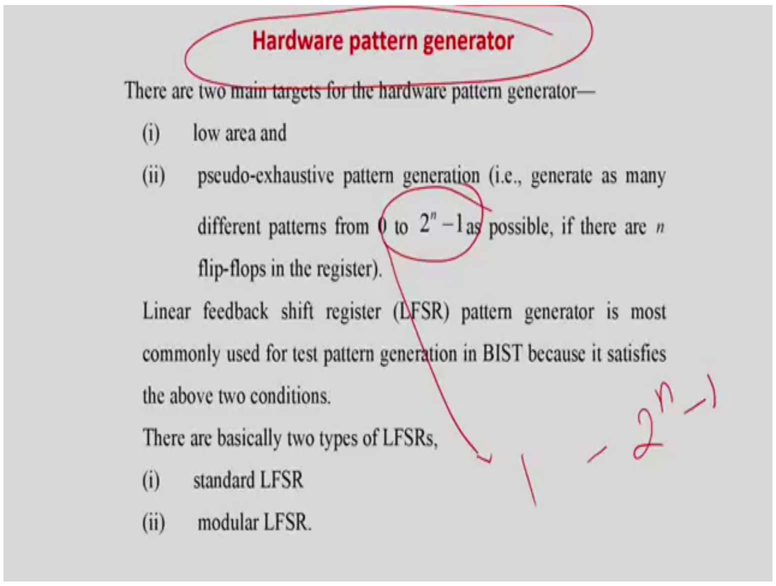
So, explicitly I will not storing the ROM these four values, I will make a compaction maybe at is a called a signature, which will be stored in the ROM with less number of bits. And whenever the circuit under test will be applied with these patterns, these outputs should be again compacted, and compared with the expected compact expected compacted response.

So, compare done compares the compacted outputs with the expected compacted response that is what is actually the ROM stores. So, comparator is very simple, it will compare the expected compacted CUT response with the golden signature from the ROM. So, basically this is your golden signature that is the expected compacted response, and basically this is the CUT outputs. They are again compacted and mastery the golden signature.

There is something called test controller, it basically decide when to start the based, when to stop the based if there is a fault, what it should do, so it is a controlling generation circuit. Suddenly, what it does if the if the device has to be tested or this device is starting up, it will actually make the mask in such away that random patterns from sorry the patterns from the hardware generator is applied. This is the control patterns, and testing is done.

And when the (Refer Time: 20:12) and after it test is ok, it will switch the input, so that inputs from the normal primary inputs come to the circuit, so that is one of the job of the test controller. If any fault occurs, it will flag off that this disease having of and something else can be actually done over these. So, these are actually all the components, I have described for this hardware.

(Refer Slide Time: 20:33)



Hardware pattern generator

There are two main targets for the hardware pattern generator—

(i) low area and

(ii) pseudo-exhaustive pattern generation (i.e., generate as many different patterns from 0 to $2^n - 1$ as possible, if there are n flip-flops in the register).

Linear feedback shift register (LFSR) pattern generator is most commonly used for test pattern generation in BIST because it satisfies the above two conditions.

There are basically two types of LFSRs,

(i) standard LFSR

(ii) modular LFSR.

Now, basically what we will do, we are now going to look in details about the hardware test pattern generator, how it has to be designed, how it has to be tested, basically how it is so such a miniature version. So, if it is generating all the possible patterns for a given n bit number n give bit input bus or if the number of input bits are n, it is generally a large sample space.

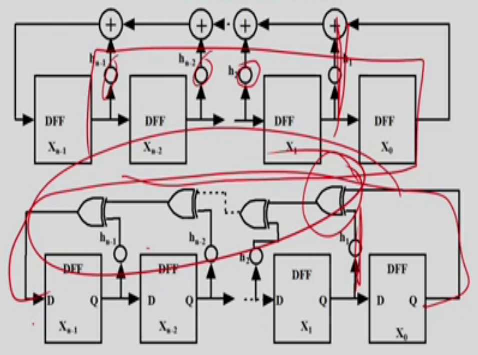
Like for example, you should not call it 0, it cannot generate from all zeros that is the only thing that is not possible to be generated, you generally generates all patterns or very huge number of patterns from 1 to 2 to the power n minus 1, if n is the number of basically your input with bus size right. So, we will see the (Refer Time: 21:10) very interesting to see that even it just with a single motive modification that I can randomize the input patterns, we will show that very very less area over a pattern generator can be designed.

So, now we are going to look, how such things can be designed by their to typically two different type of ways to do it, which we call standard linear feedback shift register or modular feedback shift register. These are the two specialized register design, which enables you to get a very huge state space of inputs from 1 to 2 to the power n minus 1, now we will be looking at that.

So, what we are trying to look at is that how can we make that register size area small for that combinational part, but still you have to generate a large spectrum of the input space that is from 0 to 2 to the power n minus 1, but as we will see 0 is not possible to be generated.

(Refer Slide Time: 21:56)

Standard LFSRs



A properly-designed LFSR can generate as a near-exhaustive set of patters, as it can cycle through distinct $2^n - 1$ states (except 0s is all flip-flops). Such a properly designed LFSR is known as a maximal length LFSR.

Now, let us look at this circuit structure, which makes this magic. So, this is one circuit structure, which is called standard linear feedback shift register. So, in this case if you will see, you will see that basically if I try to zoom it for you, so we will see that basically what happens?

So, there is a sequence of flops 1, 2, 3, 4 and up to X naught. And everybody is connected these everybody is feeding the direct flip-flop directly to the next, but only one MSB flip-flop is connected from the output of the LSB flip-flop. But, it is not directly connected as in the case of others, like these are the ordinary connections, but he is a feedback connection. And in the feedback, you will have lot of XOR gates right.

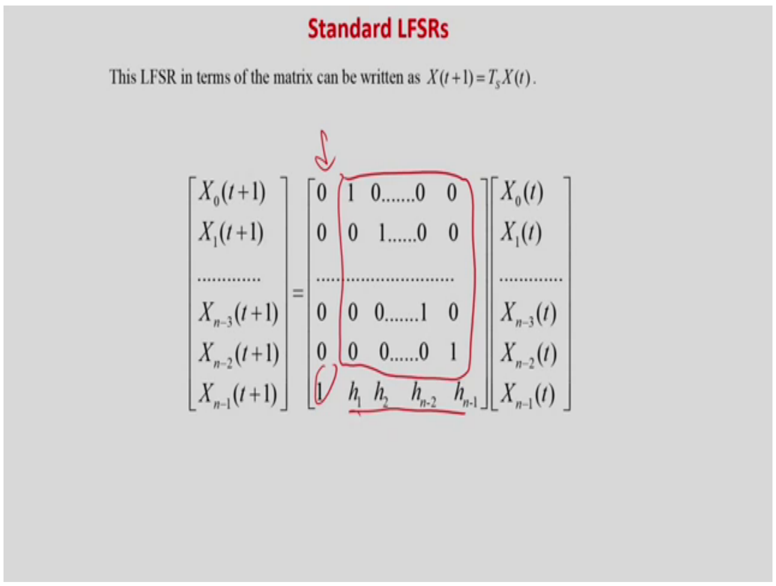
This one here is there can be an XOR gate, which will which is actually the output of X 1, XOR with the output X naught, even output of X naught 2 X 3, X 4 everything can be there or it may not be there. What I mean to say is, this is the proper structure everything will be a direct connection, simple shift register.

But, the output of the LSB register b flip-flop to the MSB flip-flop there is a feedback chain. And this output this output will be depending on X this one XOR, this one XOR, this one XOR, this one, but everywhere you may not have a XOR gate. So, depending on where you place the XOR gate, and where you do not place the XOR gate, the sequence of patterns or the in output spectrum will depend.

Here of course, you can very easily feel that all 0s cannot be generated. Like if I put on all 0 over here, so what is going to happen? Everywhere the 0 will shift, and if I put any number of XOR gates, XOR, XOR with XOR with 0s all XOR is the 0. So, it will be a 0, and it will be locked at 0.

So, any LFSRs know whether is modular or standard, if 0 is basically if given as the initial seed, it is going to lock. So, accept all 0s huge spectrum of inputs can be generated, and what sequence, what pattern depends on where and how you place this XOR gates.

