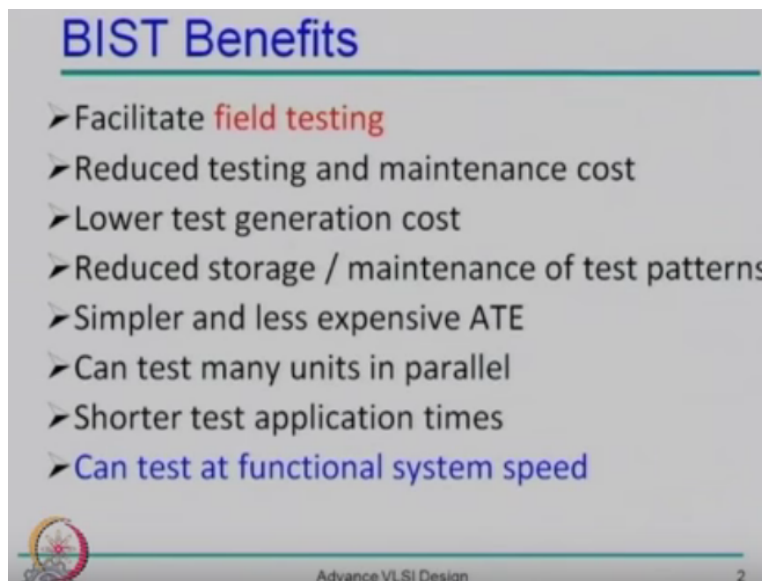


Advanced VLSI Design
Prof. Virendra K. Singh
Department of Electrical Engineering
Indian Institute of Technology- Bombay

Lecture - 38
VLSI Testing: Built-In Self-Test (BIST)

Hello, welcome to the lecture series on advanced VLSI design course. I was talking about VLSI testing in last couple of lectures. So today we will discuss little bit more in detail about Built-In Self-Test. A background of Built-In Self-Test was already discussed in the last lecture. So, we have seen that if we use Built-In Self-Test, in that case one of the very important aspect that it brings in is field testing.

(Refer Slide Time: 00:52)



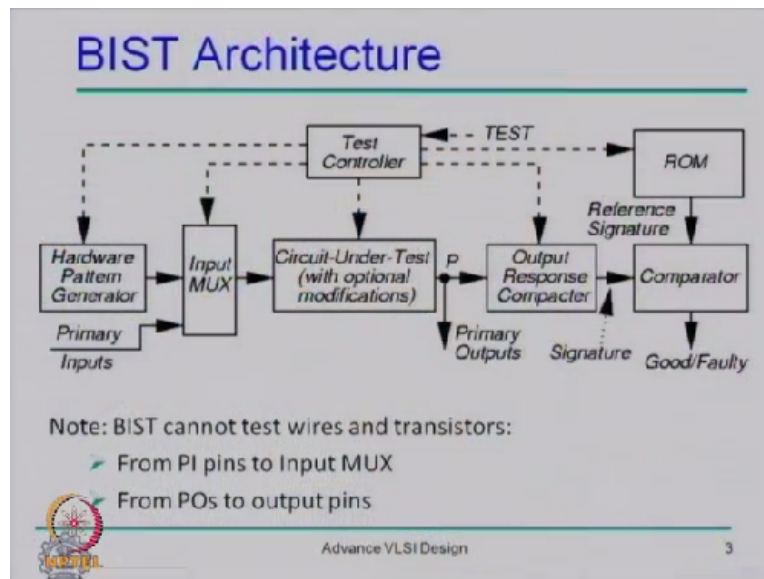
So that means here now your manufactured device, you can test as and when you want to test without using very expensive tester and the other advantage is, it can bring in R like reduce testing and maintenance cost, it will have very low test generation cost because we are not using automatic test pattern generator that takes months to generate the test. We do not need to restore the large signature and large test vector as well as the test response for the test of the device.

We need very simple automatic test equipment that can facilitate us to apply test. As I discussed in the last class that here we need only one thing that can initiate the operation and can say that yes now just start the Built-In Self-Test and tell me whether the device is good or bad. And now

because we need very simple tester that is inexpensive, so we can test large number of devices in parallel and then here because we can apply the test at a speed, the test time would be shorter.

And then here another very important feature is that we can test at functional system speed. So that means here we operate circuit at the functional speed, hence it may discover many more defects. These are couple of advantages we have.

(Refer Slide Time: 02:44)



If you look at the architecture, in that case here what we need, we need three different additional entities to help us testing the device. So you have circuit under test, you need a test generator so that means here there would be a hardware test pattern generator you need to have, you need to have a test response collector that can compare the response compared with the golden response and then here it will tell you whether chip is good or chip is faulty.

And in order to control all these activities, here we need to have a test controller. So these are the three essential parts of Built-In Self-Test we must have. So now here we have to apply the test pattern at the primary input and collect response from the primary output, so that means here the pattern generated by the Built-In Self-Test must be multiplexed with the primary inputs using a multiplexer that would be controlled by the test controller.

And then we have to color the response directly from the output of the circuit. So now here what it can test and what it cannot test. So it contests all the faults in the circuit under test, it can test all the faults in the Built-In Self-Test hardware, but because here we are not accessing directly the primary input. So, if there is a fault in this, so now here primary input line itself maybe faulty that means it is connected to ground line or open or connected to VDD.

And in that case here we may not be able to test that and then here in the same way, where the output line we are not exercising, hence we may not be able to test. So these are the parts, which cannot be tested if you use Built-In Self-Test. Okay, so now let us look at these components one by one. Let us first start with the hardware pattern generator, then we will go to the output response analyzer.

So let us see how hardware pattern generator can work. So what are the various ways. There are many ways, means sometimes it looks bit weird that we say that we want to build complete pattern generator on the circuit, that can generate deterministic test.

(Refer Slide Time: 05:12)

Pattern Generation

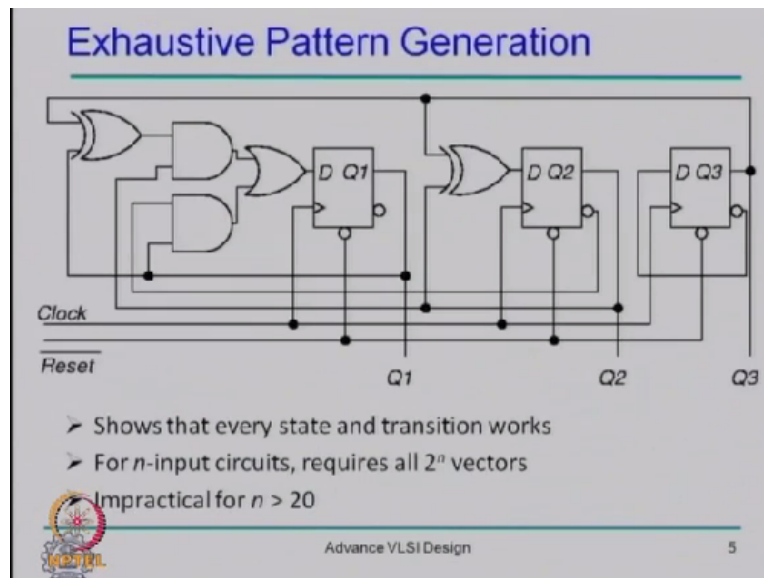
- Store in ROM – too expensive
- *Exhaustive*
- *Pseudo-exhaustive*
- *Pseudo-random* (LFSR) – Preferred method
- Binary counters – use more hardware than LFSR
- Modified counters
- Test pattern *augmentation*
 - ❖ LFSR combined with a few patterns in ROM
 - ❖ *Hardware diffracter* – generates pattern cluster in neighborhood of pattern stored in ROM

Advance VLSI Design 4

One of the simplest way that one can think of is we generate the test by using automatic test pattern generator ATPG and then store those patterns in ROM. If we do that, say we have millions of pattern, we are storing those millions of patterns on ROM, it may take huge area on

the chip and that may be too expensive. Other way is, we can generate test pattern exhaustively with simple hardware.

(Refer Slide Time: 05:34)



Like for example, this is one of the hardware that can generate exhaustive set of eight patterns and in the same way, we can build a hardware that can generate for 16 inputs, for 50 inputs, and more than that exhaustively. Again here if we generate patterns exhaustively, the problem is test application. As we have seen in very beginning that if you apply the exhaustive test pattern, in that case here it may take several centuries to apply.

Same thing holds good here, so hence we cannot apply these patterns in reasonable time or practically on the circuit, though we can generate these using some cheap hardware. Right, so this is not practical if n grows beyond 20. So then this is not very attractive scheme. So far we discussed two schemes, one is we can store pattern on the ROM and then make use of those patterns. That is impractical, the other scheme is exhaustive test that is also impractical.

