Design for Testability in Digital Integrated circuits

Bob Strunz, Colin Flanagan, Tim Hall

University of Limerick, Ireland

This course was developed with part funding from the EU under the COMETT program. The authors wish to express their thanks to COMETT.

Document rescued from the depths of internet.

Introduction and Objectives

This course provides an introductory text on testability of Digital ASIC devices. The aim of the course is to introduce the student to various techniques which are designed to reduce the amount of input test patterns required to ensure that an acceptable level of fault coverage has been obtained.

Testability In Digital Systems

Being able to design a workable system solution for a given problem is only half the battle unfortunately. We must also be able to test the system to a degree which ensures that we can have a high confidence level that it is fully functional. This is generally not a straightforward task, in very small scale digital systems, we can test exhaustively, that is to say, we can exercise the system over its full range of operating conditions. In a larger scale system, it is no longer possible to do this and therefore we must look at other strategies to ensure that the system will be properly tested.

When testing a digital logic device, we apply a stimulus to the inputs of the device and check its response to establish that it is performing correctly. The input stimulus is referred to as a *test pattern*.

In general, we observe the response of the device at its normal output pins, however, it may be that the device is specially configured during the test, to permit us to observe some internal nodes which generally would not be accessible to the user.

The response of the device is evaluated by comparing it to an expected response which may be generated by measuring the response of a known good device, or by simulation on the computer.

If the device under test (DUT) passes the test, we cannot say categorically that it is a ``good" device.

The only conclusion that we can draw from the device passing a test, is that the device does not contain any of the faults for which it was tested. It is important to grasp this point, a device may contain a huge number of potential faults, some of which may even mask each other under specified operating conditions. The designer can only be sure that the device is 100%good if it has been 100%tested, this is rarely possible in real life systems.

Faults

What type of faults are we trying to detect ? We are starting with the assumption that logically, the system performs its desired function, and that any faults occuring will be due to electrical problems associated with one or more of its component parts.

Two key concepts are of interest here, these are :

- Controllability
- Observability

During the design of a system, the designer must ensure that the test engineers have the means to set or reset key nodes in the system, that is, to *control* them. Equally as important is the requirement that the response to this control will be *observable*, that is, that we will be able to see clearly the effects of the test patterns applied.

1. Controllability - Being able to set up known internal states.
2. Combinatorial Testability - Being able to generate all states to fully exercise all combinations of circuit states.
3. Observability - Being able to observe the effects of a state change as it occurs (preferably at the system primary outputs).

Test Vector Generation

In VLSI circuits, we have a high ratio of logic gates to pins on the device, there is generally no way of accessing most of the logic, so we cannot directly probe the internals of the device. Because of this problem, we need a way of generating tests which, when applied to the inputs of a circuit, give a set of signals which indicate whether or not the device is good or faulty. The set of stimulus input and expected output pattern is called a ``Test Vector''. The test vectors distinguish between the good machine and the faulted machine. Figure 1 shows a digital device, as we can see, there is only access to the primary inputs and outputs, and therefore the device must be tested using only these ports.

Combinational Logic Test

If the combinational logic block contains no redundant logic, then the device may be tested by applying all possible 2^N possible input patterns, Where N is the number of inputs. This is termed ``Exhaustive Testing'', and is satisfactory for small circuits but rapidly becomes unweildy as the number of inputs grows. Assuming a tester capable of applying a test pattern every 100ns. Then we can calculate the test time as shown in table 1.

Inputs	Num. Tests		Test Time
20	2^{20}	10^6	0.1 Sec
40	2^{40}	10^{12}	30.5 Hours
60	2^{60}	11.8×10^{19}	58500 Years

Table 1: Exhaustive Testing Times

Looking at table 1, it is apparant that the exhaustive test strategy gets completely out of hand quite quickly, and therefore it is only of use where there are a very small number of inputs. This is also a very inefficient test strategy, most of the test patterns are actually redundant. We need a method of determining which test patterns are significant, in order to obtain a minimum set of patterns.

Various methods are listed:

1. Sensitised Path Method.
2. D-Algorithm.
3. Critical Path (L.A.S.A.R)
4. P.O.D.E.M
5. Boolean Differences

Currently, the *D-Algorithm* and its descendents, *P.O.D.E.M* and *L.A.S.A.R* are the most widely used methods.

Fault Models

A Fault Model is a description, at the digital logic level, of the effects of some fault or combination of faults in the underlying circuitry. The use of fault models has some advantages and also some disadvantages.

- Advantages
  - Technology independant.
  - Works quite well in practice.
- Disadvantages
  - May fail to identify certain process specific faults, for instance CMOS floating gates.
  - Detects static faults only.

Stuck –at faults:

Example

Single Fault.

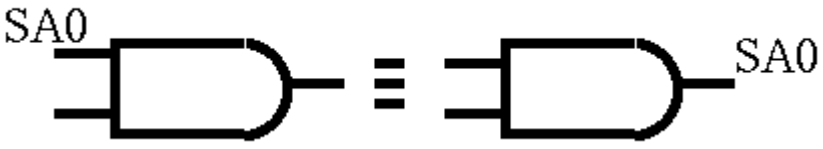
Consider an N-Input AND gate. For the STUCK-AT fault model, there are 3^(N+1) - 1 different cases of single and multiple faults, with the single fault assumption, there are only 2(N+1) stuck at faults.

AND Gate

Fault	Test at Inputs
Output SA0	1111 . . . 1
In1 SA1	0111 . . . 1
In2 SA1	1011 . . . 1
⋮	⋮
InN SA1	1111 . . . 0

Table 2: N–Input AND Fault Model

For the AND gate, any input SA0 has the same effect as the output SA0.



Output SA0 is said to *COVER* all of the input SA0 faults. Similarly, any input SA1 *COVERS* the fault output SA1.

This means that :

1. No Input SA0 faults need be included in the fault model.
2. The Output SA1 fault need not be covered either.

So, the AND gate fault model has N+1 faults, these are listed in table 2

Therefore, with N+1 faults, N+1 test vectors can completely test an N input AND gate. The N+1 faults COVER all the 2(N+1) single SA faults. It can be shown that they cover all 3^(N+1) - 1 multiple SA faults as well.

OR Gate

With an OR gate, the Output SA1 covers all Input SA1 faults. Any Input covers Output SA0. This is shown in table 3.

Fault	Test at Inputs
Output SA1	0000 . . . 0
In1 SA0	1000 . . . 0
In2 SA0	0100 . . . 0
⋮	⋮
InN SA0	0000 . . . 1

Table 3: N–Input OR Fault Model

Inverter

For an inverter, the Output SA1 covers Input SA0 and the Input SA0 covers Output SA1. This is shown in table 4

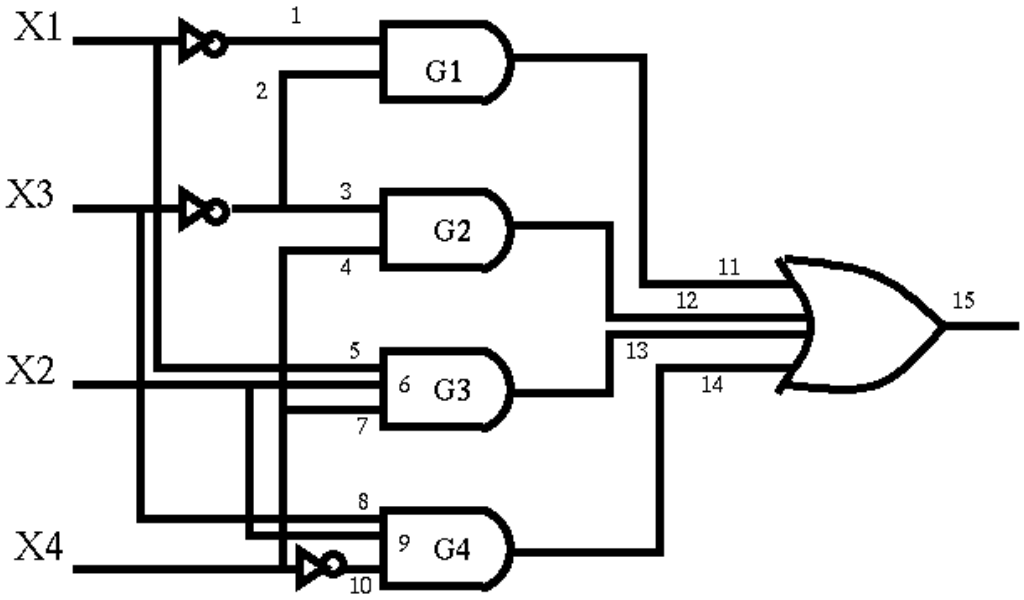
Fault	Test at Inputs
Output SA1	1
Output SA0	0

Table 4: Inverter Fault Model

Special Circuits

Some special circuits can easily have test vector sets generated for them which model both single and multiple faults. Most such circuits are generally trivial, however, the 2 input AND-OR circuit is one which is of considerable practical value, it is very common in PLA structures.