(Refer Slide Time: 23:47)



Standard LFSRs

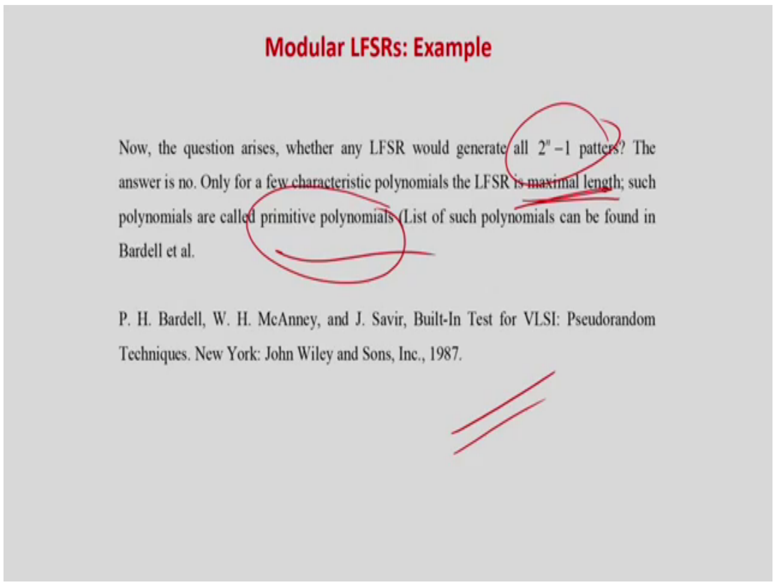
This LFSR in terms of the matrix can be written as $X(t+1) = T_s X(t)$.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ \cdots\cdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0\ldots\ldots0 & 0 \\ 0 & 0 & 1\ldots\ldots0 & 0 \\ \cdots & \cdots & \cdots\cdots & \cdots \\ 0 & 0 & 0\ldots\ldots1 & 0 \\ 0 & 0 & 0\ldots\ldots0 & 1 \\ 1 & h_1 & h_2 \quad h_{n-2} & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ \cdots\cdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

So, actually there are lot of mathematics from coding theory, which is involved which predicts that givens if I put an XOR gate here, here, and I will drop it here. What the pattern sequence generated, what is the spectrum. So, the there is a huge amount of mathematical logic and theory, which exist in coding theory literature, which tells you how you can orient is XOR gate to get this output patterns in a pre requisite fashion or I mean in a randomized fashion, but still the input spectrum.

Randomizing the sense that for a given sequence of or type of XOR gates and basically for a given type if n length, what is the sequence pattern, and what are basically which are the miss patterns, and which are the patterns, which is covered from 1 to 2 to the power n minus 1 depends on how you select these XOR gates.

(Refer Slide Time: 24:44)



Modular LFSRs: Example

Now, the question arises, whether any LFSR would generate all 2^n-1 patterns? The answer is no. Only for a few characteristic polynomials the LFSR is maximal length; such polynomials are called primitive polynomials (List of such polynomials can be found in Bardell et al.

P. H. Bardell, W. H. McAnney, and J. Savir, Built-In Test for VLSI: Pseudorandom Techniques. New York: John Wiley and Sons, Inc., 1987.

But, that theory I am not going to go in, because proves etcetera quite involved. Only the only from and for most of the test perspective, there is something called basically which we will see the name is characteristics basically what I actually call primitive polynomial, which is the term.

So, basically I will the just keep the term in mind for the time being that the theory exists which tells you that is the primitive polynomial that terminology, which tells you where you put this XOR gates, and where not to put the XOR gates. If there is no XOR gates, therefore in that case it will not depend. Like if I do not put an XOR gate over here, so it will be directly connected over here. So, this flip-flop will be just be a shifting flip-flop.

So, and the way the flip-flops are connected will define a mathematical function, which is called primitive polynomial. So, for a given n and given the primitive polynomial that primitive polynomial will tell you what are the input sequence generated, and whether you can generate the entire sequence from 1 to 2 to the power n minus 1. So, depending on your requirement, you can select a primitive polynomial, and accordingly you can build this circuit.

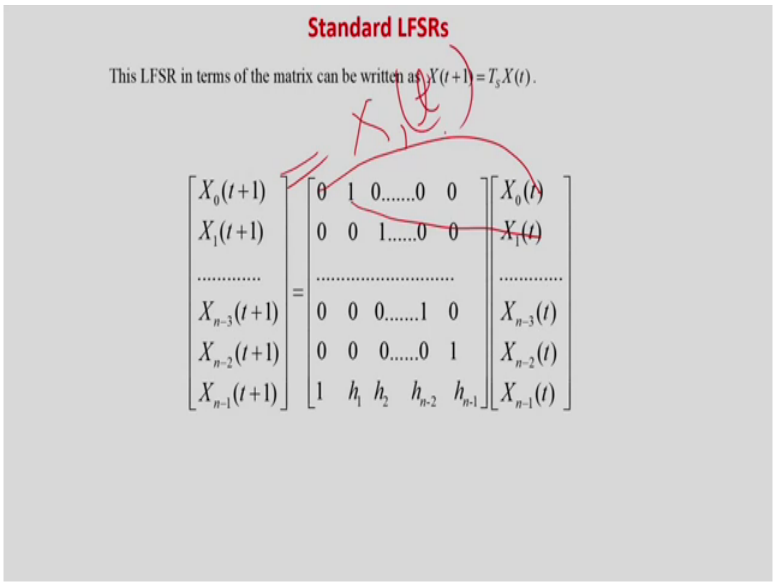
So, why the primitive polynomial works that why only for this primitive polynomial, you can generate an exhaustive set of patterns, because more or less our job is to require to generate as many patterns as possible, because then your case the coverage will be higher. So, why such primitive polynomials only generate such kind of exhausted pattern set, it is a theoretically involve subject. You can go to any coding theory book, constantly linear feedback shift registers you can know the theory behind it.

But, here what is our test engineers will do, they will given us (Refer Time: 26:01) the value of n, then it will go to the literature and find out the primitive polynomials for that. Accordingly, design the LFSR, and it will generate the huge spectrum of numbers. So, we will go by this. So, how it is represented?

So, in this way it is represented by a matrix, these are matrix which represents it. So, here basically it is a matrix with all diagonals as 1, this column everything is 0, there is a 1 over here, and the last row this is h 1 h 2 h 3 h n minus 1 and h minus 2 that is the h s. So, what are the h s? H s are basically these values. So, if you are putting a flip-flop in for this flip-flop that corresponding h will be 1. And if there is no flip-flop in this sorry no XOR gate in that case, like for example, if this XOR gate is not there, then this is not there, it is a direct feedback. So, in this case, h 1 will be equal to 0.

So, whether I am putting an XOR gate for the corresponding flip-flop or not, basically these h values will remain right. And this is the column, so all 0s and a single 1. And this is the matrix. So, this is what this actually represents this LFSR now how, will explain this? Like for example, if you look at it, so for all cases like X naught is equal to X 1 next stage, X 1 is equal to X 2, X X n minus 2 is equal to X 1. So, how it happens? So, if you look at what is the value of X naught, t t plus 1.

(Refer Slide Time: 27:20)



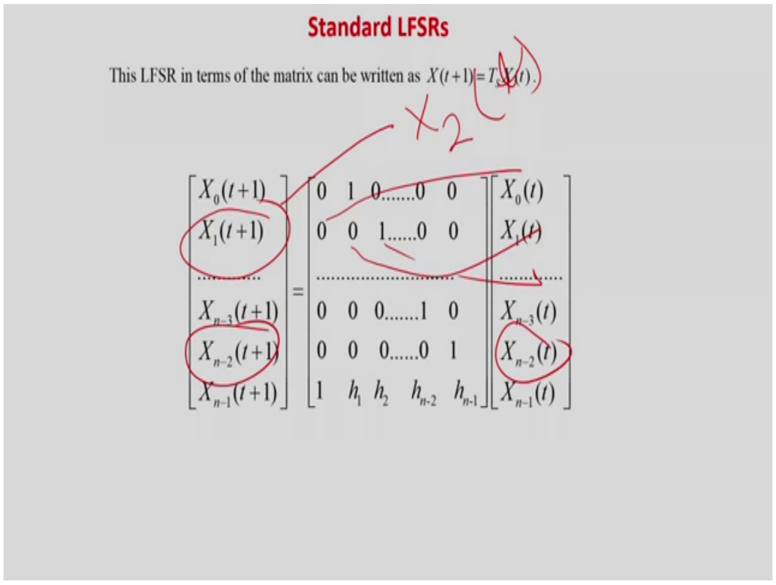
Standard LFSRs

This LFSR in terms of the matrix can be written as $X(t+1)=T_S X(t)$.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ \cdots\cdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0\ldots\ldots0 & 0 \\ 0 & 0 & 1\ldots\ldots0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0\ldots\ldots1 & 0 \\ 0 & 0 & 0\ldots\ldots0 & 1 \\ 1 & h_1 & h_2 \quad h_{n-2} & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ \cdots\cdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

So, in this case basically if you look at, this is multiplied with this, this is multiplied with this and so forth and all others are 0. So, X naught t minus 1 will be equal to X 1 of t.