(Refer Slide Time: 06:55)

Pseudo-Exhaustive Method

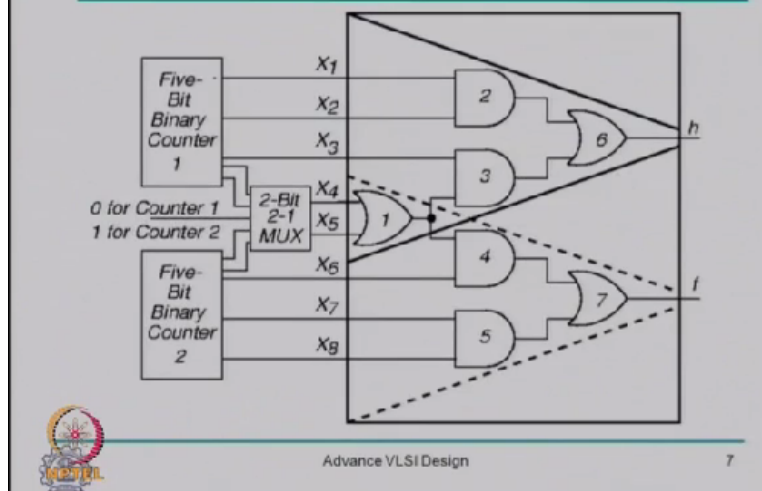
- ❖ Partition large circuit into *fanin cones*
 - Backtrace from each PO to PIs influencing it
 - Test fanin cones in parallel
- ❖ Reduced # of tests from $2^8 = 256$ to $2^5 \times 2 = 64$
 - Incomplete fault coverage



So now here the other alternate is Pseudo-Exhaustive, that means here we do not generate really exhaustive pattern, but it looks like we are generating exhaustive test pattern. For example, so what do we do we partition circuit based on the Fanin cones of the outputs and see that which are the inputs affecting that particular output. Based on the inputs which are affecting that particular output, we generate exhaustive test for that.

(Refer Slide Time: 07:28)

Pseudo-Exhaustive Pattern Generation



Like for example, this is a circuit with two outputs, so output h and output f. So this is the fan in cone of output h and this is the fan in cone of output f. So now here it has total eight inputs but here only these five inputs are affecting this output h and only these five inputs are affecting

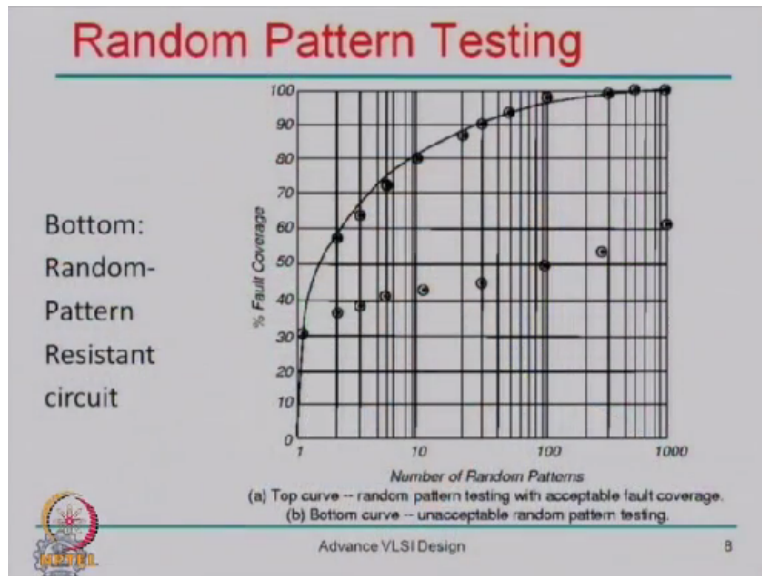
output f. So now here these three inputs and these three inputs are affecting only, the X1 to X3 are affecting only h and X6 to X7 are affecting only f.

So, if I generate exhaustive pattern for these five and exhaustive pattern for these five, in that case here I can get the similar kind of confidence that I am getting through exhaustive test. So now here, if you look at how many patterns we need, for eight input, we need 2 raise to the power 8 that means 256 patterns but here now I am dividing that in two sets each of five inputs that means here from each of the set.

I am generating 2 raise to the power 5 inputs and then there are two sets, so that means here total 64 inputs. So there is a good amount of reduction in the number of patterns we can have, but still if you go beyond say 50 inputs or 100 inputs, in that case here again this post turns out to be impractical. Hence we cannot use this approach as well. So now here what are the way I had if we want to implement Built-In Self-Test.

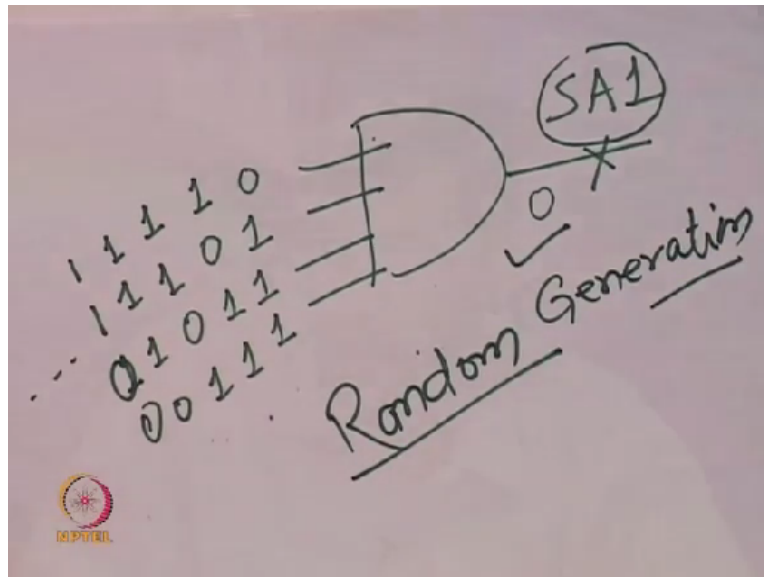
We have to generate test patterns which can be applied in reasonable time. So now here one of the approach that can be explored is random pattern test.

(Refer Slide Time: 09:21)



As we discussed in very beginning that there can be multiple pattern for a single fault or in other word one fault may have multiple test patterns. If I generate some patterns randomly, let us see for example here for this gate.

(Refer Slide Time: 09:50)



Say this is four input gate and if I want to generate a pattern for Stuck-At 1, there are various combinations because here for Stuck-At 1. I want 0 here in order to excite that. So now here there are various combinations, 0 1 1 1, there can be 1 0 1 1, there can be like here 1 1 0 1, there can be 1 1 1 0, there can be like here 1 1 0 0 and so and so for that there are so many combinations. All are able to detect this Stuck-At 1 fault.

So that means here if I generate pattern randomly, in that case here, I am likely to detect this particular fault. So that means here random generation could be one of the approaches. If you look at the fault coverage, in that case here, you may get a curve like this. So that means here very quickly you may get good coverage but after that this coverage started to become saturate. So for many of the circuits, the curve goes like this which is admissible.

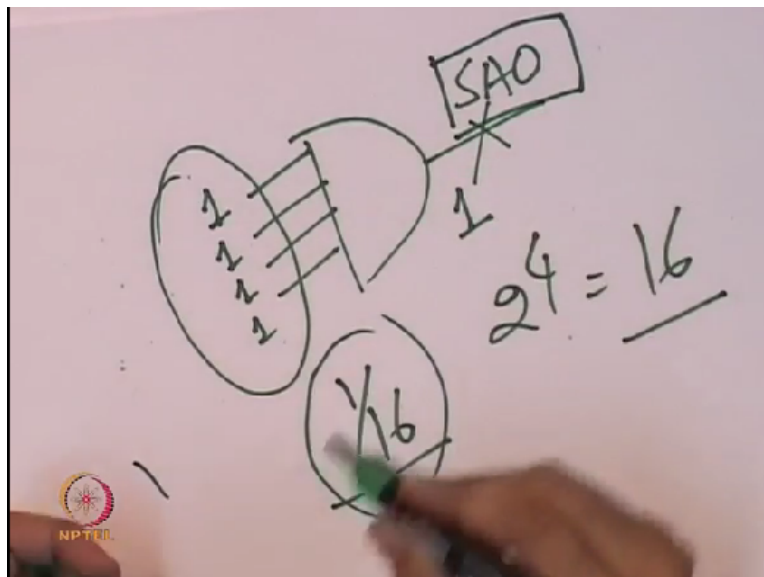
So that means here a couple of, like here for this particular circuit, hundred patterns can give you about 99% fault coverage, whereas if you want to achieve 100, in that case here, you may need to apply 1000 patterns. Look at here, the 10 pattern itself may give you 80% fault coverage. So

that means here if you apply random pattern, in that case initially you may get very high coverage.

And then slowly it will start to saturate but for some of the circuits, the curve may go like this. So this grows very fast but then here it will quickly saturate and that happens due to the circuits which have random pattern resistance. So one of the random pattern resistance that comes from like here for example, again the same I discussed, like here if I want to generate test pattern for Stuck-At 0 fault here, that means here I want 1.

And then there can be only one pattern that can excite this. So that means here I have to generate this pattern and so for the four inputs. There can be 2 raise to the power 4 that means there can be 16 patterns and the probability of generating this pattern is 1/16. And say if I generate only say five pattern, in that case here probability of getting this pattern is very low and hence here we may not be able to detect this particular fault and these kind of circuits are called as Random Pattern Resistant Fault Circuits.

(Refer Slide Time: 13:01)



So if you have large number of such kind of patterns which are needed to be generated, here you may not be able to achieve very high coverage, but by and large for most of the circuits, you are likely to get this kind of curve. But here the key issue is the pattern must be random, if they are

not random, in that case here you may not because it may be biased and you may not be able to get this curve.