Such a circuit is shown in figure 4.



The circuit of figure 4 realises the Boolean function:

$$Z = \overline{X_1}\,\overline{X_3} + \overline{X_3}X_4 + X_1X_2X_4 + X_2X_3\overline{X_4}$$

Every product term in a 2 level AND-OR circuit is a prime implicant of the function realised by the circuit. These prime implicants may be expressed using the CUBE notation,which is simply an alternative way of expressing logic functions (the cubes represent the vertices of a hypercube in a Boolean statespace).

$$\begin{array}{lcll} \text{The Cube associated with} & \overline{X_1}\,\overline{X_3} & \text{is} & (0x0x) \\ \text{The Cube associated with} & \overline{X_3}X_4 & \text{is} & (xx01) \\ \text{The Cube associated with} & X_1X_2X_4 & \text{is} & (11x1) \\ \text{The Cube associated with} & X_2X_3\overline{X_4} & \text{is} & (x110) \end{array}$$

So the function may also be realised as:

Z = {(0x0x),(xx01),(11x1),(x110)}

Consider now, the set of faults {(1/1),(2/1),(11/0)}

This completely covers all other faults in Gate G1. If (11/0) occurs, the cube {(0x0x)} disappears completely from Z leaving Z' = {(x0x1),(1x11),(x110)}

A test vector for this fault is any input pattern which causes the two functions Z and Z' to differ. Since the 0-cubes (minterms) of Z' must be a subset of those in Z, a test vector for 11/0 wil be any 0-cube in Z and not in Z'.

- T(11/0) = {Z} - {Z'}
- = {(0000),(0001),(0010),(0011)} - {(x0x1),(1x11),(x110)}
- = {(0000),(0010)}

In general, for any AND-gate's output SA0, the test set may be found by taking the difference between the dropped minterms and the other minterms. This is most easily done by expanding the dropped prime implicants to minterms and comparing them with the other prome implicants.

Testing AND gate output SA0 also tests for the appropriate OR gate output SA0 and any inverter output SA0 as well.

The test sets for the other AND gate outputs SA0 are :

- T(12/0) = {(1001)}
- T(13/0) = {(1111)}
- T(14/0) = {(0110),(1110)}

Testing for AND gate inputs SA1

Consider 1/1. This modifies Z to

Z" = {(x0xx),(x0x1),(x110)}

As before, the set of test vectors for this fault will be all those minterms in Z and not in Z" and vice versa, i.e., the set difference.

- T(1/1) = {Z} - {Z"}
- = {(00xx),(x0x1),(1x11),(x110)}-{(x0xx),(x0x1),(1x11),(x110)}
- = {(10xx)} - {(x0x1),(1x11),(x110)}

Expand {(1x0x)} to minterms and take the set difference

= {(1000),(1010)}

Similarly, the test sets for some of the other AND gate inputs SA1 are

- T(2/1)={(0110),(0101),(0111)}
- T(3/1)={(0101),(0111),(1101)}

Notice that the vectors (0101) and (0111) serve as tests for both 2/1 and 3/1.

Testing an AND gate input SA1 also tests for the OR gate output SA1, and any inverter output SA1 which lies in the path to the AND gate input. Testing the AND gate output SA1 and each input SA0 covers the AND gate. However, it also covers both the OR gate and the inverters. Thus, by testing only the AND gate we perform a complete test for all the faults in the circuit.

The Sensitized Path Method

This is a heuristic approach to generating tests for general combinational logic networks. The circuit is assumed to have only a single fault in it.

The sensitized path method consists of two parts

1. The creation of a SENSITIZED PATH from the fault to the primary output.
2. The JUSTIFICATION (or CONSISTENCY) operation, where the assignments made to gate inputs on the sensitized path are traced back to the primary inputs.

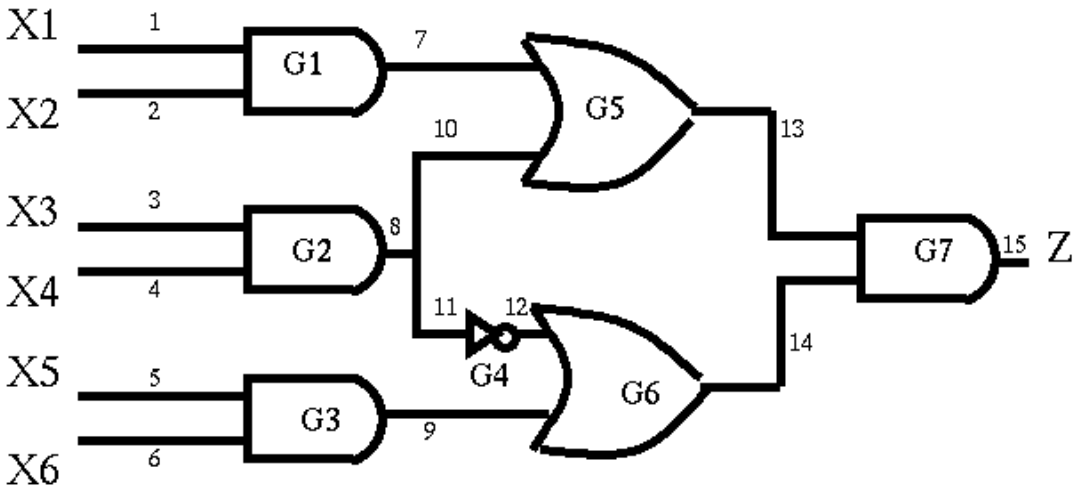


Figure 5 is an example network with an assumed fault 7/0. The sensitized path method will be used to generate a test for this fault.

PART 1. Create the sensitized path. This is done by forcing the complement of the fault on line 7 and propagating this value to the output.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1					1								
2										0			1		
3														1	0

Step 1 - Create conditions for fault to be detected on line 7, this can be achieved by applying the test vector for an AND gate output SA0 - net 1 = 1 and net 2 = 1. The value shown on net 7 in the table is that which it would carry in the absence of a fault.

Step 2 - We need to propagate the fault through G5, this may be done by setting net 10 = 0.

Step 3 - We now need to propagate through G7, this can be achieved by setting net 14 = 1.

In general the fault signal is propagated through each gate by picking a combination of the other inputs which cause the output to be solely dependent on the faulty input.

The sensitization stage is now complete because the fault has been propagated to a primary input

PART 2. Justify the assignment of values to the outputs of internal gates made in part 1 by working backwards towards the primary inputs.

Interior nets 10 and 14 have had values assigned to them, corresponding to the outputs of gates G6 and G2.

First we notice that net 10 having a value applied to it implies the values of nets 8, 11 and 12, so these are updated in step 4.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1					1			0			1	1	0
4								0			0	1			

Step 5 - Now we try to justify the assignment of a logic 1 to net 14 (output of G6). Notice that net 12 = 1 will force net 14 = 1, so this condition is automatically justified. Therefore, assign an X (don t care) to net 9.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1					1	0		0	0	1	1	1	0
5									X						

In justifying nets 8 and 9 we have created two new gates to justify, i.e., G2 and G3.