(Refer Slide Time: 27:31)



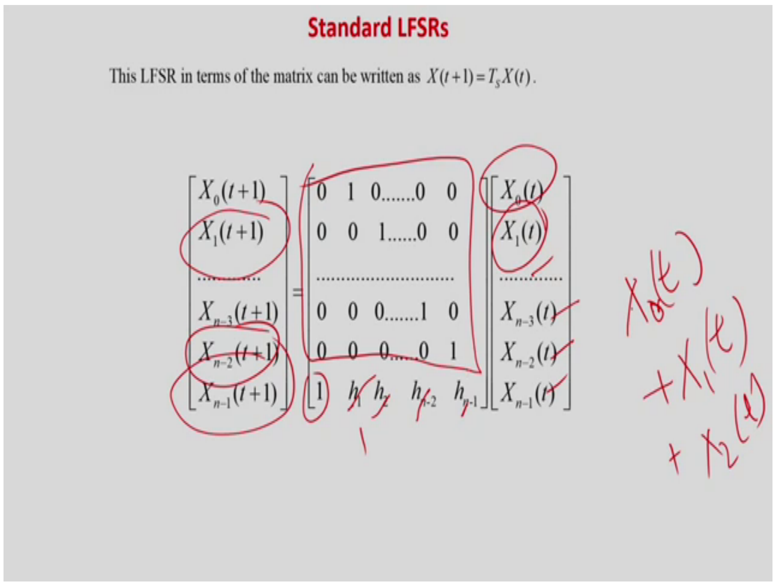
Standard LFSRs

This LFSR in terms of the matrix can be written as $X(t+1)=T_S X(t)$.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ \cdots\cdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0\ldots\ldots0 & 0 \\ 0 & 0 & 1\ldots\ldots0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0\ldots\ldots1 & 0 \\ 0 & 0 & 0\ldots\ldots0 & 1 \\ 1 & h_1 & h_2 \quad h_{n-2} & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ \cdots\cdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

Similarly, if you look at X 1 t plus 1, so in this case if you look at, so this one is multiplied by this one is multiplied by this one will be something multiplied by this. So, this one will be actually X 1, this one will be actually X 2 of t, so that way. So, similarly you can easily calculate that basically this one will also will be depend basically this one will be dependent on this and so forth.

(Refer Slide Time: 28:11)



Standard LFSRs

This LFSR in terms of the matrix can be written as $X(t+1) = T_S X(t)$.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ \cdots\cdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \ldots\ldots 0 & 0 \\ 0 & 0 & 1 \ldots\ldots 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 \ldots\ldots 1 & 0 \\ 0 & 0 & 0 \ldots\ldots 0 & 1 \\ 1 & h_1 \; h_2 & h_{n-2} & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ \cdots\cdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

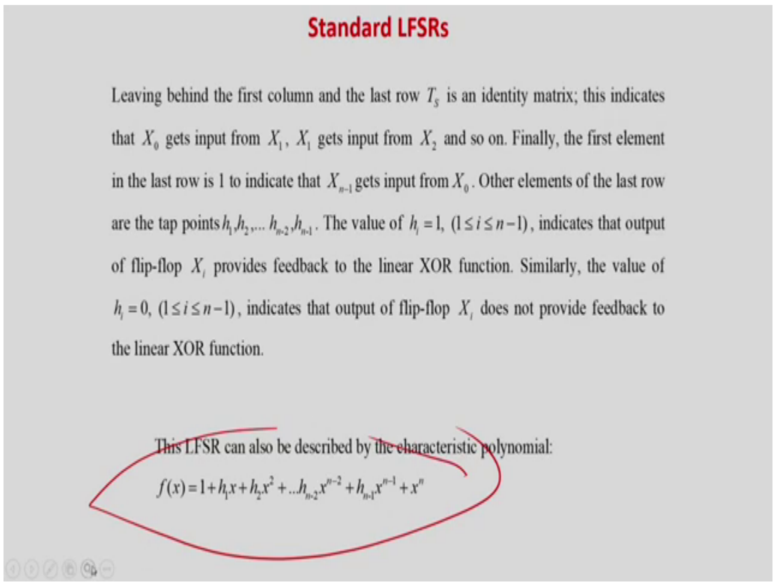
$X_0(t)$
$+X_1(t)$
$+X_2(t)$
1

So, for all the flip-flops from X naught to X n minus 2 like this will be just depending on the previous flops (Refer Time: 28:05) depend the previous flop values, so which is described by this part. Now, very interesting is how X minus n minus 1 that is the MSV flip-flop decide, it is a function of this. So, always if know XOR gates are there, still there will be a feedback over there. So, it is basically equal to X n minus 1 MSV flip-flop is equal to X naught t, now if X h is equal to a 1.

So, if X is equal to a 1 that means, there is a flip-flop over here sorry XOR gate is over here, so it will be again XOR with this gate. So, in this case h 1 is multiplied by this 1, so it will be actually equal to XOR X naught equal to t, then it will be XOR with X 1 of t, again XOR with X 2 of t and so forth, if these corresponding bits sorry if these corresponding bits are 1. So, the everything is a 1, then it will be XOR with X naught, X 1, X 2, X 3.

So, if all the flip-flops are here is in this picture, this one will be an XOR of the output of all the flip-flops. So, this row basically tells you what is the input driving for this. So, just with a little careful looking at it, you will be (Refer Time: 29:12) is equal to find out how this linear feedback shift register is represented by this matrix right.

(Refer Slide Time: 29:23)



Standard LFSRs

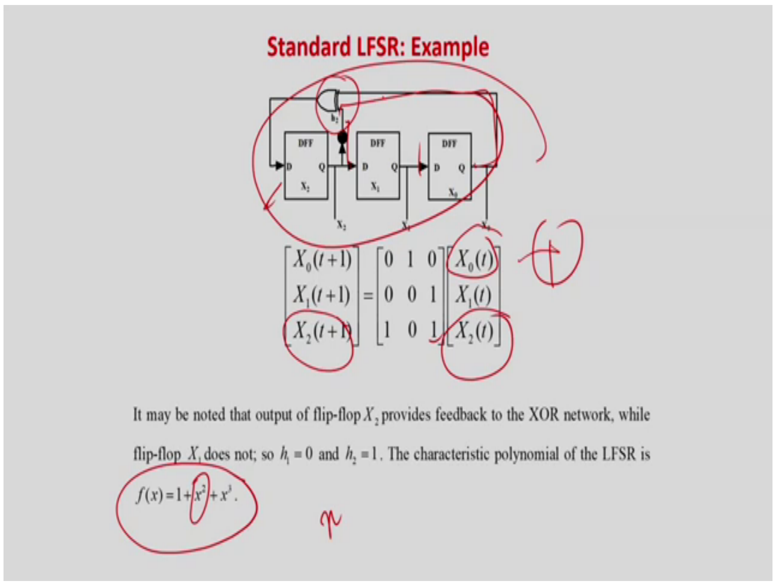
Leaving behind the first column and the last row T_S is an identity matrix; this indicates that X_0 gets input from X_1, X_1 gets input from X_2 and so on. Finally, the first element in the last row is 1 to indicate that X_{n-1} gets input from X_0. Other elements of the last row are the tap points $h_1, h_2, \dots h_{n-2}, h_{n-1}$. The value of $h_i = 1$, $(1 \le i \le n-1)$, indicates that output of flip-flop X_i provides feedback to the linear XOR function. Similarly, the value of $h_i = 0$, $(1 \le i \le n-1)$, indicates that output of flip-flop X_i does not provide feedback to the linear XOR function.

This LFSR can also be described by the characteristic polynomial:

$$f(x) = 1 + h_1 x + h_2 x^2 + \dots h_{n-2} x^{n-2} + h_{n-1} x^{n-1} + x^n$$

So, in now basically this matrix can be represent as a characteristic polynomial like this. So, I am zooming it for you, so it is just a simple way of writing x h 1 square n minus 1 and x n. So, this actually is a mathematical representation, the characteristic polynomial representation of the matrix. So, if you are not putting a XOR gate in the respective position, such h s will be actually becoming 0. So, these terms will be going out example that is better.

(Refer Slide Time: 29:45)



Standard LFSR: Example

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \end{bmatrix}$$

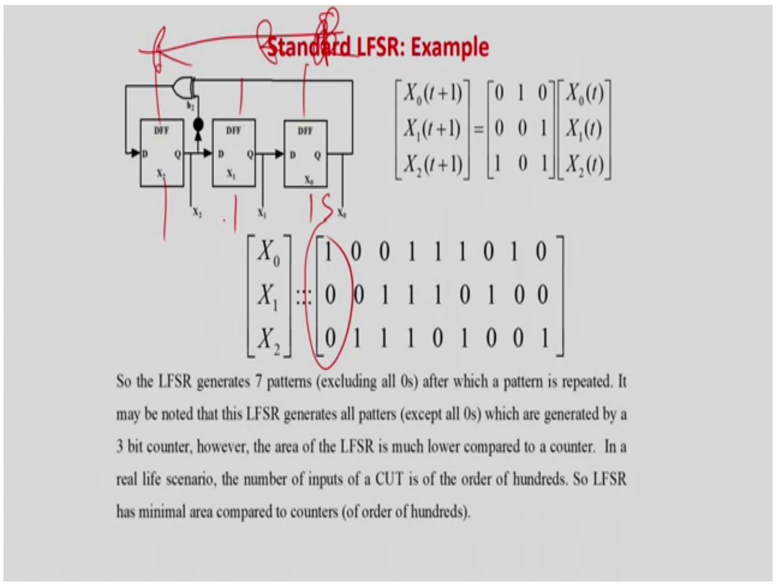
It may be noted that output of flip-flop X_2 provides feedback to the XOR network, while flip-flop X_1 does not; so $h_1 = 0$ and $h_2 = 1$. The characteristic polynomial of the LFSR is $f(x) = 1 + x^2 + x^3$.

