ECE381(CAD), Lecture 11:

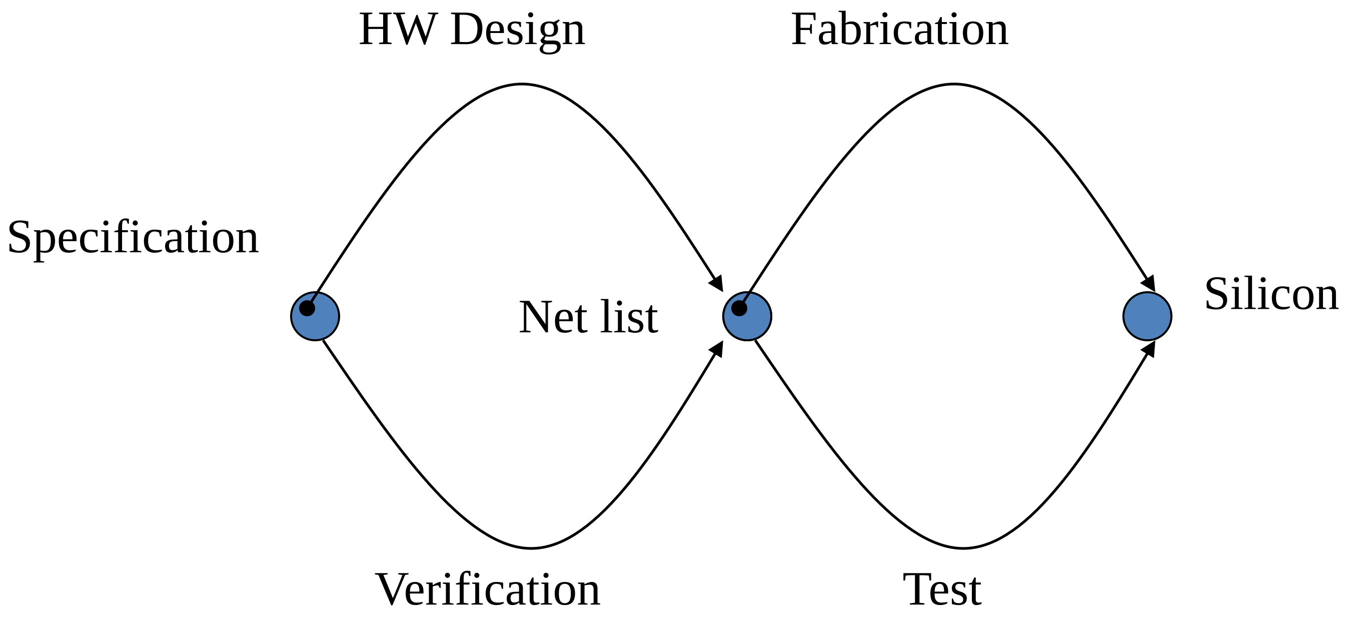
Test bench and, Random Generation, and Test Pattern Generation

Mehdi Modarressi
Department of Electrical and Computer Engineering,
University of Tehran

Verification VS. Test

- Two often confused
- Purpose of *test* is to verify that the design was manufactured properly
  - To ensure the chip is physically correct
  - Done after manufacturing
- Purpose of *Verification* is to ensure that the design has the right functionality
  - The code does exactly what it should do
  - Done in software before manufacturing

Verification and Test Reconvergence Model



- TEST
- VERIFICATION

Verification

- To make sure the code is actually does what it supposed to do
- Unlike test, here we don’t have circuit nodes to sensitize
  - We should check the functionality of the design and find wrong code pieces
- Use *Test bench* for verification

Wrong multiplexer code

```
PROCESS (a,b,s)
  IF(s=0)
     D<=a;
  ELSE
     D<=s;
END PROCESS;
```

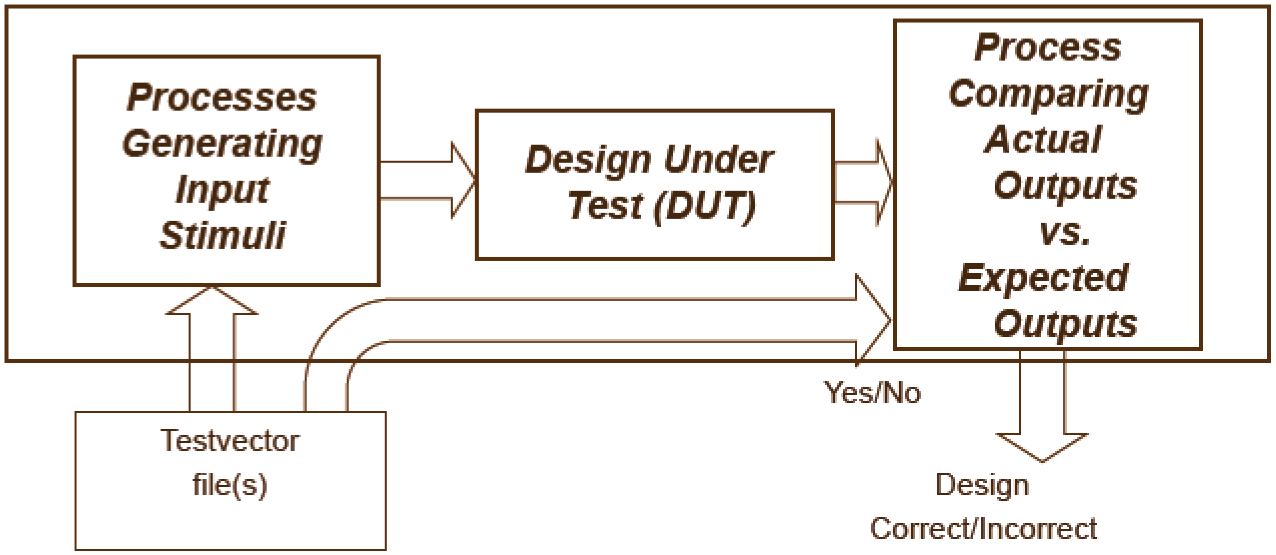
What is A Testbench?

- Test bench is a program that verifies the functional correctness of a hardware design (Design under Test= DUT).

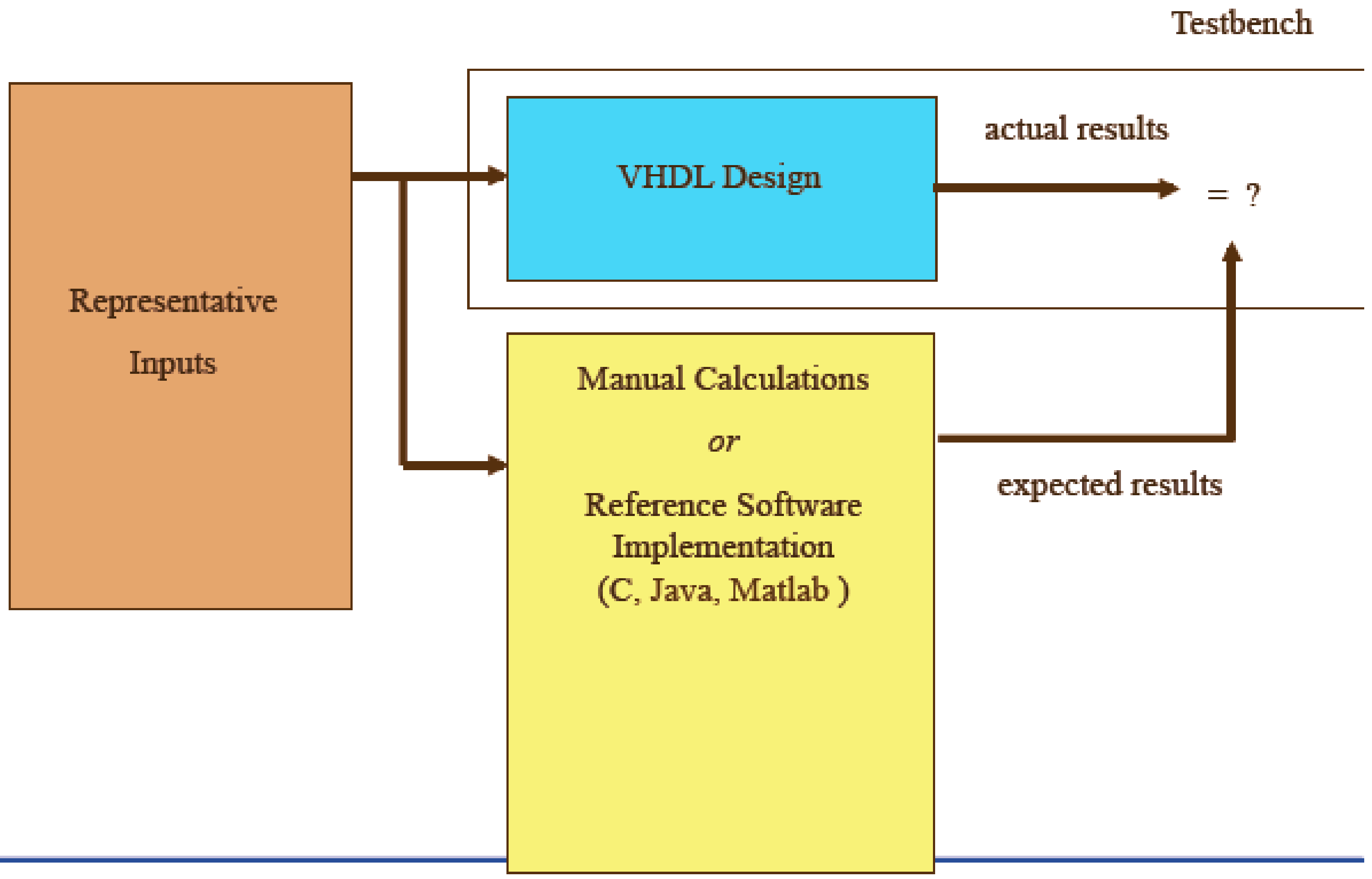
- The test bench program checks whether the hardware model does what it is supposed to do and is not doing what it is not supposed to do.

Main Functions of a Test Bench

- Generate stimulus for testing the hardware block.
- Apply the stimulus.
- Compare the generated outputs against the expected outputs.


Processes Generating Input Stimuli
Design Under Test (DUT)
Process Comparing Actual Outputs vs. Expected Outputs
Yes/No
Testvector file(s)
Design Correct/Incorrect

Test bench data



Generating Stimulus Vectors

- Vectors can be generated within the test bench program or generated elsewhere and supplied to the test bench program as an input file.

- Vectors can also be stored in a table within the test bench program.

A sample Test bench structure

```
ENTITY my_entity_tb IS
              --TB entity has no ports
END my_entity_tb;

ARCHITECTURE behavioral OF tb IS

     --Local signals and constants

  COMPONENT TestComp --All Design Under Test component declarations
            PORT (  );
  END COMPONENT;
-----------------------------------------------------
BEGIN
  DUT:TestComp PORT MAP(  ⎫   -- Instantiations of DUTs
                      );  ⎭

  testSequence: PROCESS
                                  -- Input stimuli
  END PROCESS;
 END behavioral;
```

Typical VHDL test bench

```
entity test_bench is
end;

architecture tb_behavior of test_bench is
     component design_under_test
               port ( list-of-ports-their-types-and-
     modes);
     end component;
     Local-signal-declarations;
begin
     CLOCK: process
      begin
               clock <= '0'; wait for t ns;
               clock <= '1'; wait for t ns;
     end process;
```

```
     Generate-stimulus-vectors-using-behavioral-
     constructs;
     Apply-to-entity-under-test;
     DUT: design_under_test port map ( port-
     associations );
     Monitor-output-values-and-compare-with-
     expected-values;
          if (no errors)
               report "Test bench completed!"
               severity note;
          else
              report "Something wrong!"
              severity error;
          end if;
end tb_behavior;
```

Test bench for 3-input XOR

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xor3_tb IS
END xor3_tb;

ARCHITECTURE behavioral OF xor3_tb IS
 -- Component declaration of the tested unit
 COMPONENT xor3
 PORT(
      A : IN STD_LOGIC;
      B : IN STD_LOGIC;
      C : IN STD_LOGIC;
      Result : OUT STD_LOGIC );
 END COMPONENT;