Step 6 - G3 is easy to justify, since it is X nets 5 and 6 can also be X.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1					1	0	X	0	0	1	1	1	0
6					X	X									

Step 7 - G2 may be justified by either net 3 = 0 and net 4 = X or net 3 = X and net 4 = 0, so finally we have.

step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1			X	X	1	0	X	0	0	1	1	1	0
7(a)			0	X											
7(b)			X	0											

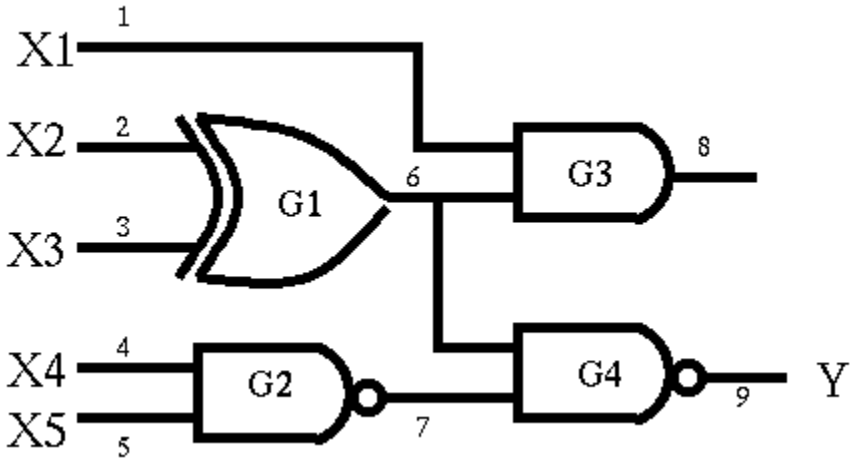
The test generation is now finished because all the primary inputs have been assigned values. T(7/0) = {(110XXX),(11X0XX)}.

Problems with the Sensitized Path Method

1. Making Choices
2. Reconvergent Fan-out Paths.

Making Choices

The sensitized path method attempts to drive a test to a single output. When the propagation routine reaches a net with fan-out it arbitrarily selects one path. Sometimes this blind choice of a path ignores easy solutions.



The NAND gate G4 of figure 5 is easy to control, its input from G2 can be set to 1 by setting PI X5 to 0. This makes an easy path to propagate faults on G1 and other to its left to a primary output.

On the other hand, gate G3 will be more difficult to control because its input (net 1) comes from other logic not directly connected to a primary input. Moreover, any test propagating through G3 must also propagate through other logic.

The path through G4 is the obvious choice, but the sensitized path heuristic has no way of recognizing this and is just as likely to choose a path through G3.

By preprocessing using the SCOAP algorithm, a hierarchy of suitable choices can be established.

Reconvergent Fan-out Paths

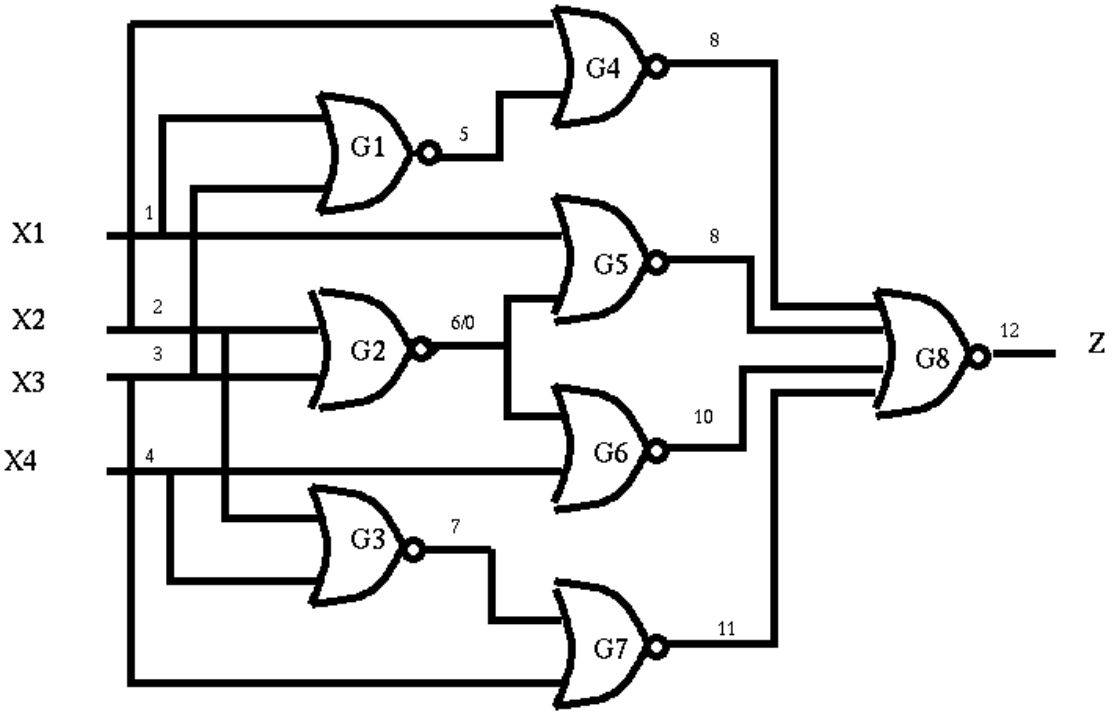
The sensitive path method is NOT guaranteed to find a test for a fault, even where such a test does exist. The principle cause of this problem is the presence of reconvergent fan-out paths in a circuit.

Consider the circuit of figure 6 with fault 6/0. The path sensitization procedure is:

Step 1 - Create conditions to detect fault on net 6.

Step 2 - Try propagating through gate G5 by assigning 0 to net 1.

Step 3 - Nets 1 and 3 are both 0 from steps 1 and 2, => net 5 = 1.



step	1	2	3	4	5	6	7	8	9
1			X	0		D			
2		0				D		$\overline{D}$	
		0	X	0		D		$\overline{D}$	
3							1	$\overline{D}$	D
		0	X	0		D	1	$\overline{D}$	D

step	1	2	3	4	5	6	7	8	9
		0	X	0		D	1	$\overline{D}$	D
4			0		X		1		
		0	0	0	X	D	1	$\overline{D}$	D
5	0	0			1		1		
	0	0	0	0	1	D	1	$\overline{D}$	D

Step 4 - Nets 5 = 1 from step 4, => net 8 = 0.

Step 5 - Now propagate net 9 to the output by setting nets 8, 10, 11 = 0

The path sensitization is now complete as the fault has been propagated to a primary output. Assignments to internal gate outputs must now be justified.

G6 = 0 and G7 = 0 need to be justified. Start with G6 = 0.

step	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0		1	1		0	0	0	0	1
6				1		1						
	0	0	0	1	1	1		0	0	0	0	1
7		0		1			0					
	0	0	0	1	1	1	0	0	0	0	0	1

Step 6 - Net 6 is SA0, so net 4 = 1 will justify net 10 = 0.

Step 7 - This implies G3 = 0, from net 2 = 0 and now net 4 = 1.

However, this assignment is INCONSISTENT because net 3 = 0 and net 7 = 0 but also net 11 = 0 according to the table. This is not correct, as the values specified on nets 3 and 7 should result in net 11 adopting a 1. The justification procedure has failed.

On examination we can see that the problem arose because we assigned a 1 to net 4. However, this is an inevitable assignment given the path we are trying to sensitize.

We could backtrack and try to sensitize a path through G6, but the same problem would arise. It appears that there is no test for 6/0. However, such a test does exist, it is T(6/0) = {(0000)}.

Why does the sensitized path method fail to find this test ?