So, if you look at this is a LFSR right. So, in this case three bits, so the matrix also a 3 cross 3. So, this part is a matrix with all the diagonal elements as 1, so that is not a problem, this is a standard. Now, if you look at, so in this case there is only one (Refer Time: 30:00) only one flip-flop, which is at the output of X 2 or here actually there is no flip-flop over here. So, if you look at this corresponds to x h 1, so this is 0. This corresponds to h 2, so it is a 1 over here. So, these two represents this flip-flop that there is a XOR gate over here, and this 0 represent there is no XOR gate over here.

Now, look at what is the value of X naught. So, X naught t plus 1 is actually equal to X 1. So, this one is very simple. Similarly, if I study the value of X 1 at t, it is also nothing but the value of X naught of t. Now, X 2 of t. Now, important is, what is the value of this one this guy, so it is basically equal to X naught this one; second bit is 0 XOR with this one, because this is a 1. So, basically you can see the output of this is XOR of this one and XOR of this one.

So, basically this is represent this circuit is represented by this. And the polynomial is 1 x square x cube. So, 1 and x cube will be anyway there, but only x square is there, x is term the term is missing, because this is not filled by XOR gate. So, I can put XOR gate here also, all the four combinations I can use, no x basically no XOR gate is not the combination to be used is all locking 0 sequence, either I can also put a XOR gate over here, I can remove this or I can also put a XOR gate over here. So, this is how I can basically do it.

(Refer Slide Time: 31:20)


DFF
D
Q
X_2
X_1
X_0
h_2

Standard LFSR: Example

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \end{bmatrix}$$

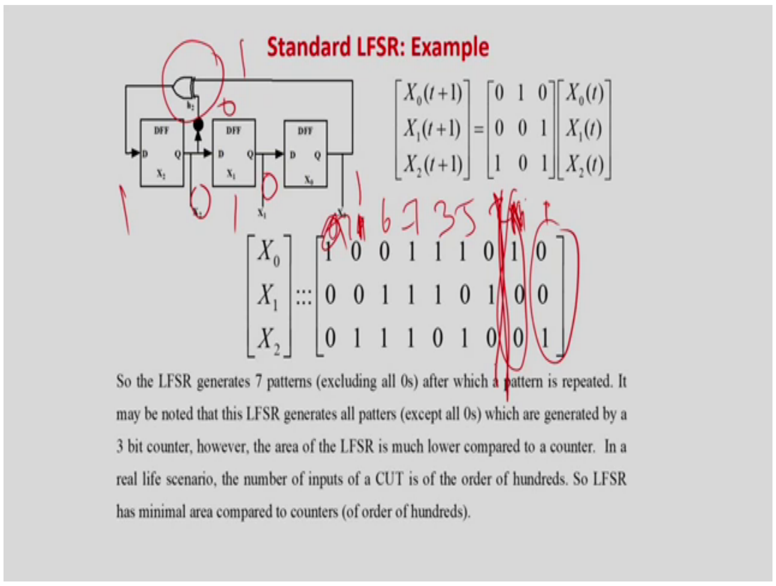
$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} ::: \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

So the LFSR generates 7 patterns (excluding all 0s) after which a pattern is repeated. It may be noted that this LFSR generates all patters (except all 0s) which are generated by a 3 bit counter, however, the area of the LFSR is much lower compared to a counter. In a real life scenario, the number of inputs of a CUT is of the order of hundreds. So LFSR has minimal area compared to counters (of order of hundreds).

Importantly what it can generate. So, first there is something called a seed, first you have to put a seed, again as I told you can easily find out that if I put this seed as 0 0 0, so you can see that it will be again a 0, because XOR 0, it will be locked. So, all 0s cannot be generated by any kind of LFSRs, but that is not a problem.

Basically if you say that I have to put a all 0 as a test pattern, then what you can do is that you can make all these reset make all these resets, first generate all the 0 patterns, and then basically generate a seed pattern. So, how can I generate the seed pattern, the seed pattern is 1 0 0. So, these are again reset, but it is a first bit X naught is a 1, so you can make it as a set. After that, you need not put any set and reset, and automatically these patterns will be generated. So, let us see how it is generated.

(Refer Slide Time: 32:06)



Standard LFSR: Example

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \end{bmatrix}$$

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} ::: \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

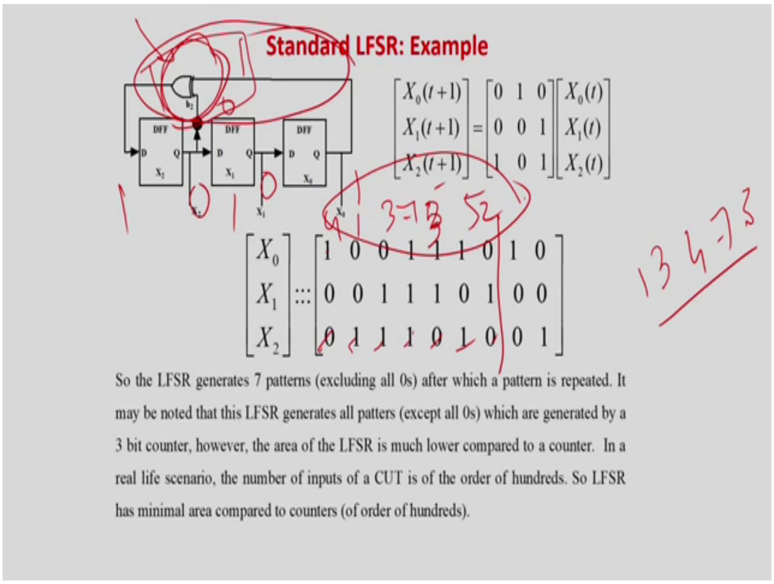
So the LFSR generates 7 patterns (excluding all 0s) after which a pattern is repeated. It may be noted that this LFSR generates all patters (except all 0s) which are generated by a 3 bit counter, however, the area of the LFSR is much lower compared to a counter. In a real life scenario, the number of inputs of a CUT is of the order of hundreds. So LFSR has minimal area compared to counters (of order of hundreds).

So, in this case it is saying it is a 1, it is a 0 and it is a 0. So, the next input will be a basically if you look at, so X naught is 0. So, next this is the first sequence, so X naught is a 1, and these are 0 0. The next batch what is going to happen, this X 1 is going to become X naught right, then X 2 will become basically X 2 will become X 1 in the next case. And what about this, this one is accurately find by this XOR gate. So, it is a 1 and a 0, so XOR is a 1, so we are going to get the value as 1.

Similarly, you can find out that this is the pattern sequence it would be generated one after another. And if you look at it, what is the sequence? It is maybe I maybe I call it as 0 4 6 7 maybe it is 3 5 2 4 something like that. So, it sorry it is a actually a 4. And after that, basically this is what is the generated. So, again it is a repetition, so it is a 4, then I call it is a 1. So, this after see you have to see that after this one, again the pattern will start repeating.

So, what basically I have found out, how many different patterns are generated. So, I started with this, and this is what is a repetition. This one I if I call it is a 4, if I call it the at the LSB, so it is a 4, and this is basically a 1. So, in this case 1 and 4, so basically I am generating all patterns up to this. So, what are the patterns I have generated? Because, from here the repetition starts, so it is a 4 1 6 7 3 and 5; so 1, 2, 3, 4, 5, 6. So, six patterns are generated by this sequence.

(Refer Slide Time: 33:47)


DFF
D
Q
X_2
DFF
D
Q
X_1
DFF
D
Q
X_0
h_2
1 0 1 0
4 1 3 7 3 5 2
1 3 4 7 3

Standard LFSR: Example

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \end{bmatrix}$$

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} ::: \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

So the LFSR generates 7 patterns (excluding all 0s) after which a pattern is repeated. It may be noted that this LFSR generates all patters (except all 0s) which are generated by a 3 bit counter, however, the area of the LFSR is much lower compared to a counter. In a real life scenario, the number of inputs of a CUT is of the order of hundreds. So LFSR has minimal area compared to counters (of order of hundreds).

1, 2, 3, 4, 5, 6 and 7. So, 2 to the power n minus 1. Just by using a simple XOR gate, I can generate it. But again, the pattern sequence you can see, it is 4 1 3 7 5 right sorry 3 5 2. So, I have no controlling the sequence. So, with this simple XOR gate, I can generate almost entire spectrum for a 3 bit keys. So, only all 0s are not generated, I accept that all other patterns are generated right. So, it is very interesting.

But, if I would have (Refer Time: 34:08) sequence something like that that 1 3 4 7 say in ascending order, so in that case is an up counter or a particular pattern counter. So, in that case, you cannot do it with a simple XOR gate, it would be quite large circuit. Just by looking at the digital design fundamentals, we can find out how to design such kind of interesting counters, like up and down counters, the particular sequence generator. So, it is never a single gate, it is a large bunch of gates.

