

Test Generation and Design for Test

Using Mentor Graphics CAD Tools

Mentor Graphics CAD Tool Suites

- IC/SoC design flow¹
- DFT/BIST/ATPG design flow¹
- FPGA design flow^{2,3}
- PCB design flow²
- Digital/analog/mixed-signal modeling & simulation^{1,2}
- ASIC/FPGA synthesis^{1,2}
- Vendor-provided (Xilinx,Altera,etc.) back end tools²

1. User-setup selection: [*eda/mentor/ICFlow2006.1*](#)
2. User-setup selection: [*eda/mentor/EN2002.3*](#)
3. User-setup selection: [*eda/mentor/FPGA*](#)

Mentor Graphics CAD Tools

(select “eda/mentor” in user-setup on the Sun network*)

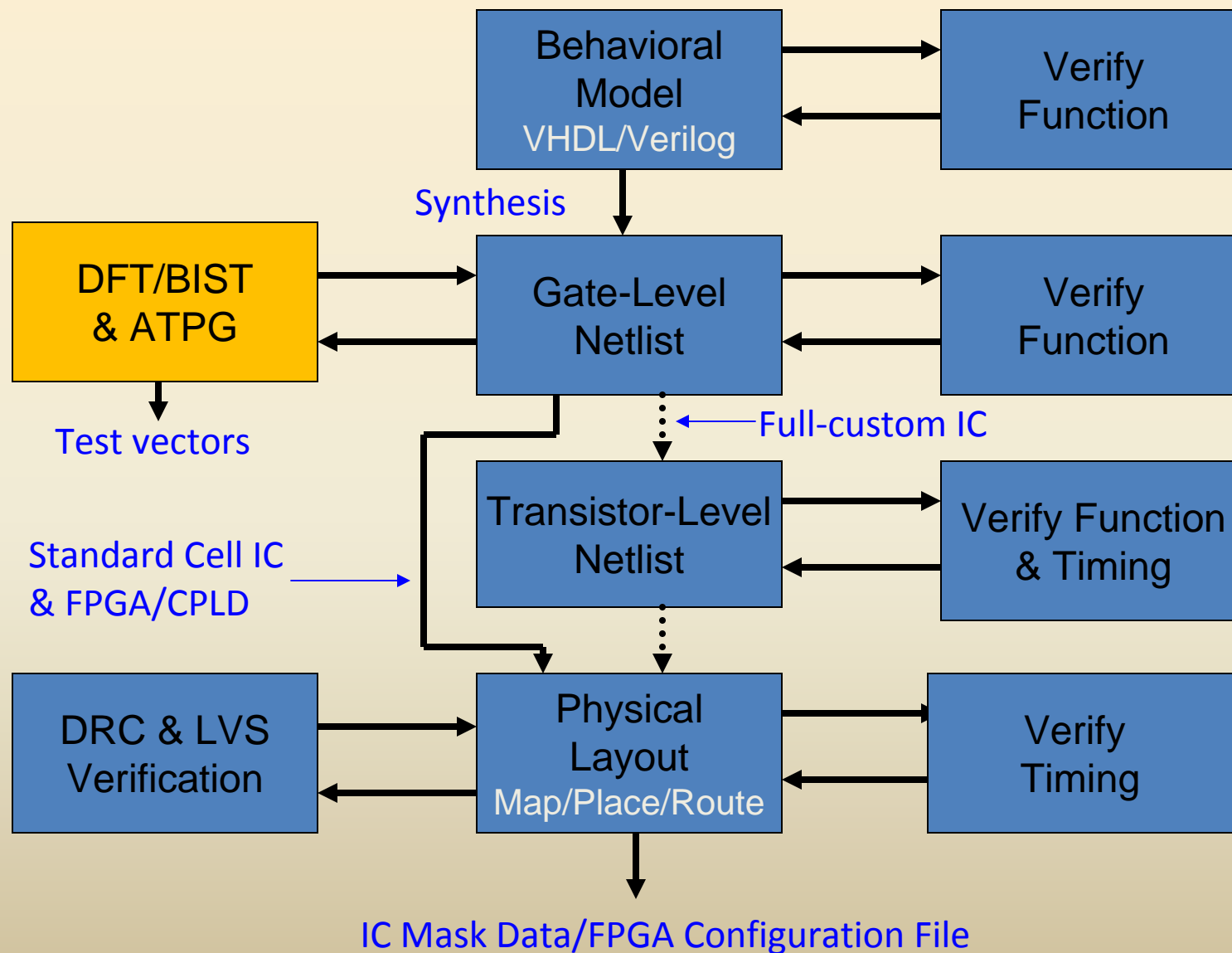
- ICFlow2006.1 – *For custom & standard cell IC designs*
 - IC flow tools (*Design Architect-IC, IC Station, Calibre*)
 - Digital/analog/mixed simulation (*Modelsim, ADVance MS, Eldo, MachTA*)
 - HDL Synthesis (*Leonardo*)
 - ATPG/DFT/BIST tools (*DFT Advisor, Flextest, Fastscan*)
 - Limited access to Quicksim II (*some technologies*)
- EN2002u3 – *For FPGA “front end” design & printed circuit boards*
 - Design Architect, Quicksim II, Quicksim Pro (*Schematic/Simulation*)
 - ModelSim & Leonardo (*HDL Simulation/Synthesis*)
 - Xilinx ISE & Altera “Quartus” tools (*Back end design*)
- FPGA (*FPGA Advantage, Modelsim, Leonardo*)

**Only one of the above three groups may be selected at a time*

Mentor Graphics ASIC Design Kit (ADK)

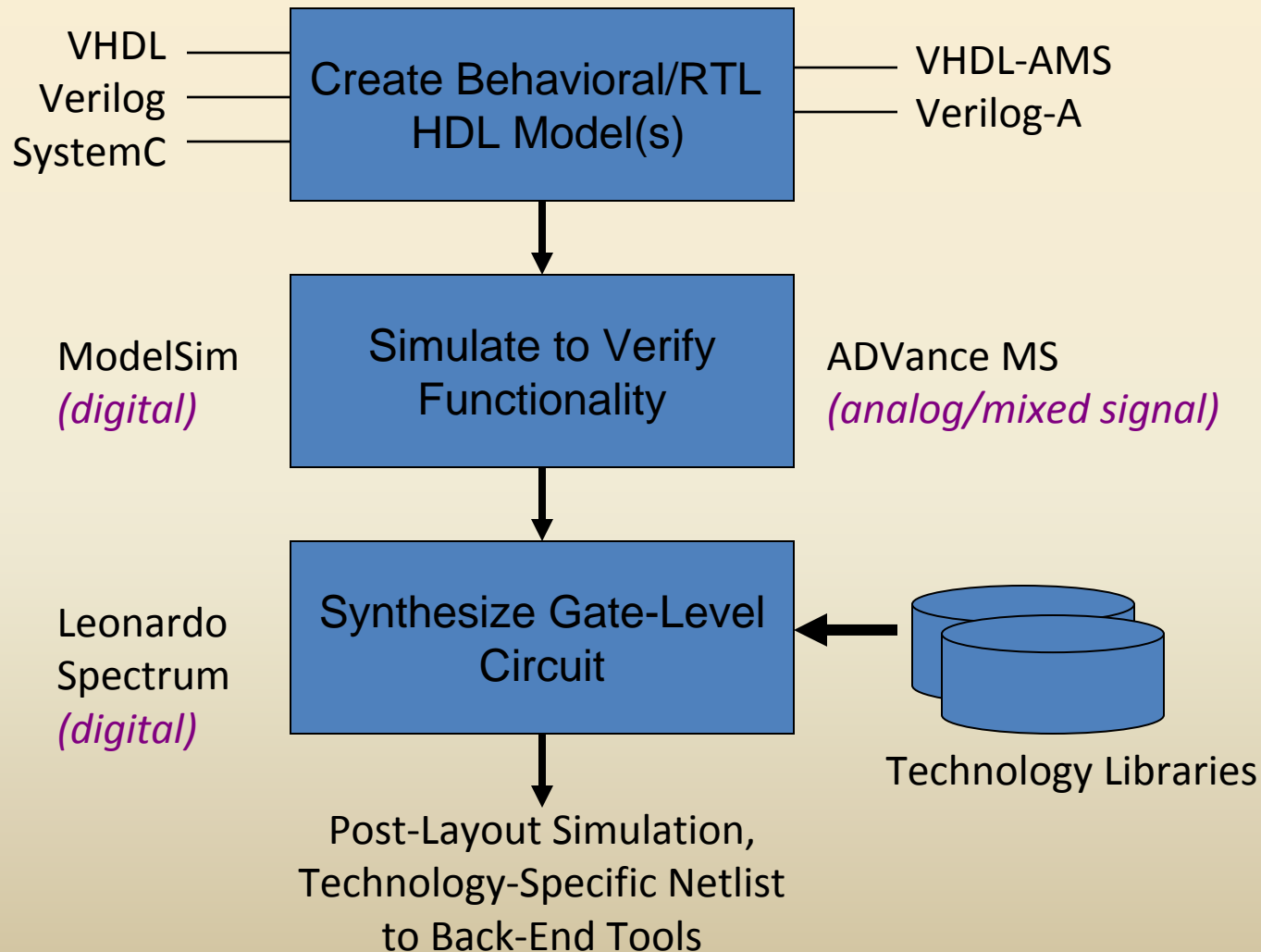
- Technology files & standard cell libraries
 - AMI: ami12, ami05 (1.2, 0.5 μm)
 - TSMC: tsmc035, tsmc025, tsmc018 (0.35, 0.25, 0.18 μm)
- IC flow & DFT tool support files:
 - Simulation
 - VHDL/Verilog/Mixed-Signal models (*Modelsim/ADVance MS*)
 - Analog (SPICE) models (*Eldo/Accusim*)
 - Post-layout timing (*Mach TA*)
 - Digital schematic (*Quicksim II, Quicksim Pro*) (exc. tsmc025, tsmc018)
 - Synthesis to standard cells (*LeonardoSpectrum*)
 - Design for test & ATPG (*DFT Advisor, Flextest/Fastscan*)
 - Schematic capture (*Design Architect-IC*)
 - IC physical design (standard cell & custom)
 - Floorplan, place & route (*IC Station*)
 - Design rule check, layout vs schematic, parameter extraction (*Calibre*)

ASIC Design Flow



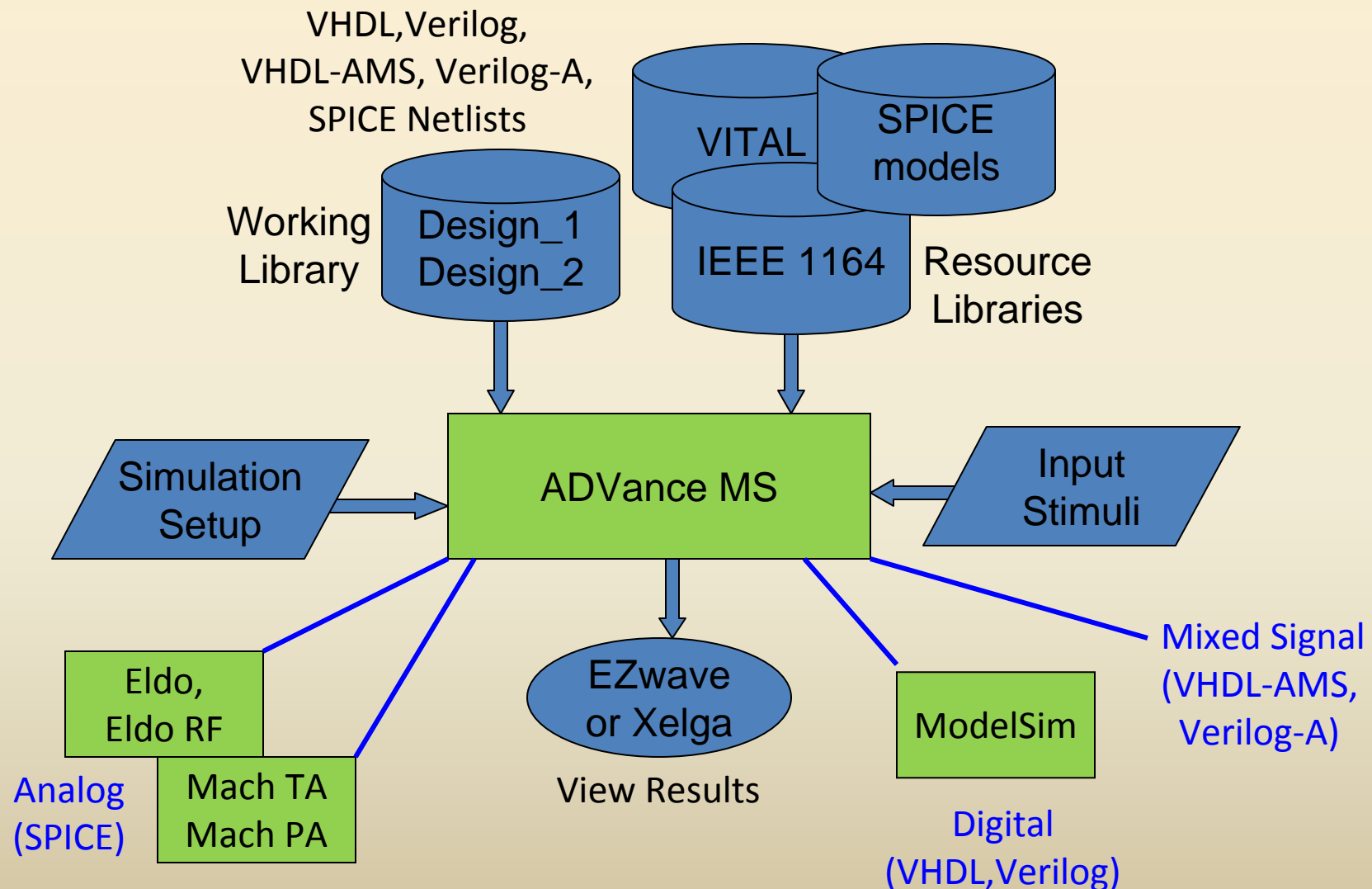
Behavioral Design & Verification

(mostly technology-independent)