So now there are two questions. First is how we generate random pattern and second thing is, if I can generate the random pattern, is it sufficient. So the answer of first is, it is very difficult to generate purely random patterns because here always you are generating these pattern with some algorithms. And hence here they may have repeatability and hence you may not be able to get this kind of curve. That is one of the issue.

Second, assume I can have very good random pattern generator that can generate but if it is random than the problem is how would I be able to compare the response of the circuit, whether my circuit is good or bad, because I do not know the response of the circuit when it is fault free because I do not know what pattern it will generate on the fly. So that means even if I have very good random pattern generator, it may not help us.

So then what is the way out. We need, by and large, the random nature of the patterns and second thing is, these pattern must be generated algorithmically so that here they have repeatability, that means we can simulate these patterns and generate the golden response of the circuit and then eventually we can compare with the golden response. So that means we have to generate these patterns algorithmically and the approach is the Pseudo-Random Pattern Generation.

(Refer Slide Time: 15:26)

Pseudo-Random Pattern Generation

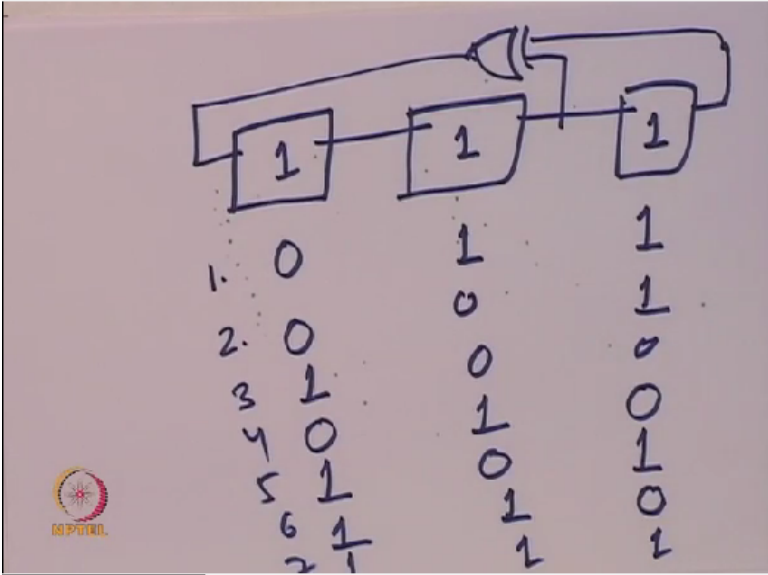
- ❖ Standard Linear Feedback Shift Register (LFSR)
 - Produces patterns algorithmically – repeatable
 - Has most of desirable random # properties
- ❖ Need not cover all 2^n input combinations
- ❖ Long sequences needed for good fault coverage

Advance VLSI Design 10

So now here, let us look at how we can generate these pseudo-random. So that means here, by large, the nature of pattern that we are generating that means here the mixture of zeros and ones that should be random, so that means it preserves the randomness and this should be deterministic. One of the very simple circuit could be the Linear Feedback Shift Register.

So if you have a shift register, say you have three flip-flops and there is a feedback from the last flip-flop with some, more or less, by intermediate value here and I feedback here and now here in this, say I initialize this to 1 1 1, then how it will progress. So initially it will have 1 1 1 value. Now after that in the second cycle, what will be the output, this is just synchronous circuit.

(Refer Slide Time: 16:29)



So now this one and one will give you zero. So next time here output would be 0 1 1, then the very next cycle, the output would be 0 0 1 because this 1 will shift here, this 1 will shift here and now in the third cycle, this 0 1 will give you 1, so this will be 1 0 0, then this 0 0 will give you 0, so you will get 0 1 0, then this 0 1 will give you 1, so this will be 1 0 1 and then here again this 0 1 will give you 1, so this will be 1 1 0 and now here this 0 1 will give you 1 and then here 1 1 1.

So now again we get back to the same venue. So now here, how many inputs it is generating, 1 2 3 4 5 6 and 7 inputs it is generating. So that means here this can with a very small circuit, we can generate a sequence of seven vectors. Exhaustively, if you want to generate from this circuit of three input, in that case here it may be eight. So that means here we are generating nearly exhaustive pattern sequence.

And now here, if you look at the placement of 0's and 1's, it is fairly random. It is much different from your counter that you may use to generate. So now here, a Linear Feedback Shift Register can be one of the Pseudo-Random Pattern Generator. So to generate the patterns algorithmically that means these are repeatable, that is very important because we can simulate that and it has most desirable random number generation property. So these are the properties we have.

Though here we want to generate as long sequence as possible because here that is good for the fault coverage. Long sequence means here it should repeat after large number of sequences but here really we do not need the exhaustive 2^n sequence, though we want long sequence. So now here what you need to have is, you have a feedback from the first flip-flop to the last flip-flop.

And if there are a n number of flip-flops we have and then here you can tack input from some intermediate points. So now here you can represent this by a matrix or by a polynomial. If you say polynomial, in that case here that can be represented by $1 + h_1X + h_2X^2 + h_3X^3 + \dots + h_{n-1}X^{n-1} + X^n$ and so forth, h_{n-1} and $X^{n-1} + X^n$ raise to the power n .

(Refer Slide Time: 19:31)

Matrix Equation for Standard LFSR

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

$$X(t+1) = T_s X(t) \quad (T_s \text{ is companion matrix})$$



So if you if you write down that as a matrix, in that case these are the output values, time $t + 1$ and these are the current value in the flip-flop, X_0 to X_{n-1} and that can be given by this matrix that is called as companion matrix. This has some properties like in the first column, the last bit is 1 that comes from the feedback from the first flip-flop to the last flip-flop. This h_1 to h_{n-1} , these values can be zero or one based on whether you are tapping that value or not.

And then the rest of this matrix is identity matrix. So this is the property of this. So X_{t+1} is T_s into X of t . The T_s is the companion matrix.

(Refer Slide Time: 20:14)

LFSR Implements a Galois Field

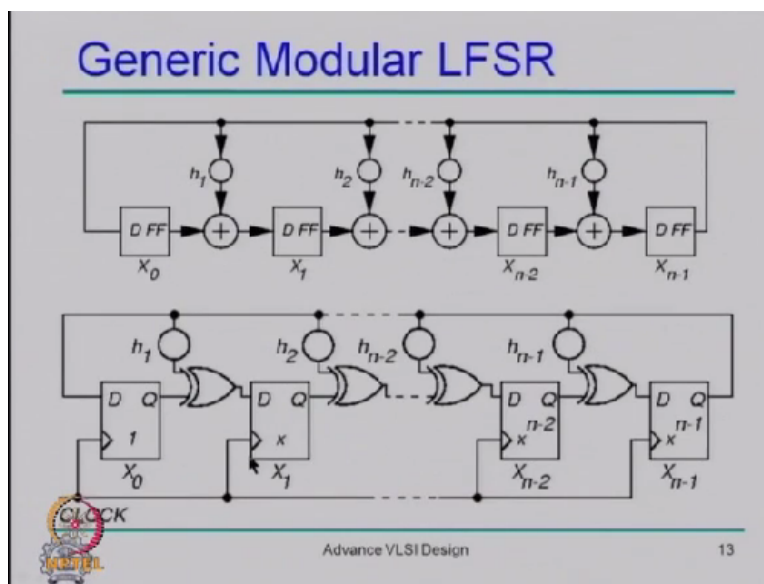
- *Galois field* (mathematical system):
 - Multiplication by x same as right shift of LFSR
 - Addition operator is XOR (\oplus)
- T_s companion matrix:
 - 1st column 0, except n th element which is always 1 (X_0 always feeds X_{n-1})
 - Rest of row n – feedback coefficients h_i
 - Rest is identity matrix I – means a right shift
- Near-exhaustive (maximal length) LFSR
 - Cycles through $2^n - 1$ states (excluding all-0)
 - 1 pattern of n 1's, one of $n-1$ consecutive 0's



This works under the Galois field theory where in the multiplication by x serve as the right shift in LFSR and addition operation is simply XOR operation or model to operation. The Ts company matrix, the properties I already explained to you this gives you the near exhaustive sequence, so that means here other than all zeros, it may give you all the sequences. So that means your cycle length may go to 2 raise to the power $n-1$.

That means it excludes 0 because once it goes to all 0 state, it may never come out from that and hence that can never be generated.

(Refer Slide Time: 20:57)



So this is one of the implementation of that. The other implementation here you have a feedback or XOR gates in the feedback path, you can generate the same property by placing. So now here in worst case, if you are tapping input from all in that case here, there can be a sequence of n XOR gates from the last flip-flop to the first flip-flop and that makes the system slower. So now here in order to improve that, the another LFSR is proposed wherein you place the XOR gate in the feed for all path and this generates the same sequence.

(Refer Slide Time: 21:39)

Modular Internal XOR LFSR

- Described by **companion matrix** $T_m = T_s^T$
- Internal XOR LFSR – XOR gates in between D flip-flops
- Equivalent to standard External XOR LFSR
 - With a different state assignment
 - Faster – usually does not matter
 - Same amount of hardware
- $X(t+1) = T_m \times X(t)$
- $f(x) = |T_m - I X|$
 $= 1 + h_1 x + h_2 x^2 + \dots + h_{n-1} x^{n-1} + x^n$
- Right shift – equivalent to multiplying by x , and then dividing by characteristic polynomial and storing the remainder



If you look at the companion matrix of this in that case, your companion matrix of the modular LFSR is the transpose of the previous one. Now, here you are reading taps. These taps from left to right, whereas in the standard LFSR, you are reading these taps from right to left. That is the difference. And now again here the same characteristic polynomial remains same as $1 + h_1x + h_2x^2 + \dots + h_{n-1}x^{n-1} + x^n$.