So, for four 3 bits, only one gate is doing it. Very interestingly in case of LFSR as we will see, even if the very high increasing the number of bit size bit width, the number of XOR gates required will be not very high. But, if you are going first (Refer Time: 34:44) pattern generator sequence using digital design final state machine based optimization sequence, it is a very very large hardware maybe hundreds of gates will be required for 32 bit input or something like that. So, here this saving is huge.

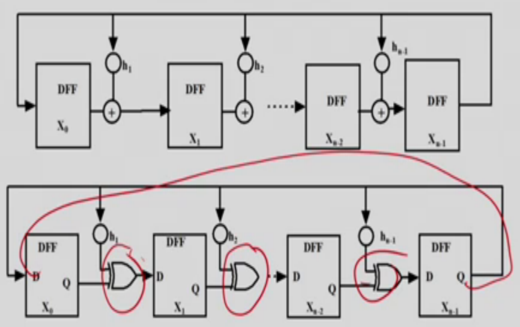
So, again I am reemphasizing that why this LFSR is very good for (Refer Time: 35:03). Because, it will for most of the cases like if you are using a primitive polynomial, entire sequence is generally generated from 1 to 2 to the power n minus 1, but the order may be slightly randomized. But, still I have no problem, because faults can be tested in any order, but the gate size in this case is only one XOR gate.

But, if you are going to generate a pre-defined sequence, this combinational clout is going to be quite high, so that is actually the big gain for LFSR seen based. That even if it is a large number of bits also, the number of XOR gates will be (Refer Time: 35:33) very very right, so that is what is the boon.

Now, some people had a problem over here in such kind of standard LFSRs, because they complained that you see this one is comprising of so many XOR gate sequence. If it is a 32 bit, there may be in the most case 32 bit input levels. So, it may be a very slow design, because something can, because the delay of the XOR gate that can happen.

(Refer Slide Time: 35:56)

Modular LFSRs



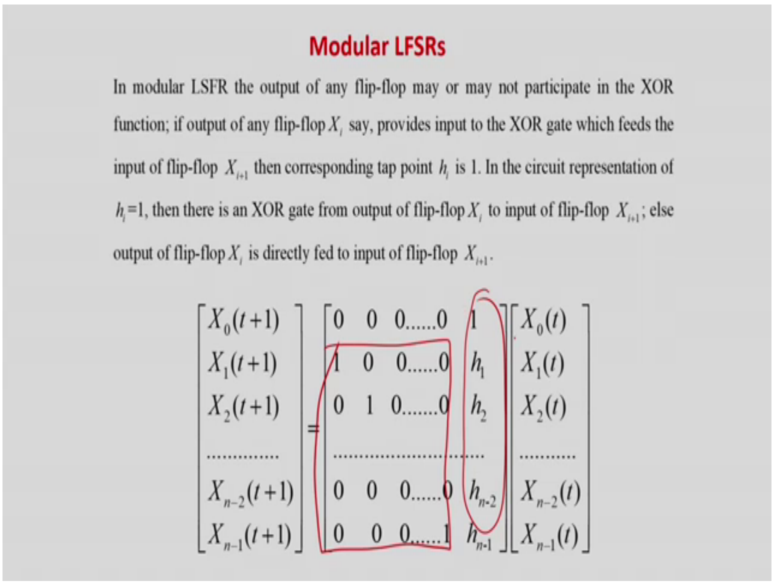
.The difference in modular LFSR compared to standard LFSR is due to the positions of the XOR gates in the feedback function; in modular LFSR XOR gates are in between adjacent flip-flops. Modular LFSR works faster than standard LFSR, because it has at most one XOR gate between adjacent flip-flops, while there can be several levels of XOR gates in the feedback of standard LFSR.

So, then people have showed that we can also design another kind of LFSRs, which are called modular LFSRs. Well this delay is reduced, but some characteristics of generating huge spectrum of inputs are possible with a very low area over it. So, in this case, what they are doing. In this case, the first flip-flop is feeding the last flip-flop directly.

But, instead of in the standard LFSRs, we are having a connection like this. But, in this case, it is not like that, you have you can put a feed XOR gate here or you may not put it. So, the control comes over here. So, these are called a modular LFSR.

Standard LFSRs means here these were all directly connected, and we are putting XOR gates in the top. But, in this case, it is slightly different, we are going a reverse about connection, we are doing basically this. So, in this case you see at (Refer Time: 36:35) between any two flip-flops, only there is a single XOR gate or no XOR gate. So, it is a more faster design.

(Refer Slide Time: 36:40)



Modular LFSRs

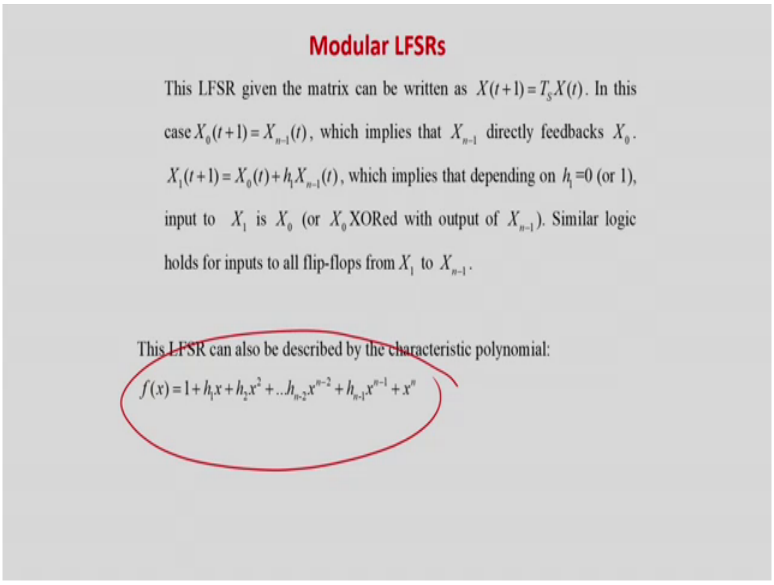
In modular LSFR the output of any flip-flop may or may not participate in the XOR function; if output of any flip-flop X_i say, provides input to the XOR gate which feeds the input of flip-flop X_{i+1} then corresponding tap point h_i is 1. In the circuit representation of $h_i=1$, then there is an XOR gate from output of flip-flop X_i to input of flip-flop X_{i+1}; else output of flip-flop X_i is directly fed to input of flip-flop X_{i+1}.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \\ \ldots\ldots\ldots\ldots \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0\ldots\ldots0 & 1 \\ 1 & 0 & 0\ldots\ldots0 & h_1 \\ 0 & 1 & 0\ldots\ldots0 & h_2 \\ \ldots & \ldots & \ldots\ldots\ldots & \ldots \\ 0 & 0 & 0\ldots\ldots0 & h_{n-2} \\ 0 & 0 & 0\ldots\ldots1 & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \\ \ldots\ldots\ldots \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

Again, it is I am extremely just a mirror map, I am not going to deal in more details, you can just look at these slides for that. Same characteristics will be generate generated, but in this case just a dual of it. So, here you will get this square matrix over here, and in this case basically this is the all 1s h s, whether you are putting XOR gate or not, so this column will determine the basically your this value for the next flop and so forth.

So, this is whatever I have discussed for standard LFSR standard LFSR, also hold for basically the modular LFSRs, only it is a (Refer Time: 37:10) mirror image like that. So, this is that means matrix representation for this.

(Refer Slide Time: 37:16)



Modular LFSRs

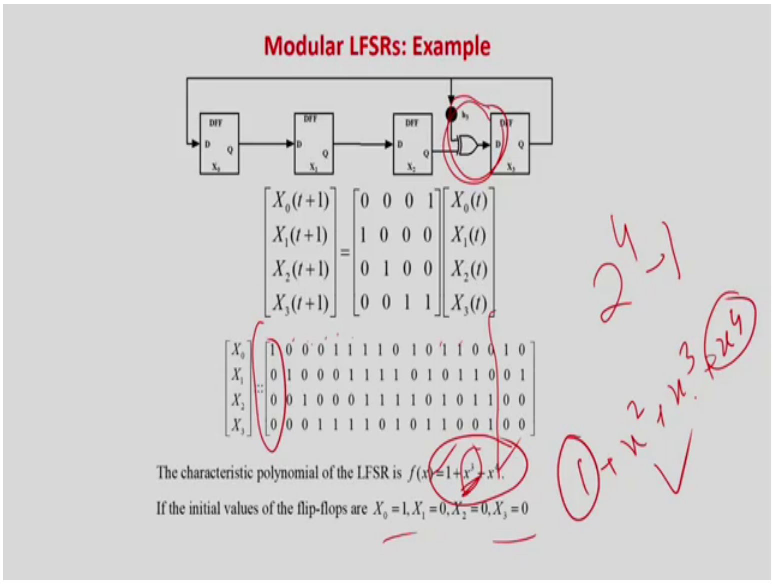
This LFSR given the matrix can be written as $X(t+1)=T_S X(t)$. In this case $X_0(t+1)=X_{n-1}(t)$, which implies that X_{n-1} directly feedbacks X_0. $X_1(t+1)=X_0(t)+h_1X_{n-1}(t)$, which implies that depending on h_1=0 (or 1), input to X_1 is X_0 (or X_0 XORed with output of X_{n-1}). Similar logic holds for inputs to all flip-flops from X_1 to X_{n-1}.