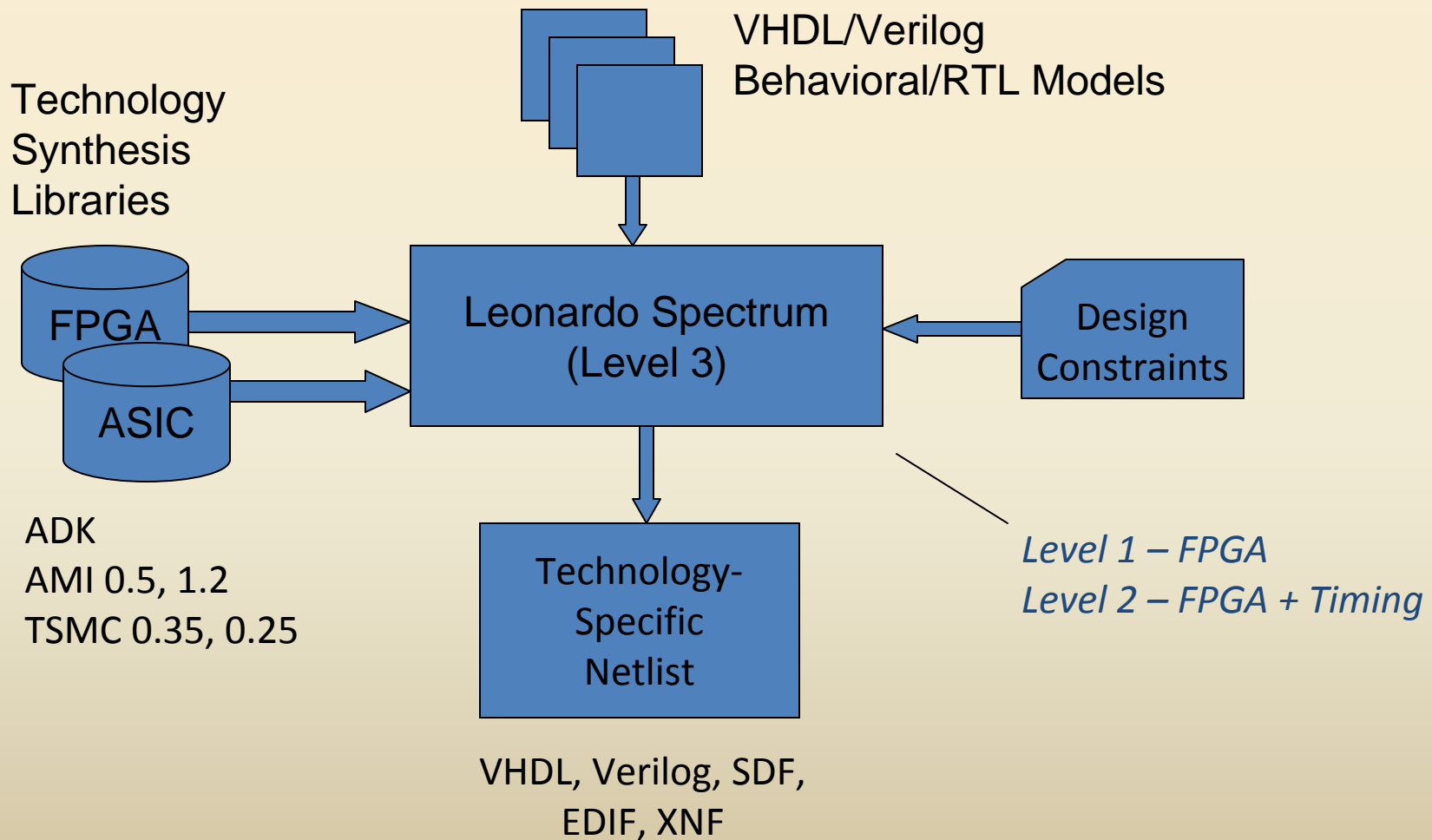


ADVance MS

Digital, Analog, Mixed-Signal Simulation



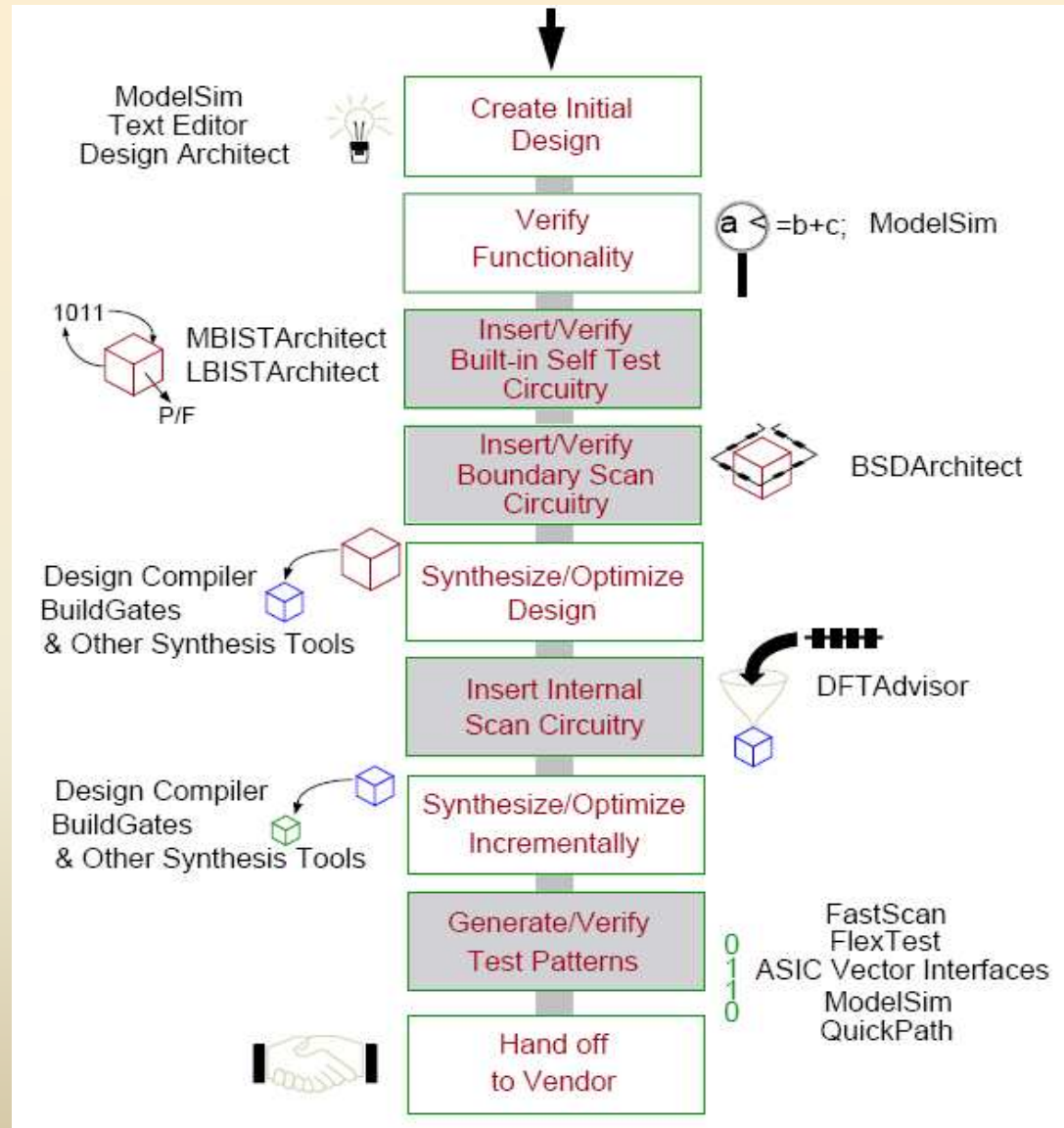
Automated Synthesis with Leonardo Spectrum



Design for test & test generation

- Consider test during the **design** phase
 - Test design more difficult after design frozen
- Basic steps:
 - Design for test (DFT) – insert test points, scan chains, etc. to improve testability
 - Insert built-in self-test (BIST) circuits
 - Generate test patterns (ATPG)
 - Determine fault coverage (Fault Simulation)

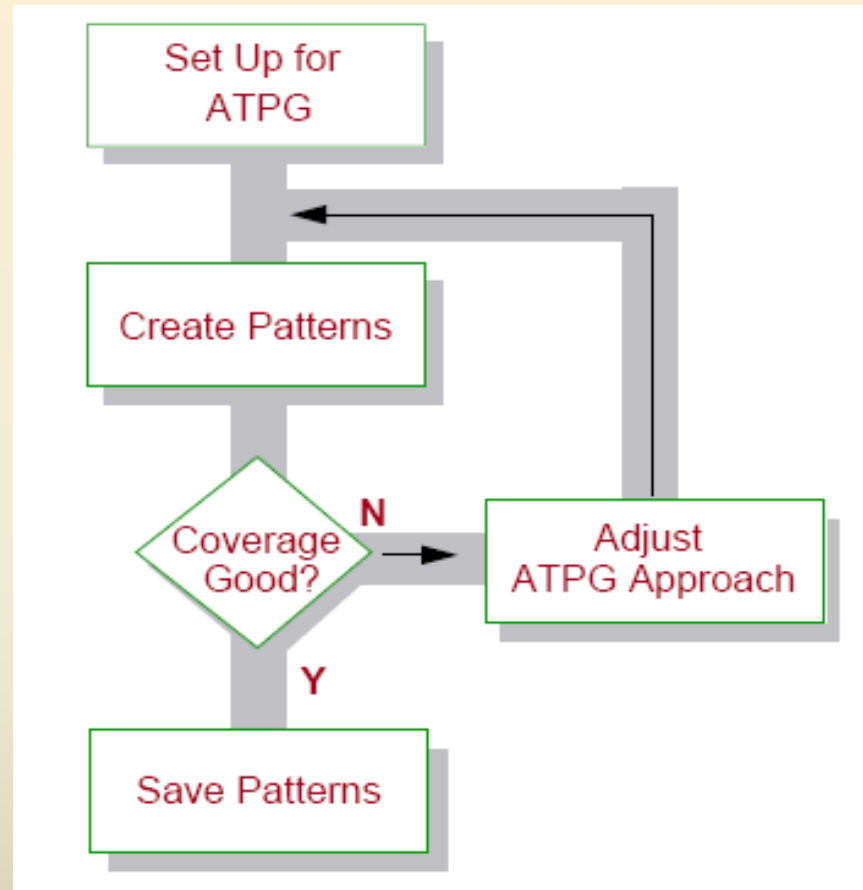
Top-down test design flow



Generate and verify a test set

- Automatic test pattern generation (ATPG)
 - apply D algorithm or other method to derive test patterns for all faults in the collapsed fault set
 - “random patterns” detect many faults – use deterministic method to detect the others (*Flextest*)
- Fault simulation
 - verify fault coverage of test patterns
 - simulate fault, apply test pattern, and observe output
 - fault detected if output different from expected value
 - repeat for each fault & test pattern combination

ATPG flow



Mentor Graphics *FlexTest/FastScan*

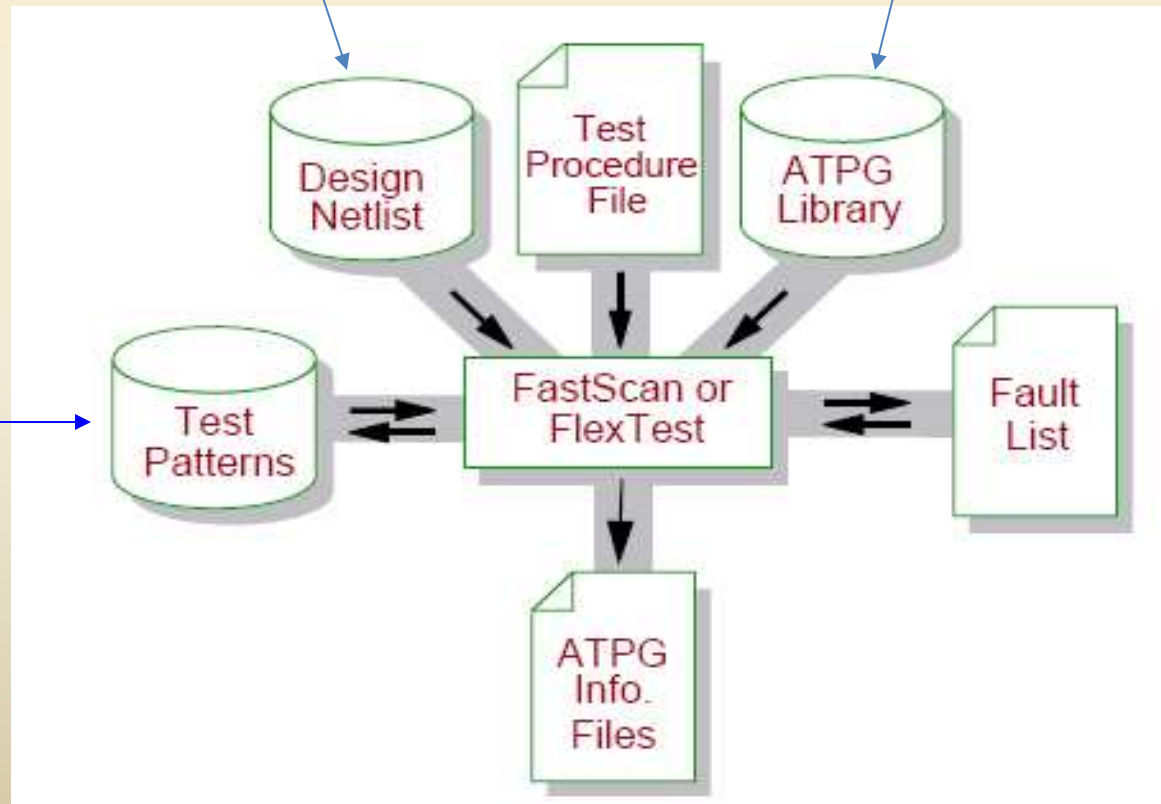
- Perform design for testability (DFT), ATPG, and fault simulation
 - **FastScan**: full-scan designs
 - **FlexTest**: non-scan through full-scan designs
- Typical flow:
 1. Implement BIST and/or DFT
 2. Generate test patterns (ATPG)
 3. Verify patterns through fault simulation

FlexTest inputs & outputs

file.v or file.vhd
(from Leonardo)

\$ADK/technology/adk.atpg

External file
or
internally
generated



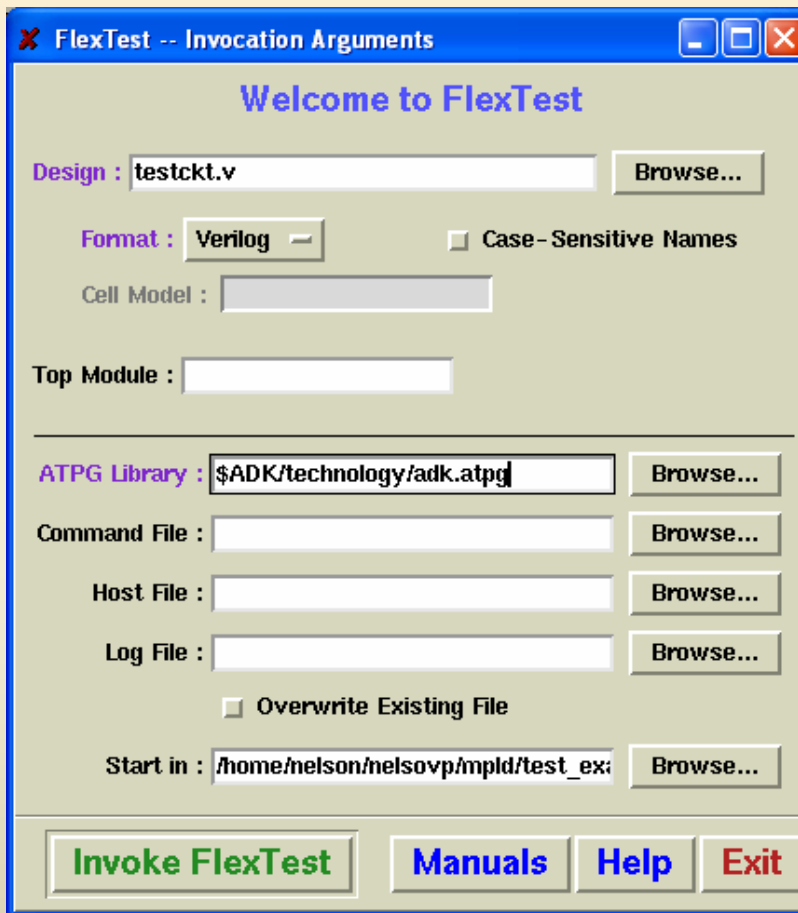
Invoking FlexTest

Command> `flextest` (and then fill out the following form)

Verilog or VHDL
Netlist →

File format →

ATPG Library →
\$ADK/technology/adk.atpg



Design : testckt.v Browse...

Format : Verilog ☐ Case-Sensitive Names

Cell Model :

Top Module :

ATPG Library : \$ADK/technology/adk.atpg Browse...

Command File : Browse...

Host File : Browse...

Log File : Browse...

☐ Overwrite Existing File

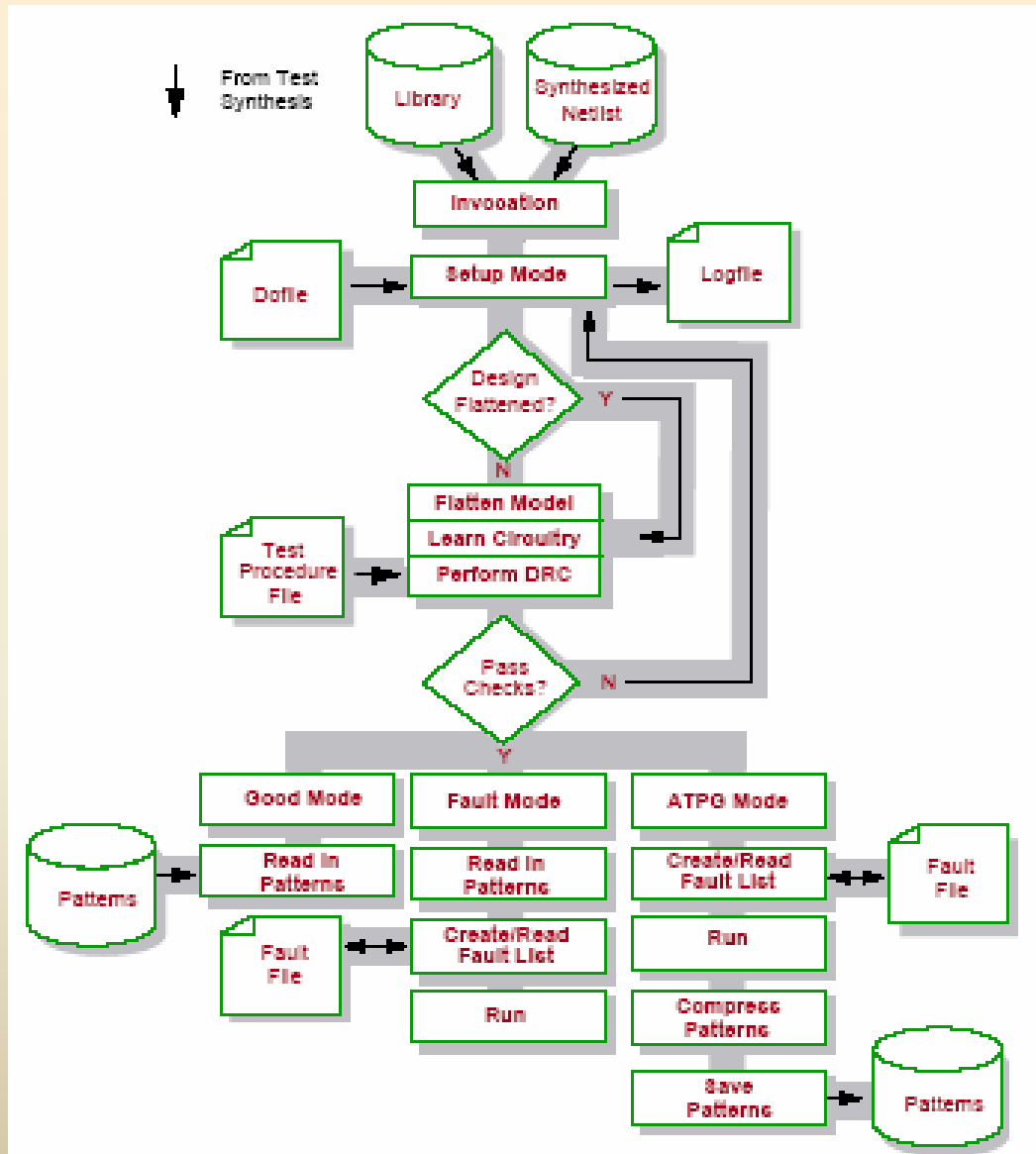
Start in : /home/nelson/nelsovp/mpid/test_ex Browse...