 -- Stimulus signals - signals mapped to the input and inout ports of tested entity
 SIGNAL test_vector: STD_LOGIC_VECTOR(2 DOWNTO 0);
 SIGNAL test_result : STD_LOGIC;
```

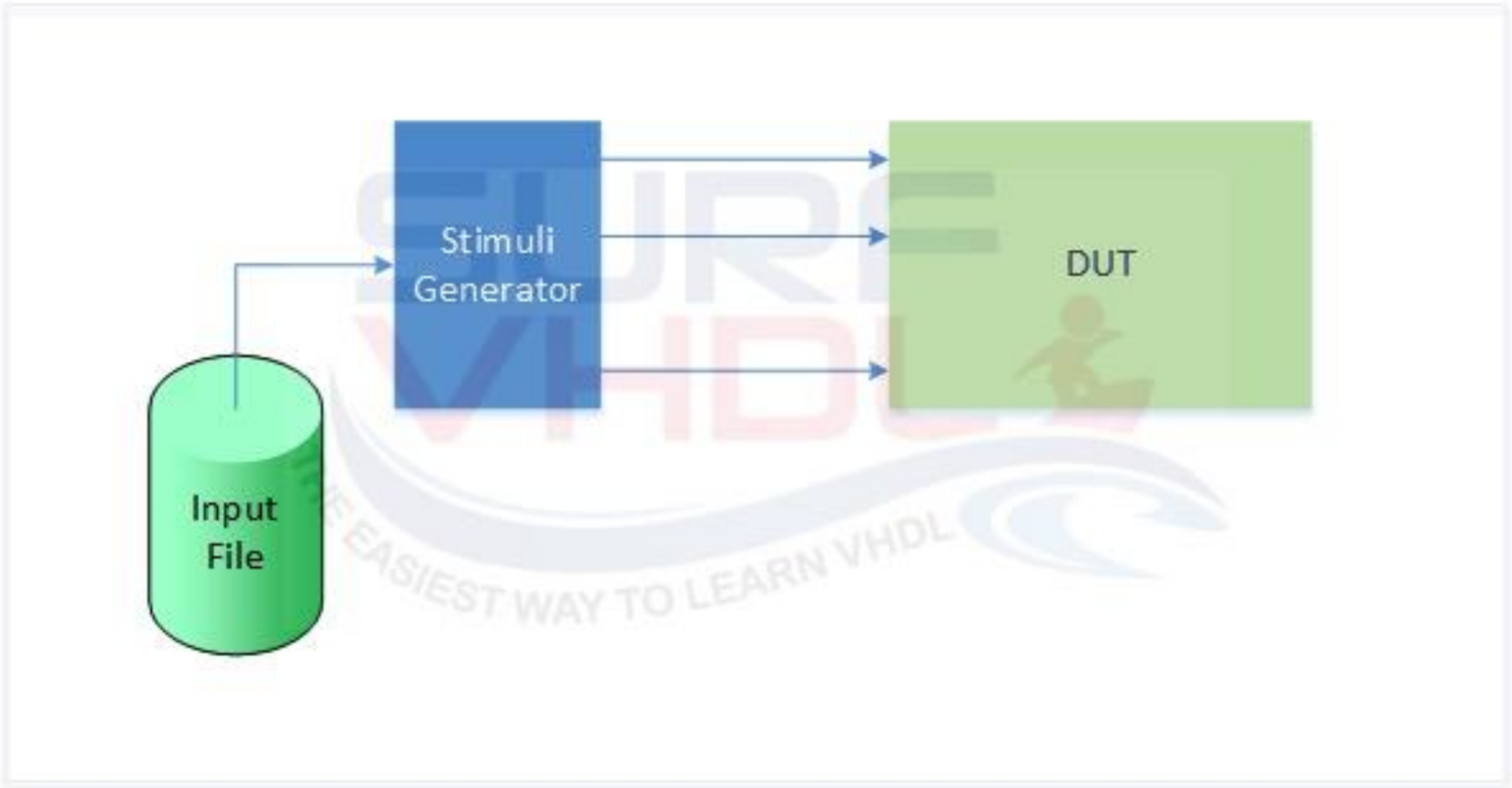
```
BEGIN
 UUT : xor3
               PORT MAP (
                             A => test_vector(2),
                             B => test_vector(1),
                             C => test_vector(0),
                             Result => test_result);
                                        );
 Testing: PROCESS
 BEGIN
     test_vector <= "000";
     WAIT FOR 10 ns;
     test_vector <= "001";
     WAIT FOR 10 ns;
     test_vector <= "010";
     WAIT FOR 10 ns;
     test_vector <= "011";
     WAIT FOR 10 ns;
     test_vector <= "100";
     WAIT FOR 10 ns;
     test_vector <= "101";
     WAIT FOR 10 ns;
     test_vector <= "110";
     WAIT FOR 10 ns;
     test_vector <= "111";
     WAIT FOR 10 ns;
 END PROCESS;
END behavioral;
```

Clock for testbench

- In case the DUT accepts a clock input, we can make it in the Test bench by a process statement

```
CLOCK: process
  begin
      clock <= '0'; wait for t ns;
      clock <= '1'; wait for t ns;
 end process;
```

Test data from file



File I/O

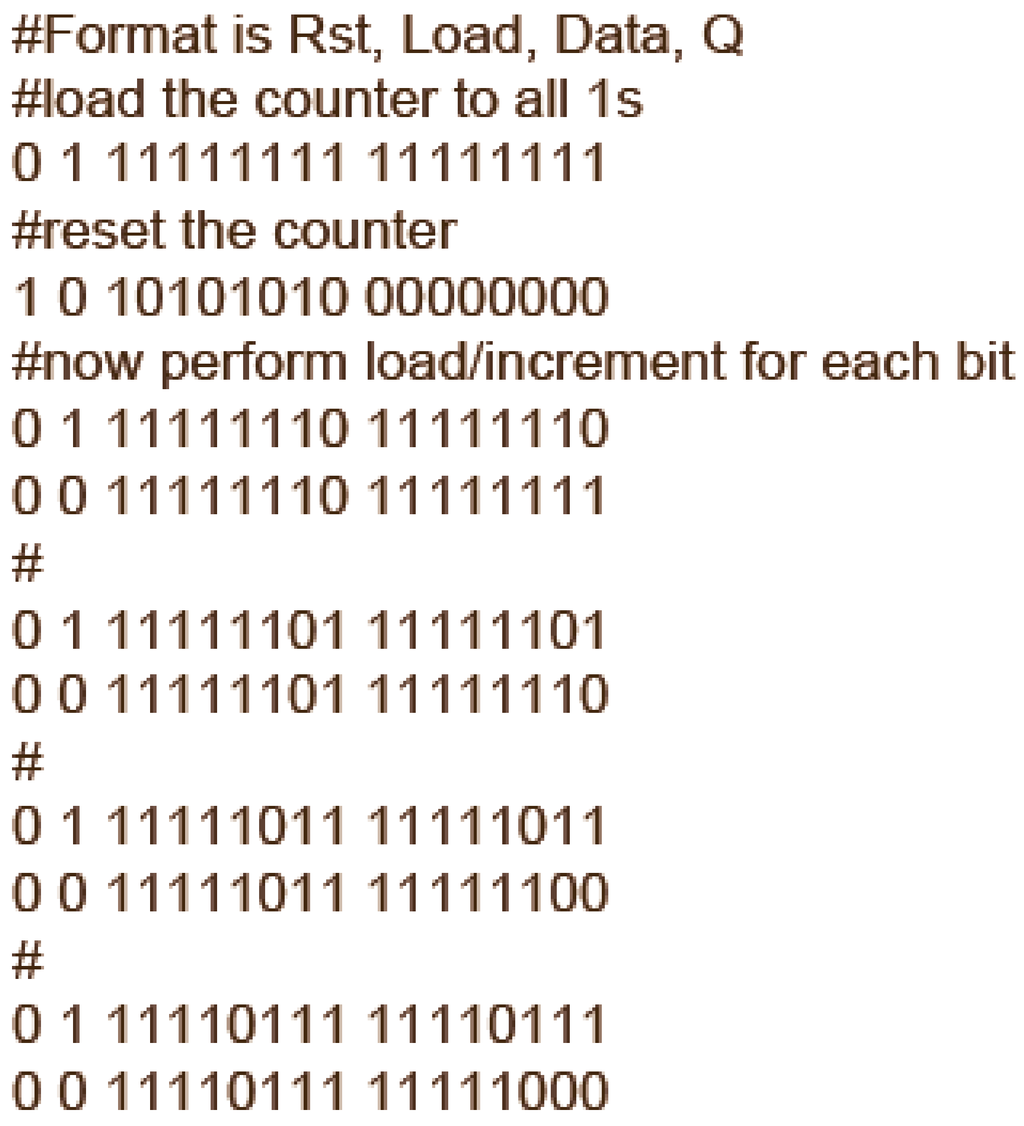
- Test a counter with load and reset

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY loadCnt IS
PORT (
  data:   IN STD_LOGIC_VECTOR (7 DOWNTO 0);
  load:   IN STD_LOGIC;
  clk:    IN STD_LOGIC;
  rst:    IN STD_LOGIC;
  q:      OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END loadCnt;
```

Architecture of the counter

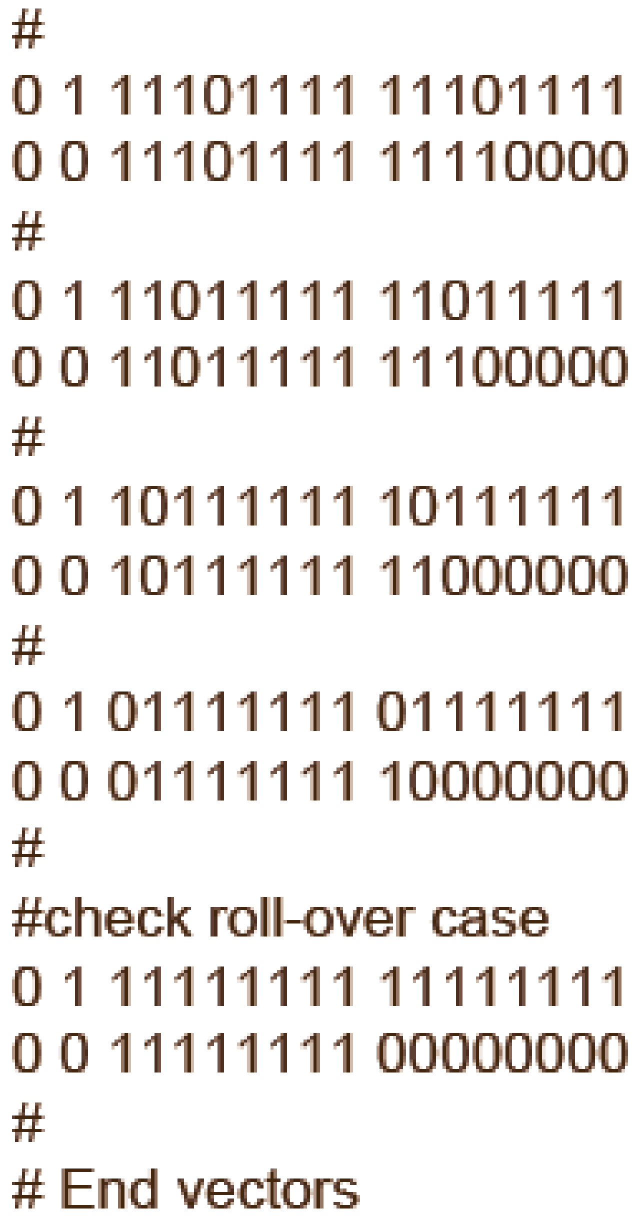
```
ARCHITECTURE rtl OF loadCnt IS

SIGNAL cnt: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
 counter: PROCESS (clk, rst)
BEGIN
   IF (rst = '1') THEN
    cnt <= (OTHERS => '0');
   ELSIF (clk'event AND clk = '1') THEN
    IF (load = '1') THEN
     cnt <= data;
    ELSE
     cnt <= cnt + 1;
    END IF;
   END IF;
 END PROCESS;
 q <= cnt;
END rtl;
```

Test vector saved in a file



```
#Format is Rst, Load, Data, Q
#load the counter to all 1s
0 1 11111111 11111111
#reset the counter
1 0 10101010 00000000
#now perform load/increment for each bit
0 1 11111110 11111110
0 0 11111110 11111111
#
0 1 11111101 11111101
0 0 11111101 11111110
#
0 1 11111011 11111011
0 0 11111011 11111100
#
0 1 11110111 11110111
0 0 11110111 11111000
```