(Refer Slide Time: 22:13)

Primitive Polynomials

- Want LFSR to generate all possible $2^n - 1$ patterns (except the all-0 pattern)
- Conditions for this – **must have a primitive polynomial**:
 - **Monic** – coefficient of x^n term must be 1
 - Modular LFSR – all D FF's must right shift through XOR's from X_0 through X_1, \dots , through X_{n-1} , which must feed back directly to X_0
 - Standard LFSR – all D FF's must right shift directly from X_{n-1} through X_{n-2}, \dots , through X_0 , which must feed back into X_{n-1} through XORing feedback network



If you want to achieve very long sequence, in that case here the polynomial that you implement or characteristic polynomial that is given by this expression. It should be a primitive polynomial. And what are the conditions for the primitive polynomial. One of the conditions is that it must be

a Monic, so that means here coefficient of X^n term must be 1. So that means here it always should have $1 + x$ raise to the power n term.

If you look at here in that case here it should have $1 + x$ raise to the power n term. These terms may or may not be there.

(Refer Slide Time: 22:56)

Primitive Polynomials

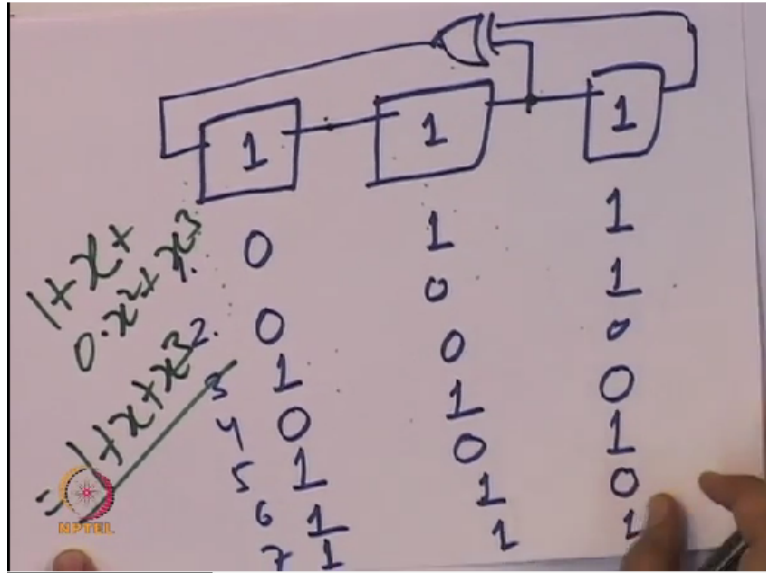
- Characteristic polynomial must divide the polynomial $1 - x^k$ for $k = 2^n - 1$, but not for any smaller k value

Advance VLSI Design 15

Then the other condition is that the characteristic polynomial must divide the one - x raise to the power k or $1 + x$ raise to the power k . So, like in this previous example, I can write the characteristic polynomial of this as $1 + h_1x$, so we are tapping from here, so in that case here this is 1 into $x +$ we are not tapping anything from here. So 0 into x square + x cube. So this is $1 + x + x$ cube.

Now here let us see whether this is primitive polynomial or not. This is generating the longest sequence of 7, hence it should be a primitive polynomial.

(Refer Slide Time: 23:37)



So what are the conditions. So condition says that this should be a factor of $1 - x$ raise to the power k or $1 + x$ raise to the power k because it follows the modular theory and where k is 2 raise to the power $n-1$ where n is the number of flip-flops that we have. So it should be 1 raise to the power k where k is 2 raise to the power $n-1$. Here in this case n is 3 hence k would be 7 because there are 3 flip-flops I am using.

So that means here it should be a factor of $1 + x$ raise to the power 7 and that must have $1 + x$ raise to the power n as mandatory term in that, right where n is here 3 , so that means here that must have $1 + x$ raise to the power 3 . Let us look at the factor of this. Factor of this would be $1 + x$, $1 + x + x^2 + x^3$ and $1 + x^2 + x^3$. These are the factor of $1 + x$. Keep in mind, we are using the modular theory.

So that means $x + x$ is 0 because $+$ is XOR operation. So now here, let us look at which factor qualifies for this. This factor, does it qualify, it does not qualify because here it does not have x raise to the power 3 term. So, here this is unqualified term. Now look at this. This has $1 + x^3$. So that means here this is qualified term. This has $1 + x^3$ term, so this is qualified polynomial for LFSR, this is primitive.

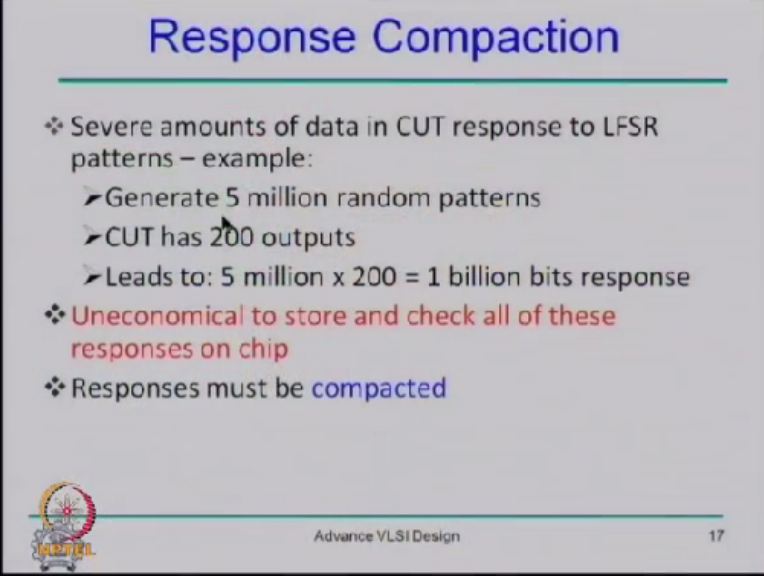
(Refer Slide Time: 25:20)

As we know that here we are generating the Pseudo-Random Test Patterns, these are repeatable, so we can do the simulation and we know what would be the real output of the fault-free circuit. Now the question is, we can one way is we can store this response and read-only memory and compare. Again here, if you store everything on read-only memory, in that case here the number of bits you need in ROM would be very large, hence that will consume lot of area.

So what we need to do. What we can look for is, we can look for is the reduction of volume of that data and one of the way is rather than using the real data, we can compact the data, generate some signature, like for example when you go to bank, nowadays bank do have the photograph and they match the photograph and all these things but still if you write a check, they do not look at the photographs, they just compare the signature.

So signatures are representatives, so you can have the compact storage of the signature and then you can compute.

(Refer Slide Time: 30:35)



Response Compaction

- ❖ Severe amounts of data in CUT response to LFSR patterns – example:
 - Generate 5 million random patterns
 - CUT has 200 outputs
 - Leads to: 5 million x 200 = 1 billion bits response
- ❖ **Uneconomical to store and check all of these responses on chip**
- ❖ Responses must be **compacted**

Advance VLSI Design 17

So now here let us say for example is you have 5 million random patterns generated, there are 200 outputs. In that case, you need 1 billion bits and then here, this is very uneconomical to store one 1 billion bits. So now here we need compact. So what could be the way to generate the signature.

(Refer Slide Time: 30:56)

Definitions

- **Aliasing** – Due to information loss, signatures of good and some bad machines match
- **Compaction** – Drastically reduce # bits in original circuit response – lose information
- **Compression** – Reduce # bits in original *circuit response* – *no information loss* – *fully invertible* (can get back original response)
- **Signature analysis** – Compact good machine response into *good machine signature*. Actual signature generated during testing, and compared with good machine signature
- **Transition Count Response Compaction** – Count # transitions from $0 \rightarrow 1$ and $1 \rightarrow 0$ as a signature



One of the problems could be like even if you go to bank, somebody else may sign and that looks like same as your signature. So that means there is a little possibility that other person's signs that matches with your signature. That is known as Aliasing. So that means here due to information loss, a signature of good and bad circuit matches, that is known as aliasing. So now here when you generate signature.

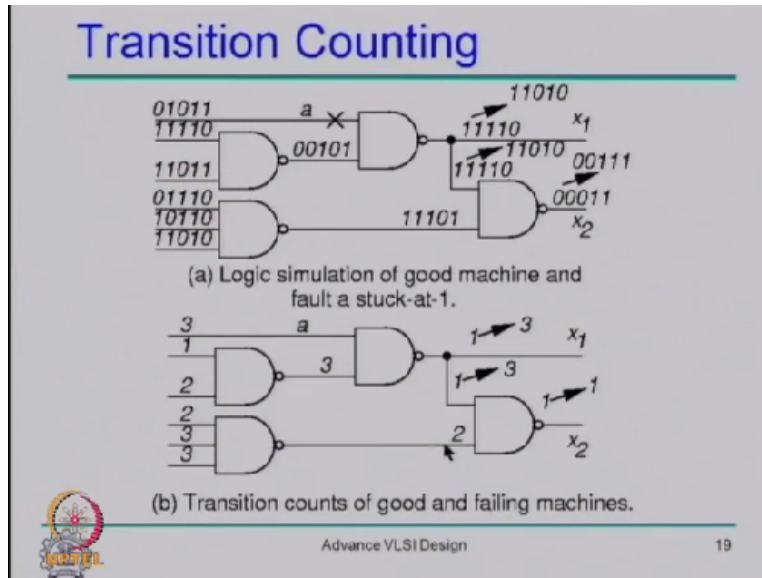
Here, we have to be careful that here aliasing should be as little as possible or ideally there should not be aliasing. Then we define two terms, one is the compaction and another is compression. Compaction is drastic reduction in bits in the original circuit response and then here we lose some information. So that means here this is irreversible process. From the compacted response, we cannot get back the original response.