Invoke FlexTest Manuals Help Exit

To bypass the above form:

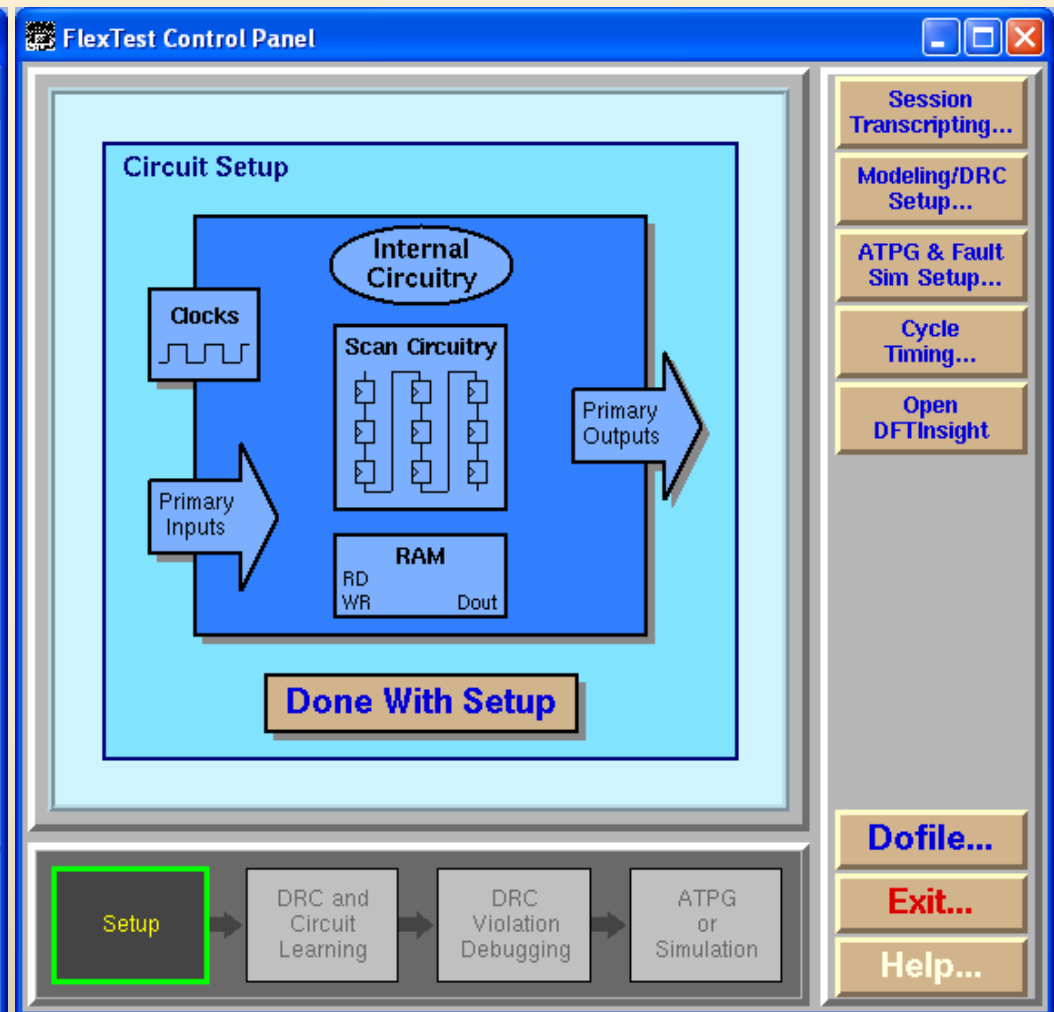
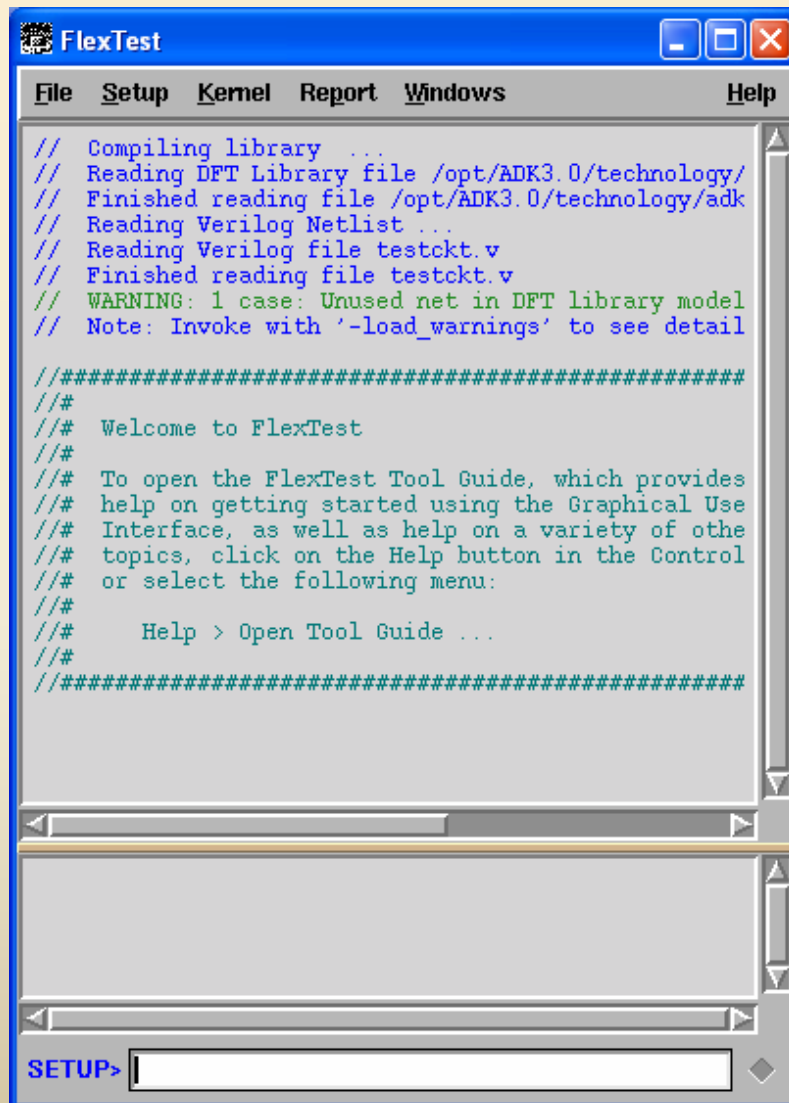
Command> `flextest testckt.v -verilog -lib $ADK/technology/adk.atpg`

Flextest/Fastscan Flow

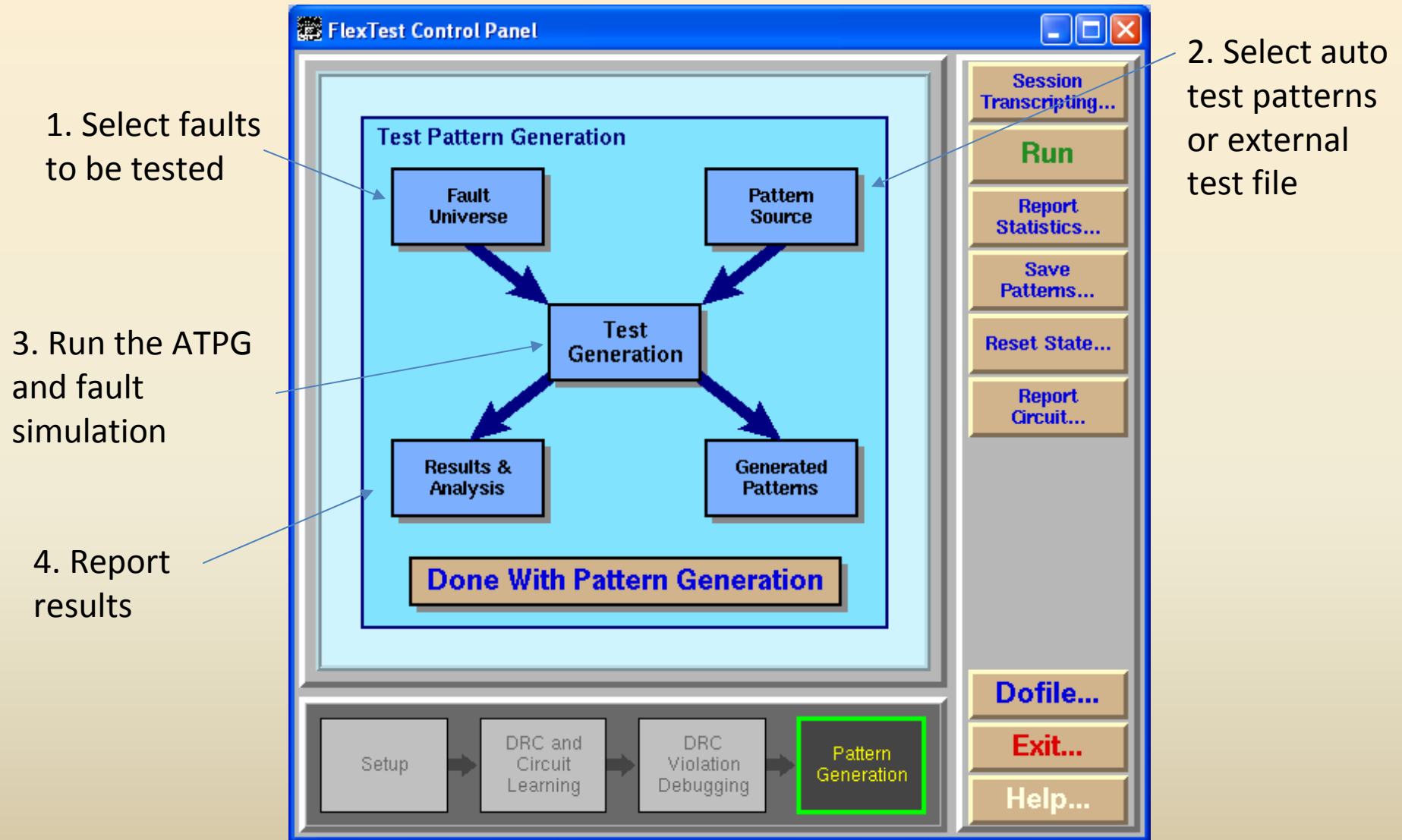


>set system mode setup

FlexTest control panel



FlexTest ATPG control panel



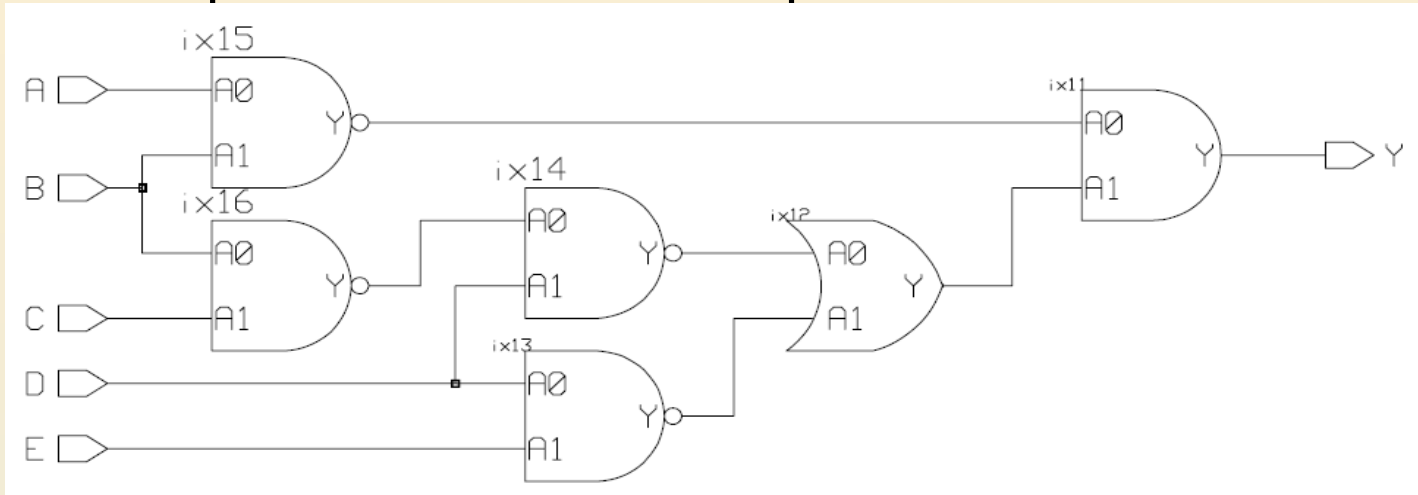
Fault Simulation

- Deliberately induce faults to determine what happens to circuit operation
- Access limited to primary inputs (PIs) & primary outputs (POs)
- Apply pattern to PIs at start of test cycle
- At end of test cycle, compare POs to expected values
- Fault detected if POs differ from correct values
- Fault coverage = detected faults/detectable faults

Fault simulation with external file selected as “Pattern Source” (“Table Pattern” option)

// fastscan test pattern file – define inputs

PI A
PI B
PI C
PI D
PI E
PO Y



// test patterns – bits in above order

000100
010000
011111
100111
100010

Note: These were “random” patterns

Flextest fault simulation results

| | | | | | | | | |
|---|----|----------|---|----|----------|---|----|----------|
| 1 | RE | /ix14/A1 | 0 | DS | /ix16/Y | 0 | DS | /ix12/A1 |
| 1 | RE | /ix13/A0 | 0 | DS | /ix14/A1 | 0 | DS | /ix13/Y |
| 1 | DS | /ix15/A1 | 1 | DS | /Y | 0 | DS | /Y |
| 1 | DS | /B | 1 | DS | /ix11/Y | 0 | DS | /ix11/Y |
| 1 | DS | /D | 0 | DS | /B | 0 | DS | /ix11/A0 |
| 0 | DS | /D | 1 | DS | /ix14/A0 | 0 | DS | /ix15/Y |
| 1 | DS | /ix11/A1 | 1 | DS | /ix16/Y | 0 | DS | /ix11/A1 |
| 1 | DS | /ix12/Y | 0 | DS | /ix16/A1 | 0 | DS | /ix12/Y |
| 1 | DS | /ix12/A1 | 0 | DS | /C | 1 | UO | /ix16/A1 |
| 1 | DS | /ix13/Y | 0 | DS | /ix16/A0 | 1 | UO | /C |
| 0 | DS | /ix13/A1 | 0 | DS | /ix12/A0 | 1 | UO | /ix16/A0 |
| 0 | DS | /E | 0 | DS | /ix14/Y | 1 | UC | /ix11/A0 |
| 0 | DS | /ix13/A0 | 1 | DS | /ix15/A0 | 1 | UC | /ix15/Y |
| 1 | DS | /ix12/A0 | 1 | DS | /A | 0 | UC | /ix15/A0 |
| 1 | DS | /ix14/Y | 1 | DS | /ix13/A1 | 0 | UC | /A |
| 0 | DS | /ix14/A0 | 1 | DS | /E | 0 | UC | /ix15/A1 |