Test vectors (C'ntd)



```
#
0 1 11101111 11101111
0 0 11101111 11110000
#
0 1 11011111 11011111
0 0 11011111 11100000
#
0 1 10111111 10111111
0 0 10111111 11000000
#
0 1 01111111 01111111
0 0 01111111 10000000
#
#check roll-over case
0 1 11111111 11111111
0 0 11111111 00000000
#
# End vectors
```

testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_textio.all;

LIBRARY std;
USE std.textio.all;

ENTITY loadCntTB IS
END loadCntTB;
ARCHITECTURE testbench OF loadCntTB IS

COMPONENT loadCnt
  PORT (
     data:          IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
     load:          IN   STD_LOGIC;
     clk:           IN   STD_LOGIC;
     rst:           IN   STD_LOGIC;
     q:             OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
END COMPONENT;
```

testbench

```
FILE vectorFile: TEXT OPEN READ_MODE is "vectorfile.txt";

SIGNAL Data:    STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL Load:    STD_LOGIC;
SIGNAL Rst:     STD_LOGIC;
SIGNAL Qout:    STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL Qexpected: STD_LOGIC_VECTOR(7 DOWNTO 0);

SIGNAL  TestClk:        STD_LOGIC := '0';
CONSTANT ClkPeriod:     TIME := 100 ns;
BEGIN

-- Free running test clock
  TestClk <= NOT TestClk AFTER ClkPeriod/2;

-- Instance of design being tested
  u1: loadCnt PORT MAP (Data => Data,
                        load => Load,
                        clk => TestClk,
                        rst => Rst,
                        q => Qout
                       );
```

Reading from file

```
-- File reading and stimulus application

  readVec: PROCESS
    VARIABLE    VectorLine:     LINE;
    VARIABLE    VectorValid:    BOOLEAN;
    VARIABLE    vRst:           STD_LOGIC;
    VARIABLE    vLoad:          STD_LOGIC;
    VARIABLE    vData:         STD_LOGIC_VECTOR(7 DOWNTO 0);
    VARIABLE    vQ:            STD_LOGIC_VECTOR(7 DOWNTO 0);
    VARIABLE    space:          CHARACTER;
```

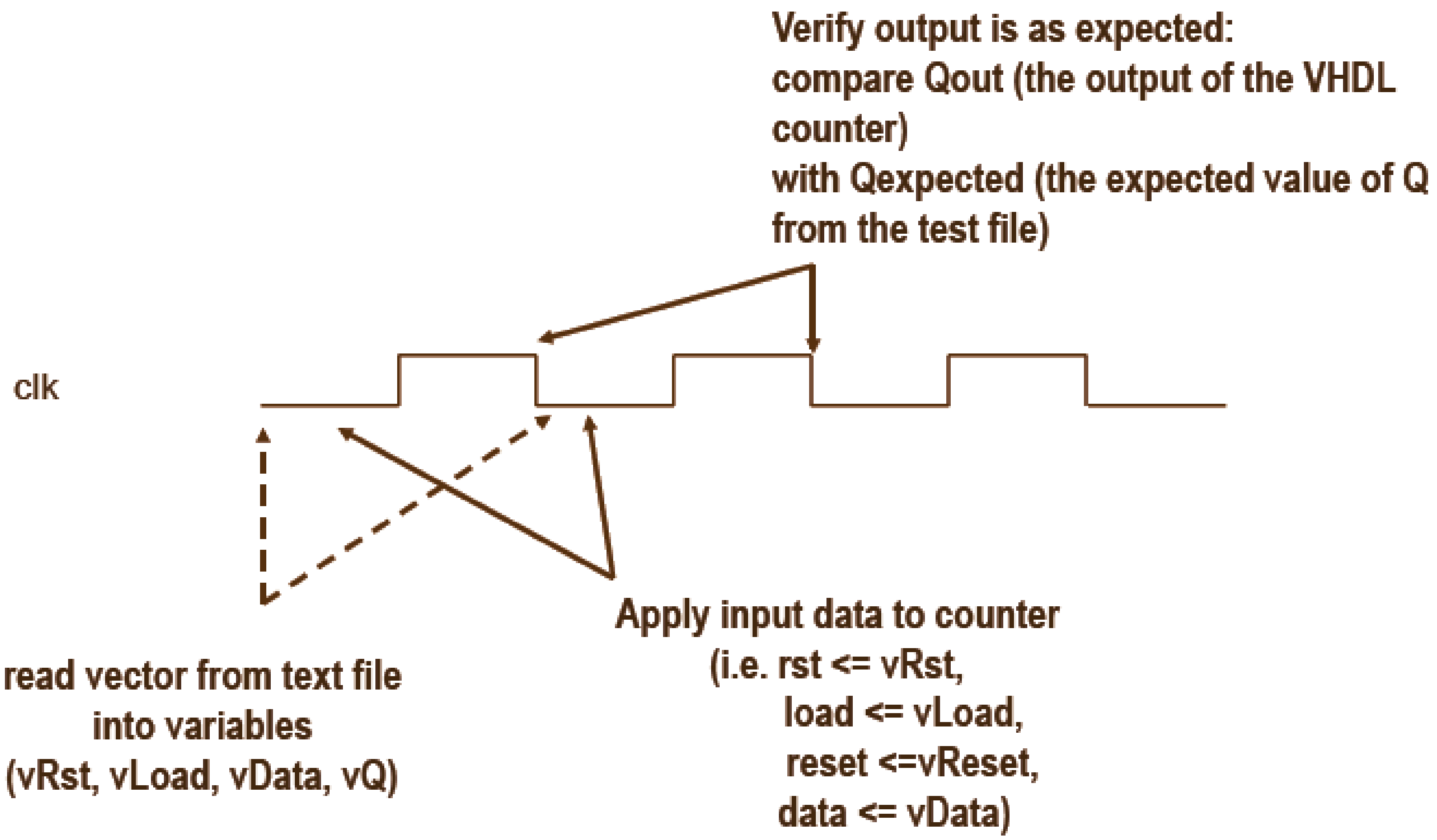
Reading from file- C’ntd

```
BEGIN
   WHILE NOT ENDFILE (vectorFile) LOOP
     readline(vectorFile, VectorLine); -- put file data into line

     read(VectorLine, vRst, good => VectorValid);
     NEXT WHEN NOT VectorValid;
     read(VectorLine, space);
     read(VectorLine, vLoad);
     read(VectorLine, space);
     read(VectorLine, vData);
     read(VectorLine, space);
     read(VectorLine, vQ);

     WAIT FOR ClkPeriod/4;
     Rst <= vRst;
     Load <= vLoad;
     Data <= vData;
     Qexpected <= vQ;