The other term that you people are well familiar with is compression, like here you do ZIP. So where in you compress the data and then here you can again expand it to the same original data. So those are lossless. Signature analysis is another term that we define. What it says is that here compact good machine response into good machine signature, actual signature generated during the testing and compared with the good machine signature.

That is, so we have to compact the good machine signature, store somewhere that needs lesser memory space and then we again generate the signature on the fly from the circuit under test and

match with the stored response. One of the way is that here we can use the transition count in the bit stream.

(Refer Slide Time: 33:00)



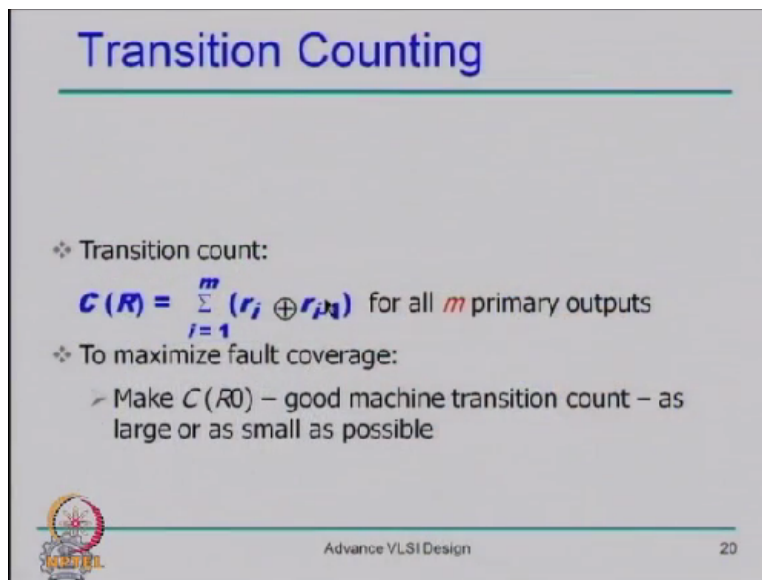
Like here for example, if this is the circuit, if I apply a bit sequence here in five different cycles, then here what I will get is, I will get this as response when it is fault free and when it is faulty. So now here when it is fault free and faulty, here both of the responses do have different transition. So here if you look at, there is only one transition from 0 to 1. Whereas in this, here you can see there can be to two transitions, one is from 1 to 0 then 0 to 1.

And then again 1 to 0, so there are three transitions. On the same way at the second output, here there is one transition and here there is also one transition. So if you look at only this output, in that case here you have distinguishable transition of fault free and faulty machine, whereas on output 2, you may not have distinguishable number of transitions for fault free and faulty circuit. So if you happen to have only one output say X2, in that case here.

If you use the transition count, you cannot distinguish whether machine is faulty or fault free, that is known as aliasing. Whereas if you have only say X1, in that case here, you can distinguish that. So what we want is that here aliasing should be as small as possible and here in this, I do not want to go in detail the aliasing analysis of this, but here the aliasing probability goes like this. So if there are n number of inputs that you are compacting in K.

So for the very low and very large, like here if the number of transition is either zero or one, in that case, there are only few possibilities. Sorry this will go from here. Now here, but here when the number of transitions are somewhere in the middle range, the aliasing probability is very high.

(Refer Slide Time: 35:16)



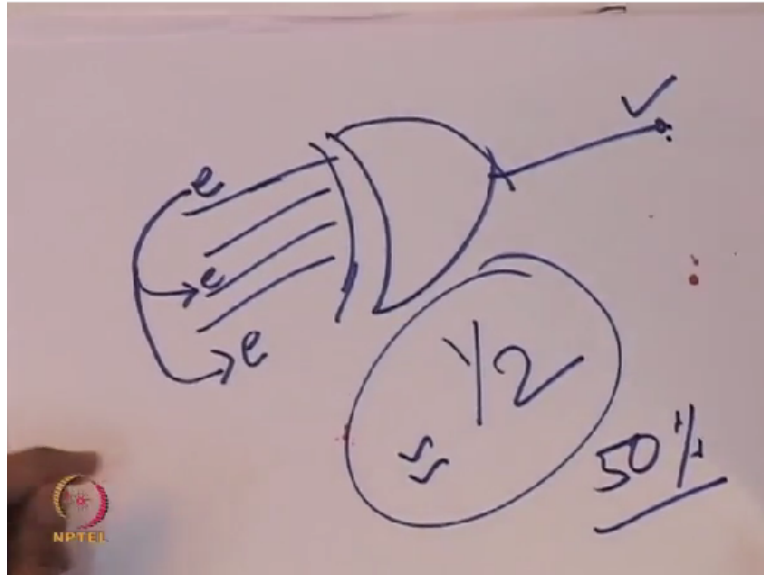
Transition Counting

- ❖ Transition count:
$$C(R) = \sum_{i=1}^m (r_i \oplus r_{i+1})$$
 for all m primary outputs
- ❖ To maximize fault coverage:
 - Make $C(R)$ – good machine transition count – as large or as small as possible

Advance VLSI Design 20

So transition count is though it is very easy because here you need to XOR the previous input and the current input, generate the output using this one XOR gate and then here you can count these using a counter and so the transition count is one of the easy mechanism but aliasing probability is very high. The other solution is that you can use simply an XOR gate.

(Refer Slide Time: 35:46)



So that means here all inputs here you can compact in one, the problem with XOR gate is that if your fault effect propagates at one of the input, in that case here always you will get fault effect at output, but if fault effect propagates at two places, then here it will mask because the generated signature would be same as the good machine signature and hence it will go unnoticed.

If it propagates to odd number of inputs, in that case here, you will get distinguishable signature, hence you can detect. So this can detect the error which propagates to 1 or odd number of inputs but this cannot detect if your fault effect propagates to even number of outputs. So now here if you look at the aliasing probability, in that case here aliasing probability would be roughly half or 50%, that is very high.

So now here we have to device. Transition count was one of the methods, use of one XOR gate is also another method but here the aliasing probability is very high. So most of the time, it is not very usable method. so now here the other way is we can use the cyclic redundancy code which were in use in communication systems for longtime. So what do they do, when they transmit a bit stream.

They generate some redundant bits from these data bits and send along with the data and then when you receive at the output, you regenerate that. If there is a match, in that case here, you can say that your transmission was good; otherwise, there was an error in the transmission. Same

concept we can use here. We can generate the CRC codes from the given bit stream.

(Refer Slide Time: 37:59)

LFSR for Response Compaction

- Use *cyclic redundancy check code* (CRCC) generator (LFSR) for response compacter
- Treat data bits from circuit POs to be compacted as a decreasing order coefficient polynomial
- CRCC divides the PO polynomial by its characteristic polynomial
 - Leaves remainder of division in LFSR
 - Must initialize LFSR to *seed value* (usually 0) before testing
- After testing – compare signature in LFSR to known good machine signature
- Critical: Must compute good machine signature

Advance VLSI Design 21

So now here also we can treat data bits from the circuit primary output and we can compact that in the decreasing order of polynomial which can be given by LFSR, means the circuit that we can use to compact is again here the LFSR kind of circuit where in we have linear feedback shift register. So now here when you scan it through the LFSR, here it will generate a redundant code.

(Refer Slide Time: 38:36)

Example Modular LFSR Response Compacter

Characteristic Polynomial $x^5 + x^3 + x + 1$

- LFSR seed value is "00000"

Advance VLSI Design 22

So say this is LFSR whose characteristic polynomial is $x^5 + x^3 + x + 1$ and then here, through this XOR gate, here you are combining the input bit stream. Say this is the input bit stream and initially say we initialized this feedback shift register to all zero values. Keep in mind, in LFSR

we do not have any input bit stream but here it accepts the data input.

(Refer Slide Time: 39:13)

Polynomial Division

Inputs	x^0	x^1	x^2	x^3	x^4
Initial State	0	0	0	0	0
1	1	0	0	0	0
0	0	1	0	0	0
Logic	0	0	1	0	0
Simulation:	0	0	0	1	0
1	1	0	0	0	1
0	1	0	0	1	0
1	1	1	0	0	1
0	1	0	1	1	0

Logic simulation: $Remainder = 1 + x^2 + x^3$

0 1 0 1 0 0 0 1

$0 \cdot x^0 + 1 \cdot x^1 + 0 \cdot x^2 + 1 \cdot x^3 + 0 \cdot x^4 + 0 \cdot x^5 + 0 \cdot x^6 + 1 \cdot x^7$

Advance VLSI Design
23

Now here let us look at what it will generate. So if you look at, it starts from all zero states and then when you will receive this bit stream 1 0 0 0 1 0 1 0, then here based on the LFSR characteristic polynomial, it will give you the output and at the end after this 8 bits, here we will have 1 0 1 1 0. So this would be available in the flip-flops. So we have five flip-flops. This would be data in the flip-flop.