Test coverage = 38 detected/48 faults = 79%

DS – fault detected in simulation

RE – redundant fault

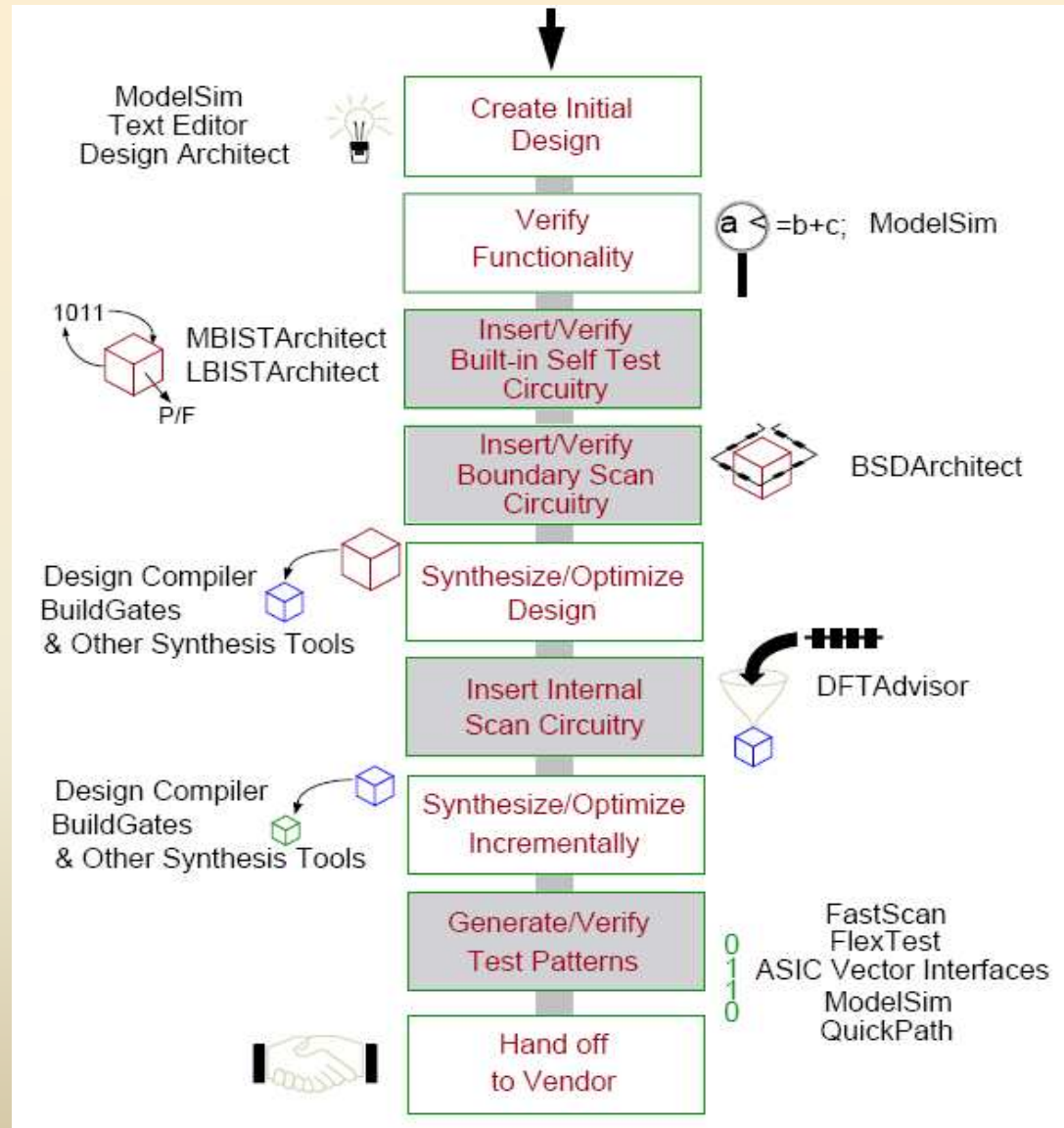
UO – unobserved fault

UC – uncontrolled fault

Design for Test

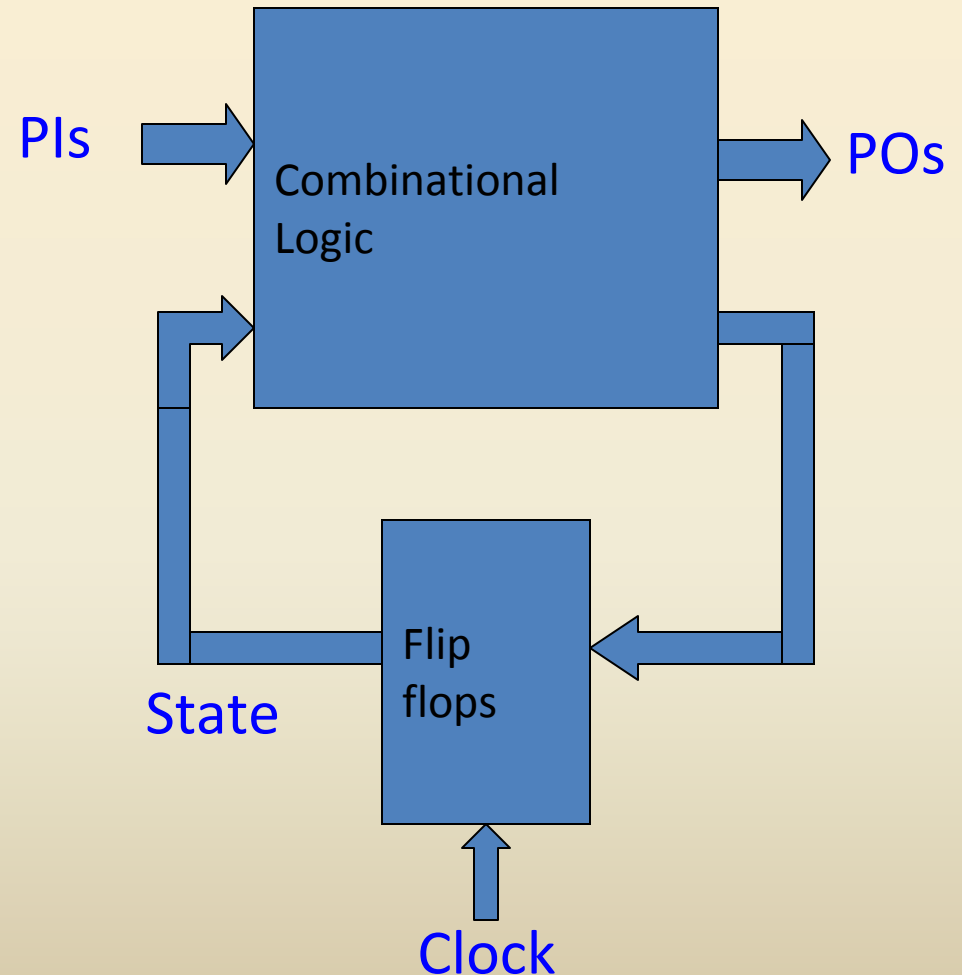
Scan Test

Top-down test design flow

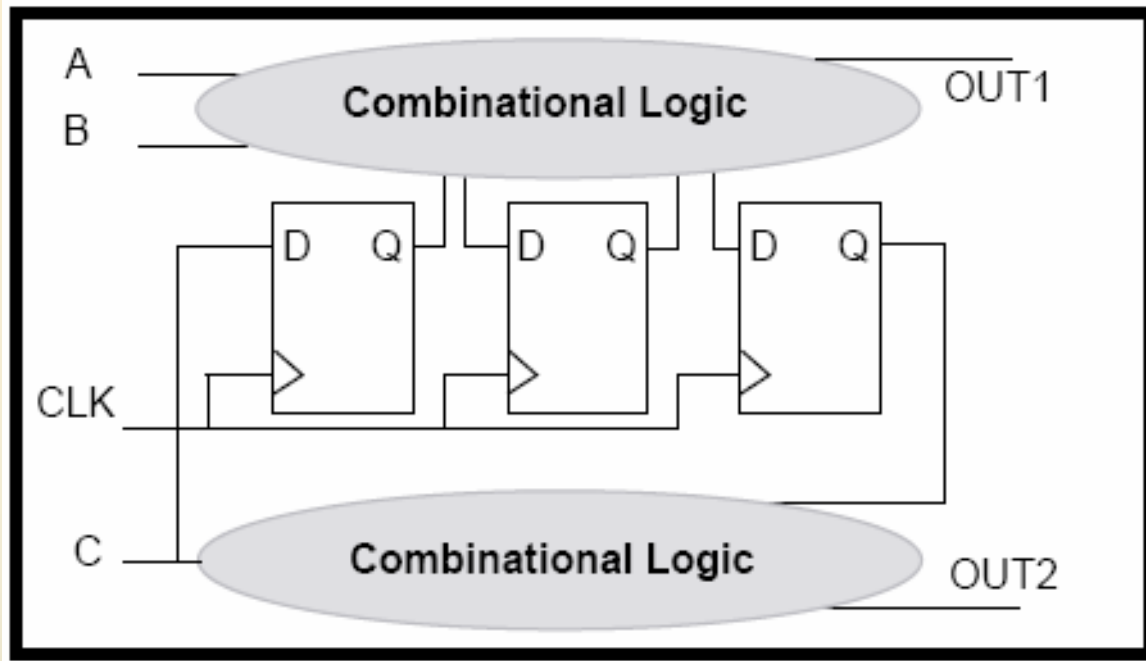


Sequential circuit testing problem

- External access only to PIs and POs
- Internal state is changed indirectly
- For N PIs and K state variables, must test 2^{N+K} combinations
- Some states difficult to reach, so even more test vectors are needed



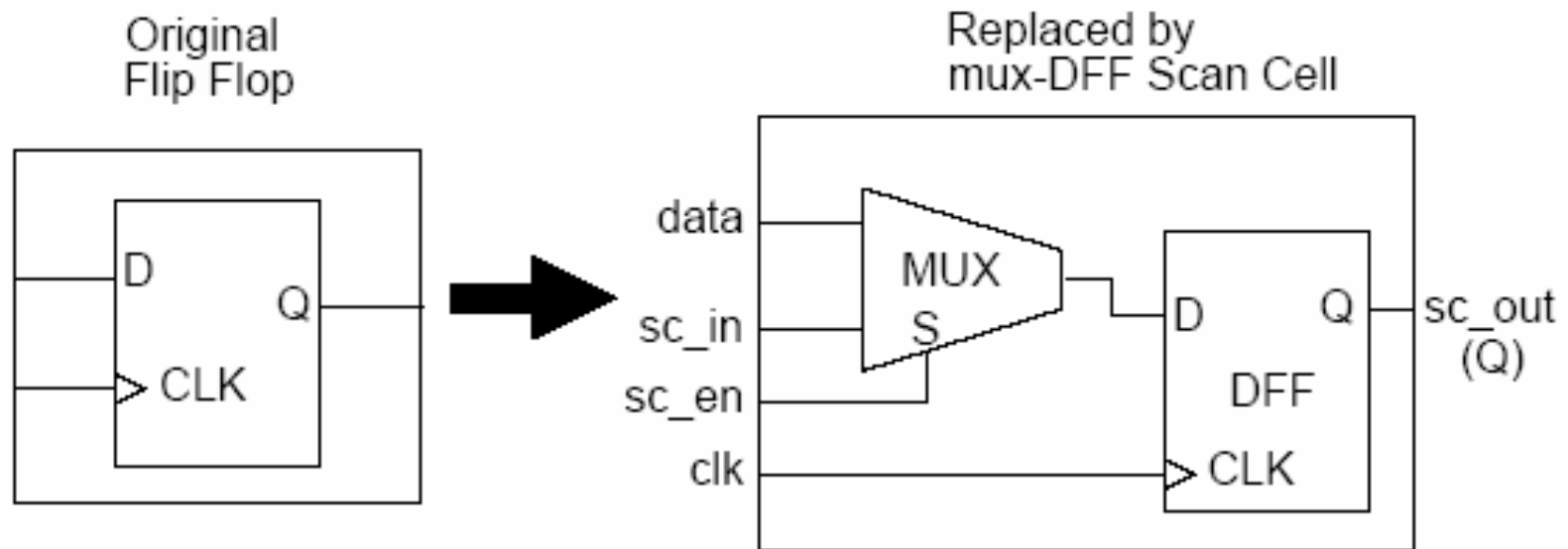
Design for Test (DFT)