     WAIT FOR (ClkPeriod/4) * 3;
  END LOOP;
```

Comparing the output

```
-- Process to verify outputs
  verify: PROCESS (TestClk)
  variable ErrorMsg: LINE;
  BEGIN
    IF (TestClk'event AND TestClk = '0') THEN
      IF Qout /= Qexpected THEN
        write(ErrorMsg, STRING'("Vector failed "));
        write(ErrorMsg, now);
        writeline(output, ErrorMsg);
      END IF;
    END IF;
  END PROCESS;
END testbench;
```

Test timing


Verify output is as expected:
compare Qout (the output of the VHDL counter)
with Qexpected (the expected value of Q from the test file)
clk
read vector from text file into variables (vRst, vLoad, vData, vQ)
Apply input data to counter (i.e. rst <= vRst,
load <= vLoad,
reset <=vReset,
data <= vData)

Testbench

- Test bench data may be read from file or generated randomly at runtime
- Runtime methods can use the VHDL random generation function (uniform) or using an LFSR
  - Will be introduced soon
- A key question is how a good a benchmark is?
  - How many parts of the design is tested by a set of test input?
- A good way is to calculate the *code coverage* of the testbench

Random generation

- By naturally occurring sources
  - Measure the source value and use as random
  - Time, noise, …
- By an algorithm
  - A computer algorithm generates the numbers
  - Example: $X_{n+1}=(aX_n+b) \bmod m$
    - The initial value of X (i.e. X_0) can be the seeds of the random
    - Used in many programming languages (compiler runtime libraries)
    - Example: For MS Visual C++, a=214013 and b=2531011 and $m=2^{32}$
- LFSR is one the most common algorithmic random generators

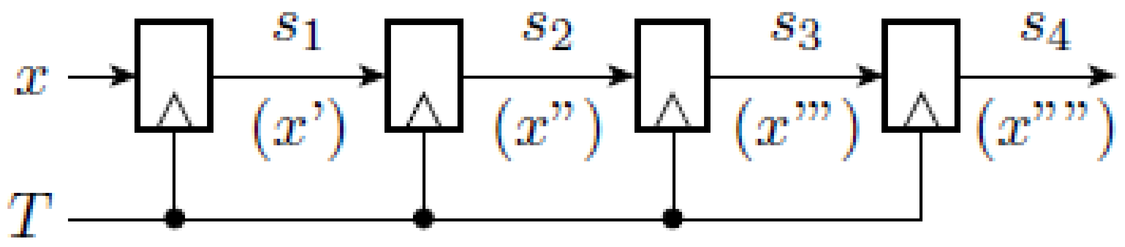
Pseudo random generation

- LFSR: Linear-Feedback Shift Register
- Fast random generation with simple hardware
- Generates a sequence of numbers that approximates the properties of random numbers.
- The sequence is fully deterministic, i.e., it can

  be repeated based on an initial state.
- The period of the sequence may be made very large (typically, 2^n-1, where n is an internal state size)

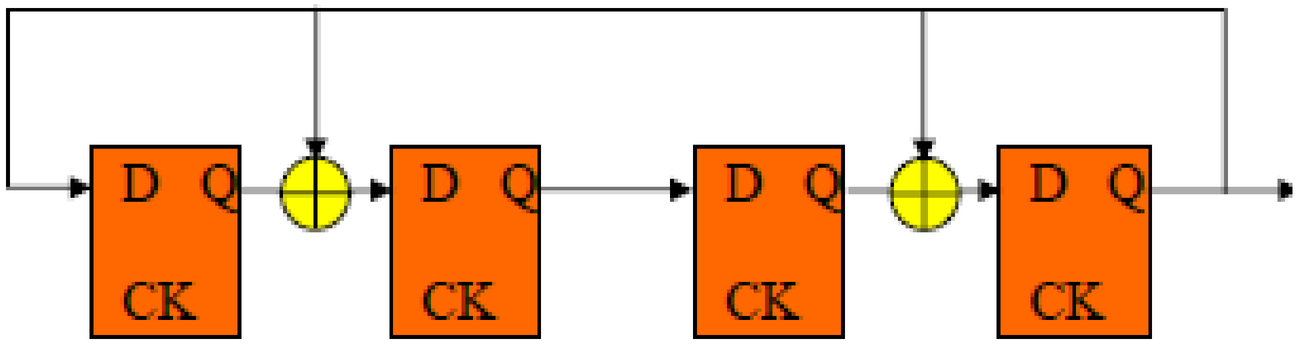
Shift register

- A shift register is a chain of edge triggered flip-flops in which during each shift:
  - Each successor flip-flop takes the value of its predecessor
  - At the begin one bit is added and at the end one bit gets lost



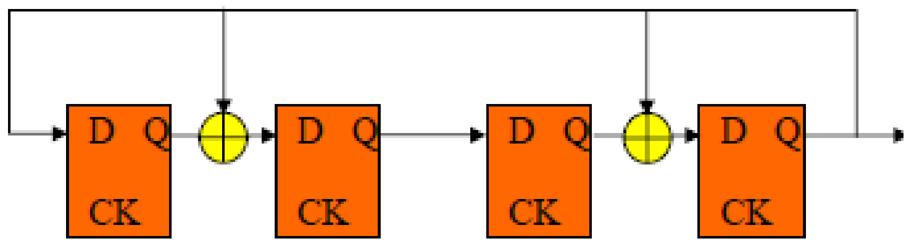
LFSR

- A linear feedback shift register (LFSR) is a shift register, in which in addition, the output value of the last bit is added modulo-2 to selected bit positions
- A modulo-2 addition adds two bits without calculating the carry and is realized by an XOR gate

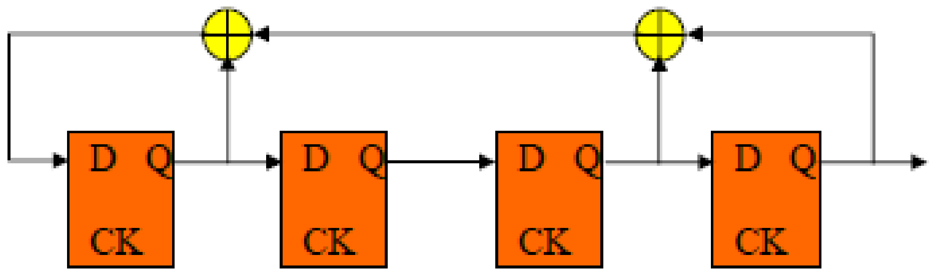


Two types of LFSRs

- Internal



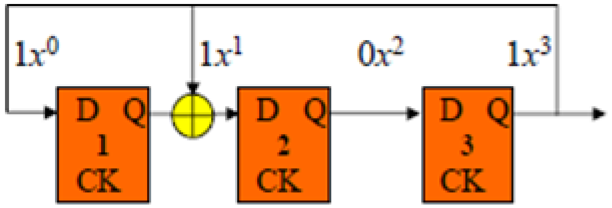
- External



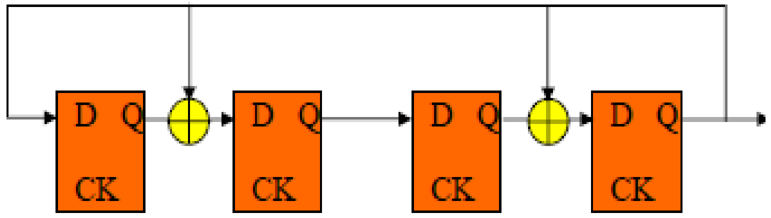
- Characteristic polynomial
  - defined by XOR positions
  - $P(x) = x^4 + x^3 + x + 1$ *in both examples*

LFSR

- Characteristic polynomial of LFSR
- *n = # of FFs = degree of polynomial*
- XOR feedback connection to FF $_i$ $\Leftrightarrow$ *coefficient of* x^i
  - coefficient = 0 if no connection
  - coefficient = 1 if connection
- Two coefficients always included in characteristic polynomial:
  - x^n *(degree of polynomial & primary feedback)*
  - $x^0 = 1$ *(principle input to shift register)*



$$P(x) = x^3 + x + 1$$



$$P(x) = x^4 + x^3 + x + 1$$

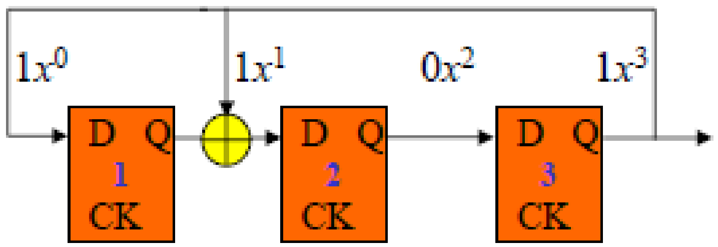
LFSR

- An LFSR generates periodic sequence
  - must start in a non-zero state,
- The maximum-length of an LFSR sequence is 2*n -1*
  - does not generate all 0s pattern (gets stuck in that state)
- The characteristic polynomial of an LFSR generating a maximum-length sequence is a ***primitive polynomial***

Primitive LFSR

- Example: Characteristic polynomial is $P(x) = x^3 + x + 1$

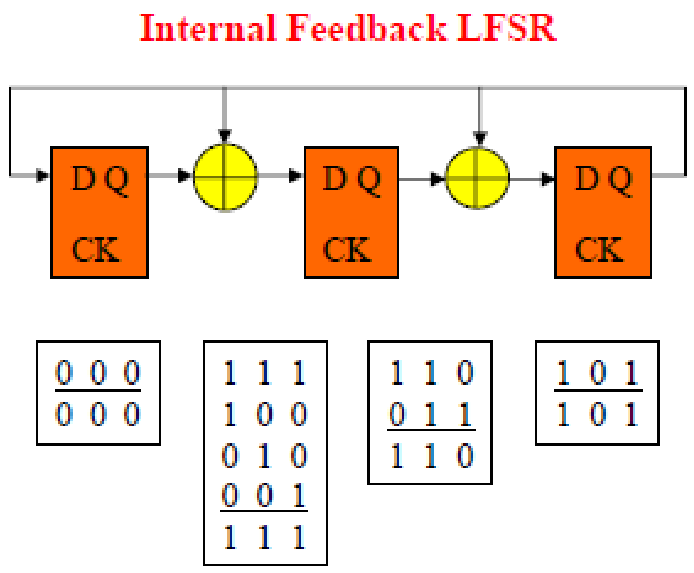
- Beginning at all 1s state
  - 7 clock cycles to repeat
  - maximal length = 2^n-1
  - polynomial is primitive



	1	1	1	1
	1	0	1	2
	1	0	0	3
	0	1	0	4
	0	0	1	5
	1	1	0	6
1)	0	1	1	7
	1	1	1	

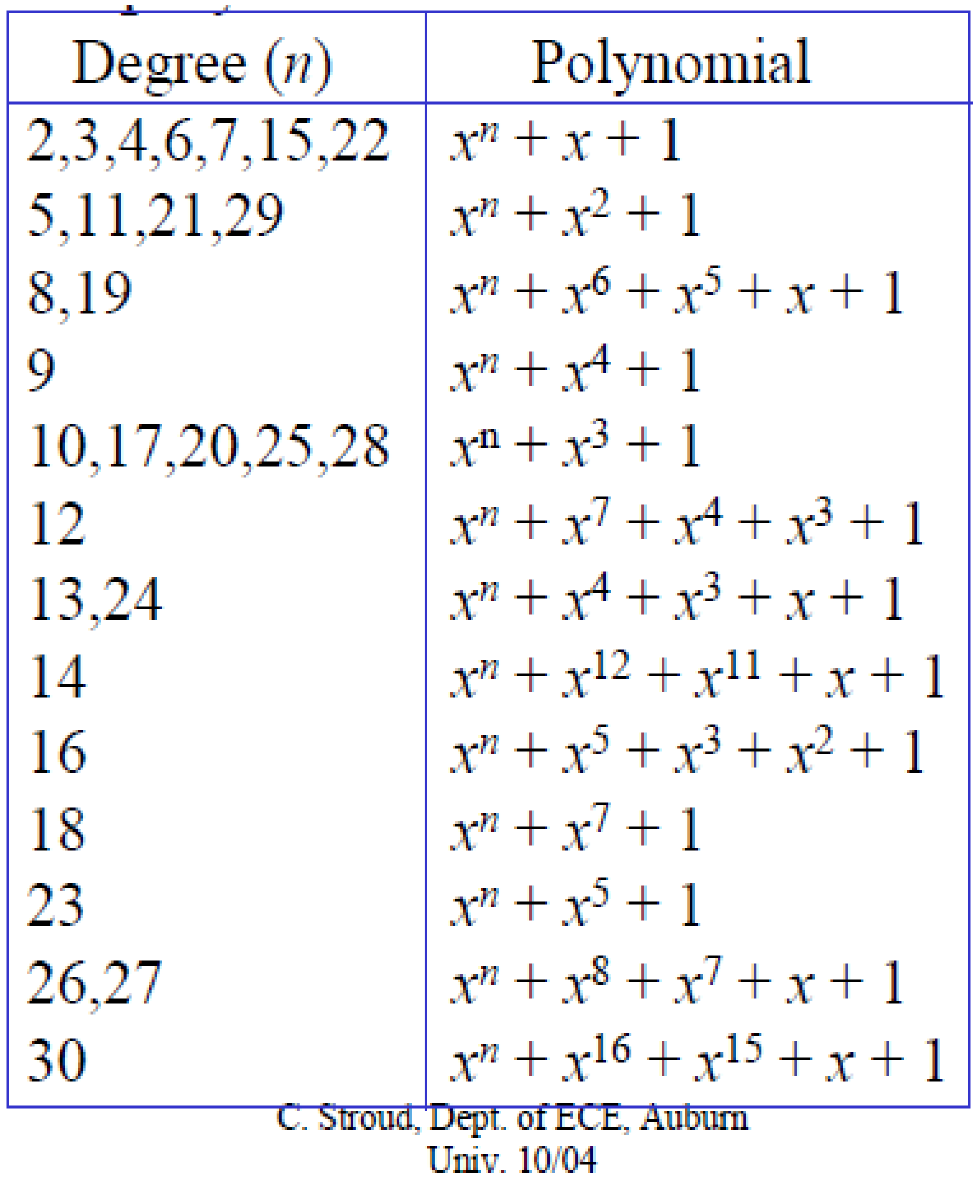
Primitive LFSR

- Non-primitive polynomials produce sequences < 2^n-*1*
- Example of non-primitive polynomial:



LFSR

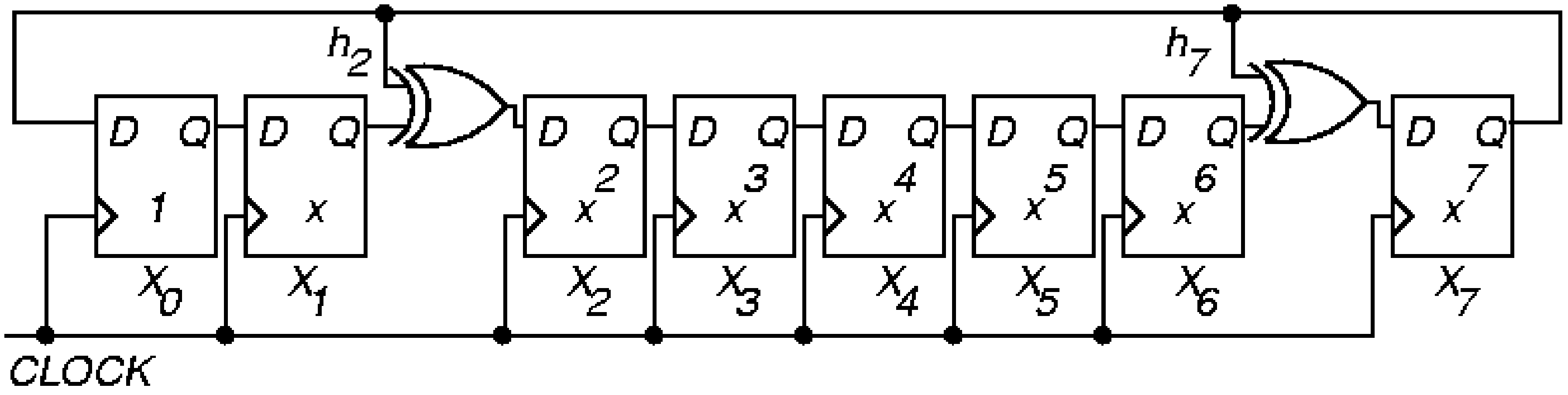
- Primitive polynomials with minimum # of XORs



Degree (n)	Polynomial
2,3,4,6,7,15,22	$x^n + x + 1$
5,11,21,29	$x^n + x^2 + 1$
8,19	$x^n + x^6 + x^5 + x + 1$
9	$x^n + x^4 + 1$
10,17,20,25,28	$x^n + x^3 + 1$
12	$x^n + x^7 + x^4 + x^3 + 1$
13,24	$x^n + x^4 + x^3 + x + 1$
14	$x^n + x^{12} + x^{11} + x + 1$
16	$x^n + x^5 + x^3 + x^2 + 1$
18	$x^n + x^7 + 1$
23	$x^n + x^5 + 1$
26,27	$x^n + x^8 + x^7 + x + 1$
30	$x^n + x^{16} + x^{15} + x + 1$

C. Stroud, Dept. of ECE, Auburn Univ. 10/04

Example Modular LFSR



- $f(x) = 1 + x^2 + x^7 + x^8$
- Read LFSR tap coefficients from left to right

Code coverage

- Coverage referrers to the percent of the design that has been exercised by a testbench
- For simulation, a Test bench applies data to a DUT, test data generated by the Test bench reach certain parts of a circuit and may never create activities in other parts
- A good Test bench uses a minimum set of data to cover testing of the largest possible portion of the circuit
- Several methods to calculate code coverage of a testbench

Code coverage

- ModelSim have a powerful integrated tool that calculates the code coverage
  - Statement coverage
  - Condition coverage
  - Toggle coverage
  - ….
- See the ModelSim code coverage document on the course webpage

Code coverage

Statement coverage

- Statement coverage is a measure of the number of executable statements within the model that have been executed during the simulation run.

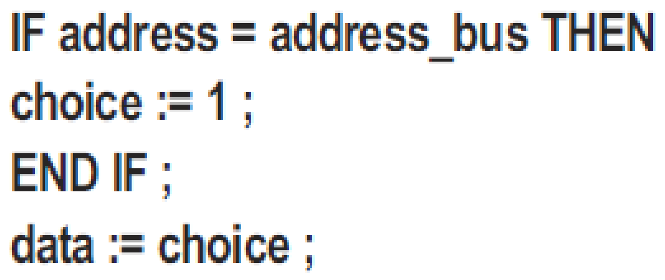
Toggle coverage

- Counts changes of state on specified nodes, in other words different values a signal takes
  - Maybe a signal is activated many times by a testbench, but always is assigned the same values
  - A good test bench should assign many different values to verify

Code coverage

Branch coverage

- Branch coverage, sometimes referred to as decision coverage, counts the execution of different paths of conditional expressions and case statements
- **Example:** If the Test bench forced address to always equal *address_bus*, the statement coverage for the below code is 100%, since every statement would be executed.
- However, branch coverage would only be 50%, since the FALSE branch would not have been taken during simulation.