I can write this as a polynomial that means 1 into x raise to the power 0 + 0 into x raise to the power 1 + 1 into x raise to the power 2 + 1 into x raise to the power 3 + 0 into x raise to the power 4 that combines to $1 + x^2 + x^3$. In the same way, I can write a polynomial for this input bit stream that is your 1 0 0 0 1 0 1 0 and that I can represent by polynomial.

(Refer Slide Time: 40:24)

Symbolic Polynomial Division

$x^5 + x^3 + x + 1$

x^7	$+ x^3$	$+ x$
x^7	$+ x^5$	$+ x^3 + x^2$
	x^5	$+ x^2 + x$
	$x^5 + x^3$	$+ x + 1$
	$x^3 + x^2$	$+ 1$

remainder → $x^3 + x^2 + 1$

Remainder matches that from logic simulation of the response compacter!

Advance VLSI Design
24

The same thing you can obtain by division operation. So now here say this was $x^7 + x^3 + x$ is the polynomial of this data. This is $x + x^3 + x^7$. So this is input bit stream, that is divided by the characteristic polynomial.

(Refer Slide Time: 40:44)

Example Modular LFSR Response Compacter

Characteristic Polynomial $x^5 + x^3 + x + 1$

- LFSR seed value is "00000"

Advance VLSI Design
22

So if you look at here the characteristic polynomial of this LFSR is $x^5 + x^3 + x + 1$. And then if you use the modular theory and do the division, in that case here you will get remainder as $x^3 + x^2 + 1$. That is same as the remainder in the flip-flops. So hence if you divide this by the characteristic polynomial, you will get the remainder. So now here if this remainder matches with the correct response that we obtained after the logic simulation.

In that case, your circuit is good and otherwise circuit maybe bad. This may result into some sort of aliasing because here now aliasing would be much better because you have multiple flip-flops. So now if you look at the aliasing, say there is a bit stream of n bits and there are k flip-flops. So now here you can have 2 raise to the power k combinations of bits generated from this one and out of 2 raise to the power n combination.

So now here the 2 raise to the power n divided by 2 raise to the power k pattern will result into the same pattern. Out of these pattern, 2 raise to the power $n - k$, out of that, one pattern maybe good and other patterns are just aliased pattern. So $- 1$ is the good pattern. Out of this 2 raise to the power n are the total pattern but out of that one pattern is good which is compacted, others are bad.

So this would be 2 raise to the power $n - 1$, this is the probability of aliasing or probability of masking. This if you say if n is greater than k , in that case here, this is equal to 2 raise to the power $- k$. So this gives us very good observation that here now if you use LFSR. In that case here masking probability will not depend on the number of inputs and that will depend only on the number of flip-flops that you have in the circuit.

(Refer Slide Time: 42:58)

Handwritten mathematical derivation on a whiteboard:

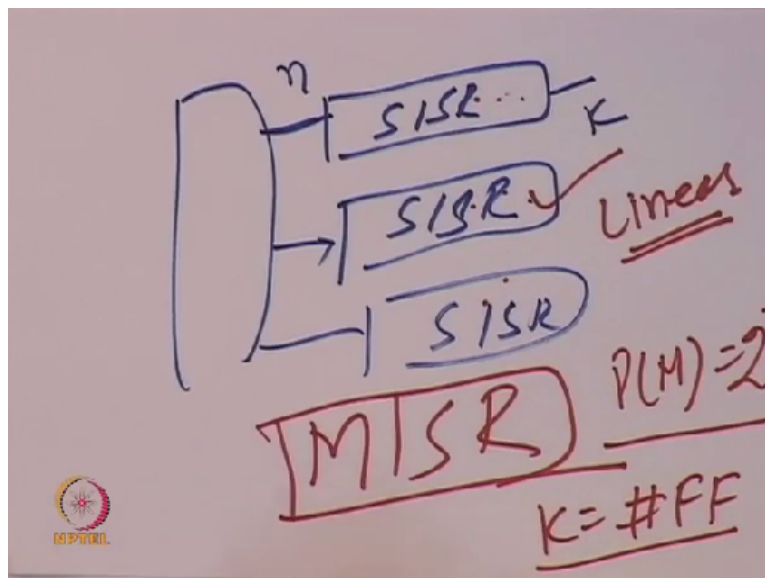
- Top left: n and 2^n circled.
- Top right: k and 2^k circled.
- Middle: $\frac{2^n}{2^k}$
- Bottom left: $P(M) = \frac{2^{n-k} - 1}{2^n - 1}$
- Bottom right: $n \gg k$ and 2^{n-k} circled.

So if you want to reduce that masking probability, you can have more number of flip-flops in the circuit. So now here so far what we discuss is that for every primary output, you need to

have one LFSR or one signature register that is typically known as single input signature register SISR and so if you have singling input, so this is your circuit output, you have one signal input signature register.

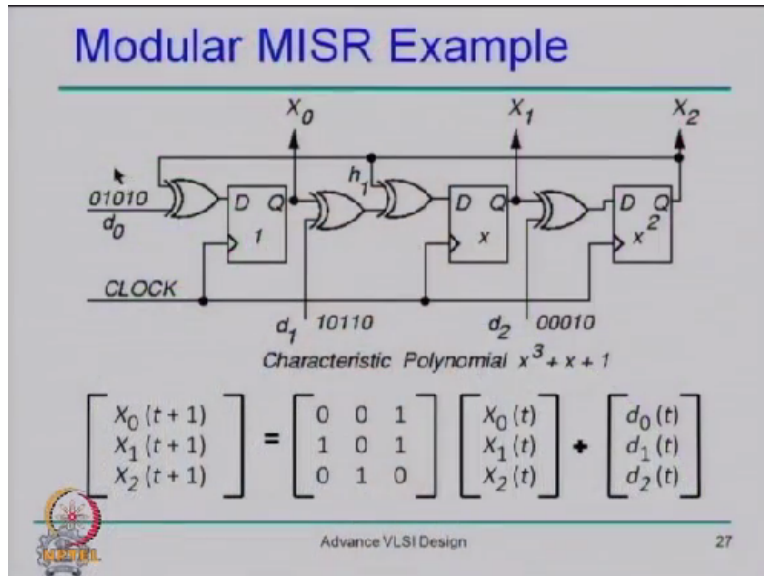
For another one, you have again single input signature register and so now here say you are getting n inputs in n number of cycles and then you are compacting this in K bits where K is the number of flip-flops that you have in SISR. Now here it has k flip-flop here, k flip-flop here, k flip-flop here, so now here hardware overhead is too much.

(Refer Slide Time: 44:00)



But if you look at this SISR, this is a simple Linear Feedback Shift Register, this is linear system. If it is linear system, in that case it should obey principle of superposition. So now here if it follows that, in that case here, now what you can do is, you can combine all these SISR into one and then you can form a Multiple Input Signature Register.

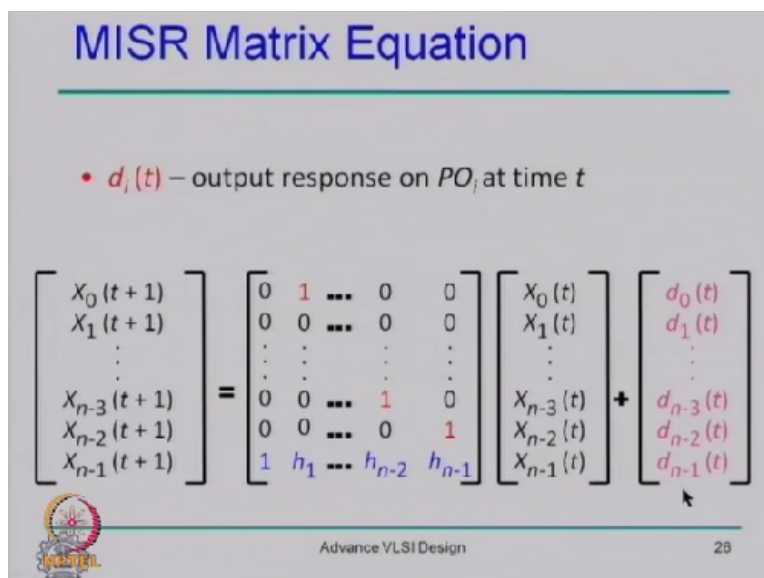
(Refer Slide Time: 44:31)



So that means here in multiple input signature register, one input you are multiplexing here, another input you are multiplexing here, another input you are multiplexing here. So if you have K number of flip-flops, in that case here inputs from k number of primary outputs of the circuit, you can multiplex here and now here if you want to find out, because this is linear system and this is the companion matrix of this LFSR.

Then you can write equation for the output in the next cycle $X_0(t+1)$, that would be companion matrix multiplied by the current values + the input arrival at this point in time, so D_0 , D_1 and D_2 .