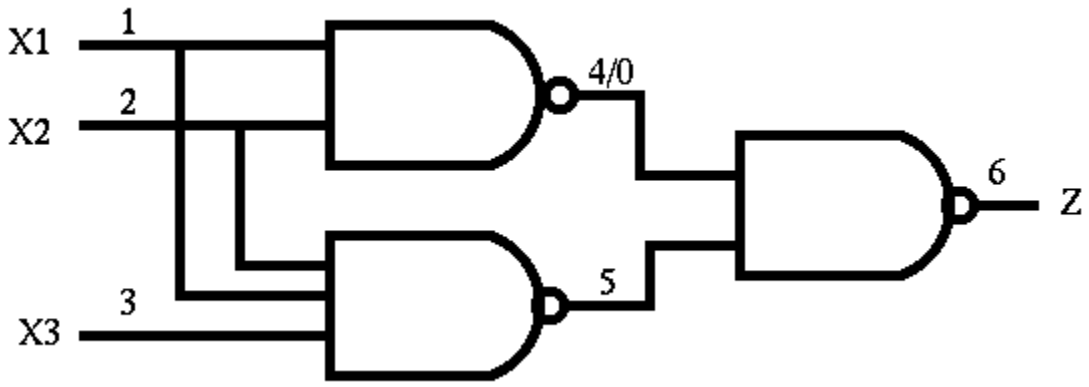
If we examine (0000) we see that this vector sensitizes a path through both G5 and G6 simultaneously. The problem with the sensitized path method is that it only ever attempts to sensitize a SINGLE path. For reconvergent fan-out networks this will usually cause problems during the justification stage.

The problem arises because, having completed the sensitization, we must, during justification, trace back along the alternative path(s) to the position with the fault. This will require us to justify a gate one of whose inputs is faulty, which severely restricts our choice of inputs. Usually this leads to the type of difficulties encountered in this example.

Redundancy and Undetectability

If no test vector exists for a fault then that fault is UNDETECTABLE. Undetectable faults are usually caused by redundant logic in a circuit, usually gates inserted to remove hazard conditions.

Figure 7 is an example of a circuit containing redundant logic which has an undetectable fault.



Fault 3/1 is undetectable because in order to propagate it to net 6 we require X1 = 1 and X2 = 1 but X1 AND X2 = 0.

Are undetectable faults a problem ? Yes, because they can MASK other faults.

Example

Fault Masking.

(110) is a test for 1/0. However, if 3/1 is simultaneously present in the circuit, this test will fail.

Undetectable faults arise from REDUNDANT logic. Consider the equation of the circuit shown above,

Y = X1 AND X2. Satisfy yourself that this is so.

The D-algorithm

The D-algorithm is a modification of the sensitized path method. Unlike the latter it is guaranteed to find a test vector for a fault if one exists.

The D-algorithm has been specified very formally, and is suitable for computer implementation. It is the most widely used test vector generator.

The primary difference between the D-algorithm and the sensitized path approach is that the D-algorithm always attempts to sensitize every possible path to the primary outputs.

D-Notation

This is a compact way of specifying how faults propagate through a circuit. D is a composite signal, it implies that in the good machine a 1 is to be found at the node holding D, whereas in the faulted machine a 0 is to be found at that node. Not(D) is analogously defined.

		Faulted Machine	
		0	1
Good Machine	0	0	$\overline{D}$
	1	D	1

``D" stands for ``discrepancy signal".

Singular Cover

The Singular Cover of a logic gate is a compact representation of its truth table. i.e., 2-input AND gate.

Truth table

a	b	F
0	0	0
0	1	0
1	0	0
1	1	1

Singular Cover

a	b	F
0	X	0
X	0	0
1	1	1

Each row of the singular cover is called a CUBE. The set of cubes which contain 0 as the output value is called the P0set. The set of cubes containing 1 as the output value is called the P1 set. For the AND gate:

$$p_0 = \{(0X0), (X00)\}$$
$$p_1 = \{(111)\}$$

Another way of thinking of the singular cover of a function F - it is the union of the prime implicants of F and those of Not(F).

Primitive D-Cubes of Failure (P.D.C.F.s)

A Primitive D-Cube of Failure for a fault in a circuit is a set of inputs to the circuit which bring the fault to the circuit output..

To generate the P.D.C.F. for a fault

- Generate singular covers for the circuit in both its faulted and fault-free states.
- Intersect the P0 cubes of the fault free cover with the F1 cubes of the faulted cover and intersect the P1 cubes with the F0 cubes.

F1 and F0 play analogous roles in the faulted cover to P1 and P0 in the fault-free cover.

Intersection is defined by intersecting each element of the cubes according to the following table.

		Faulted Machine		
		0	1	X
Good Machine	0	0	$\overline{D}$	0
	1	D	1	1
	X	0	1	X

Note - D and Not(D) are only allowed on *OUTPUT pins for P.D.C.F. intersection.*

Example

Form the P.D.C.F.s for a 2-input AND gate where input 1 is SA0.

We already have the singular cover for the fault-free AND gate, it is

$$p_0 = \{(0X0), (X00)\}$$
$$p_1 = \{(111)\}$$

For the faulted AND gate 1/0 the cover is

1	2	3
X	X	0

i.e., F1 is the empty set and F0 contains all input combinations

$$p_0 \cap f_1 = \{(0X0), (X00)\} \cap \{\}$$
$$= \{\}$$
$$p_1 \cap f_0 = \{(111)\} \cap \{(XX0)\}$$
$$= \{(11D)\}$$

Now, performing the intersections

$$
\begin{aligned}
p_0 \cap f_1 &= \{(0X0),(X00)\} \cap \{(1X1)\} \\
&= \{(10\overline{D})\} \\
p_1 \cap f_0 &= \{(111)\} \cap \{(0X0)\} \\
&= \{\}
\end{aligned}
$$

So the only P.D.C.F. for 1/0 is {(11D)}.

Example

Form the P.D.C.F.s for a 2-input AND gate where input 2 is SA1.

For the faulted AND gate 2/1 the cover is:

1	2	3	
0	X	0	f_0
1	X	1	f_1

Intersecting

$$
\begin{aligned}
p_0 \cap f_1 &= \{(0X0),(X00)\} \cap \{(1X1)\} \\
&= \{(10\overline{D})\} \\
p_1 \cap f_0 &= \{(111)\} \cap \{(0X0)\} \\
&= \{\}
\end{aligned}
$$

The only P.D.C.F. for 2/1 is {(10 Not(D))}

Propagation D-Cubes

The propagation D-Cubes of a gate are those which cause the output of a gate to depend solely on one or more of its inputs (usually one). This allows a fault on this input to be propagated through the gate.

The propagation D-cubes for a 2-input AND gate are

a	b	y
1	D	D
D	1	D
D	D	D
1	$\overline{D}$	$\overline{D}$
1	1	$\overline{D}$
$\overline{D}$	$\overline{D}$	$\overline{D}$

To generate the propagation D-cubes, intersect region P0 of a gate's cover with region P1 according to the following table

		p_1		
		0	1	X
p_0	0	0	D	0
	1	$\overline{D}$	1	1
	X	0	1	X

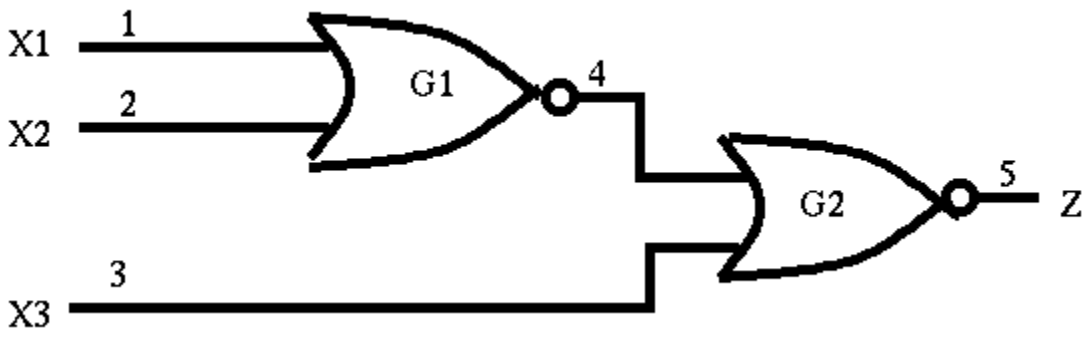
In general it is possible to have up to 2^(2N-1) propagation D-cubes for an N-input gate, so normally only those cubes with a single D in the inputs are stored. Cubes with in the inputs are easily formed by complementing all the in a cube, and cubes with more than one D in the inputs can be formed by intersecting the covers.