```
IF address = address_bus THEN
choice := 1 ;
END IF ;
data := choice ;
```

Code coverage

- Code coverage gives us a metric to decide how a Test bench is good and comprehensive
- However, a good coverage does not guarantee that all possible errors can be found by the testbench

- TEST
- VERIFICATION

Test

- Must ensure that all transistors and links work correctly
- How? Like testbenches, we should feed input test vectors to input ports and observe output ports to see if
  - Exhaustive or random vectors
- Uses special expensive machines to test ICs called Automatic Test Equipment

Test

- Exhaustive: use all possible input combinations
  - Full coverage but test time is prohibitive
  - 2^n vectors for an n-input circuit
- Random: a set of selected input vectors
  - May not find all faults
- Solution: automatic test pattern generation (ATPG)
  - Find the minimum subset of vectors that can cove all possible defects
  - Currently, the D-Algorithm and its descendants like PODEM are the most widely used methods for test vector generation

Fault model

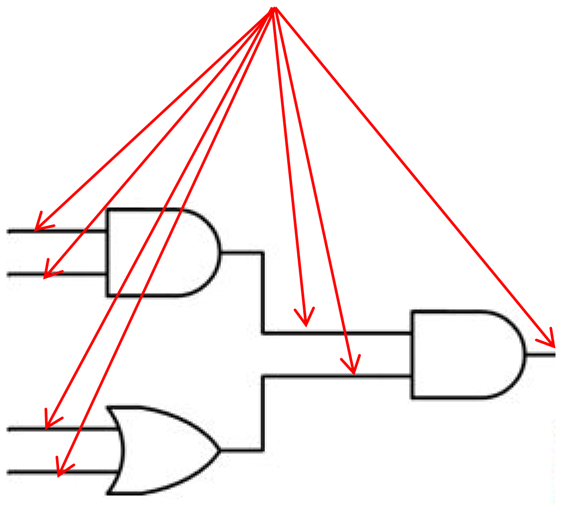
- Each transistor may have many faults
  - Some parts that must be connected may be open
  - Some parts that must not be connected may be connected (shortcut)
  - The channel may be always on