This LFSR can also be described by the characteristic polynomial:

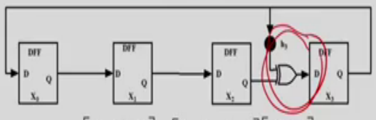
$$f(x)=1+h_1x+h_2x^2+..h_{n-2}x^{n-2}+h_{n-1}x^{n-1}+x^n$$

Then there will also be a similarly characteristics polynomial, and other theory is more or less similar. So, you can just look over these slides, and you will get a feeling.

(Refer Slide Time: 37:24)



Modular LFSRs: Example



$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \\ X_3(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \\ X_3(t) \end{bmatrix}$$

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} :: \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The characteristic polynomial of the LFSR is $f(x)=1+x^3+x^4$.

If the initial values of the flip-flops are $X_0=1, X_1=0, X_2=0, X_3=0$

2^4-1

$1+x^2+x^3+x^4$ ✓

So, in this case is also an example of a modular LFSR in which case also (Refer Time: 37:28) they have taken a 4 bit number. So, if you are counting it is 1, 2, 3, 4, 5, 6, 7 this is the seed, so the seed is repeating over here. So, if you look at, it is a 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 14, 15. So, 15 numbers are generate entire spectrum, 2 to the power 4 minus 1, because 0s are not generated.

So, even by just by putting a single XOR gate, you are able to generate the entire spectrum for 4 bit input cases. The characteristics polynomial is this, basically 1 and 4 will always be there, there is only one XOR gate. So, the XOR factor is there. So, initial values are this, this is seed vectors, and this can be easily generated, you can easily verify by this slide.

So, what emphasizing that for 3 bits, 4 bits, 5 bits. Similarly, it can be shown that with very very few XOR gates, you can generate the entire spectrum of inputs. But, if you are going to make a control generation just like we are doing digital design final state machine generation, it is a huge circuit that cannot be used in based, but LFSRs are very very well (Refer Time: 38:27) for based ok.

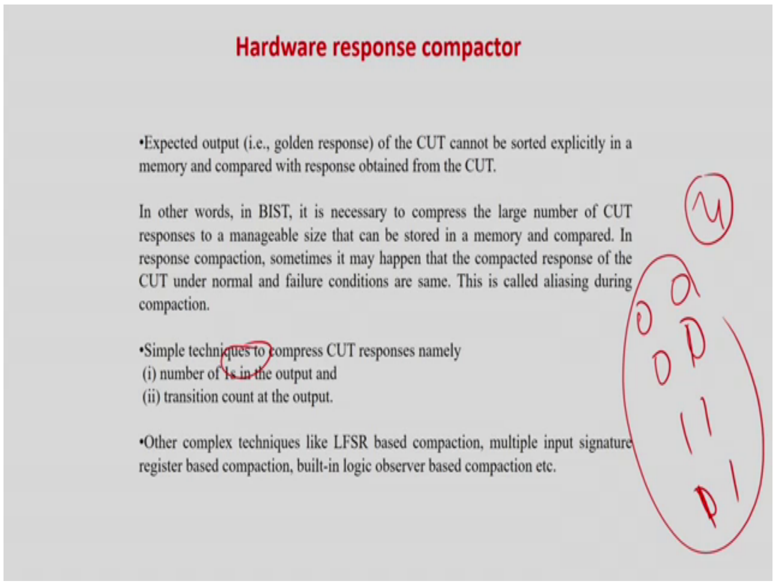
Now, the question arises can any LFSR would generate these patterns, as I told you it is no, only a few characteristics polynomial the LFSR is maximal length that is entire spectrum is generated, they are called primitive polynomials. This is actually the paper, so here lot of proofs are presented that what are the primitive polynomials for what length you should select, what is the area impact on that. Because, based on the polynomial size or polynomial type, you have to put the XOR gates correct.

So, if you are putting such kind of XOR gates, then the size may vary. Like for example, in this case the primitive polynomial, you can directly find out that I require a single XOR gate. But, if it is say x 1 plus x square plus x cube plus x 4, so (Refer Time: 39:06) this bit and this one, no XOR gates are required, but here two XOR gates should be required. If it is a primitive polynomial, you know that two XOR gates are required, but it may be generating the entire spectrum, just I was giving you an example.

So, just by your requirement, you look at the characteristics polynomial available for different ends and you select and do it. So, test engineers basically do not that way directly involved in the theoretical proof, rather they know the theory words. So, you select the required means ATPG vectors, which you know which will be generated by this LFSR, you can select a primitive polynomial.

And for certain cases, if you do not require the entire spectrum of values to be generated, maybe only four or five different patterns to be generated, then accordingly you can put the XOR gate and design the LFSRs; but, the key thing is that this size is very very less, only a few XOR gates will do the (Refer Time: 39:50) that is why LFSRs have become a boon for BIST. So, I can also say that if LFSR were not there, BIST was a very difficult problem, because I require a very low area (Refer Time: 39:59) test pattern generator, and which is being possible by LFSRs.

(Refer Slide Time: 40:04)


Hardware response compactor

•Expected output (i.e., golden response) of the CUT cannot be sorted explicitly in a memory and compared with response obtained from the CUT.

In other words, in BIST, it is necessary to compress the large number of CUT responses to a manageable size that can be stored in a memory and compared. In response compaction, sometimes it may happen that the compacted response of the CUT under normal and failure conditions are same. This is called aliasing during compaction.

•Simple techniques to compress CUT responses namely
(i) number of 1s in the output and
(ii) transition count at the output.

•Other complex techniques like LFSR based compaction, multiple input signature register based compaction, built-in logic observer based compaction etc.

So, your problem of input generation is solved. So, (Refer Time: 40:06) LFSR, it will generate the maximum spectrum of inputs and you are happy with it. Now, what about the output? Because, LFSRs they are generating the entire spectrum of inputs for you in a random fashion. Then for each pattern, should I store the output response, I can do it. But, in that case, your ex expected signature that is wrong, where actually the values has to be stored for the golden outputs required that is the expected output of the CUT will be very very large.

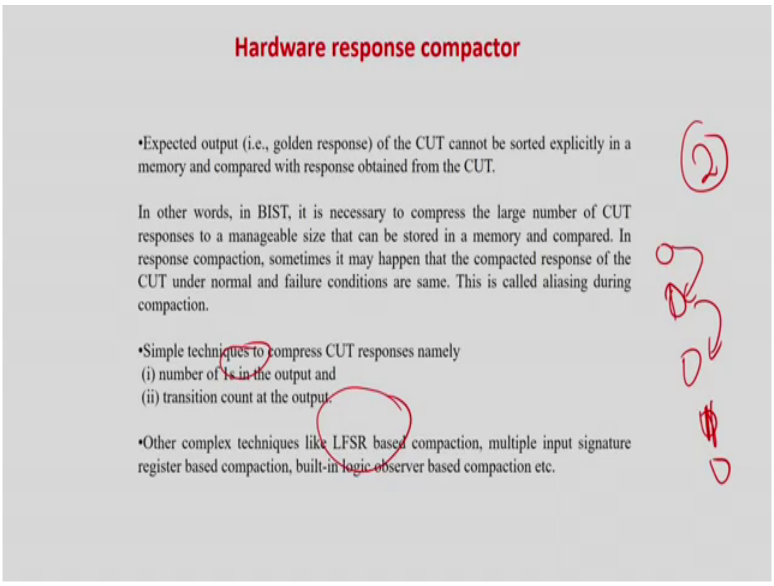
So, what we do is basically, we go for compaction that is we block make a block off some of the input, outputs or even all the outputs we can make a block that is given output expected output sequence, we make some kind of a compaction arrangements, and we represent by 1 or 2 bits in a very compact manner that is called a compaction, but it is the lossy compaction.

Lossy compaction means, for some of the cases, the errors may be fault may be masked. But, still to make the any optimization, we have to do it, and also theory exists that there are lot of good compaction algorithms or techniques available, where the aliasing is very very low. We are not going into more depth on this, but rather we try to see very two simple type of compactions, and see what is the aliasing, and what how it can be avoided.

So, (Refer Time: 41:10) simple compactions will look at this lecture that is called the one count and transition count. One count means if they decide the output sequence is I told you maybe 0 0 0 1 1 1 1 0 something like that, there will be counting how many 1s are there that is 1 4. So, we will store the value of 4 that is in this whole expected output values, the number of 1s are 4. So, you just count the number of 1s.

So, if some fault happens like in which case this is made as a 0 and this is made as a basically and this is made as a 1, your fault should be aliased, because you are just counting the number of 1s. Two faults occur, which may cancel each out, then it is gone. So, another type of is called the transition value. So, transition value means how many basically transition happens.