D-Intersection

The D-Intersection is the method used to build sensitized paths. It is a set of rules which show how D signals at the outputs of gates intersect with the propagation D-cubes of other gates, allowing a sensitized path to be constructed.

Example

Generate a test for 2/0 in the circuit of figure 8



2/0 has P.D.C.F. {(01 Not(D))}. To transmit the Not(D) on net 4 through G2 we must try to match (i.e., intersect) the specification with one of the propagation D-cubes for G2. Such a match is possible if the propagation D-cube (0D Not(D)) is used.

	1	2	3	4	5
P.C.D.F.	0	1		$\overline{D}$	
Prop. D–cube			0	$\overline{D}$	D
Intersection	0	1	0	$\overline{D}$	D

The full set of rules for the D–intersection is as follows. Let $A = (a_1, a_2, \cdots, a_n)$, $B = (b_1, b_2, \cdots, b_n)$ be D–cubes where $a_i, b_i \in \{0, 1, X, D, \overline{D}\}$, $1 \leq i \leq n$. The D–intersection, denoted by $A \bigcap B$ is as follows

1. $X \cap a_i = a_i$
2. if $a_i \neq X$ and $b_i \neq X$ then

$$a_i \cap b_i = \begin{cases} a_i & \text{if } a_i = b_i \\ \phi & \text{otherwise} \end{cases}$$

3. $A \bigcap B = \Phi$, i.e., the null intersection or empty cube, if for any i, $a_i \cap b_i = \phi$

Examples

$$(1X1D0) \bigcap (X\overline{D}1D0) = (X\overline{D}1D0)$$
$$(01\overline{D}X1) \bigcap (00XD1) = (0\phi DD1) = \Phi$$

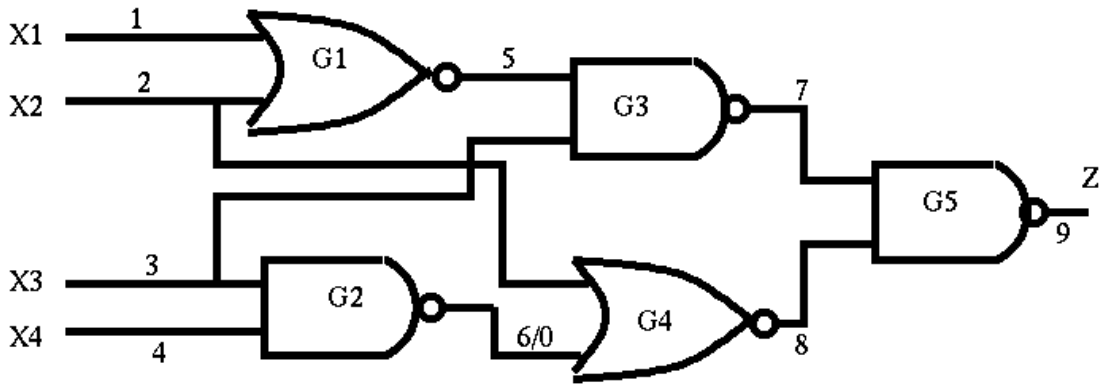
For purposes of intersection blank entries in a table correspond to X's.

The Full D-Algorithm

1. Choose a P.D.C.F. for the fault under consideration.
2. Sensitize all possible paths from the faulty gate to a primary output of the circuit. We do this by successively intersecting the P.D.C.F. of the fault with the propagation D-cubes of successor gates. The process is called the ``D-Drive".
3. Justify the net assignments made during the D-drive by intersecting the singular covers of gates in the justification path with the expanding D-cube. This is called the ``Consistency Operation".

Example

Use the D-algorithm to generate a test for 6/0 in the network shown in figure 9.



D-Drive

Step 1 - Select P.D.C.F. for 6/0.

Step 2 - Intersect with propagation D-cube for NOR gate G4.

Step 3 - Intersect with propagation D-cube for NAND gate G5.

At this stage a primary output has been reached, so the D-drive is complete

step	1	2	3	4	5	6	7	8	9
1			X	0		D			
2		0				D		$\overline{D}$	
		0	X	0		D		$\overline{D}$	
3							1	$\overline{D}$	D
		0	X	0		D	1	$\overline{D}$	D

step	1	2	3	4	5	6	7	8	9
		0	X	0		D	1	$\overline{D}$	D
4			0		X		1		
		0	0	0	X	D	1	$\overline{D}$	D
5	0	0			1		1		
	0	0	0	0	1	D	1	$\overline{D}$	D

Consistency Operation

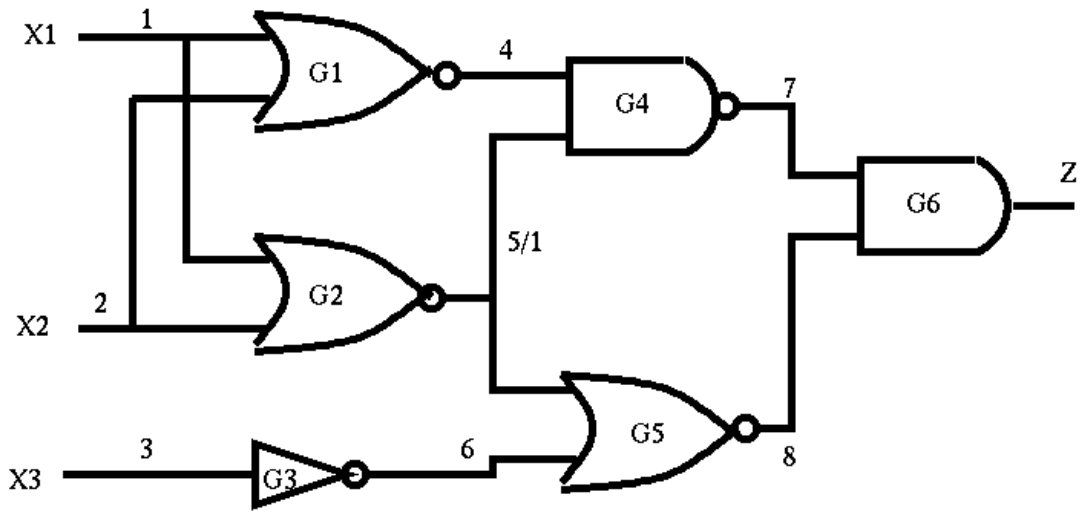
Step 4 - Justify the assignment G3 = 1. By examining the cover for a NAND gate we see that (0X1) will serve.

Step 5 - Justify the assignment G1 = X. Any combination of inputs will serve, so we choose (001) arbitrarily.

All primary inputs have been assigned values, so the consistency operation is complete and we have a test for 6/0.

This is an easy example because no inconsistencies were encountered. When they are, it is necessary to backtrack to the last point at which an arbitrary decision was made and make another choice at that point. This can lead to a lot of BACKTRACKING and can make the algorithm very slow if a lot of alternative paths must be examined.

Figure 10 is an example circuit where inconsistencies force backtracking.



D-Drive

Step 1 - Select P.D.C.F. for 5/1.

Step 2 - Carry out implications of this choice, net 4 becomes 1.

Step 3 - Propagate through G4 by using one of the propagation D-cubes of the NAND gate.