(Refer Slide Time: 45:17)



So here again the companion matrix will remain same, it should follow same property that we discussed earlier. So now here, if you use LFSR that is known as Multiple Input Signature Resistor, MISR and if you use, then again here the masking probability will remain same as 2 raise to the power - k where k is the number of flip-flops we have in the circuit. So now these can give you very compact response, it needs very less hardware.

(Refer Slide Time: 46:06)

Transition Counting vs. LFSR

- LFSR aliases for *f sa1*, transition counter for *a sa1*

Pattern <i>abc</i>	Responses			
	Good	<i>a sa1</i>	<i>f sa1</i>	<i>b sa1</i>
000	0	0	1	0
001	1	1	1	0
010	0	1	1	0
011	0	1	1	0
100	0	0	1	1
101	1	1	1	1
110	1	1	1	1
111	1	1	1	1
	Signatures			
Transition Count	3	3	0	1
LFSR	001	101	001	010


Advance VLSI Design 28

If you compare with the previous approach that we discussed with the transition count, in that case here, you can say that in a circuit, this is the good value, there are three inputs and this is the good value and if there is Stuck-At 1 fault at A input, Stuck-At fault at B input and Stuck-At fault at F output. These are the responses. So here if you see, now this output if you use the transition count, it may not be able to detect the output, so this can give you the difference between the transition count and LFSR.

(Refer Slide Time: 46:38)

Summary

- ❖ LFSR pattern generator and MISR response compacter – preferred BIST methods
- ❖ BIST has overheads: test controller, extra circuit delay, Input MUX, pattern generator, response compacter, DFT to initialize circuit & test the test hardware
- ❖ BIST benefits:
 - At-speed testing for delay & stuck-at faults
 - Drastic ATE cost reduction
 - Field test capability
 - Faster diagnosis during system test
 - Less effort to design testing process
 - Shorter test application times

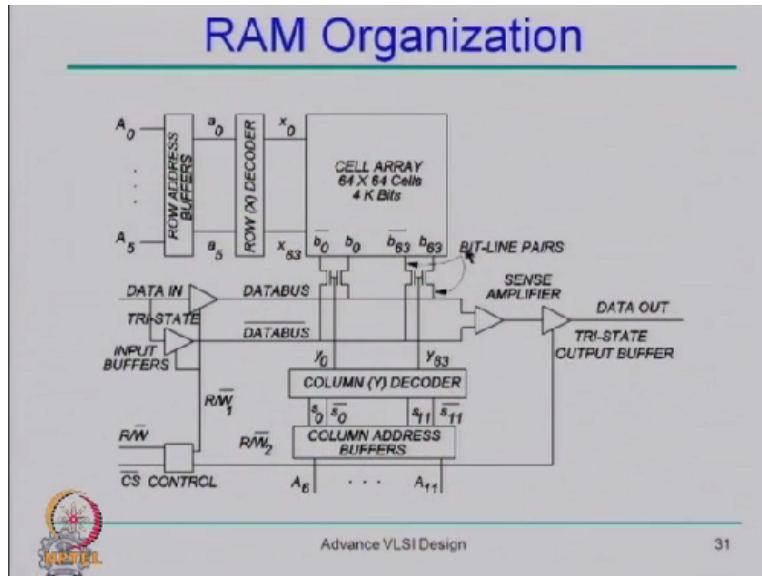


Advance VLSI Design 29

Okay so in summary here, I can say that LFSR, means the Built-In Self-Test is wonderful approach that can give you facility to apply test at any point in time, that means your chip can test itself and hence we can use that for the field test and so this allows you the at-speed test and field test. Okay this completes the Built-In Self-Test. Now so far we discussed about the test for Stuck-At 0 fault.

Let us briefly discuss about another kind of circuit that we have and those are say memory type of the circuit. So in memory, here it is a very special type of circuit, wherein it does very specific job, it stores value, that value can be zero or one, it should retain that value until it is rewritten by the system.

(Refer Slide Time: 47:41)



So if you look at the memory structure, in that case here, it will have the cell, it will have the row decoder and column decoder to enable one of the particular cell and you want to read by using the sense amplifier or if you want to write, you can write on to that particular cell.

(Refer Slide Time: 48:00)

Test Time

Size n bits	Number of Test Algorithm Operations			
	n	$n \times \log_2 n$	$n^{3/2}$	
1 Mb	0.06	1.26	64.5	18.3 hr
4 Mb	0.25	5.54	515.4	293.2 hr
16 Mb	1.01	24.16	1.2 hr	4691.3 hr
64 Mb	4.03	104.7	9.2 hr	75060.0 hr
256 Mb	16.11	451.0	73.3 hr	1200959.9 hr
1 Gb	64.43	1932.8	586.4 hr	19215358.4 hr
2 Gb	128.9	3994.4	1658.6 hr	76861433.7 hr

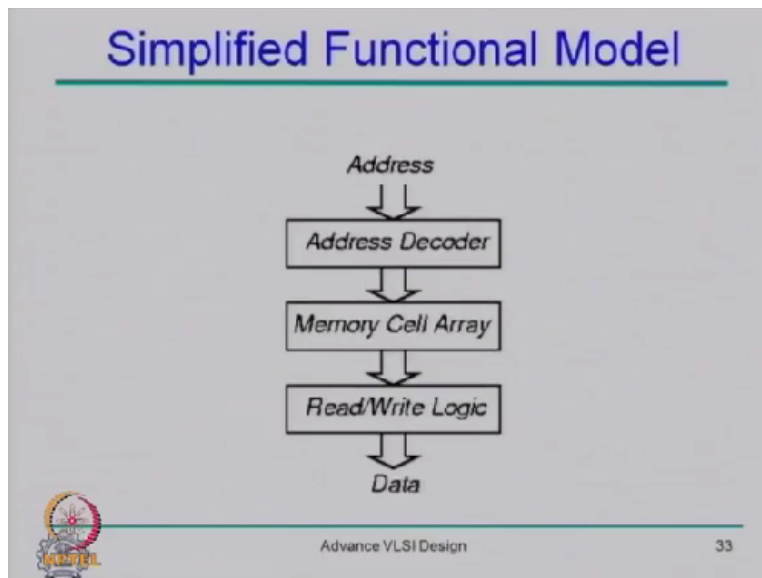
Advance VLSI Design 32

So now here look at how complex algorithm I can afford. One thing is, shall I go for the logic test kind of approach wherein you model every fault as Stuck-At 0 or Stuck-At 1. We do not need that essentially for memory because it does only restricted operation. So now here say if I have n number of cells and assume that I do one operation with every cell, in that take case here the complexity of my algorithm that I run would be n , so that means here n operations I do.

Let say I run a test at say 16 megahertz, in that case here if your memory is 1MB, it will take 0.06 second; if it is 4MB, it will take and now here you will easily have 2GB, 4GB, 8GB, 16GB, kind of memory and it will take say if it is 2GB, 128 seconds if you run your test. If you look at, if the complexity grows to $n \log n$ or n raise to the power 3 by 2 or n square that means here you do n square operations with every cell, in that case here it may go to several hours and that is impractical.

So that means here you have to devise a mechanism to test memory which has complexity as n so that means here you can have fixed number of operations with each and every memory cell.

(Refer Slide Time: 49:44)




So now here as I said that here memory does some restricted function. Now we can generate a simplified functional model for that. So you place address for the memory, address decoder will decode the address, it will enable one of the memory cell and then you will read or write that and if you are reading in that case here, you will get data out.

So now there can be a fault in the decoder so that means your decoder may enable 2 cells or multiple cells so that may not enable any of the cells. Your memory cell that may stuck to always logic 0 that may stuck to logic 1 or there may be a coupling between the memory cell so that means here if I am reading or writing one memory cell, it will also affect the another memory cell.

(Refer Slide Time: 50:38)

Reduced Functional Faults

	Fault
SAF	Stuck-at fault
TF	Transition fault
CF	Coupling fault
NPSF	Neighborhood Pattern Sensitive fault*

 Advance VLSI Design 34

So if you look at the functional fault model, in that case here I can categorize these fault model in four different categories, one is Stuck-At fault so that means here any element can stuck to logic 1 or stuck to logic 0, so that means it may not have transition. Transition fault means here, it may not make a transition from 0 to 1 or 1 to 0.

So that means if you have stored 1 in that case, it may not go to state 0 or there can be a coupling between to 2 cells or if there is some specific pattern around your cell, in that case here, it may invert the value of the cell.