Flip flop states are difficult to set from PIs A & B

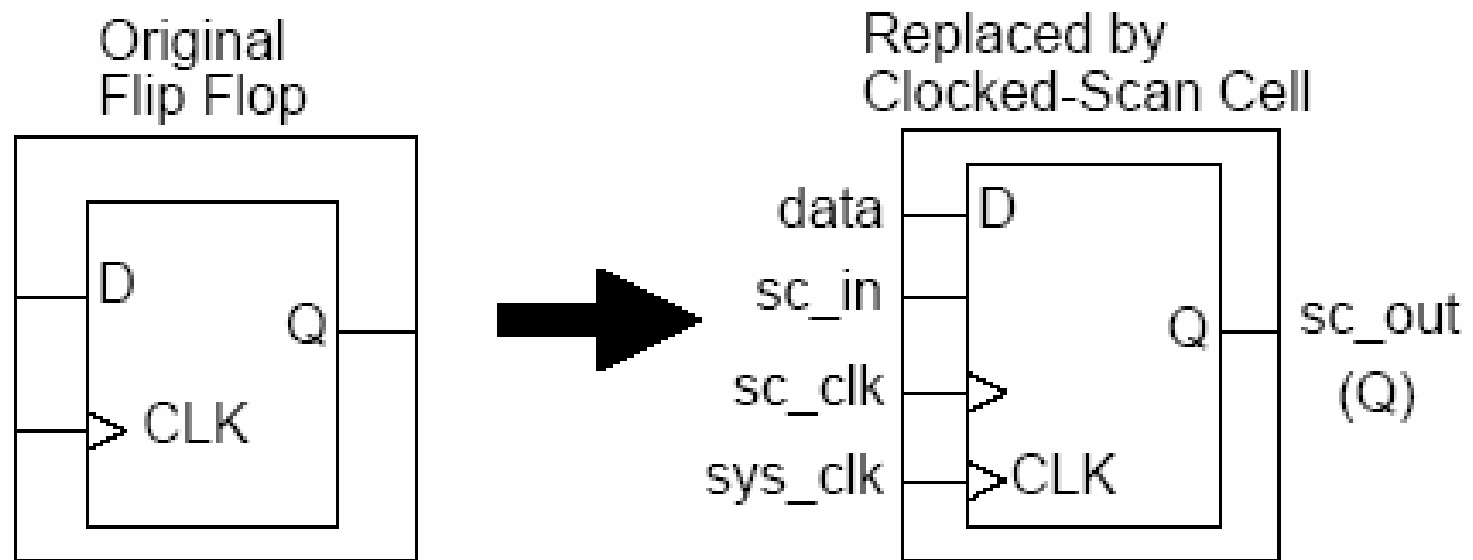
Scan type: mux_scan

Figure 3-10. Mux-DFF Replacement



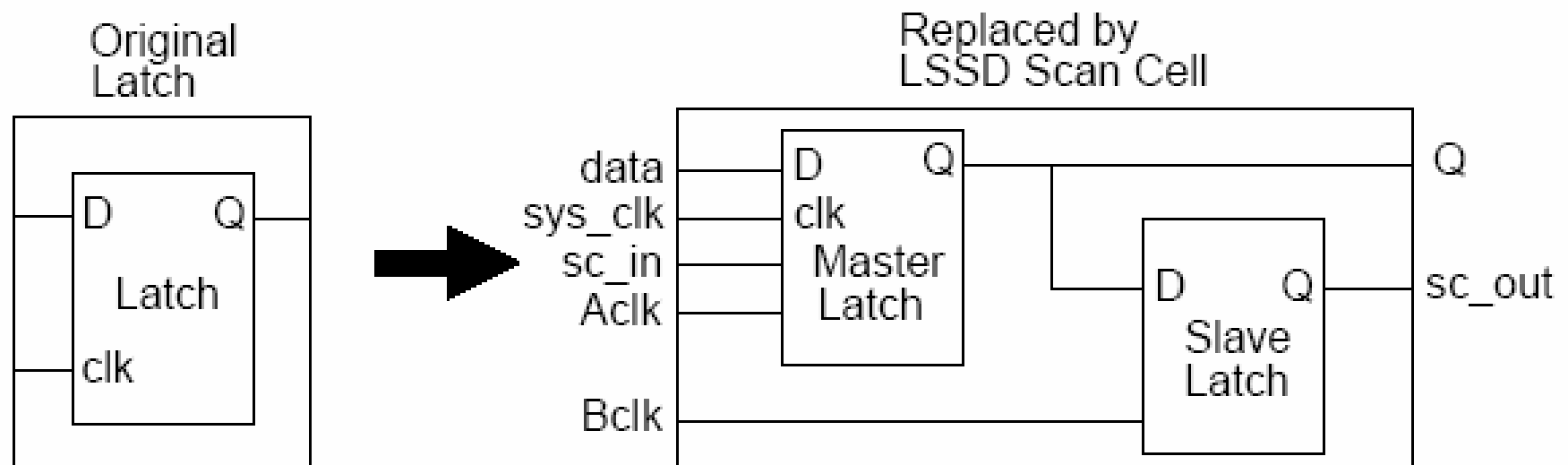
Scan type: clocked_scan

Figure 3-11. Clocked-Scan Replacement

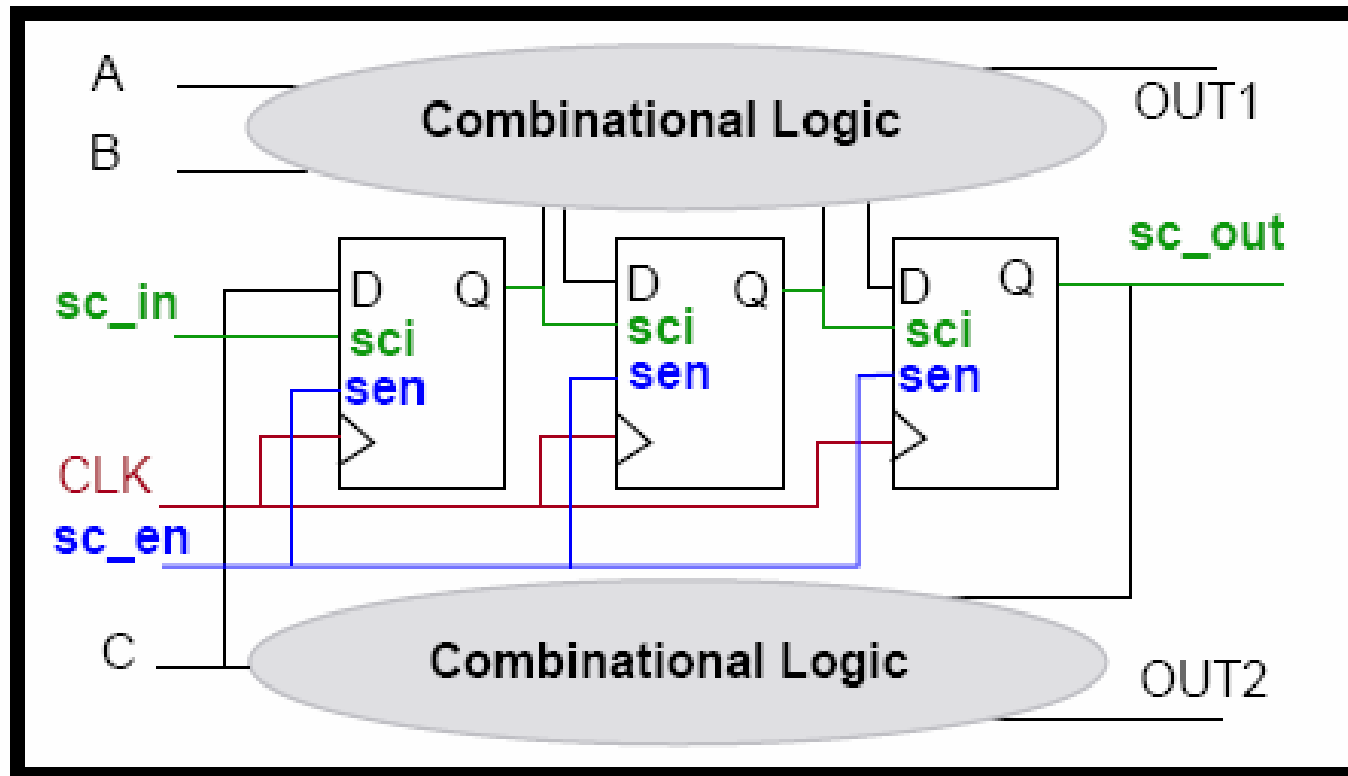


Scan type: Lssd

Figure 3-12. LSSD Replacement

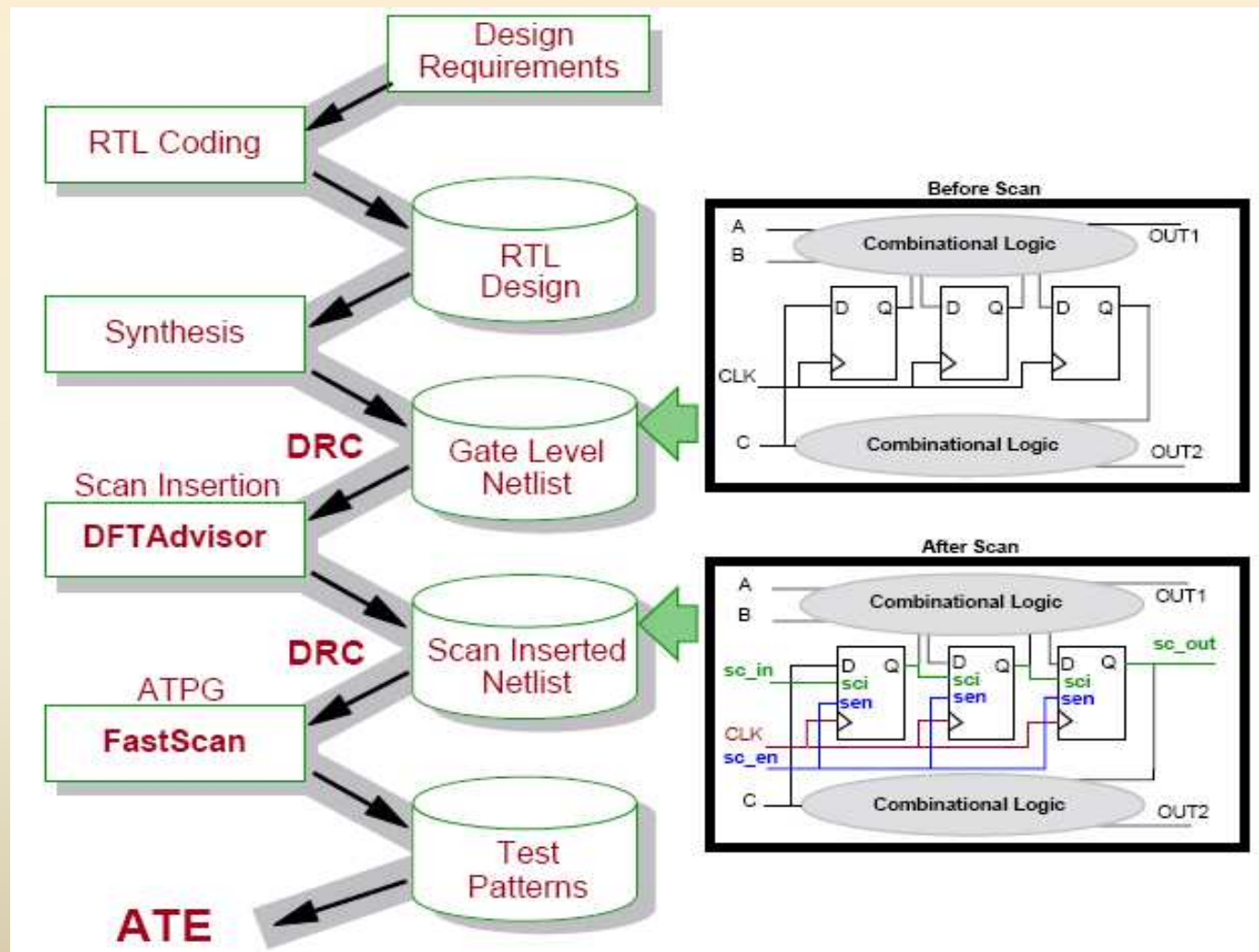


DFT: Scan Design

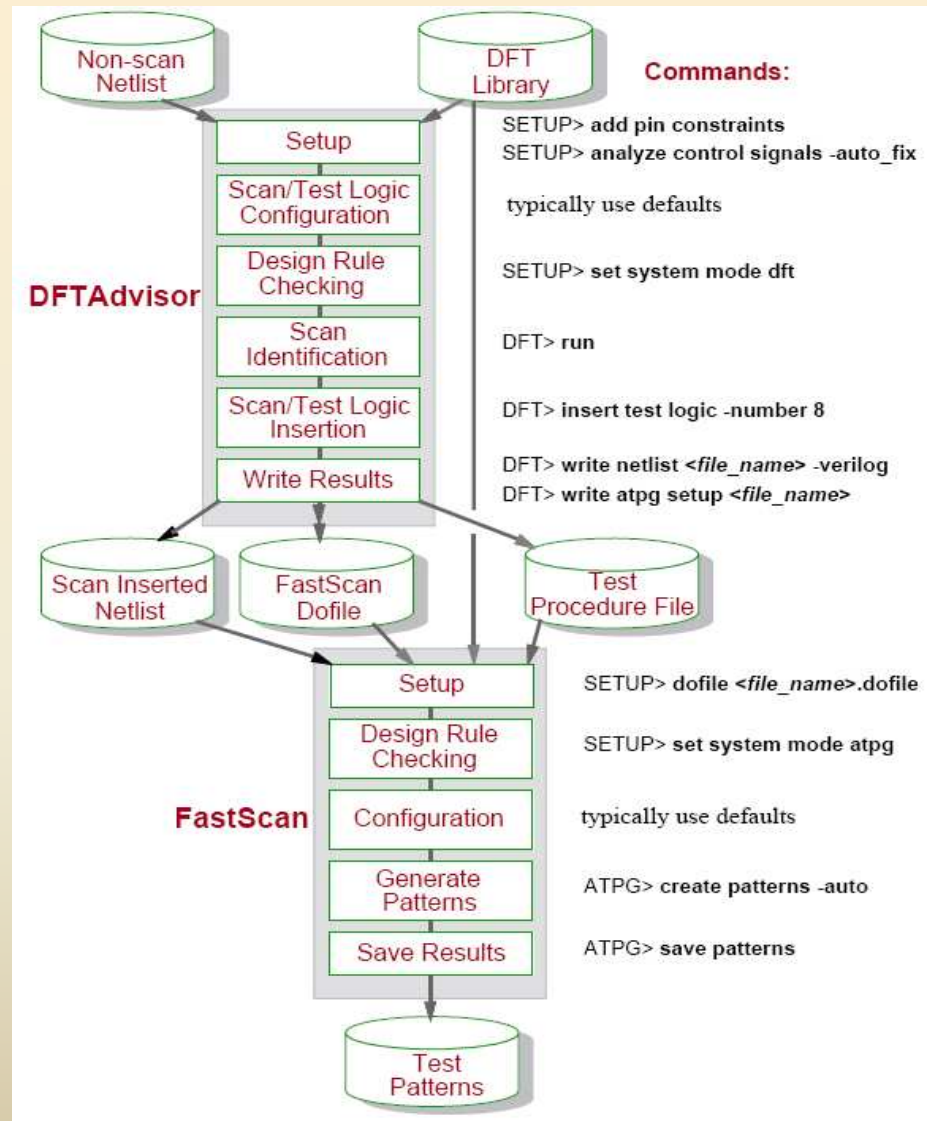


Flip flops replaced with “scan” flip flops
Flip flop states set via “scan input” **sc_in**

DFTadvisor/FastScan Design Flow



DFT test flow and commands



Example DFTadvisor session

- Invoke:
 - `dftadvisor –verilog count4.v –lib $ADK/technology/adk.atpg`
- Implement scan with defaults (full scan, mux-DFF elements):
 - `set system mode setup`
 - `analyze control signals –auto`
 - `set system mode dft`
 - `run`
 - `insert test logic`
 - `write netlist count4_scan.v –verilog`
 - `write atpg setup count4_scan`

Example FastScan session for a circuit with scan chains

- Invoke:
`fastscan -verilog count4_scan.v -lib $ADK/technology/adk.atpg`
- Generate test pattern file:
 - `dofile count4_scan.dofile` (defines scan path & procedure)
 - `set system mode atpg`
 - `create patterns -auto`
 - `save patterns`

-- Example: count4.vhd 4-bit parallel-load synchronous counter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all; USE ieee.numeric_std.all; --synthesis libraries

ENTITY count4 IS
    PORT (clock,clear,enable,load_count : IN STD_LOGIC;
          D: IN unsigned(3 downto 0);
          Q: OUT unsigned(3 downto 0));
END count4;

ARCHITECTURE rtl OF count4 IS
    SIGNAL int : unsigned(3 downto 0);
BEGIN
    PROCESS(clear, clock, enable)
    BEGIN
        IF (clear = '1') THEN
            int <= "0000";
        ELSIF (clock'EVENT AND clock='1') THEN
            IF (enable = '1') THEN
                IF (load_count = '1') THEN
                    int <= D;
                ELSE
                    int <= int + "01";
                END IF;
            END IF;
        END IF;
    END PROCESS;
    Q <= int;
END rtl;
```

Binary counter (4-bit)

Synthesized by
Leonardo

```
count4.v - WordPad
File Edit View Insert Format Help

//
// Verilog description for cell count4,
// Thu Sep 15 13:29:54 2005
//
// LeonardoSpectrum Level 3, 2005a.82
//
module count4 ( clock, clear, enable, \output ) ;

    input clock ;
    input clear ;
    input enable ;
    output [3:0]\output ;

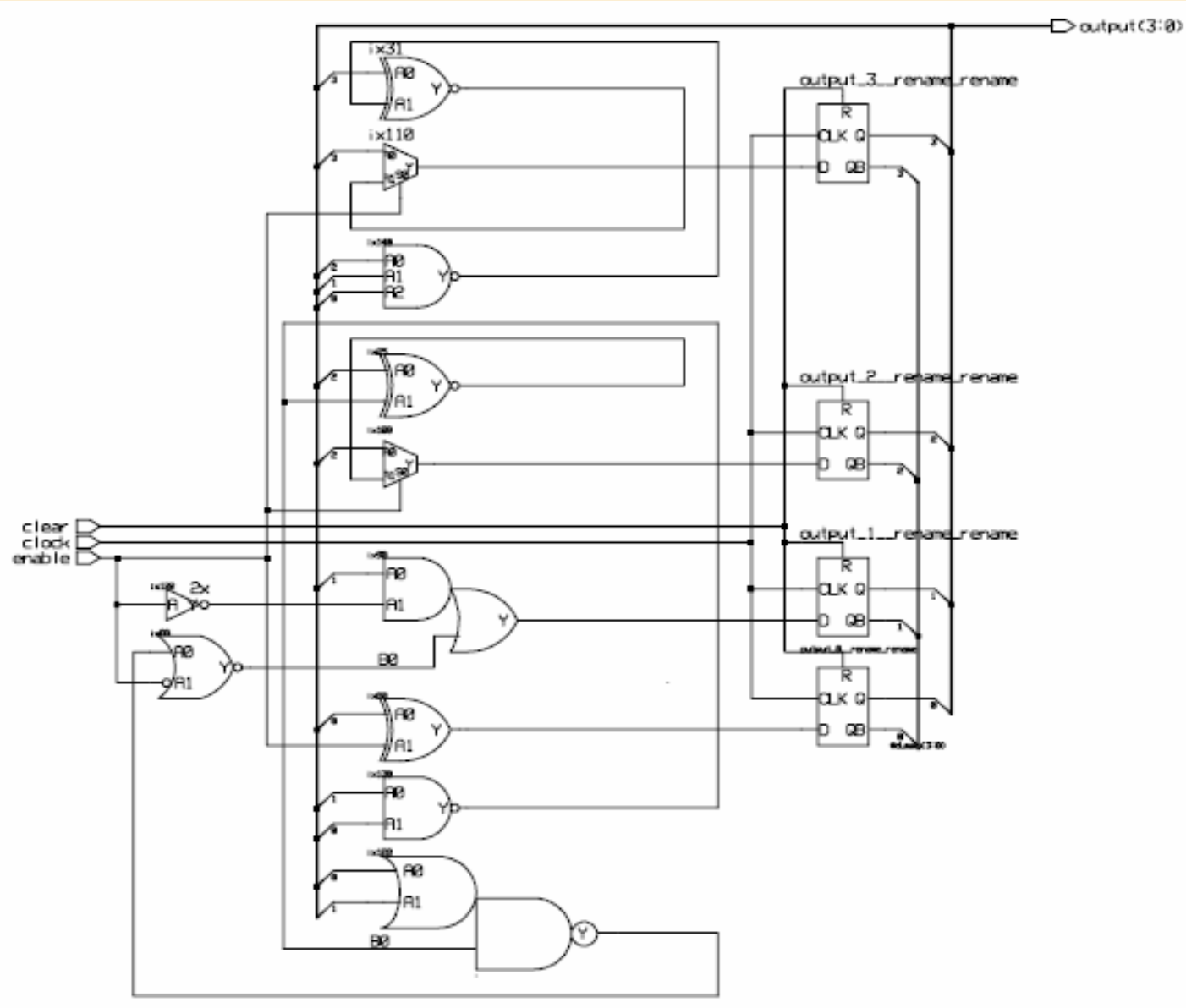
    wire nx24, nx30, nx79, nx87, nx89, nx99, nx109, nx121, nx127, nx129, nx139;
    wire [3:0] \ $dummy ;

    dffr output_0_rename_rename (.Q (\output [0]), .QB (\ $dummy [0]), .D (nx79)
        , .CLK (clock), .R (clear)) ;
    inv02 ix122 (.Y (nx121), .A (enable)) ;
    dffr output_1_rename_rename (.Q (\output [1]), .QB (\ $dummy [1]), .D (nx89)
        , .CLK (clock), .R (clear)) ;
    ao21 ix90 (.Y (nx89), .AO (\output [1]), .A1 (nx121), .BO (nx87)) ;
    oai21 ix128 (.Y (nx127), .AO (\output [0]), .A1 (\output [1]), .BO (nx129)
        ) ;
    nand02_2x ix130 (.Y (nx129), .AO (\output [1]), .A1 (\output [0])) ;
    dffr output_2_rename_rename (.Q (\output [2]), .QB (\ $dummy [2]), .D (nx99)
        , .CLK (clock), .R (clear)) ;
    mux21_ni ix100 (.Y (nx99), .AO (\output [2]), .A1 (nx24), .SO (enable)) ;
    xnor2 ix25 (.Y (nx24), .AO (\output [2]), .A1 (nx129)) ;
    dffr output_3_rename_rename (.Q (\output [3]), .QB (\ $dummy [3]), .D (nx109)
        , .CLK (clock), .R (clear)) ;
    mux21_ni ix110 (.Y (nx109), .AO (\output [3]), .A1 (nx30), .SO (enable)) ;
    xnor2 ix31 (.Y (nx30), .AO (\output [3]), .A1 (nx139)) ;
    nand03 ix140 (.Y (nx139), .AO (\output [2]), .A1 (\output [1]), .A2 (
        \output [0])) ;
    xor2 ix80 (.Y (nx79), .AO (\output [0]), .A1 (enable)) ;
    nor02ii ix88 (.Y (nx87), .AO (nx127), .A1 (enable)) ;

endmodule

For Help, press F1
```

count4 – without scan design



Binary counter (4-bit)

Synthesized by
Leonardo

*DFTAdvisor
Changed to
Scan Design*

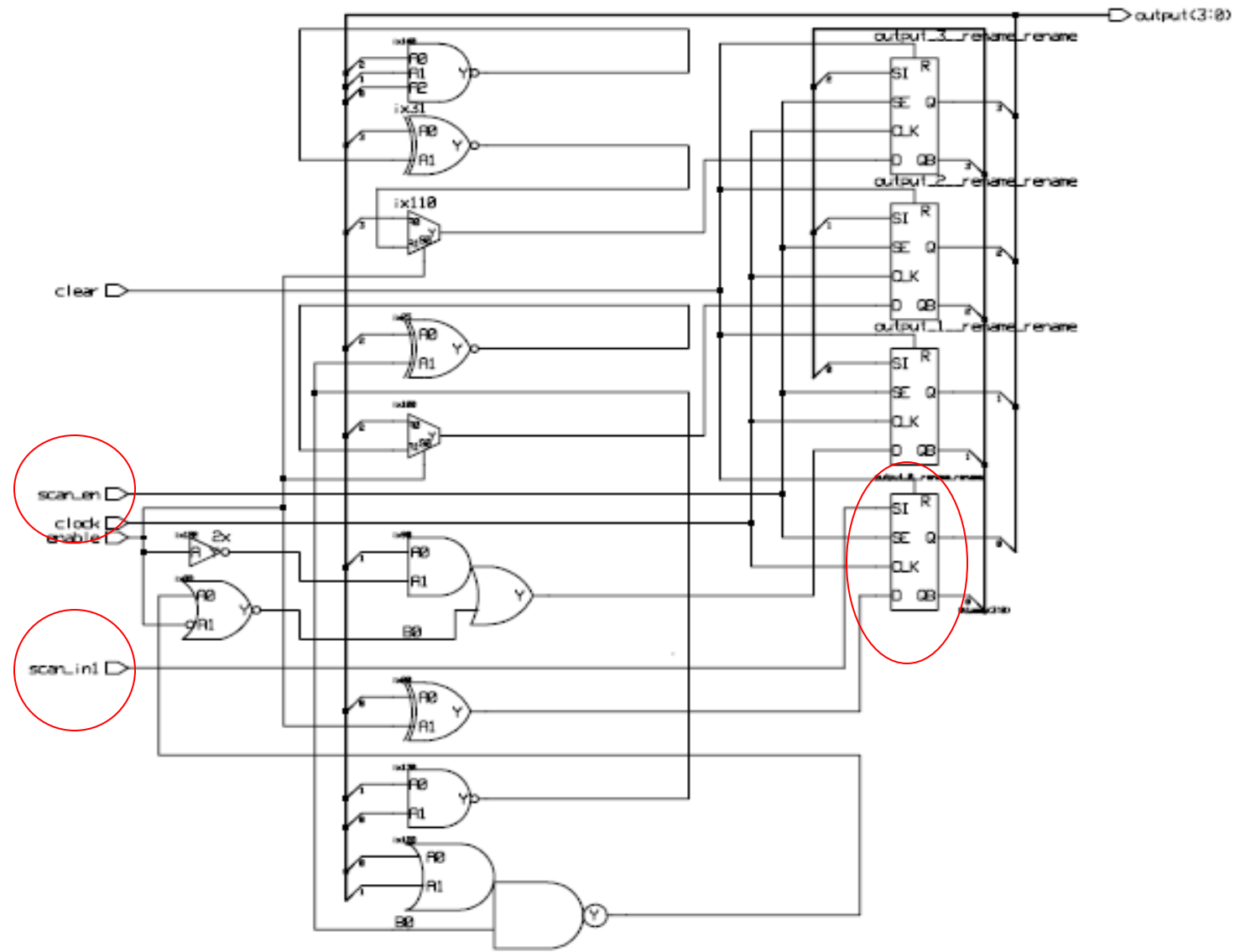
```
count4_scan.v - WordPad
File Edit View Insert Format Help

/*
 *   DESC: Generated by DFTAdvisor at Wed Nov 30 17:01:15 2005
 */
module count4 ( clock , clear , enable , \output , scan_in1 , scan_en );
input  clock , clear , enable , scan_in1 , scan_en ;
output [3:0] \output ;
    wire nx139 , nx30 , nx109 , nx24 , nx99 , nx127 , nx129 , nx87 , nx89 , nx121 , nx79 ;
    wire [3:0] \$dummy ;

    sffr_ni output_0_rename_rename (.D ( nx79 ) , .SI ( scan_in1 ) , .SE
        ( scan_en ) , .CLK ( clock ) , .R ( clear ) , .Q ( \output [0] ) ,
        .QB ( \$dummy [0] ));
    inx02 ix122 (.A ( enable ) , .Y ( nx121 ));
    sffr_ni output_1_rename_rename (.D ( nx89 ) , .SI ( \$dummy [0] ) , .SE
        ( scan_en ) , .CLK ( clock ) , .R ( clear ) , .Q ( \output [1] ) ,
        .QB ( \$dummy [1] ));
    ao21 ix90 (.AO ( \output [1] ) , .A1 ( nx121 ) , .B0 ( nx87 ) , .Y ( nx89 ));
    oai21 ix128 (.AO ( \output [0] ) , .A1 ( \output [1] ) , .B0 ( nx129 ) ,
        .Y ( nx127 ));
    nand02 2x ix130 (.AO ( \output [1] ) , .A1 ( \output [0] ) , .Y ( nx129 ));
    sffr_ni output_2_rename_rename (.D ( nx99 ) , .SI ( \$dummy [1] ) , .SE
        ( scan_en ) , .CLK ( clock ) , .R ( clear ) , .Q ( \output [2] ) ,
        .QB ( \$dummy [2] ));
    mux21_ni ix100 (.AO ( \output [2] ) , .A1 ( nx24 ) , .S0 ( enable ) , .Y
        ( nx99 ));
    xnor2 ix25 (.AO ( \output [2] ) , .A1 ( nx129 ) , .Y ( nx24 ));
    sffr_ni output_3_rename_rename (.D ( nx109 ) , .SI ( \$dummy [2] ) , .SE
        ( scan_en ) , .CLK ( clock ) , .R ( clear ) , .Q ( \output [3] ) ,
        .QB ( \$dummy [3] ));
    mux21_ni ix110 (.AO ( \output [3] ) , .A1 ( nx30 ) , .S0 ( enable ) , .Y
        ( nx109 ));
    xnor2 ix31 (.AO ( \output [3] ) , .A1 ( nx139 ) , .Y ( nx30 ));
    nand03 ix140 (.AO ( \output [2] ) , .A1 ( \output [1] ) , .A2
        ( \output [0] ) , .Y ( nx139 ));
    xor2 ix80 (.AO ( \output [0] ) , .A1 ( enable ) , .Y ( nx79 ));
    nor02ii ix88 (.AO ( nx127 ) , .A1 ( enable ) , .Y ( nx87 ));
endmodule