Step 4 - Propagate through G5 by using a propagation D-cube of the NOR gate.

step	1	2	3	4	5	6	7	8	9
1	1	X			$\overline{D}$				
2	1	X		1					
	1	X		1	$\overline{D}$				
3				1	$\overline{D}$		D		
	1	X		1	$\overline{D}$		D		
4					$\overline{D}$	0		$\overline{D}$	
	1	X		1	$\overline{D}$	0	D	$\overline{D}$	
5							ϕ	ϕ	ϕ
	1	X		1	$\overline{D}$	0	ϕ	ϕ	ϕ

Step 5 - We now have D on net 7 and Not(D) on net 8. No propagation D-cube exists for this combination of inputs on an AND gate (G6), so Nulls are entered instead.

This causes the intersection to fail, so the D-drive has failed. We need to back up and choose another value for either net 7 or net 8. This is a justification step for G6.

Step 6 - We arbitrarily choose to modify net 8 and select to set it to 0. This action invalidates any inputs driving G5 which may have been chosen during the D-drive. In this instance the value on net 6 becomes invalid and is dropped.

step	1	2	3	4	5	6	7	8	9
	1	X		1	$\overline{D}$		D	0	
6							ϕ	ϕ	ϕ
	1	X		1	$\overline{D}$		ϕ	ϕ	ϕ

Step 6 also fails because of the lack of a propagation D-cube. Again we must back up and try another choice of input.

Step 7 - This time we try net 8 = 1. A propagation D-cube does exist for this combination of inputs on G6, so the intersection is successful. This completes the D-drive.

step	1	2	3	4	5	6	7	8	9
	1	X		1	$\overline{D}$		D	1	
7							D	1	D
	1	X		1	$\overline{D}$		D	1	D

Consistency Operation

If a gate has a logic value assigned to its output but is missing values at its inputs it must be justified. Gates G5 and G3 need justification.

Step 8 - Justify G5. This gate has a fault on one input, which can be matched using an X.

Step 9 - Justify G3. This completes the consistency operation.

step	1	2	3	4	5	6	7	8	9
	1	X		1	$\overline{D}$		D	1	
7							D	1	D
	1	X		1	$\overline{D}$		D	1	D

Other Test Generation Methods

In this section, we will briefly examine some other test generation methods.

L.A.S.A.R.

L.A.S.A.R. - Logic Automated Stimulus And Response. Otherwise known as the Critical Path method. Unusual in that all circuits to be analyzed using this technique must first be converted to NAND equivalent form. Very similar to the justification part of the D-algorithm. It works back from assumed values on primary outputs towards the inputs. Used in the HILO simulator.

P.O.D.E.M.

Path-Oriented DEcision Making. This algorithm was designed to generate tests for error detection/correction circuits. These circuits typically contain large numbers of XOR gates which slow down the D-algorithm. Has been found to work as well as or better than the D-algorithm for most general circuits. Works from primary inputs towards fault site and primary outputs.

Boolean Differences

This approach was popular amongst researchers in the 1960's, but has not survived.

Design for Testability

There are various techniques in common usage which help the designer of digital systems to ensure that his system will be testable. In the following sections, we shall consider some of these.

Ad-Hoc Techniques

In this section we shall list a set of what might be termed design for testability rules. These are not architecture design techniques per-se, but rather a set of guidelines.

1. A global reset signal is important, this brings all of the internal memory elements to a known state.
2. Long counter chains should be broken. A 10 bit counter needs 1024 cycles to test it fully, if divided into 2 5 bit counters, only 32 cycles are required. (Plus a few (approx 18) cycles for testing the decode logic. This approach may be difficult with fast synchronous counters with lookahead carry.
3. Bring difficult to test internal nodes, out to device pins. This may also be difficult, as pads are usually at a premium.
4. On board clock generators should be replacable by an external test clock signal, this will allow external control of the clock during test.
5. Never, Ever use asynchronous design, this can lead to RACE conditions, and lots of other nasty problems. Only ever use the CLEAR input on a flip flop for the global reset signal.

Structured Techniques

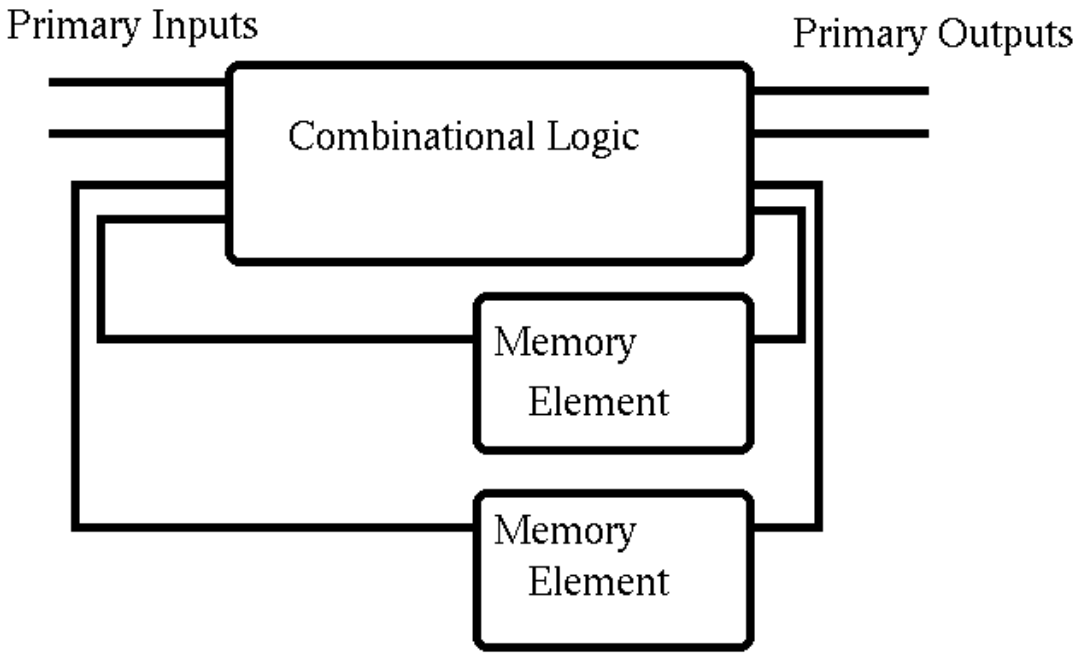


Figure 11 illustrates a canonical model of a sequential circuit. What the structured design for testability techniques do, is to break the feedback path. This allows access to the memory elements from external pins.

Scan Paths

Since IO Pins are generally expensive in terms of Silicon area, and are in short supply, the memory elements (flip flops or latches) are usually connected in a shift register chain for test purposes. This is called a *SCAN PATH* design.