(Refer Slide Time: 41:54)



Hardware response compactor

•Expected output (i.e., golden response) of the CUT cannot be sorted explicitly in a memory and compared with response obtained from the CUT.

In other words, in BIST, it is necessary to compress the large number of CUT responses to a manageable size that can be stored in a memory and compared. In response compaction, sometimes it may happen that the compacted response of the CUT under normal and failure conditions are same. This is called aliasing during compaction.

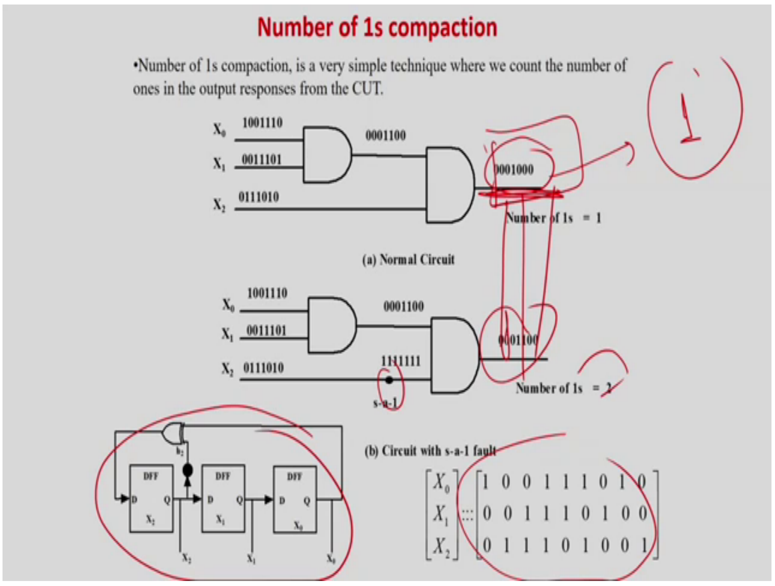
•Simple techniques to compress CUT responses namely
(i) number of 1s in the output and
(ii) transition count at the output.

•Other complex techniques like LFSR based compaction, multiple input signature register based compaction, built-in logic observer based compaction etc.

So, for example, if the input output sequence is 0 1 I am assuming a 1 bit is output 0 0 0. So, what is the transition? One transition and another transition, so you would be maintaining the transition count as 2. But also, you can find out that this can also have some aliasing effects like if something balance is out, maybe it becomes a 0 and this becomes 1. So, faults are there, but the transition count will be 2, and errors will not be detected.

But, you will see for many of the cases however, (Refer Time: 42:15) simple compactions can detect the errors, but there are more complicated testing again, based on LFSR, which actually can minimize the compaction like anything, but that theories are more involved will not go into that. If you are interested, you can find out the entire literature on LFSRs on built in self-test or basically in coding theory, we will find out the entire spectrum of literature on that. But, we are going to see today simple compaction techniques and what it results we are going to look at it.

(Refer Slide Time: 42:40)



Number of 1s compaction

•Number of 1s compaction, is a very simple technique where we count the number of ones in the output responses from the CUT.

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} \vdots \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

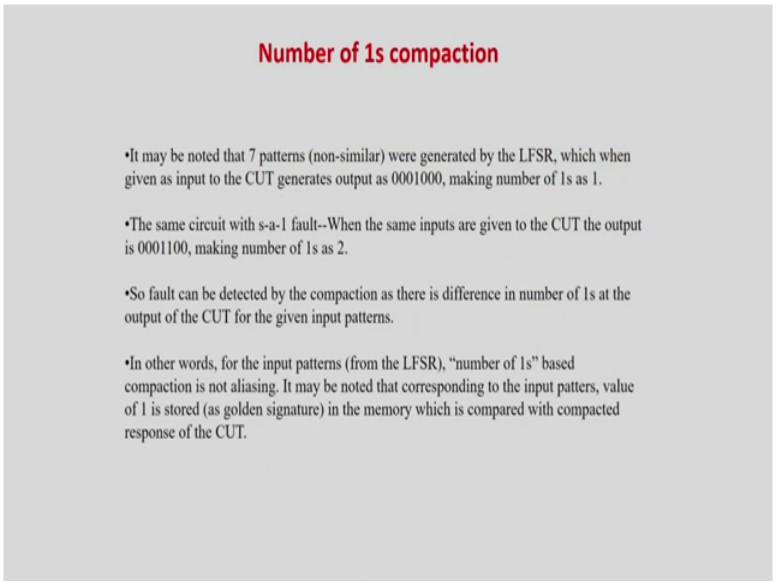
So, for example, is a simple circuit, which we are taking, and we are taking a simple stuck-at one fault over here, and we are taking this circuit, and we are generating this patterns. And our compaction technique, we are taking is counting the number of 1s. So, we are you applying these patterns, we will find out that the number of 1s are 1.

Now, I will (Refer Time: 43:00) zoom it for your help, so I am doing it, so I am zoomed. This is the input, and also putting a stuck-at 1 over here. And if you do the operation, you will find that the number of 1s are 2 at the output. Basically for this input sequence, so this is the input sequence generated by this LFSR, already which we have designed, we are applying patterns to that, and this is the output.

And for a fault, you will see that the number of output number of 1s has become a 2. So, the compaction was in this case, the expected answer is a 1, I am not going to represent explicitly all the values in the ROM. Only I will say that for this pattern set, I am going to store 1 as the expected response in the ROM.

So, if the stuck-at fault happens over here, the number of 1s will be 2, so the error this error will be detect or this fault will be detected, but there can be lot of aliasing cases also. In this case, it works. So, this is basically compaction. Instead of throwing the signal explicitly as it expected value expected response and comparing bit by bit, we are not going to do it. Rather for this block, we are going to take the total number of one cell, and then store that as a in the ROM as a golden signature.

(Refer Slide Time: 43:48)



Number of 1s compaction

•It may be noted that 7 patterns (non-similar) were generated by the LFSR, which when given as input to the CUT generates output as 0001000, making number of 1s as 1.

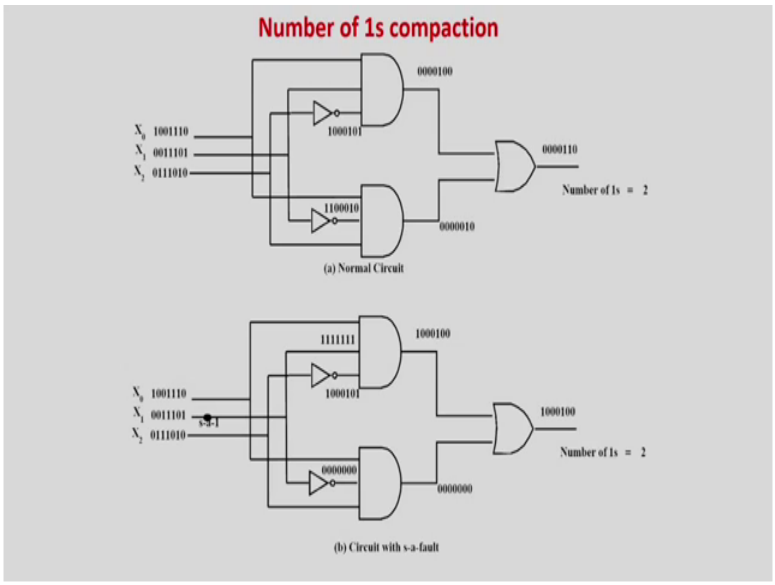
•The same circuit with s-a-1 fault--When the same inputs are given to the CUT the output is 0001100, making number of 1s as 2.

•So fault can be detected by the compaction as there is difference in number of 1s at the output of the CUT for the given input patterns.

•In other words, for the input patterns (from the LFSR), "number of 1s" based compaction is not aliasing. It may be noted that corresponding to the input patters, value of 1 is stored (as golden signature) in the memory which is compared with compacted response of the CUT.

That is what actually I have written in this slide, you can easily read a get what I have told you.

(Refer Slide Time: 44:03)

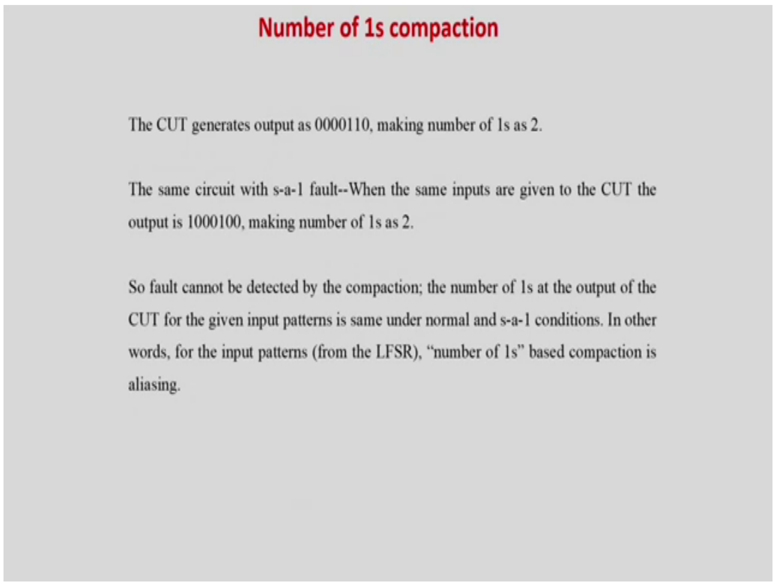


Another example. So, another circuit I have taken, again same input patterns I am giving over here, these are the patterns, and this is the output, number of 1s are 2 for this circuit also. Again, here I am going to take a stuck-at 1 fault. So, this is going over here, so this is 1 and the outputs will be all 0s, if you compare the number of this what is the output.