For Help, press F1
```

count4 – scan inserted by DFTadvisor



Test file: scan chain definition and load/unload procedures

```
scan_group "grp1" =  
  scan_chain "chain1" =  
    scan_in = "/scan_in1";  
    scan_out = "/output[3]";  
    length = 4;  
  end;  
  procedure shift "grp1_load_shift" =  
    force_sci "chain1" 0;  
    force "/clock" 1 20;  
    force "/clock" 0 30;  
    period 40;  
  end;  
  procedure shift "grp1_unload_shift" =  
    measure_sco "chain1" 10;  
    force "/clock" 1 20;  
    force "/clock" 0 30;  
    period 40;  
  end;
```

```
  procedure load "grp1_load" =  
    force "/clear" 0 0;  
    force "/clock" 0 0;  
    force "/scan_en" 1 0;  
    apply "grp1_load_shift" 4 40;  
  end;  
  procedure unload "grp1_unload" =  
    force "/clear" 0 0;  
    force "/clock" 0 0;  
    force "/scan_en" 1 0;  
    apply "grp1_unload_shift" 4 40;  
  end;  
end;
```

Test file: scan chain test

// send a pattern through the scan chain

CHAIN_TEST =

 pattern = 0;

 apply "grp1_load" 0 = (use grp1_load proc.)

 chain "chain1" = "0011"; (pattern to scan in)

 end;

 apply "grp1_unload" 1 = (use grp1_unload proc.)

 chain "chain1" = "1100"; (pattern scanned out)

 end;

end;

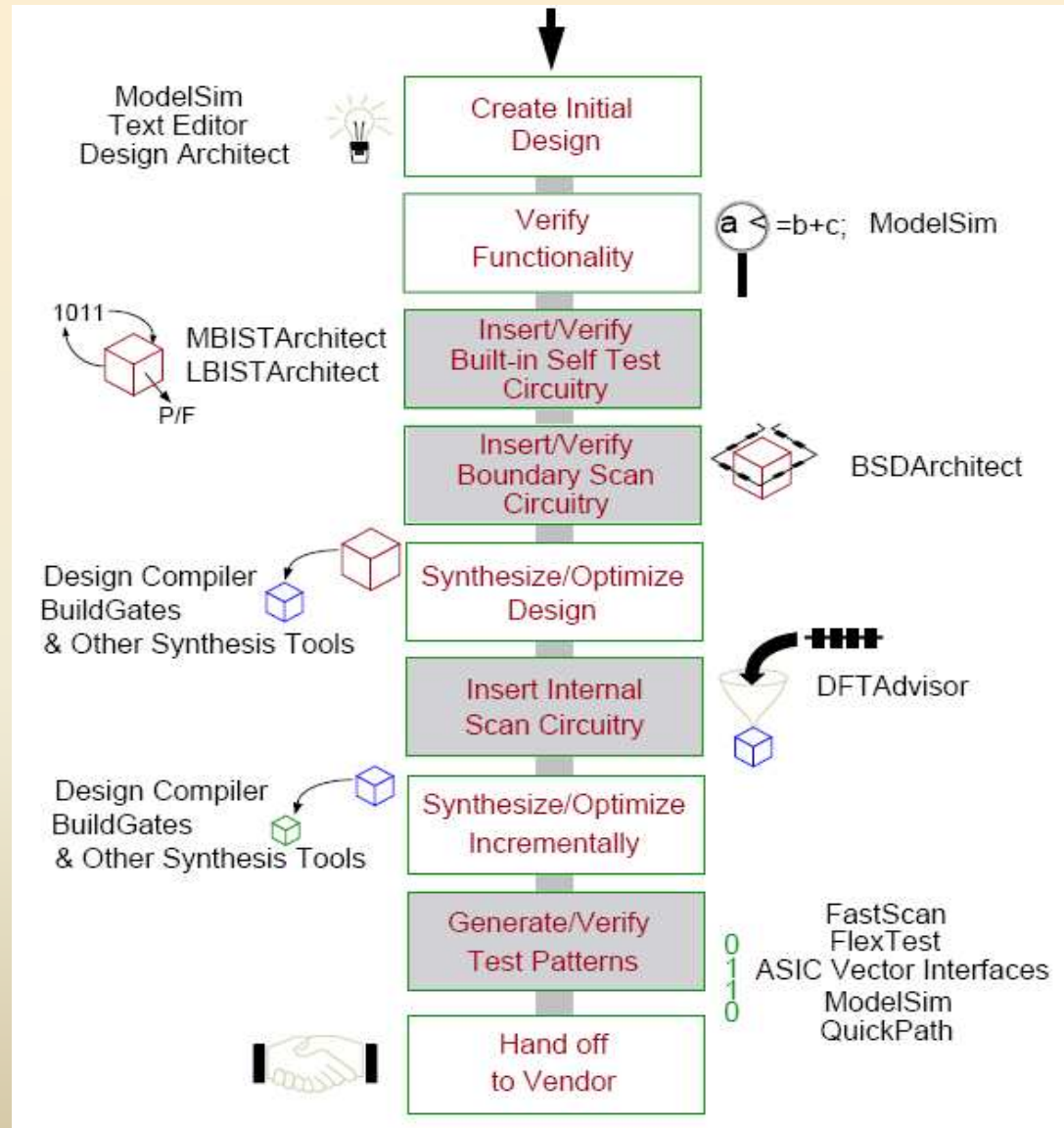
Test file: sample test pattern