  - The threshold voltage may be so high that never lets the transistor turn on
  - …..
- How can we test them all?
  - Modelling the faults
- 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
- Advantages

Fault model

- Many different faults have the same result at the gate level
  - Example: Short channel, low threshold voltage, unwanted connection to VDD and many other defects make the output of a gate to be always '1' regardless of its inputs
- Two famous fault models that reflect the behaviour of many defects:
  - Stuck-at-1 (sa1): the gate output is faulty and is always 1
  - Stuck-at-0 (sa0): the gate output is faulty and is always 0
- There are other models that shortcut or stuck-at-open, but have less coverage than sa0 and sa1

Test

- In testing a digital circuit, we must make sure that no signal (input and output of gates) has the sa0 or sa1 faults
- A set of test vectors to test all signals for sa0 and sa1
- Signals to test: circuit inputs, circuit outputs, the output of each gate, the input(s) of each gate



Test pattern generation

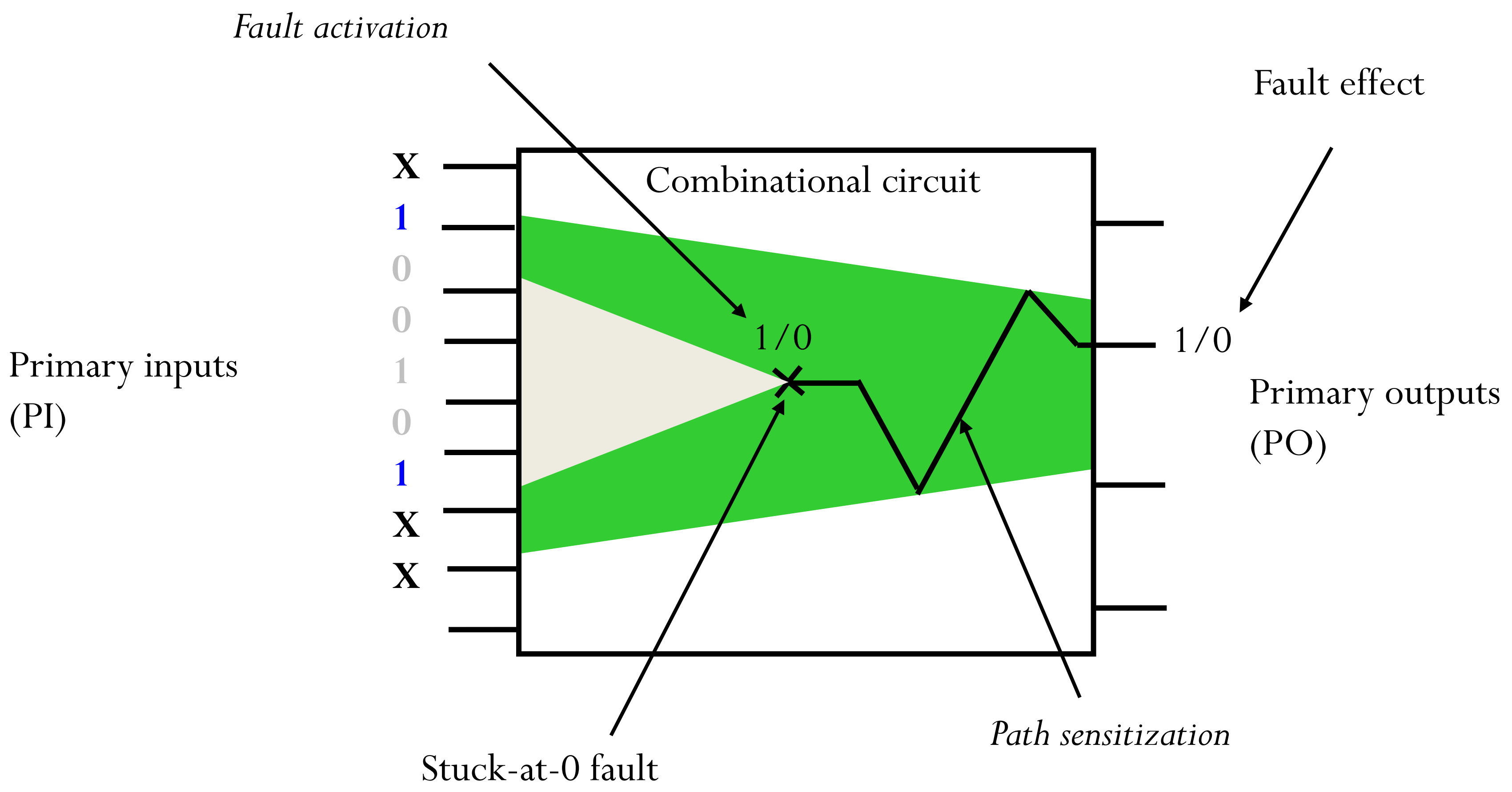
- How to find the minimum set of vectors to test all signals for sa0 and sa1? By D-algorithm
- Uses the D notation
- D is a state: it implies that in the good circuit the signal value is 1, but in the faulted circuit it is 0