So, surprisingly in this case, the number of 1s are also 2. So, if you look at it, the why the aliasing has happened. So, simple, if you look at it. So, if somehow I could have managed this, so you can look at it. So, in this case this is 1 bit, the MSB has been flipped over here, if you look at is 1 bit flip from 0 to 1. And on the contrary, this is (Refer Time: 44:44) from 1 to 0. So, they are balancing each other.

So, the number of 1 still remains 2, but still there is a fault. Because, the (Refer Time: 44:49) bit can itself tell you that there is a fault, which is occur, but it will not be captured, because we are not expressing the storing the full vector right. Then you can tell me, which is a better compaction method, in fact it depends on size circuits to circuits. So, from here, it may appear that counting number of 1s would have been a is a very bad solution, we can take something else like transition count.

(Refer Slide Time: 45:10)



Number of 1s compaction

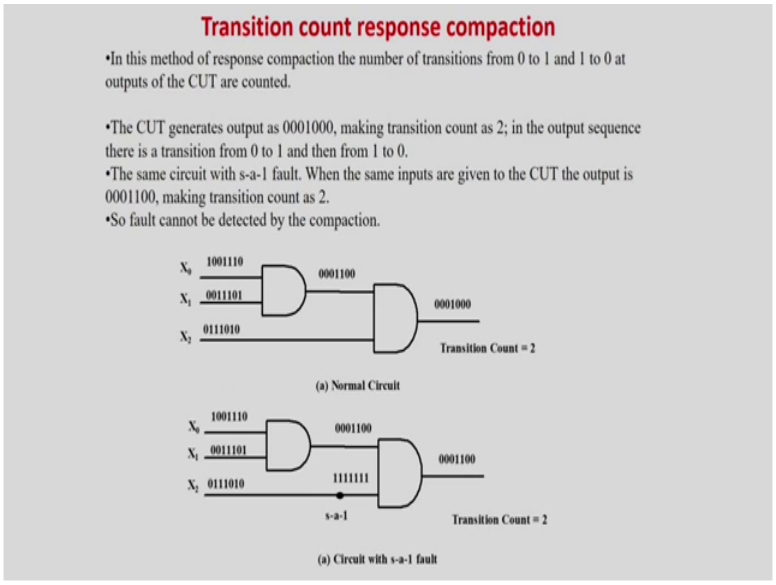
The CUT generates output as 0000110, making number of 1s as 2.

The same circuit with s-a-1 fault--When the same inputs are given to the CUT the output is 1000100, making number of 1s as 2.

So fault cannot be detected by the compaction; the number of 1s at the output of the CUT for the given input patterns is same under normal and s-a-1 conditions. In other words, for the input patterns (from the LFSR), "number of 1s" based compaction is aliasing.

Anyway, this is the story for this slide.

(Refer Slide Time: 45:16)



Transition count response compaction

- In this method of response compaction the number of transitions from 0 to 1 and 1 to 0 at outputs of the CUT are counted.

- The CUT generates output as 0001000, making transition count as 2; in the output sequence there is a transition from 0 to 1 and then from 1 to 0.
- The same circuit with s-a-1 fault. When the same inputs are given to the CUT the output is 0001100, making transition count as 2.
- So fault cannot be detected by the compaction.

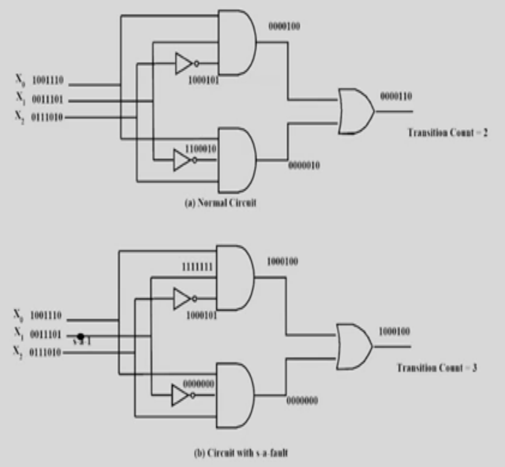
Now, here so transition count. So, I am just going to give you a story about transition count. So, from here, it may appear that sorry from there, it may appear that number of 1s counting is bad, it is aliasing. So, let us see about the transition count. I am taking the whole circuit, which you have considered previously.

So, in this case, so this is the normal this is the input sequence from the LFSR, this is the what is the expect (Refer Time: 45:32) output. So, you can see the number of transitions count will be two; 1 and 2. Again, all stuck-at 1s this is a stuck-at 1 value, you will see that this is what is the output. Here also, this transition count is a 2, because 0 1, then no transition 2.

So, in this case, counting 1s, because there is one 1, there is two 1s, it could have been detected by the counting 1 based aliasing, but transition count it fails in this case. Those then I can say that transition count is a bad example, or a bad way of handling this type of errors, or the bad compaction? (Refer Time: 46:02) answer is no.

(Refer Slide Time: 46:03)

Transition count response compaction



If you take this one, where the number of one counts failed, here it will succeed. So, here if you are looking at it, so we know that this is the normal 1, see transition count is 2. So, this is the second circuit we are using. But here, the error is you know that, this is what is the error output for this stuck-at fault. So, in this case if you look at the transition count is 1, then again 2 and then 3. So, the transition count is 3. So, in this circuit, the transition count based compaction works.

So, what we tells basically that for a given circuit and a given fault, which aliasing method will work better, it depends. So, here what I have tried to show you that, but these are very very not very good kind of aliasing techniques. As I told you there are lot of theories exist for LFSR based techniques for basically compaction with very low aliasing within very low (Refer Time: 46:46) but they are more theory intensive. So, I have not covered in this lecture, but as a future or part or an additional, you can read over there.

But, what I have shown you basically is that compaction can actually go for fault detection with of course a compromise, but the area (Refer Time: 46:59) is very very less. Because, here you just want to require the count, so it is a log complexity which comes in, and here basically obviously, same thing when you are going for counts, so it is a log complexity. As a transition count or the number of one counts, so it is a log complexity. And if the variable size is n, only log 2 n bits are equal to story. So, the area for the ROM is lowered, but still there can be some kind of a compaction (Refer Time: 47:20).

The lot of theory exists, which tries to modify this, so that they can get good compaction algorithms, but again area is very important. I cannot have a very large area, and explicitly store all of them and get a 0 compaction and I can get very good result that is not actually allowed in BIST because, in BIST, everything is inside the chip, and you cannot make your tester circuit more large then the circuit itself.

Then it can happen that if your (Refer Time: 47:42) circuit is much larger than your cut itself, then the probability of occurrence of faults in such circuits also increases. Because, what is the theory or the statistics it say is that if the circuit is large, then the probability of occurrence of faults also in this case becomes larger right.

But, if you have a very smaller circuit, then you can properly design it with much more spacing between the vias and the (Refer Time: 48:00) and you can have slightly less probability of faults there. So, certain tweaks we can do, or even if you are not doing any tweaks also, a smaller circuit tweaks to going to have less probability of failures.

So, therefore, we want to keep the circuit of the tester much smaller for both for lower probability of errors, as well as for the cost. Because, you are nobody is going to pay you for the test, everybody is going to pay you for the circuit functional circuit, so you want to bring it smaller.

So, therefore basically what we have seen in class that built in self-test is very important to boost the confidence of the working of the circuit. But, for that it is very important is that we should have a very low area over pattern generator and very low over area low area over a response analyzer. So, pattern generator is very good well solved by LFSRs. Because, by with a very little XOR gate into business, you can generate the entire spectrum of inputs for the outputs what is the problem, you have to go for compaction.

So, here actually lot of aliasing happens and lot of theoretical works are done, so that you can get a low area overhead using LFSRs etcetera, but the compaction is lower. So, anyway that we are not going into details in this module, because it is a coverage of design verification and test, so you can look at the very dedicated embedded system testing courses, which deals more on this.

So, with this, we come to end of the lecture on BIST. In the next two lectures, we are going to look at fault tolerance. BIST happens at the startup of the circuit. Offline test happens, when the chip is manufactured. But, if some problem happens, when the chip is doing its normal operation, then who is going to solve it. There is some testing test the circuit itself, while it is doing its normal operation that is called concurrent testing or online testing, which is mainly required for fault tolerance. The next two lectures will be dedicated to that.

Thank you.