```
// one of 14 patterns for the counter circuit
pattern = 0;                (pattern #)
  apply "grp1_load" 0 =      (load scan chain)
    chain "chain1" = "1000"; (scan-in pattern)
  end;
  force  "PI" "00110" 1;     (PI pattern)
  measure "PO" "0010" 2;     (expected POs)
  pulse  "/clock" 3;         (normal op. cycle)
  apply "grp1_unload" 4 =    (read scan chain)
    chain "chain1" = "0110"; (expected pattern)
  end;
```

Built-In Self Test

Smith Text: Chapter 14.7

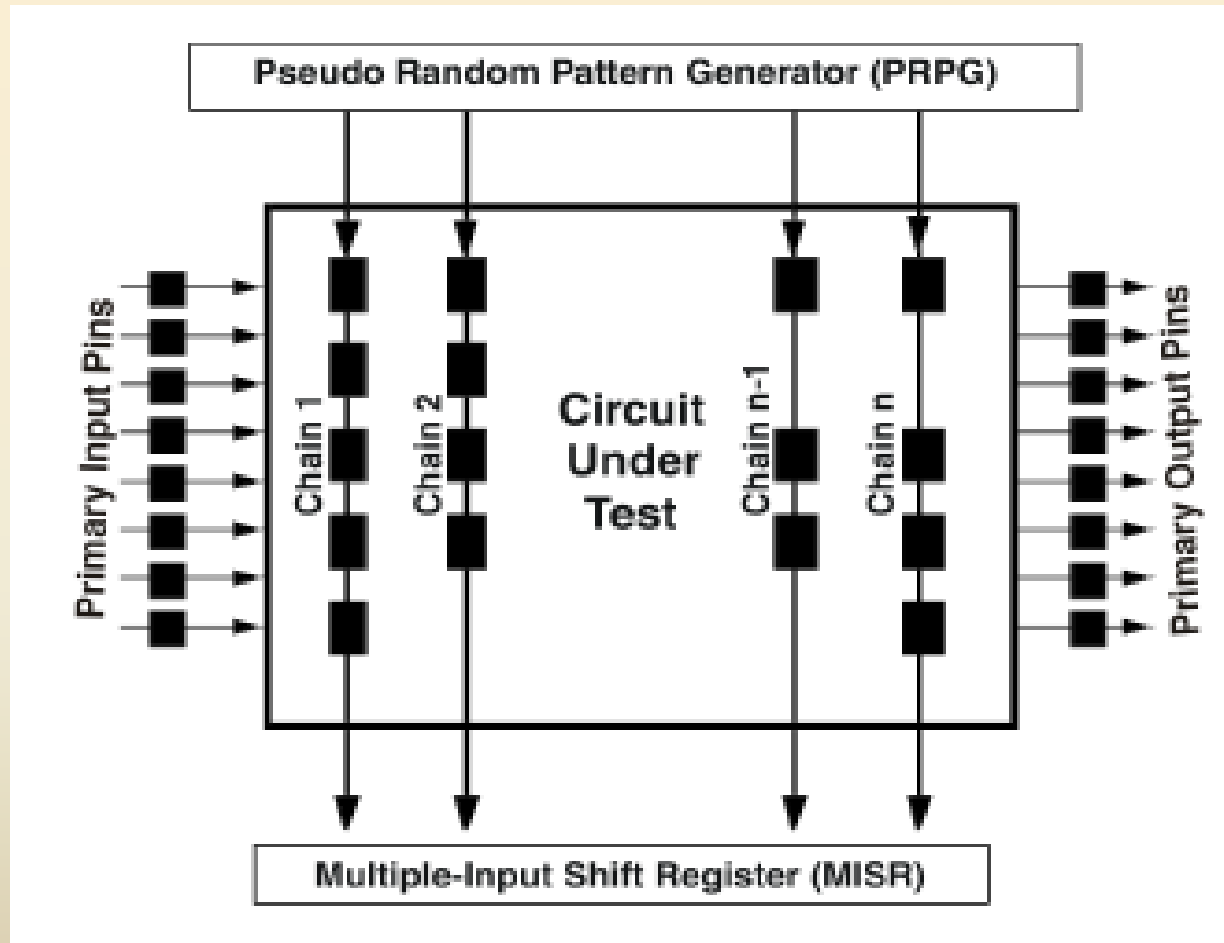
Top-down test design flow



Built-In Self-Test (BIST)

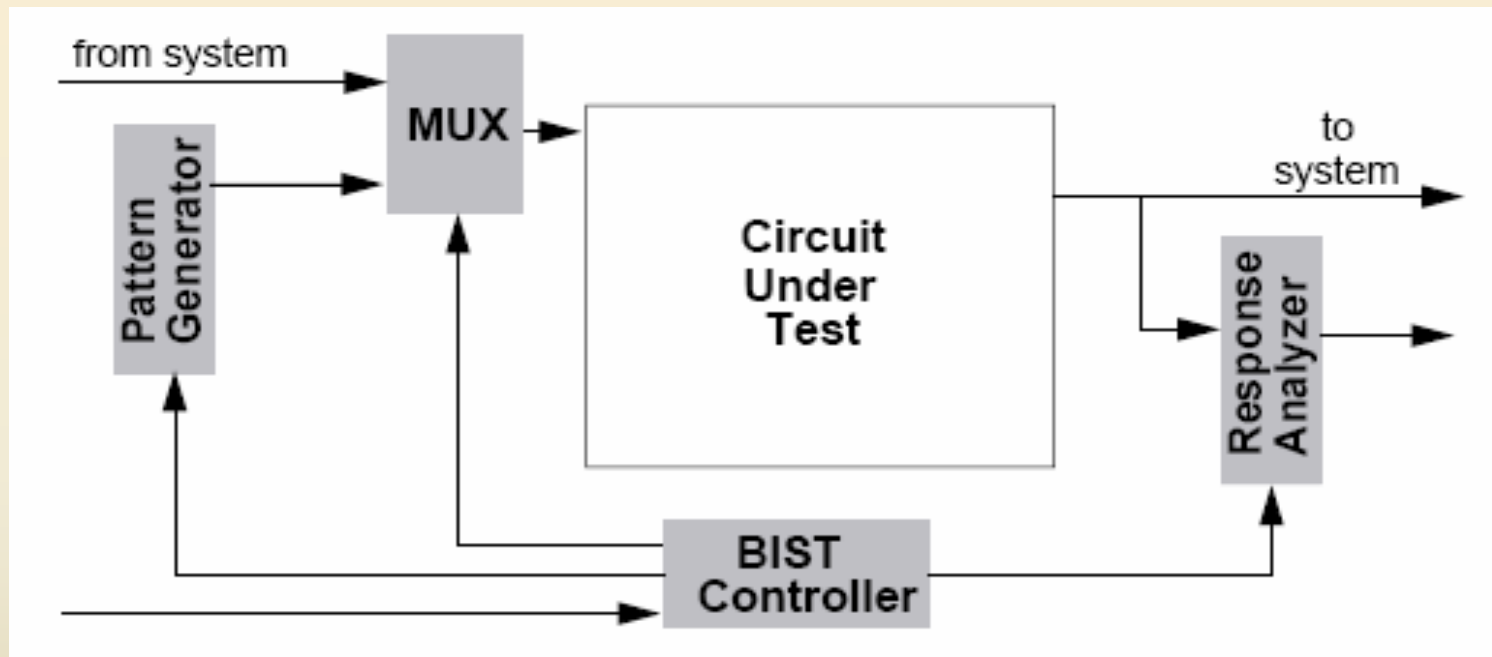
- Structured-test techniques for logic ckts to improve access to internal signals from primary inputs/outputs
- BIST procedure:
 - generate a test pattern
 - apply the pattern to “circuit under test” (CUT)
 - check the response
 - repeat for each test pattern
- Most BIST approaches use pseudo-random test vectors

Logic BIST general architecture



Source: Mentor Graphics "LBISTArchitect Process Guide"

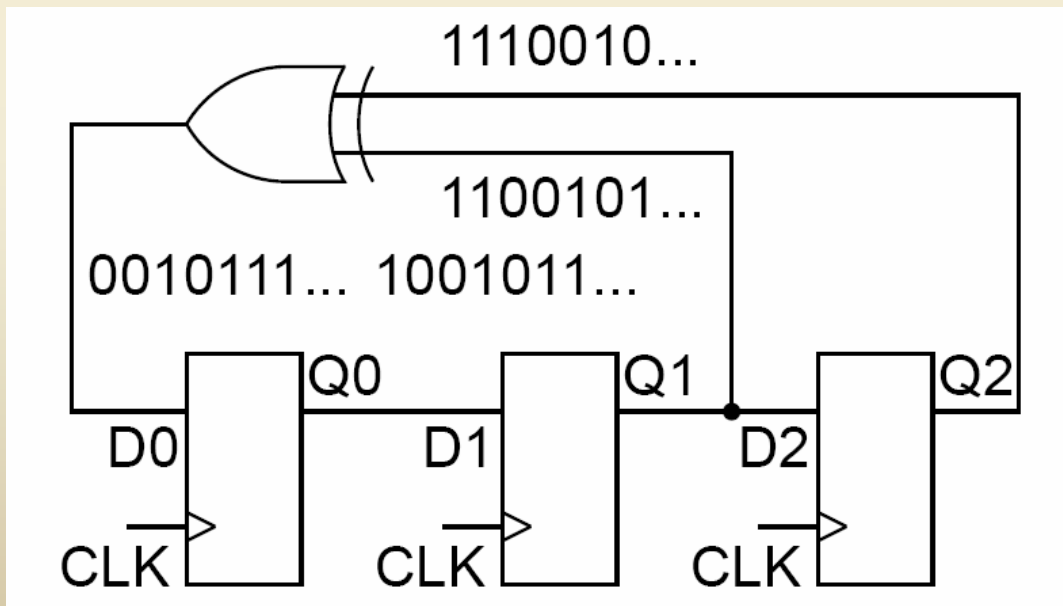
Circuit with BIST circuitry



Source: Mentor Graphics "LBISTArchitect Process Guide"

Linear Feedback Shift Register (LFSR)

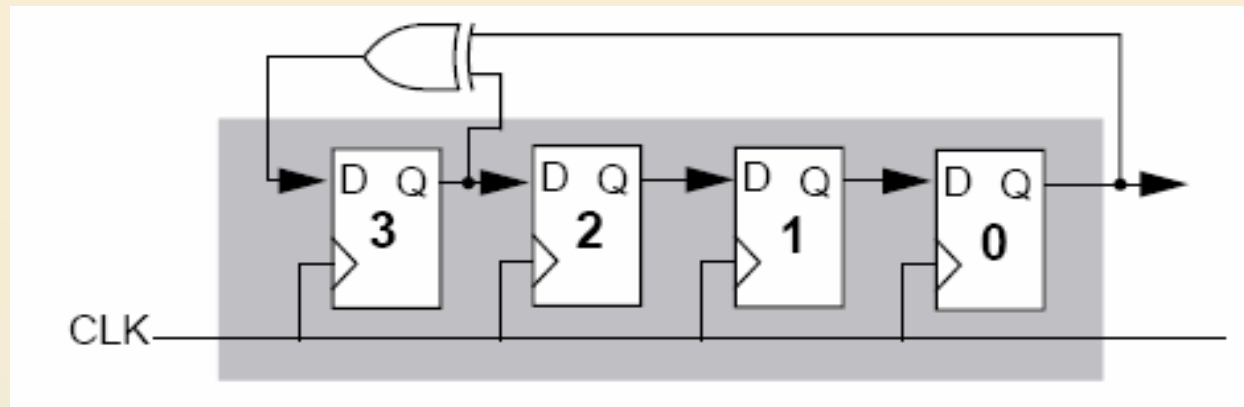
- Produce pseudorandom binary sequences (PRBS)
- Implement with shift register and XOR gates
- Selection of feedback points allows n-bit register to produce a PRBS of length $2^n - 1$



LFSR produces
pattern:
7,3,1,4,2,5,6
(PRBS length 7)

Text figure 14.23

4-stage LFSR with one tap point

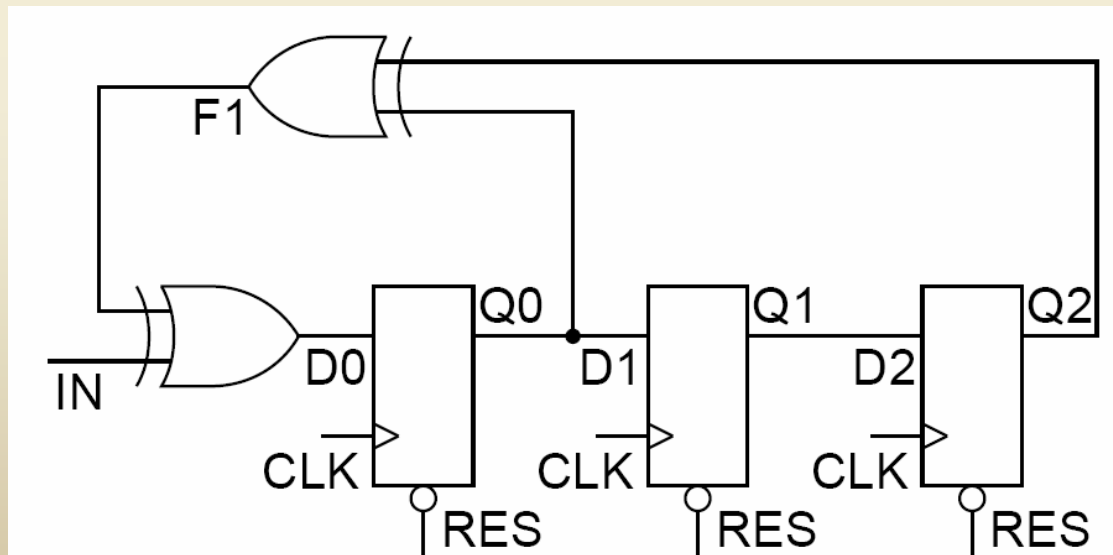


| <u>State</u> | <u>Pattern</u> | <u>State</u> | <u>Pattern</u> |
|--------------|----------------|--------------|----------------|
| 0 | 1000 | 8 | 1101 |
| 1 | 1100 | 9 | 0110 |
| 2 | 1110 | 10 | 0011 |
| 3 | 1111 | 11 | 1001 |
| 4 | 0111 | 12 | 0100 |
| 5 | 1011 | 13 | 0010 |
| 6 | 0101 | 14 | 0001 |
| 7 | 1010 | 15 | 1000 |

Source: Mentor Graphics "LBISTArchitect Process Guide"

Serial Input Signature Register (SISR)

- Use an LFSR to compact serial input data into an n-bit “signature”
- For sufficiently large n, two different sequences producing the same signature is unlikely
- Good circuit has a unique signature



Initialize LFSR
to '000' via RES.

Signature formed via shift & add

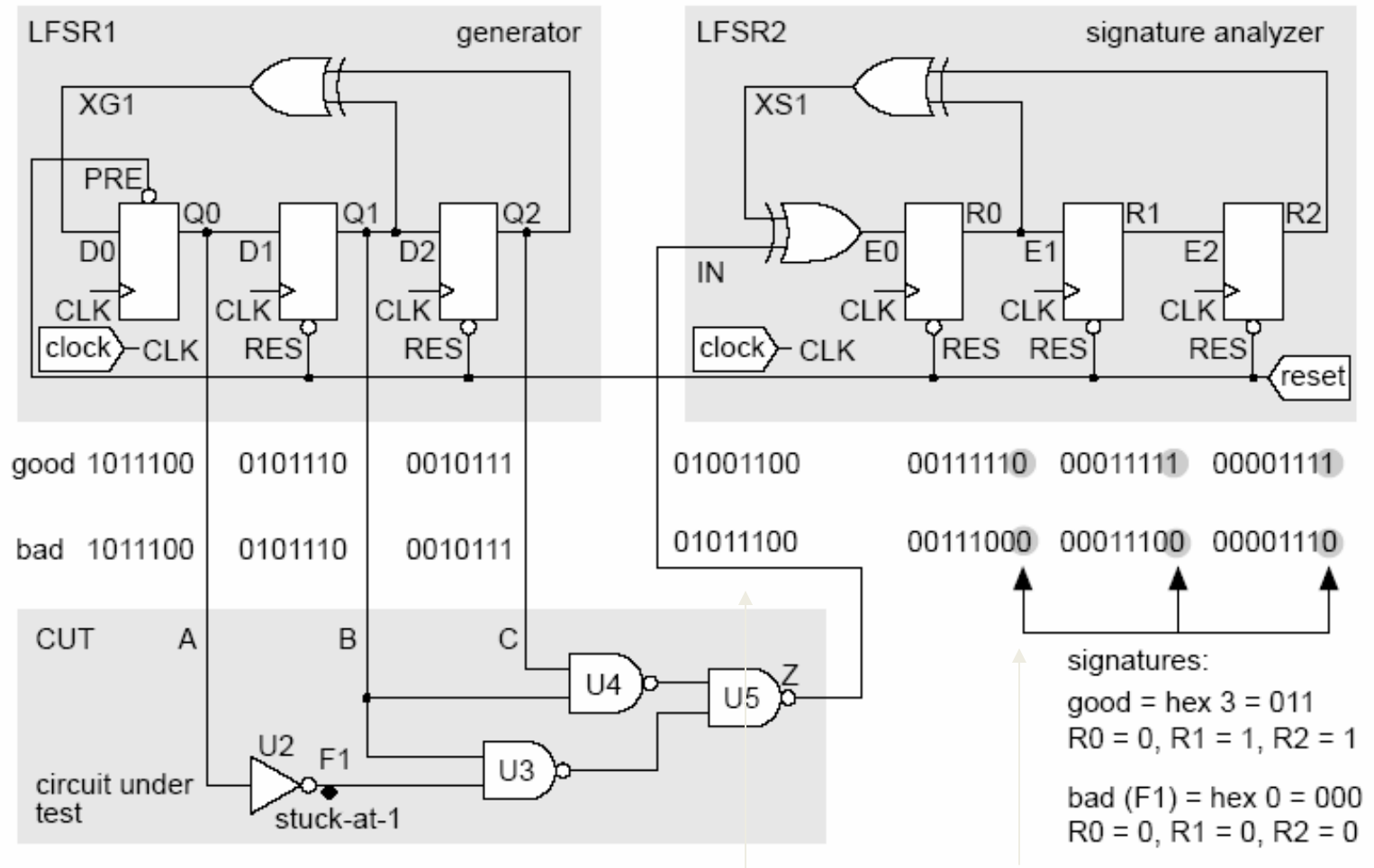
Text figure 14.24

BIST Example (Fig. 14.25)

Pattern generator

Signature analyzer

Generated
test
patterns



Circuit under test

Output
sequences

Signatures

Aliasing

- Good and bad circuits might produce the same signature (“aliasing”) – masking errors
- Previous example:
 - 7-bit sequence applied to signature analyzer
 $2^7 = 128$ possible patterns
 - 3-bit signature register: $2^3 = 8$ possible signatures
 - $128/8 = 16$ streams can produce the good signature: 1 corresponds to good circuit, 15 to faulty circuits
(assume all bit streams equally likely)
 - $128-1 = 127$ streams correspond to bad circuits
 - $15/127 = 11.8\%$ of bad bit streams produce the good signature, and therefore will be undetected
(Probability of missing a bad circuit = 11.8%)

Aliasing – Error Probability

- Given test sequence length L & signature register length R
- Probability of aliasing is:

$$p = \frac{2^{L-R} - 1}{2^L - 1}$$

- For $L \gg R$:

$$p \approx 2^{-R}$$

- Use long sequences to minimize aliasing

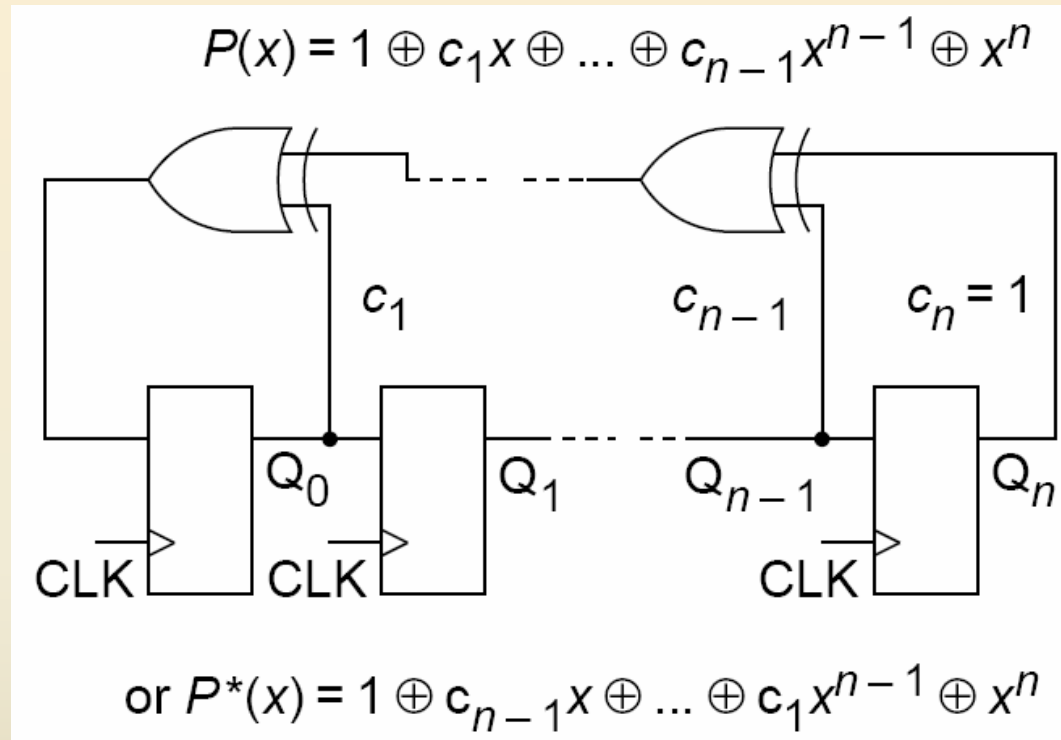
LFSR Theory (chap 14.7.5)

- Operation based on polynomials and Galois-field theory used in coding
- Each LFSR has a “characteristic polynomial”
 - Called a “primitive polynomial” if it generates a maximum-length PRBS
 - General form: $P(x) = c_0 \oplus c_1x^1 \oplus \dots \oplus c_nx^n$
 c_k always 0 or 1, $\oplus = \text{xor}$
 - Reciprocal of $P(x)$ is also primitive:
 $P^*(x) = x^n P(x^{-1})$
- LFSR can be constructed from $P(x)$ or $P^*(x)$

Primitive polynomial examples