- D' is the reverse: a good signal is 0, but the faulty version is 1

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

D algorithm

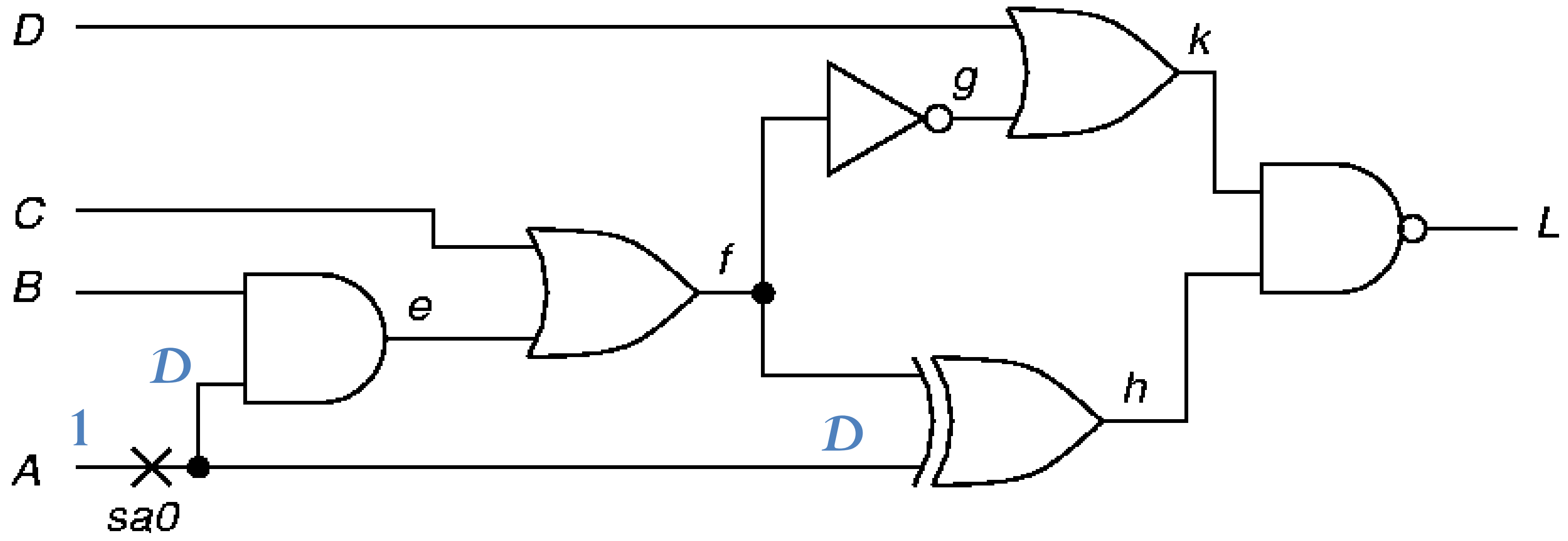
- Check all circuit nodes for sa0:
  - Fault activation: Set the node value to D (make its value 1 to detect if it is stuck at 0 or not)
  - Path sensitization: Set the other signal values to propagate D to the circuit outputs
  - Justification: Set the circuit inputs to generate all required intermediate values

What is a Test?



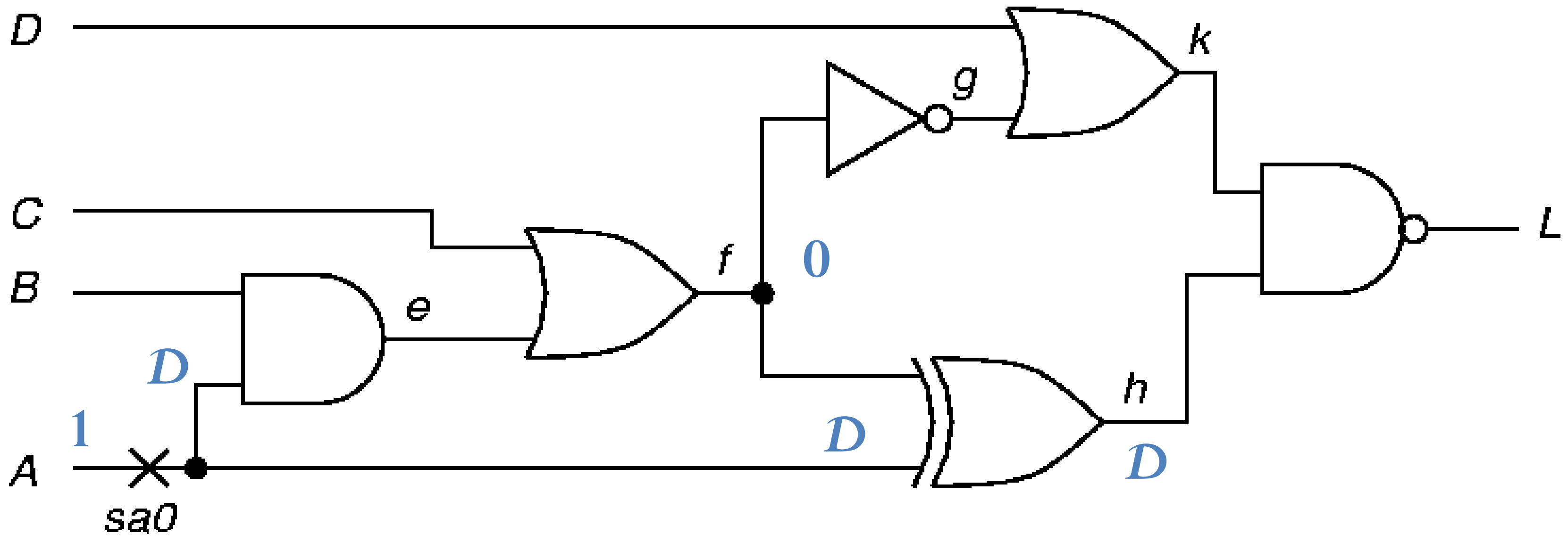
Example: Fault *A* sa0

- **Step 1 – *Fault activation* – Set *A = 1***



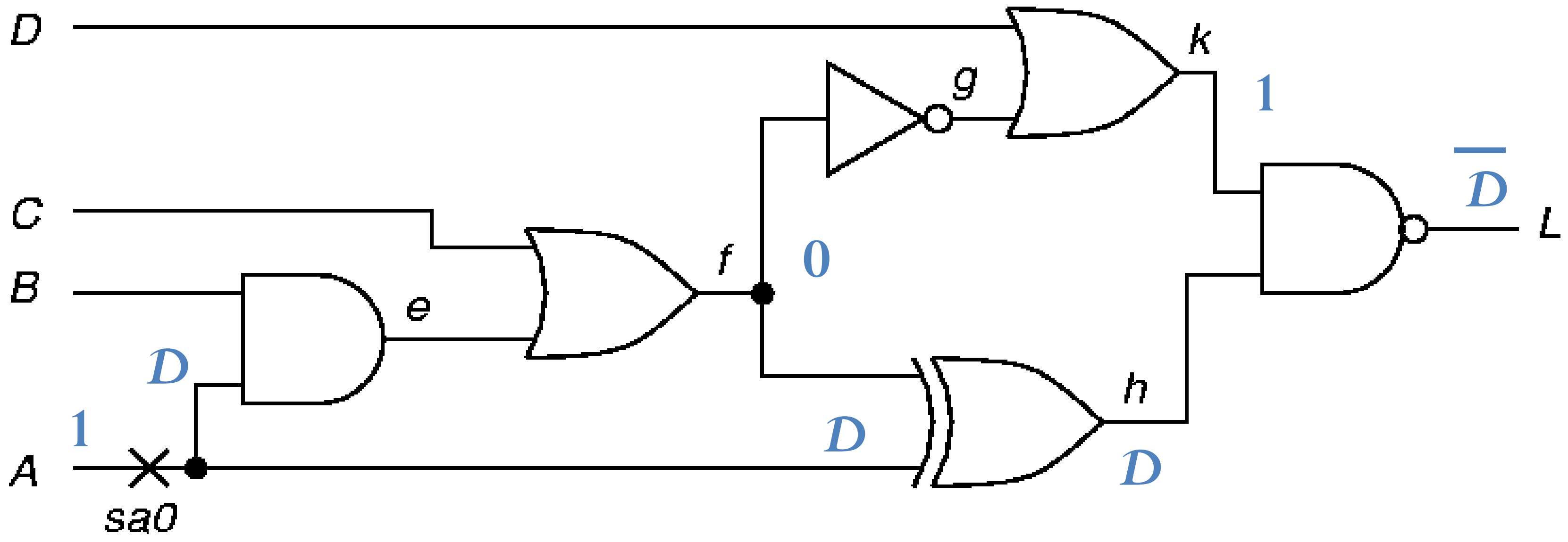
Example Continued

- **Step 2 – *D-Drive* – Set *f* = 0**



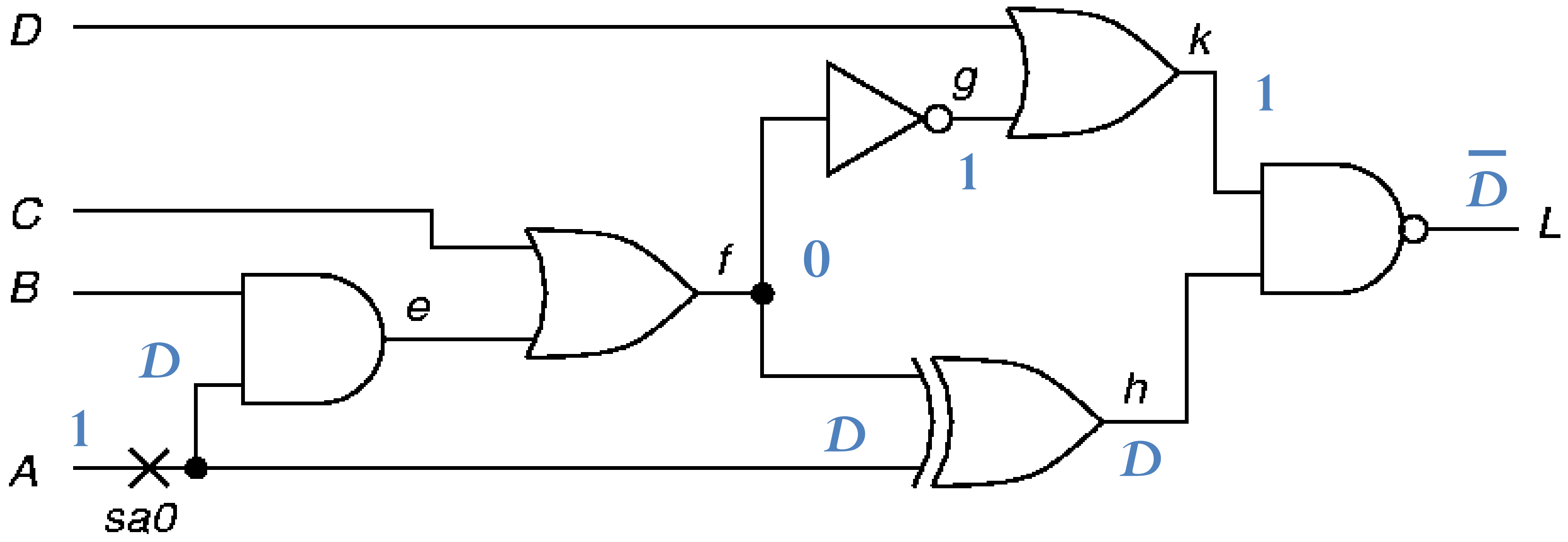
Example Continued

- **Step 3 – *D-Drive* – Set *k* = 1**



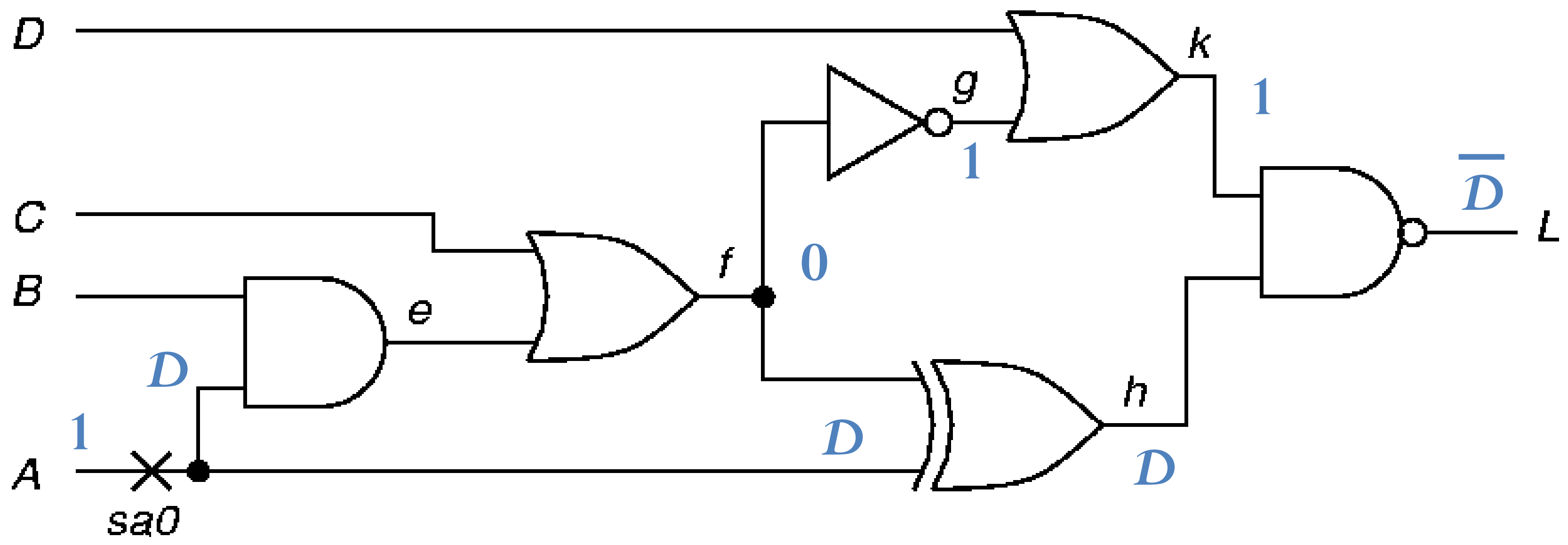
Example Continued

- **Step 4 – *Consistency* – Set *g* = 1**



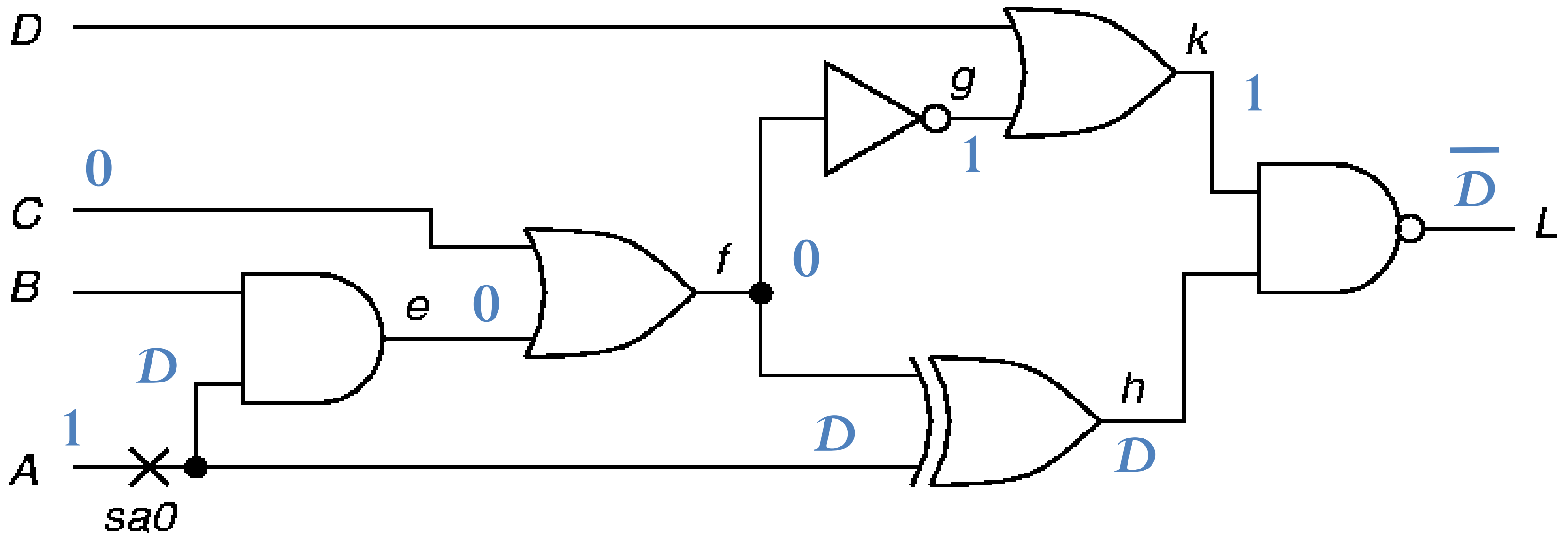
Example Continued

- **Step 5 – *Consistency* – *f* = 0 Already set**



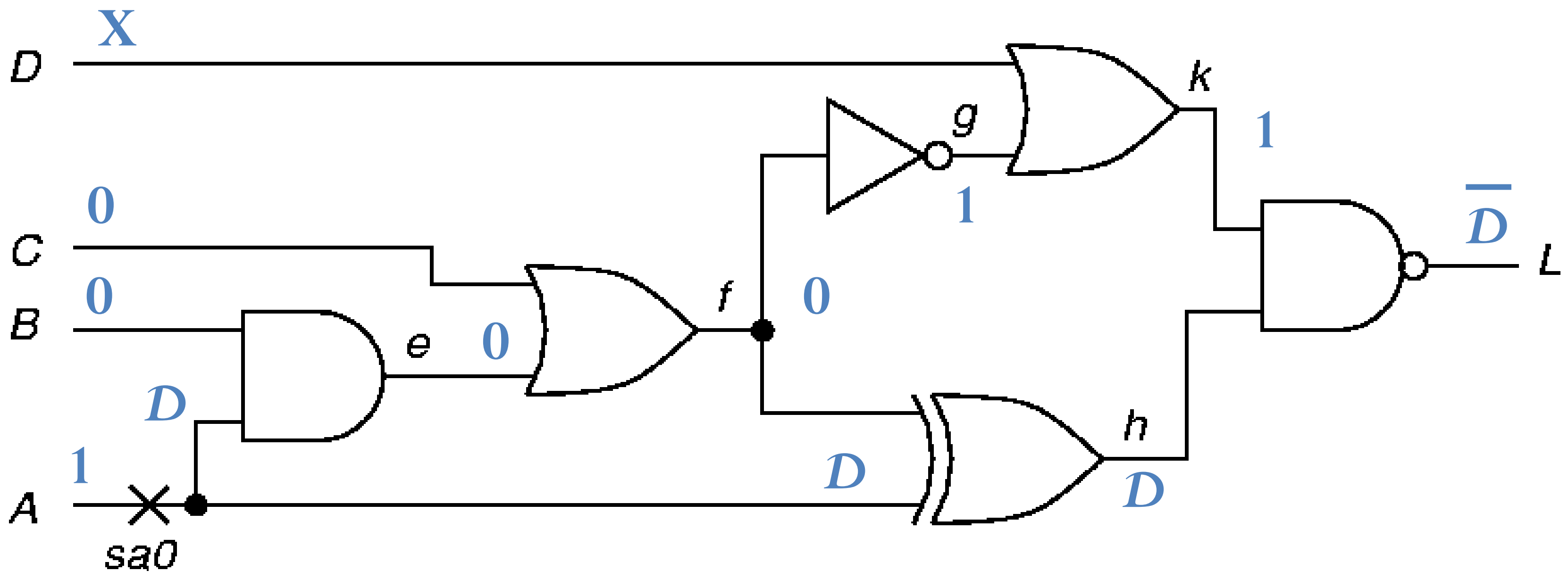
Example Continued

- **Step 6 – *Consistency* – Set *c* = 0, Set *e* = 0**

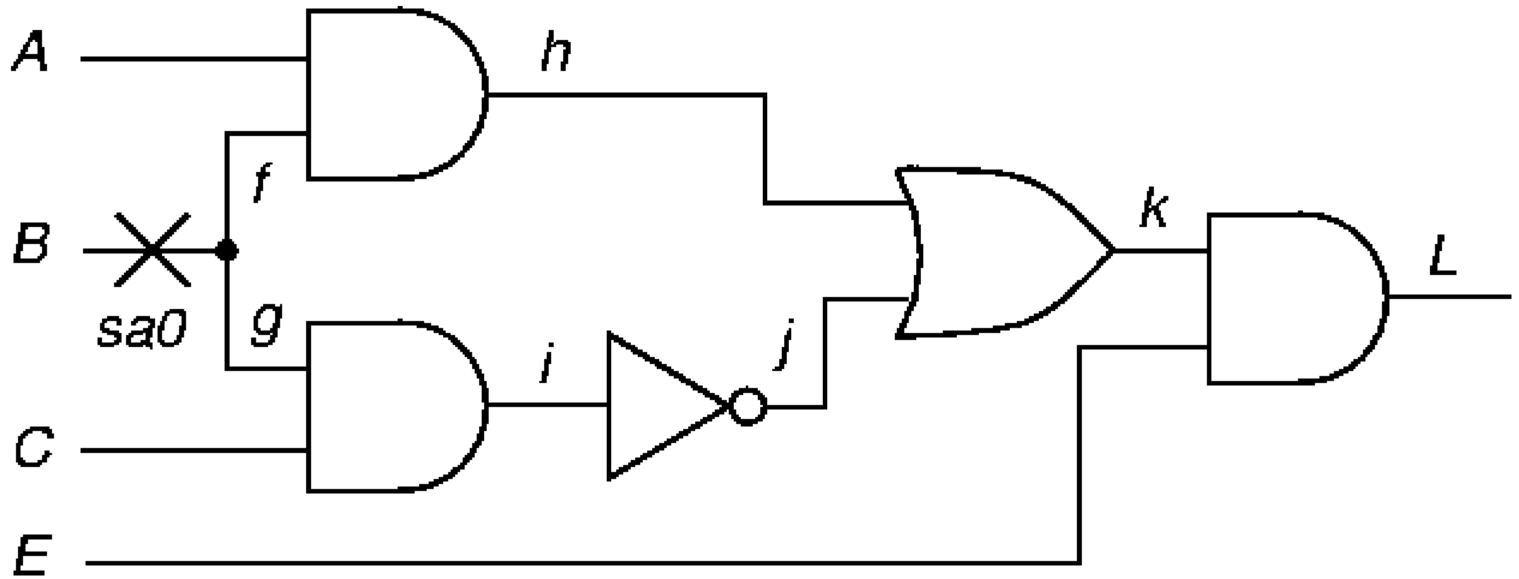


Example: Test Found

- **Step 7 – *Consistency* – Set *B* = 0**