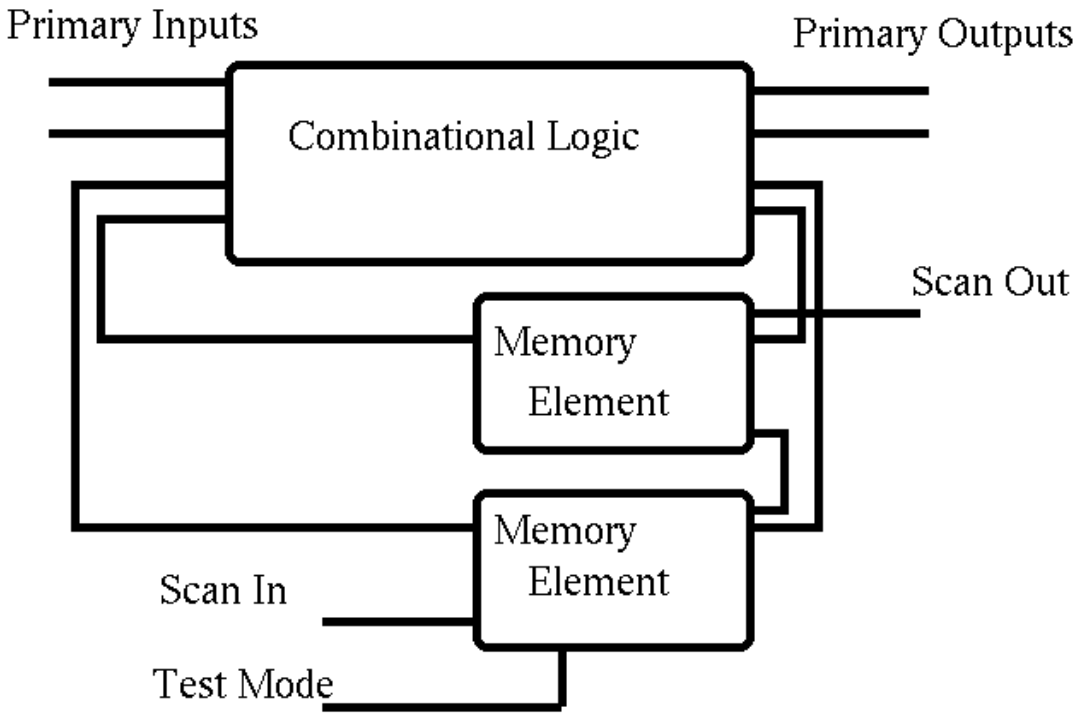


Figure 12 illustrates the canonical system of figure 11 with a Scan Path added to it. During test, a binary sequence is shifted into the Scan_Ininput. Tests can be generated for the combinational logic block, by treating the memory element outputs as primary inputs, and the memory element inputs as primary outputs.

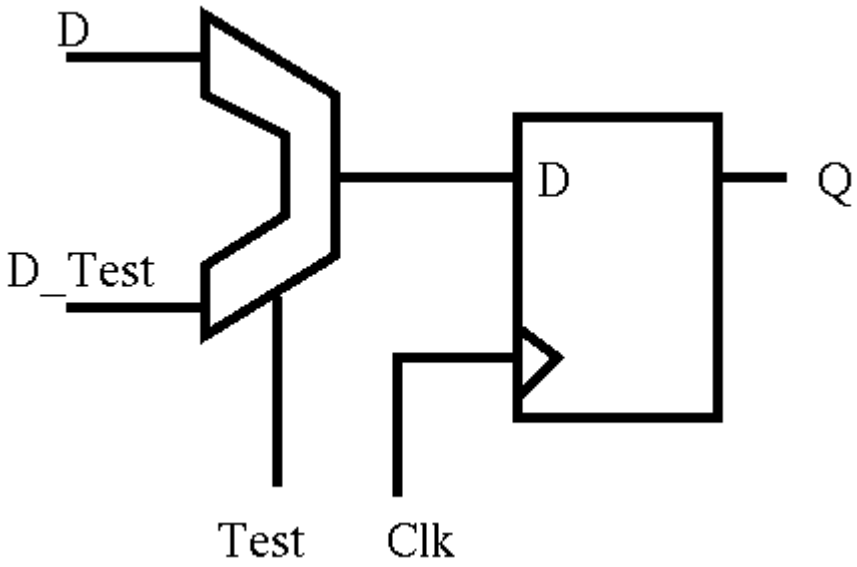
The shift register chain is first tested by shifting a 1 0 1 0 1 ...pattern through it, Once the shift register chain has been demonstrated to be working correctly, test patterns can be shifted through it. Having shifted in a test pattern, the device can be taken out of Scan Mode, and 1 cycle of normal operation performed, to check that the combinational block is working with that pattern. The test results may then be shifted out in scan mode.

Scan Path Implementation

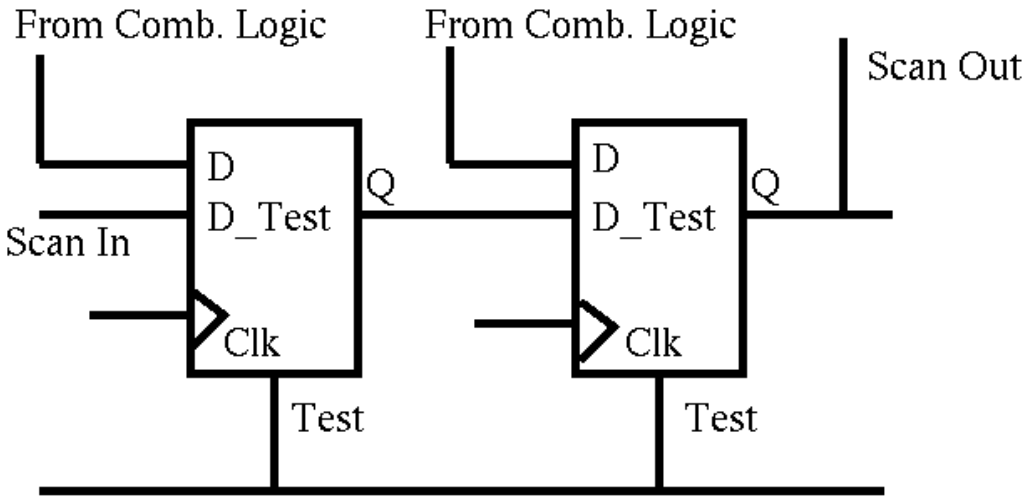
Scan path designs may be implemented in various ways

Stanford Scan Path Design

In this design, the memory elements are made up of a flip-flop, with an extra multiplexer added as shown in figure 13.



The flip-flops are then connected as shown in figure 14.



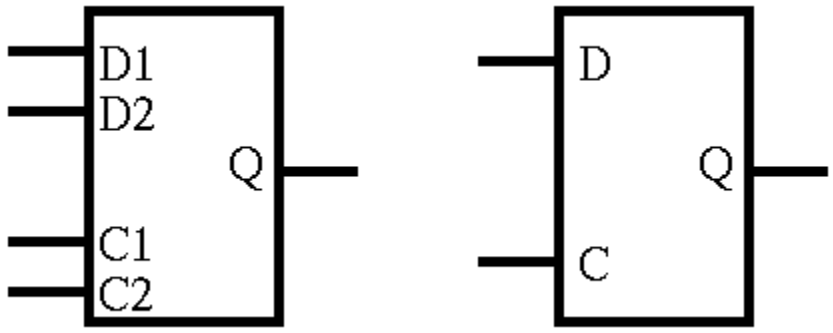
To test the design :

1. Set test = 1.
2. Load test pattern into flip-flops by clocking.
3. Set test = 0.
4. Apply 1 clock pulse, results are clocked into the same flip-flops which held the test pattern.
5. Set test = 1 and clock out the result.

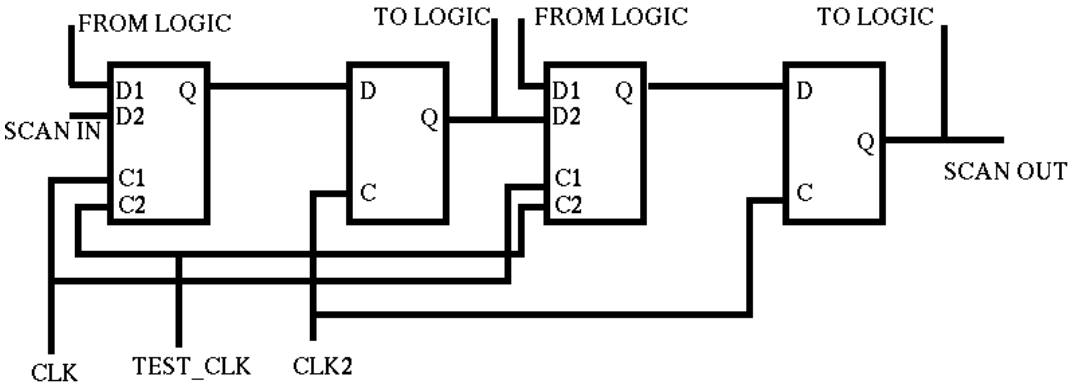
Latch Based Designs

Latch based designs attempt to eliminate circuit race hazards. A completely hazard free circuit is easier to test and more reliable. The most important technique was developed by IBM and is called *Level Sensitive Scan Design* or LSSD.