(Refer Slide Time: 51:16)

March Test Elements

```
M0: { March element  $\updownarrow$ (w0) }
  for cell := 0 to n - 1 (or any other order) do
    write 0 to A [cell];

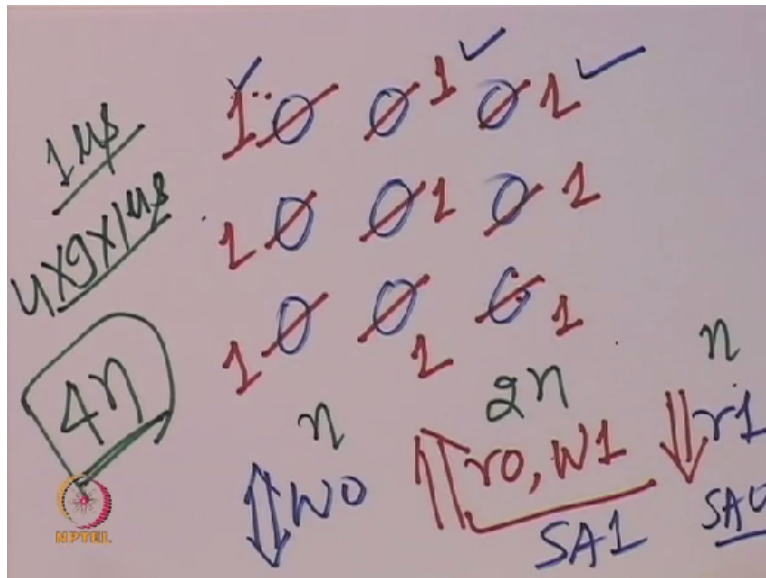
M1: { March element  $\uparrow$ (r0, w1) }
  for cell := 0 to n - 1 do
    read A [cell]; { Expected value = 0 }
    write 1 to A [cell];

M2: { March element  $\downarrow$ (r1, w0) }
  for cell := n - 1 down to 0 do
    read A [cell]; { Expected value = 1 }
    write 0 to A [cell];
```

 Advance VLSI Design 35

So now here in order to test these, here the simplest approach that was explored was, what do you do is, you initialize memory in some particular state. Say you have 3 x 3 array, so you initialize all the these, so you can go in any order and initialize all these bits to 0. So that means here, you are writing 0 to all the memory states and you can go in any order of address. So this is like you are marching from the first location to the last location.

(Refer Slide Time: 51:58)



Then what do you do after that. Next time, you come back and see. First read the earlier written value. If it is 0, in that case that cell is good, that means that can store value 0. So there is no Stuck-At 1 fault there. So now what you do is, you read this value, that is 0 and then here you again rewrite that to 1. So now here you read that value and then here you write 1. Again here the second one you do the same thing, third one you do the same thing, fourth, fifth, sixth, seventh.

In case any of the cell is stuck to logic 0, in that case that was not able to make a transition to 1 and then you can detect that immediately here. So now here you in this you can detect Stuck-At 1 fault but here you are not able to detect Stuck-At 0 fault. What do you do in the third time, you can come back and now here read this values. So if you read this value, again here, if it is 1 in that case, this is correct; if this is 1, this is correct.

So again here you march from first location to last location. So that means here you are reading 1, so this can detect Stuck-At 0 fault. So, now here how many operations. So now here, this one,

first you initialize this memory in all 0 states or all 1 states. Then here in one order of address or may be either from first to last or last to first, here you read first the earlier value, compare with the earlier written value and then change to the inverse value. So read 0 write 1.

You finish this then the next time here, again you start either from the last value to come down to the first or first to last and then let us say you come from last to first and you read all these values. This can detect all Stuck-At fault. And now here how many operations with one cell you need to do. Here for all these cells, you are doing n number of operation. Here with every cell, you are doing 2 operations.

In that case, you have 2n number of operations and here n number of operations. So that means here total 4n number of operations you are doing, so that means here you are accessing if say 1 microsecond is the memory access time and you have said 9 memory cells, in that case you need 4 x 9 x 1 microsecond as the test time for this memory. That is known as March Test. So here in this notation, here we say this is March 0, March 1 and March 2.

(Refer Slide Time: 55:08)

<h2>March Tests</h2>	
Algorithm	Description
MATS	{ ↑ (w0); ↓ (r0, w1); ↑ (r1) }
MATS+	{ ↑ (w0); ↑ (r0, w1); ↓ (r1, w0) }
MATS++	{ ↑ (w0); ↑ (r0, w1); ↓ (r1, w0, r0) }
MARCH X	{ ↓ (w0); ↑ (r0, w1); ↓ (r1, w0); ↑ (r0) }
MARCH C-	{ ↑ (w0); ↑ (r0, w1); ↑ (r1, w0); ↓ (r0, w1); ↓ (r1, w0); ↑ (r0) }
MARCH A	{ ↓ (w0); ↑ (r0, w1, w0, w1); ↑ (r1, w0, w1); ↓ (r1, w0, w1, w0); ↓ (r0, w1, w0) }
MARCH Y	{ ↓ (w0); ↑ (r0, w1, r1); ↓ (r1, w0, r0); ↑ (r0) }
MARCH B	{ ↑ (w0); ↑ (r0, w1, r1, w0, r0, w1); ↑ (r1, w0, w1); ↓ (r1, w0, w1, w0); ↓ (r0, w1, w0) }

So there are various March Tests were proposed, you can go through the various March Tests.

(Refer Slide Time: 55:14)

March Test Fault Coverage

Algorithm	SAF	ADF	TF	CF in	CF id	CF dyn	SCF
MATS	All	Some					
MATS+	All	All					
MATS++	All	All	All				
MARCH X	All	All	All	All			
MARCH C-	All	All	All	All	All	All	All
MARCH A	All	All	All	All			
MARCH Y	All	All	All	All			
MARCH B	All	All	All	All			



And if you look at what they detect. The very basic tests that is known as MATS that I discussed that it has complexity as $4n$ and it can detect all Stuck-AT fault. This can detect some of the address decoder fault, this may not detect all address decoder fault. Whereas the MATS + that is augmented by again here W0 in the third March, so that means you are writing back zero and here you are going in the forward direction.

Here you are going in the reverse direction, that make sure that you will also detect all the address decoder related fault. So now here this can detect all these faults.

(Refer Slide Time: 55:57)

March Test Complexity


Algorithm	Complexity
MATS	$4n$
MATS+	$5n$
MATS++	$6n$
MARCH X	$6n$
MARCH C-	$10n$
MARCH A	$15n$
MARCH Y	$8n$
MARCH B	$17n$




If you look at the complexity of various algorithm, in that case here this will be the complexity.

(Refer Slide Time: 56:03)


MATS+ Example Cell (2,1) SA0 Fault




(a) Good machine
after M0.




(b) Good machine
after M1.




(c) Good machine
after M2.



(d) Bad machine
after M0.




(e) Bad machine
after M1.



(f) Bad machine
after M2.

MATS+: { M0: $\updownarrow(w0)$; M1: $\uparrow(r0, w1)$; M2: $\downarrow(r1, w0)$ }



Advance VLSI Design


40

(Refer Slide Time: 56:05)


MATS+ Example

Multiple AF: Addressed Cell Not Accessed; Data
Written to Wrong Cell


- Cell (2,1) is not addressable
- Address (2,1) maps onto (3,1), and vice versa
- Cannot write (2,1), read (2,1) gives random data




(a) Good machine
after M0.




(b) Good machine
after M1.




(c) Good machine
after M2.




(d) Bad machine
after M0.



(e) Bad machine
after M1 for cell (2, 1).




(f) Bad machine
after M1.



(g) Bad machine
after M2.

MATS+: { M0: $\updownarrow(w0)$; M1: $\uparrow(r0, w1)$; M2: $\downarrow(r1, w0)$ }

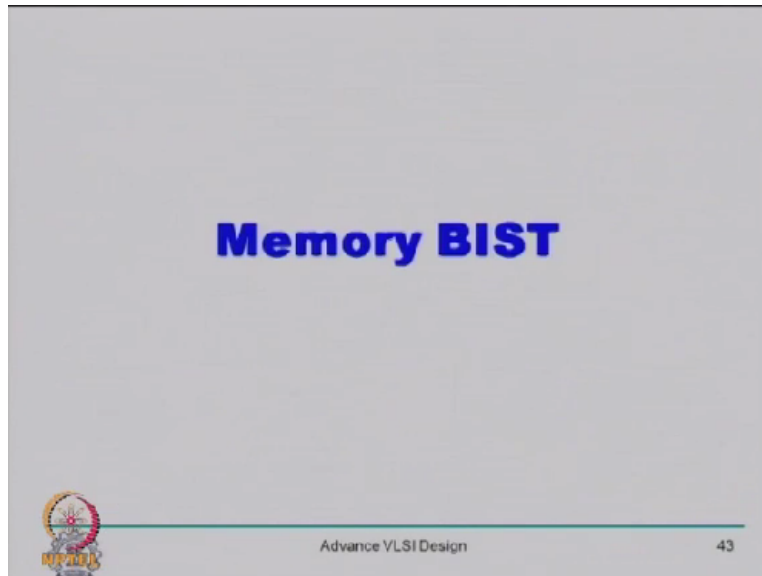


Advance VLSI Design

42

This example, I already given you, so I will skip.

(Refer Slide Time: 56:06)



So now here if you look at the memory, in that case here, we are following certain pattern and if you follow certain pattern, in that case here always it is easy to implement that as a Built-In Self-Test. So what we are doing, we are generating address right from the first to last or last to first and you are writing some single bit or we are reading single bit for that and so now here for that you need to have a pattern generator.

This pattern generator is nothing but it is address generator, so now you need to have a counter or LFSR that can generate the rising address and falling address and then you are writing that value to that particular memory state. So what is important here to detect a fault is you have to go in certain order of address and when you come back you have to come back exactly in the reverse order of address.

So the memory is one of the circuit that is most favored by Built-In Self-Test and practically now all the memories are coming with the Built-In Self-Test and that is why when you power up your laptop, most of the time you might have observed that it will start to test the memory. So that here you can always make sure that your memory is good. Sometimes we have some additional rows and columns.

So once you identify a bad row or bad column, you can replace that bad column or bad row by the redundant row or column and hence here you can full make use of your memory. Thank you

very much. Good day.