- **Test: *A* = 1, *B* = 0, *C* = 0, *D* = X**

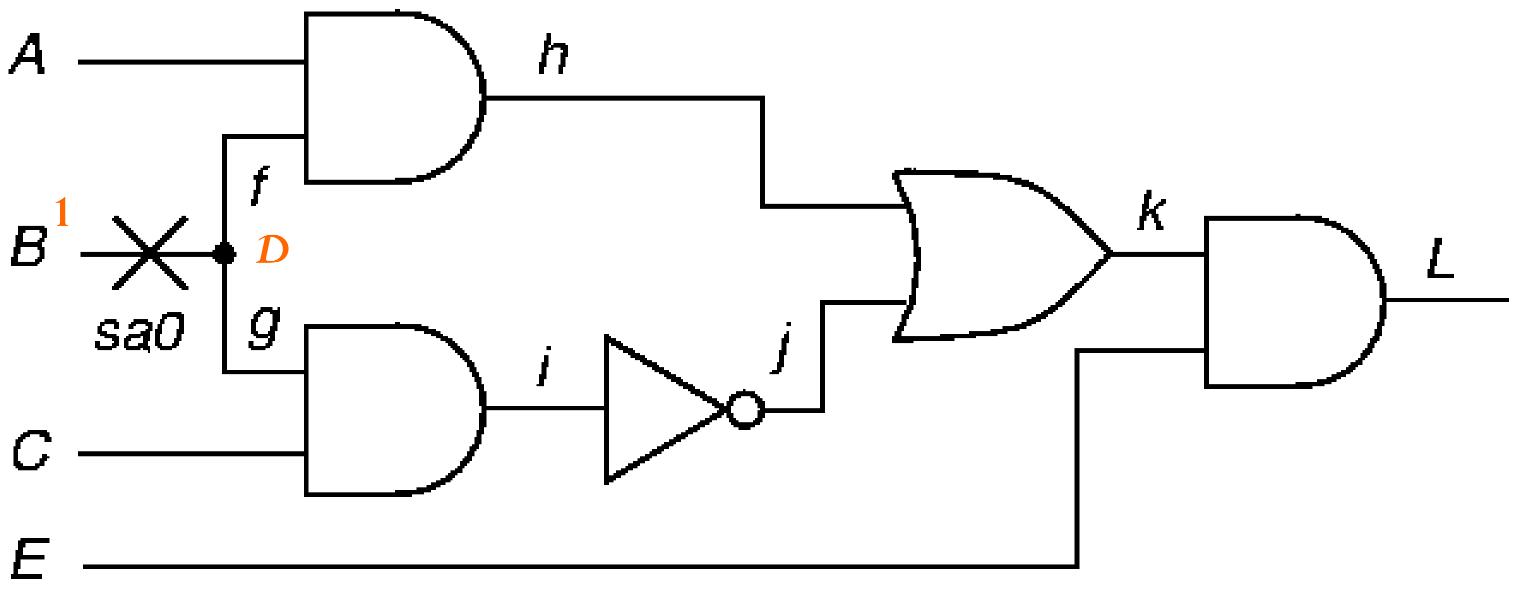


Another example



An ATPG Example

1 **Fault activation**
2 **Path sensitization**
3 **Line justification**

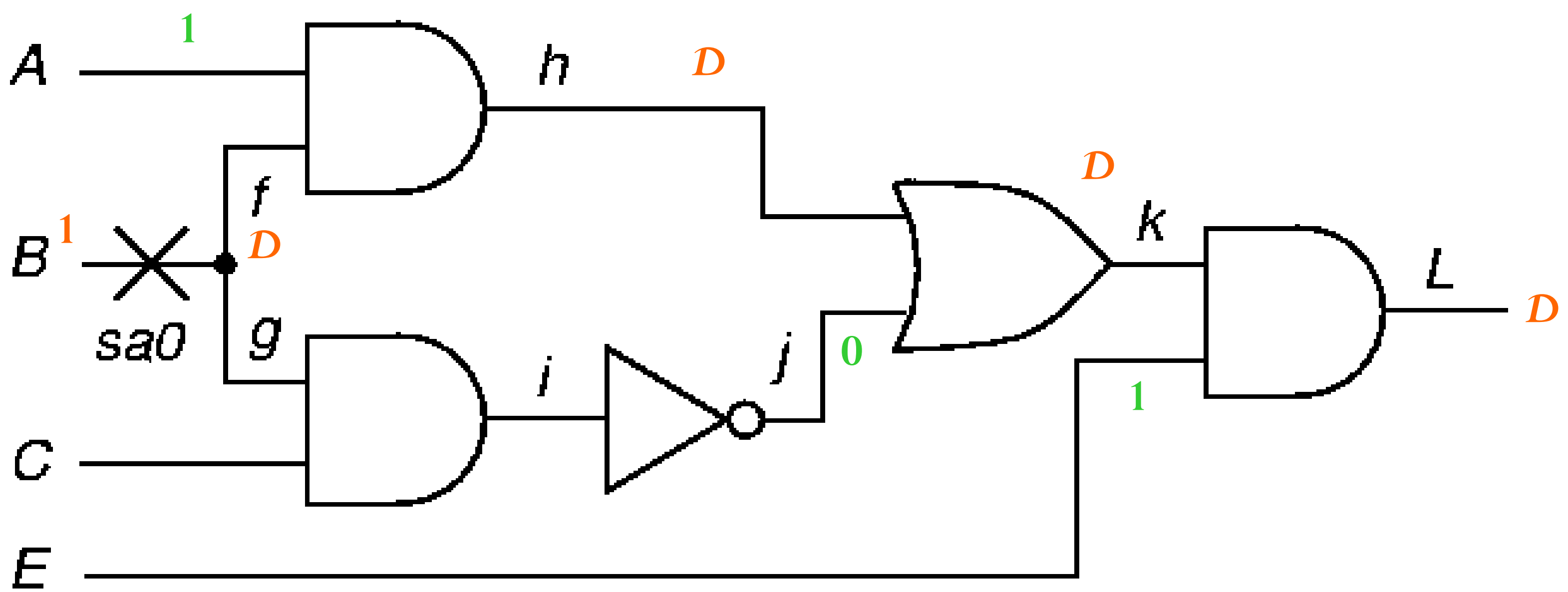


ATPG Example (Cont.)

1 Fault activation

2 **Path sensitization**

3 Line justification

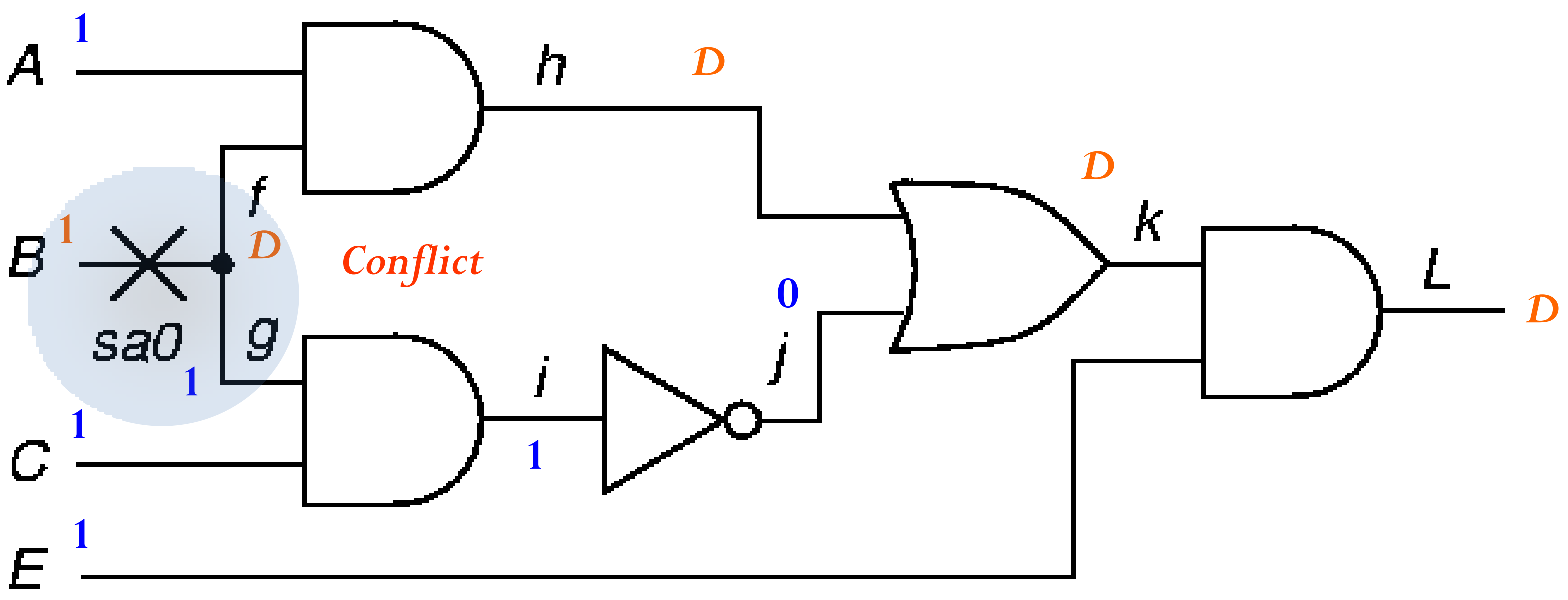


ATPG Example (Cont.)

1 Fault activation

2 Path sensitization

3 **Line justification**



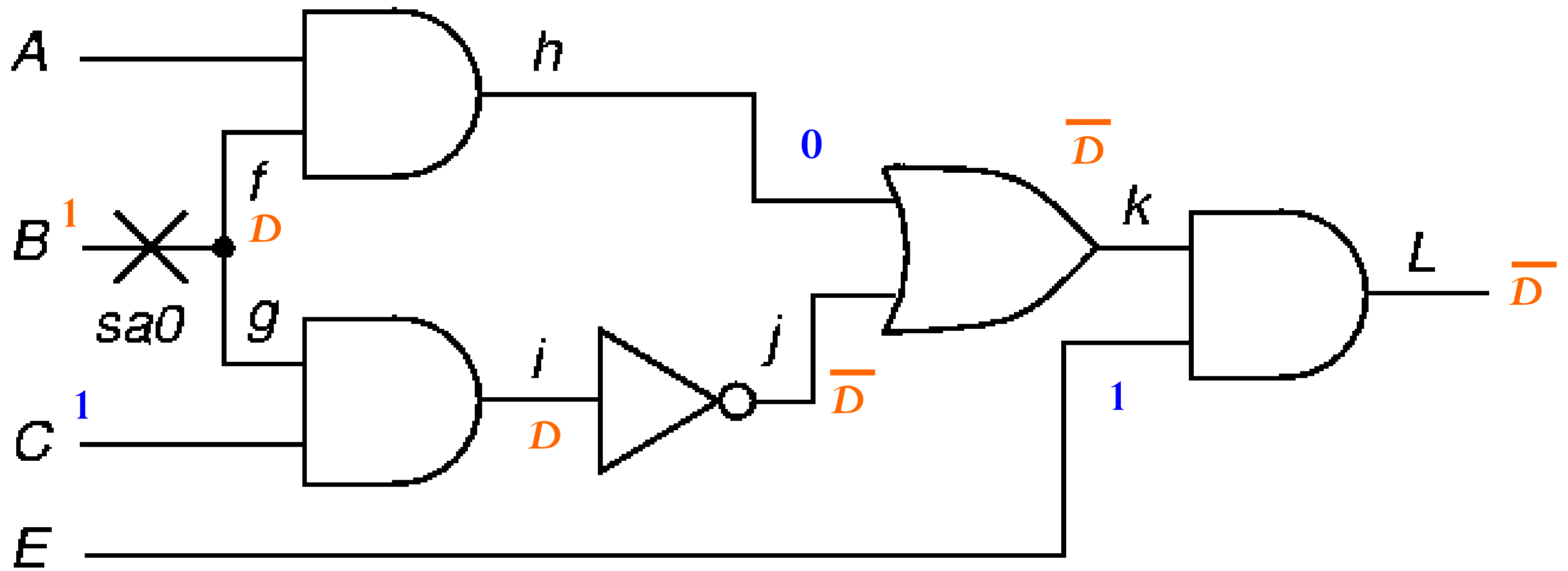
ATPG Example (Cont.)

1 Fault activation

2 **Path sensitization**

3 **Line justification**

Backtrack

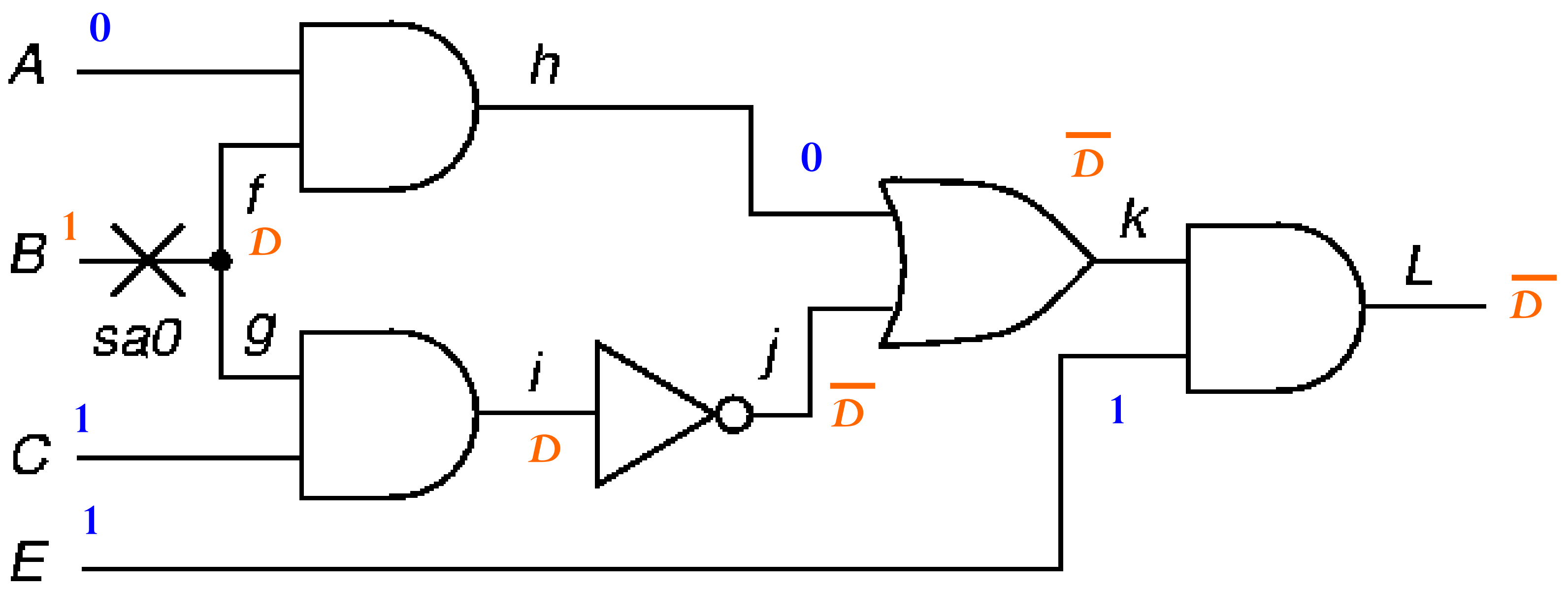


ATPG Example (Cont.)

1 Fault activation

2 Path sensitization

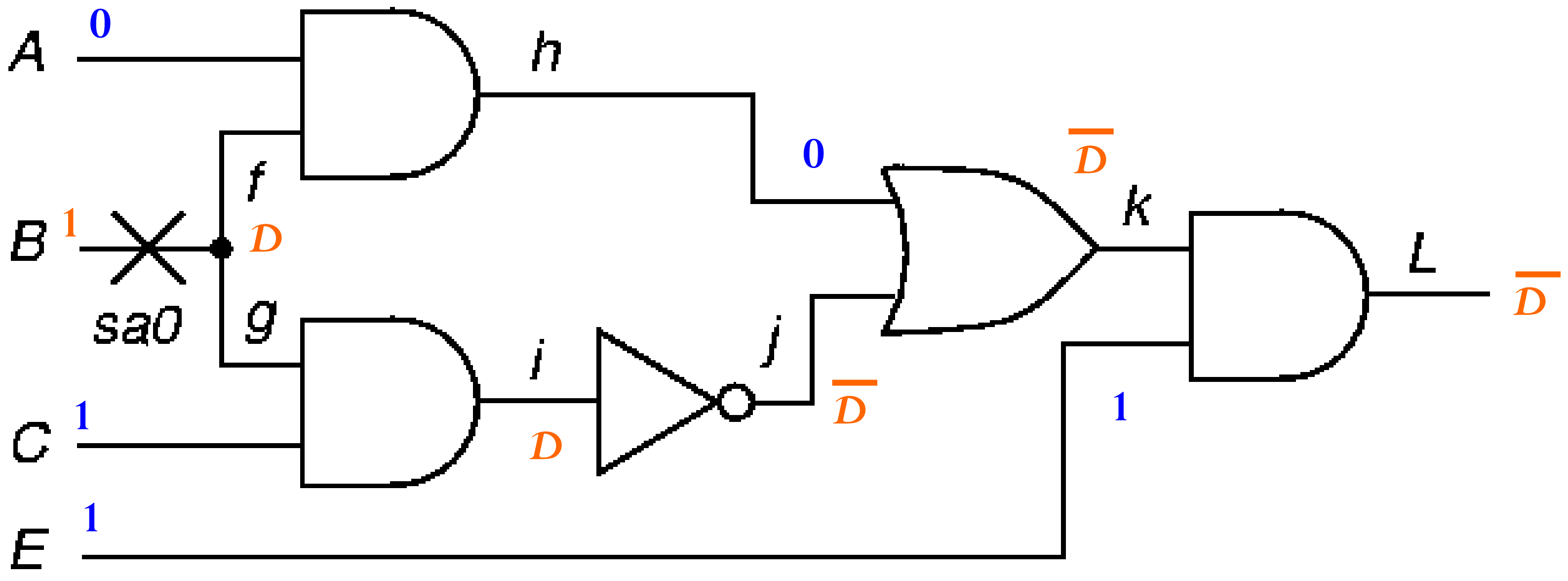
3 **Line justification**



Test found

ATPG Example (Cont.)

- Vector 0111 can detect
  - sa0 at B and i
  - sa1 at j and k and L
- If we apply 0111 to inputs and got D' (i.e. the output was 0) how to detect which node is faulty?



D- algorithm’s application

- D algorithm just detect faults
- It sometimes cannot say the fault location
- Useful to see if the entire circuit is functional or not
  - For all nodes make a vector to detect sa0 and one vector to detect sa1
  - Each vector may cover other faults
- For location, one can use other methods