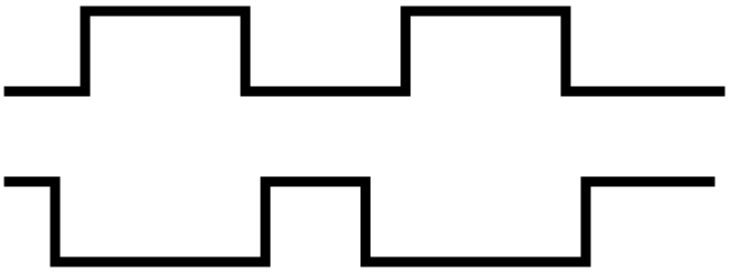
In LSSD, each memory element consists of 2 latches, the L1 latch and the L2 latch. The L1 latch is a 2 port latch, with 2 clock inputs as shown in figure 15.



Input D1 is controlled by clock signal C1, when C1 is high, then D1 is connected to Q Normally, D1 is connected to the outputs of the system combinational logic, and D2 to the scan path. Latch L2 is a standard D-type latch. The system is connected together as shown in figure 16.



This is one possible LSSD structure, it is designed by directly replacing flip-flops in an IC. Each latch pair is used exactly like a Master-Slave flip-flop in normal operation. A 2 phase, nonoverlapping clock is used on CLK1 and CLK2 as shown in figure 15.



The test procedure is as follows.

1. Apply pulses to TSTCLK and CLK2 in order to shift a bit pattern into the circuit.
2. Apply 1 CLK1 pulse followed by 1 CLK2 pulse to run the test through the circuit.
3. Clock the result out using TSTCLK and CLK2.

This is only 1 technique which may be used to design LSSD circuits.

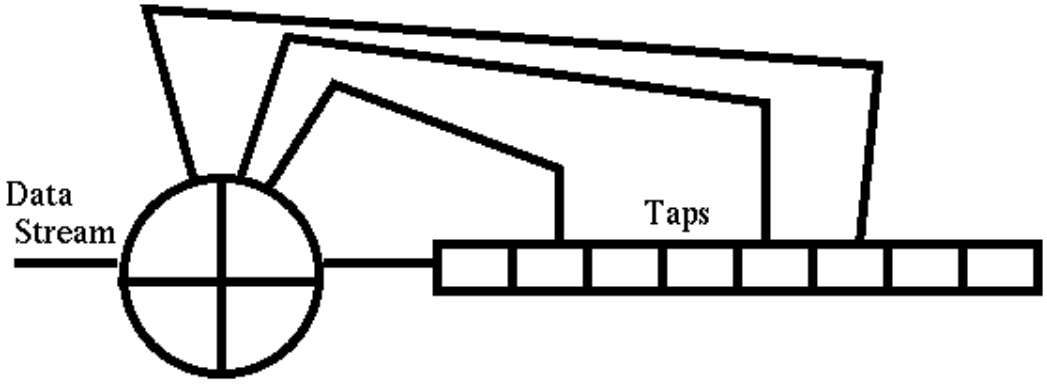
Self Test

Various forms of self test techniques are used in modern Digital IC design. In the following sections we shall look at some of these.

Signature Analysis

Signature Analysis is a data compression technique, it takes very long sequences of bits from a unit under test and compresses them into a unique N-bit signature which represents the circuit. A good circuit will have a unique signature, and a faulty one will deviate from this.

Signature analysis is based on Linear Feedback Shift Registers (LFSR), basically, the memory elements in the system are reconfigured in test mode, to form an LFSR, as shown in figure 18.



The summation unit (+) performs modulo 2 addition (according to the rules of addition in GF(2)) on the incoming bit stream and the taps coming back from the LFSR. (XOR Gates).

A bit stream is fed in to the register (which is initially all 0's, because you remembered to put in the global reset signal !). After N clock pulses, the register will contain the signature of the data stream. Hewlett Packard, among others, make signature analysers for this purpose. Such a machine can trap all 1 bit errors, it is however possible that 2 or more errors will mask each other. The probability of two different data streams yielding the same signature is given by.

$$P_{err} = \frac{2^{n-m} - 1}{2^n - 1}$$

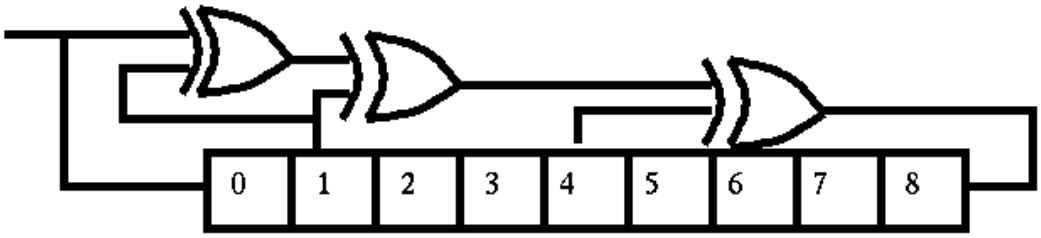
Where m is the length of the LFSR and n is the length of the sequence, for n tending to infinity this tends to.

$$P_{err} \approx \frac{1}{2^m}$$

So by making m large, the probability of a bad sequence being masked is small. Hewlett Packard use m = 16, giving Perr = 1.5E-5 and have not found this error probability to pose a problem in practice.

Generating Self Test Patterns

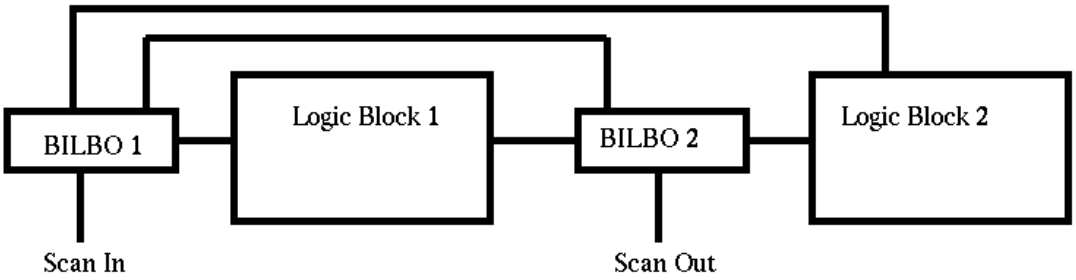
LFSR's corresponding to primitive polynomials over GF(2) make good sources of pseudo-random patterns (PRBS). As an example, consider figure 19 where the sequence length will be 2^16 -1 distinct patterns (all zeros is not allowed).



To perform in-situ testing of a logic network, we could place one of these registers at its input, and some signature analysis circuitry at the output. The LFSR generates random binary sequences which are fed through the network under test, and analysed by the signature analysers.

BILBO

BILBO, is a rather unfortunate acronym for *Built In Logic Block Observation*, it implements the signature analysis idea, in practice. The memory elements in the system are connected in a Scan Path as shown in figure 20.



Each BILBO can act as.

- A Scan Path shift register.
- An LFSR generating random patterns.
- A multi-input signature analyser.

Provided that the start state is known, and that a known number of clock cycles are injected, the finish state will be a known pattern.

Test Procedure with BILBO's

Testing using BILBO's is carried out as follows.

For logic block 1.

1. BILBO 1 is initialised to a non-zero initial state by shifting in a pattern through Scan_In.
2. BILBO 1 is configured as a PRBS generator and BILBO 2 as a multi-input signature analyser.
3. N clock pulses are applied.
4. BILBO 2 is configured as a Scan-Path, and the result is shifted out through Scan_Out.

To test logic block 2, BILBO 2 becomes the sequence generator, and BILBO 1 the signature analyser.

The quality of the tests generated (fault coverage) must be determined by prior fault simulation. The final signature may be determined by checking a known good part (Dangerous !) or by logic simulation.

The translation was initiated by strunzb@itdsrv1.ul.ie on Mon Jan 23 16:44:16 WET 1995