- $P(x) = 1 \oplus x^1 \oplus x^3$
 - Order: $n = 3$
 - Coefficients: $c_0=1, c_1=1, c_2=0, c_3=1$
 - LFSR feedback taps: $s = 0, 1, 3$
(non-zero coefficients)
- $P^*(x) = 1 \oplus x^2 \oplus x^3$

“Type 1” LFSR schematic

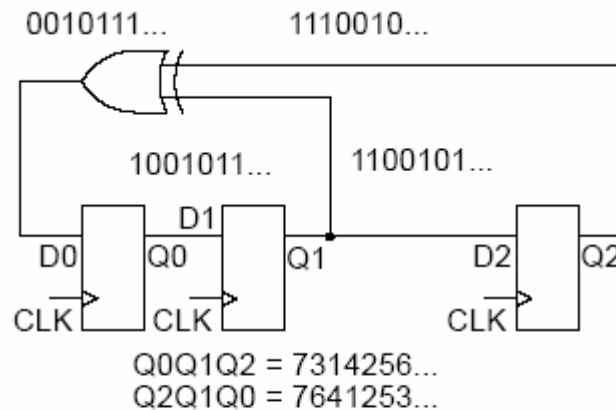


If $ck=1$ add feedback connection & xor gate in position k

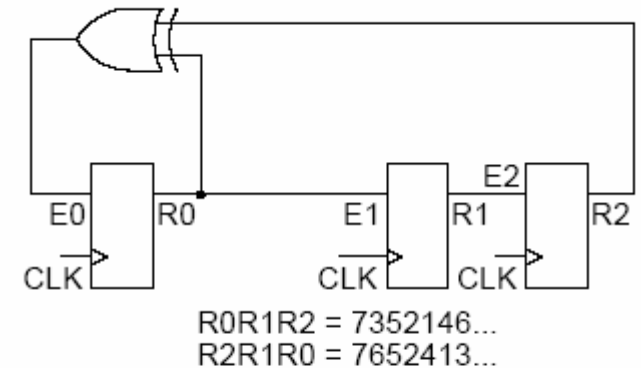
Four LFSR structures for every primitive polynomial

Type 1

- external XOR
- easy to build from existing registers
- Q outputs delayed by 1 clock (test seq's are correlated)



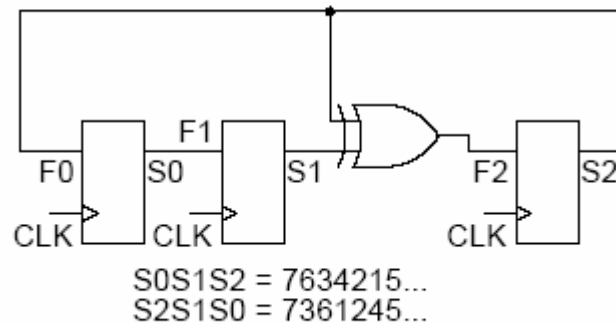
Type 1, $P^*(x)$ (a)



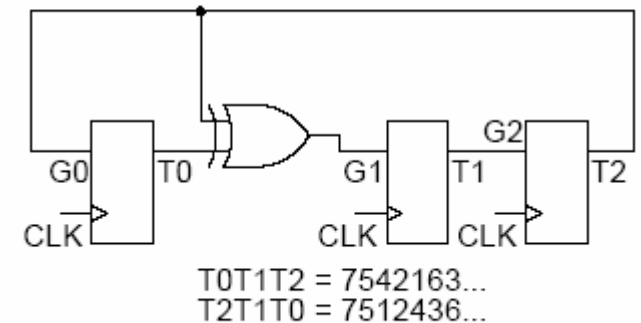
Type 1, $P(x)$ (b)

Type 2

- internal XOR
- fewer series XORs (faster)
- outputs not correlated
- usually used for BIST



Type 2, $P(x)$ (c)



Type 2, $P^*(x)$ (d)

$$P(x) = 1 \oplus x \oplus x^3$$

$$P^*(x) = 1 \oplus x^2 \oplus x^3$$

Common LFSR Configurations

Table 2-1. Common LFSR Configuration

| LFSR Length | Primitive Polynomial | Tap Points (“in”/Type2) | Tap Points (“out”/Type1) |
|--------------------|-----------------------------|--------------------------------|---------------------------------|
| 8-bits | $x^8+x^4+x^3+x^2+1$ | 6, 5, 4 | 4, 3, 2 |
| 16-bits | $x^{16}+x^5+x^4+x^3+1$ | 13, 12, 11 | 5, 4, 3 |
| 24-bits | $x^{24}+x^7+x^2+x+1$ | 23, 22, 17 | 7, 2, 1 |
| 32-bits | $x^{32}+x^{22}+x^2+x+1$ | 31, 30, 10 | 22, 2, 1 |

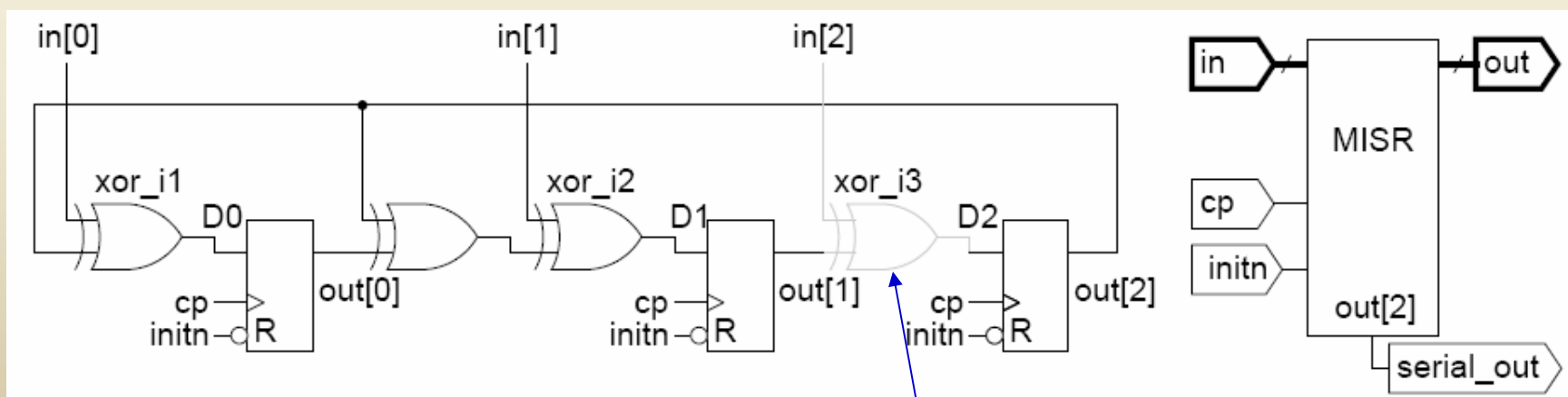
Source: Mentor Graphics “LBISTArchitect Process Guide”

Also see Figure 14.27 and Table 14.11 in the Smith Text

Multiple-Input Signature Register (MISR)

- Reduce test logic by using multiple bit streams to create a signature
- BILBO (built-in logic block observer) – uses MISR as both PRBS generator and signature register

Example: MISR from Type 2 LFSR with $P^*(x) = 1 \oplus x^2 \oplus x^3$

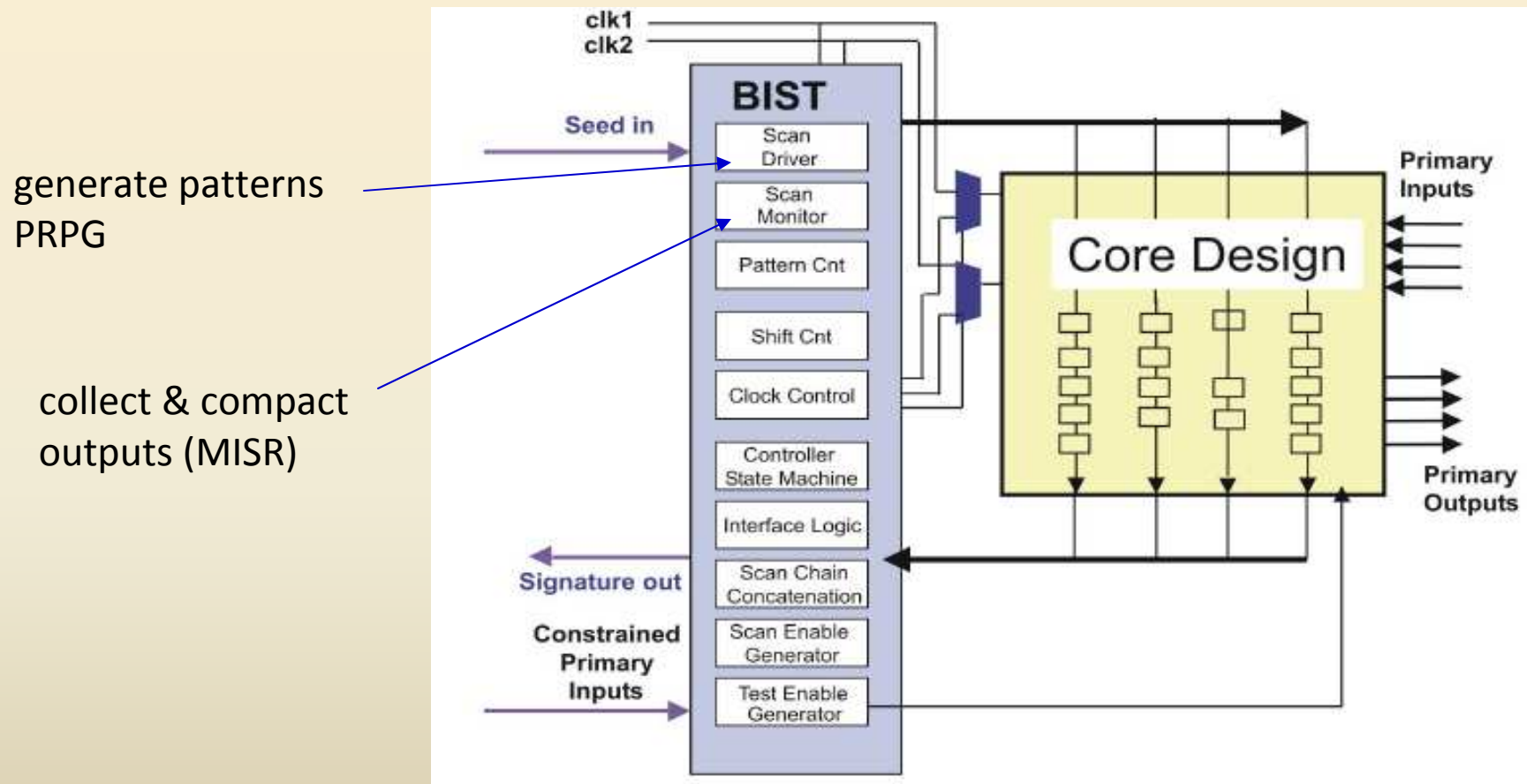


omit xor_i3 if only 2 outputs to test

Mentor Graphics Tools

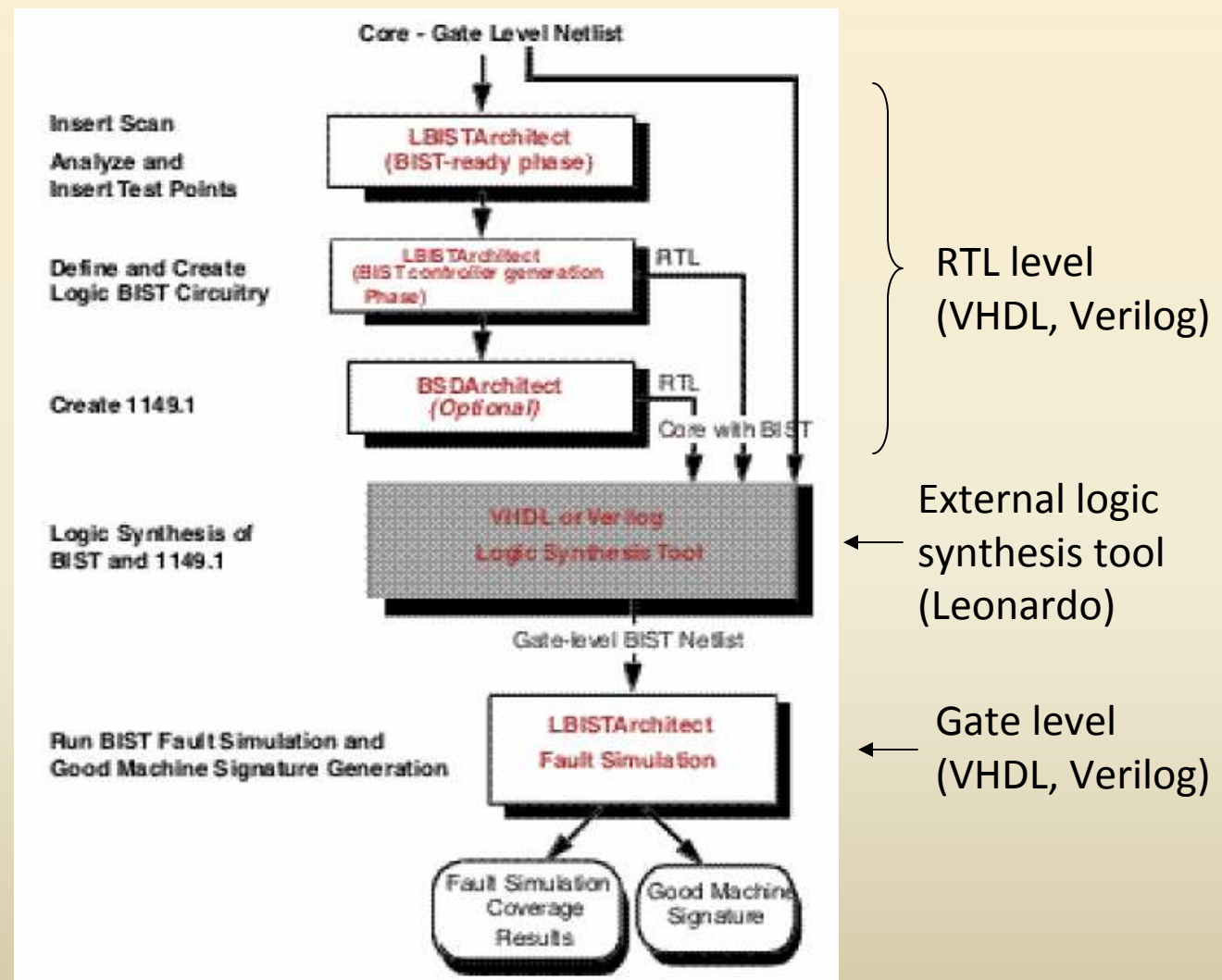
- LBISTArchitect
 - logic BIST design & insertion
 - Reference: “LBISTArchitect Process Guide”
- MBISTArchitect
 - memory BIST design & insertion

Architecture produced by LBISTArchitect



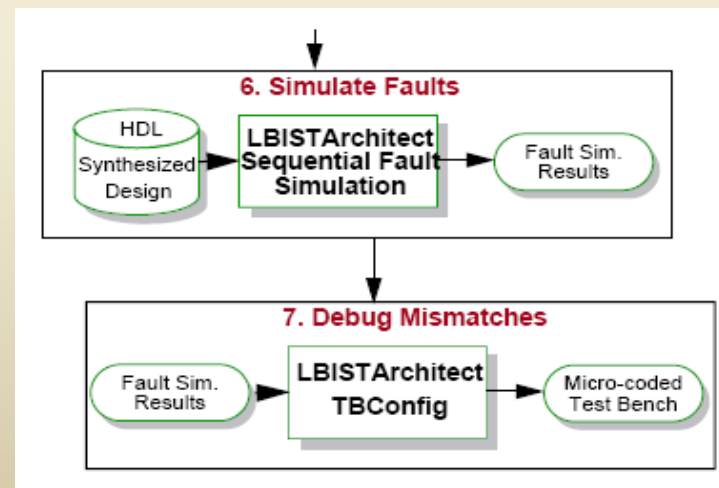
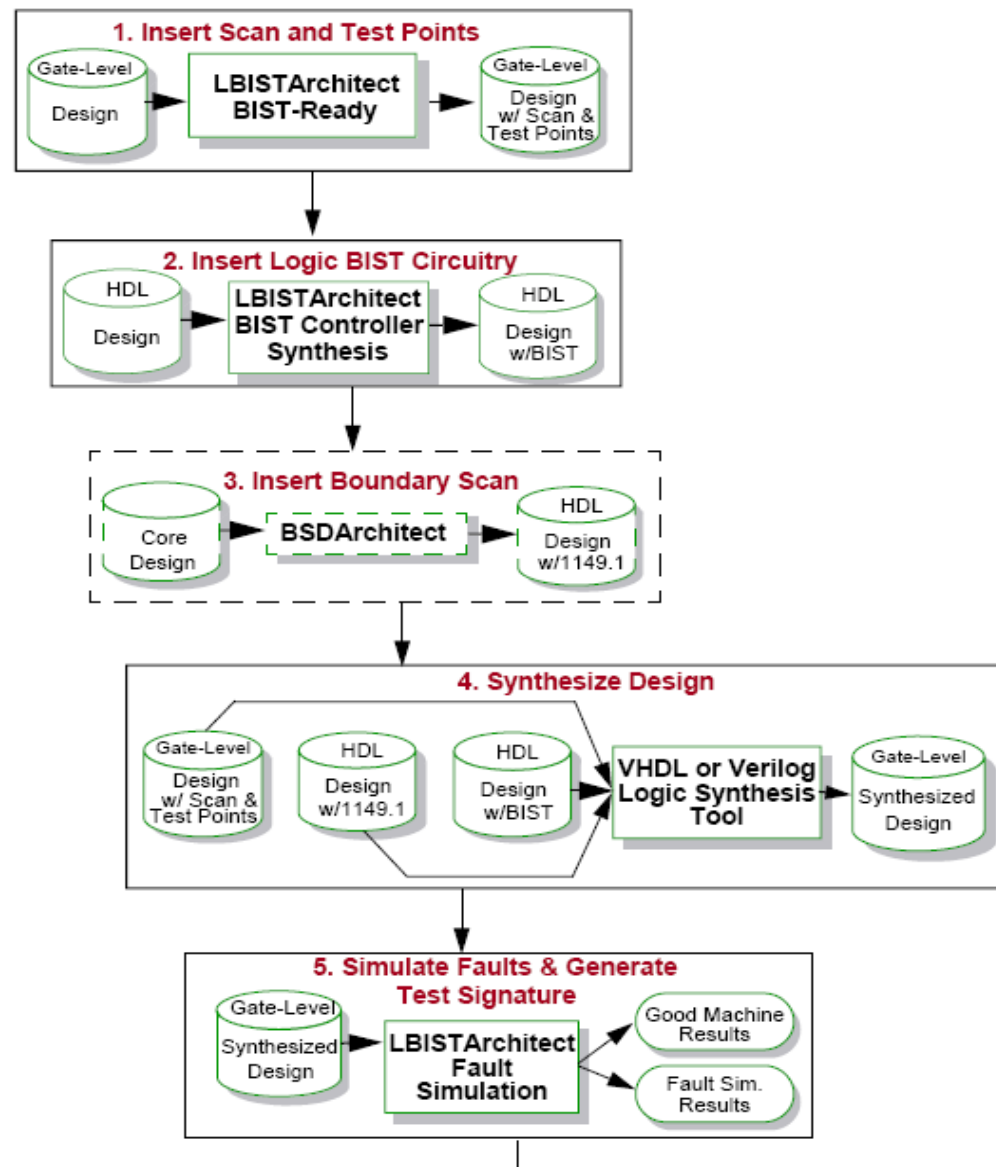
Source: Mentor Graphics "LBISTArchitect Process Guide"

Logic BIST design flow



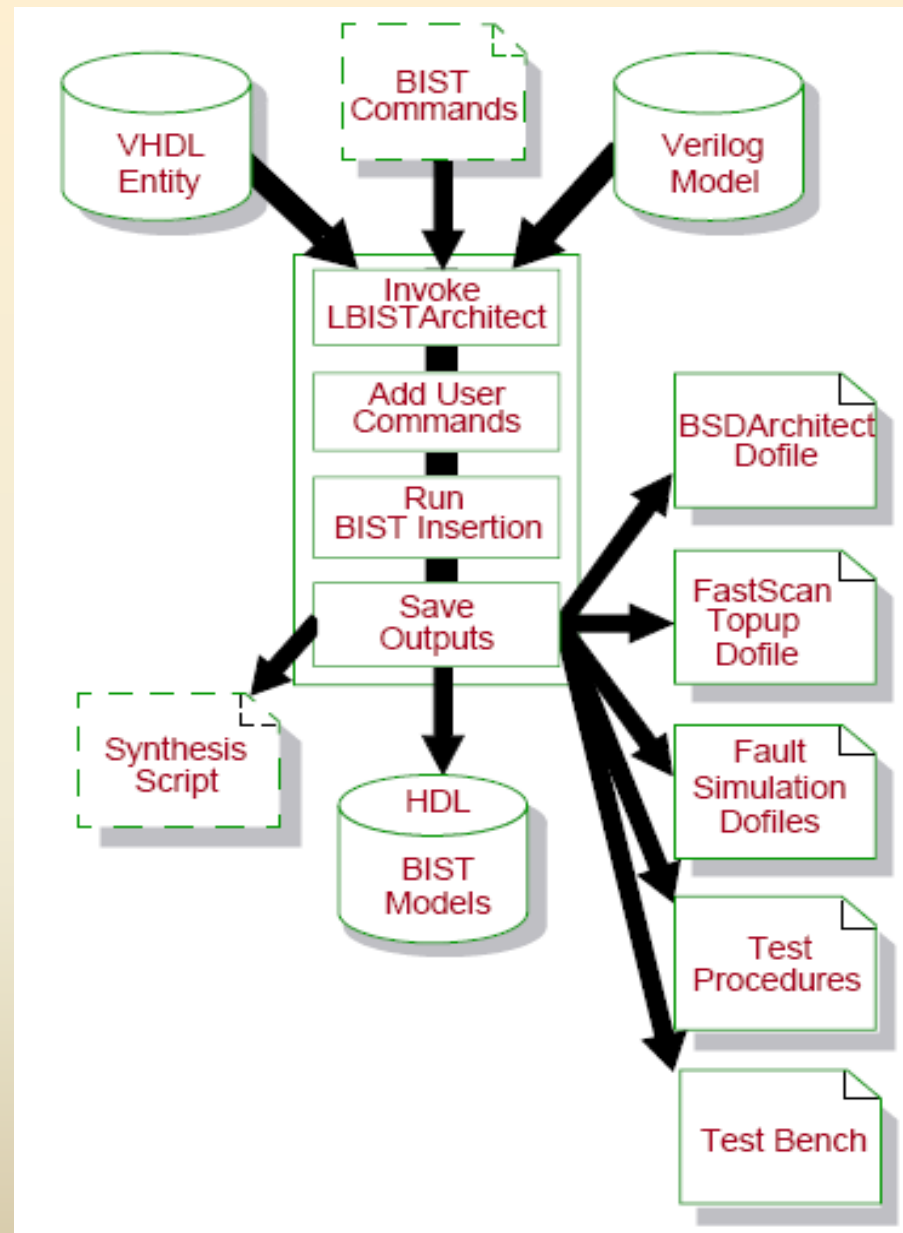
Source: Mentor Graphics "LBISTArchitect Process Guide"

Logic BIST flow



Source: Mentor Graphics "LBISTArchitect Process Guide"

Logic BIST insertion flow



Logic BIST design phases

- BIST-Ready:
 - check design for testability
 - insert scan circuits & test points
- BIST Controller Generation:
 - produce synthesizable RTL model (VHDL, Verilog)
 - includes scan driver/PRPG, scan monitor/MISR
- Boundary Scan Insertion (optional)
 - BSDarchitect can tie 1149.1 to logic BIST
 - inserts boundary scan ckts & TAP controller

LOGIC BIST design phases (2)

- Fault simulation & signature generation
 - determine fault coverage of BIST patterns
 - generate signature of “good circuit”
- Sequential fault simulation (optional)
 - determine fault coverage of BIST hardware
- BIST verification (optional)
 - generate test bench for full simulation
- Manufacturing diagnostics (optional)
 - generate info to assist in fault diagnosis

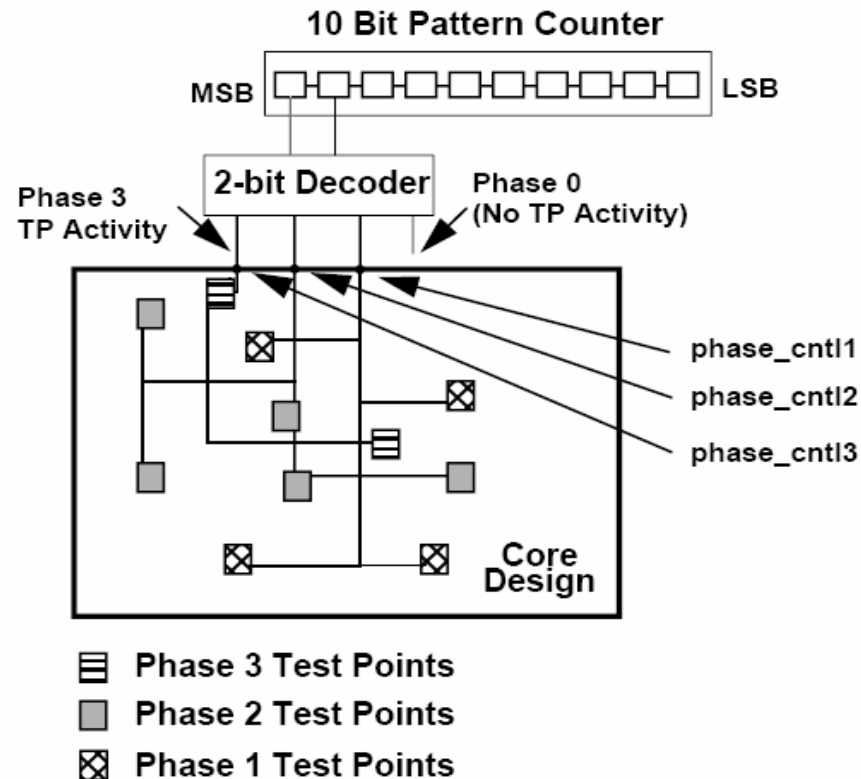
BIST-ready phase: test point insertion

- Add control test points to gain access to inputs of difficult-to-test gates
- Add observe test points to gain access to outputs of difficult-to-test gates
- MTPI: Multiphase Test Point Insertion
 - break test into phases (ex. 256 patterns each)
 - activate only test points used in a phase
 - add points to improve detection of faults not detected in previous test phases

MTPI Example

Table 2-2. Four-Phase Test Point Control Activity

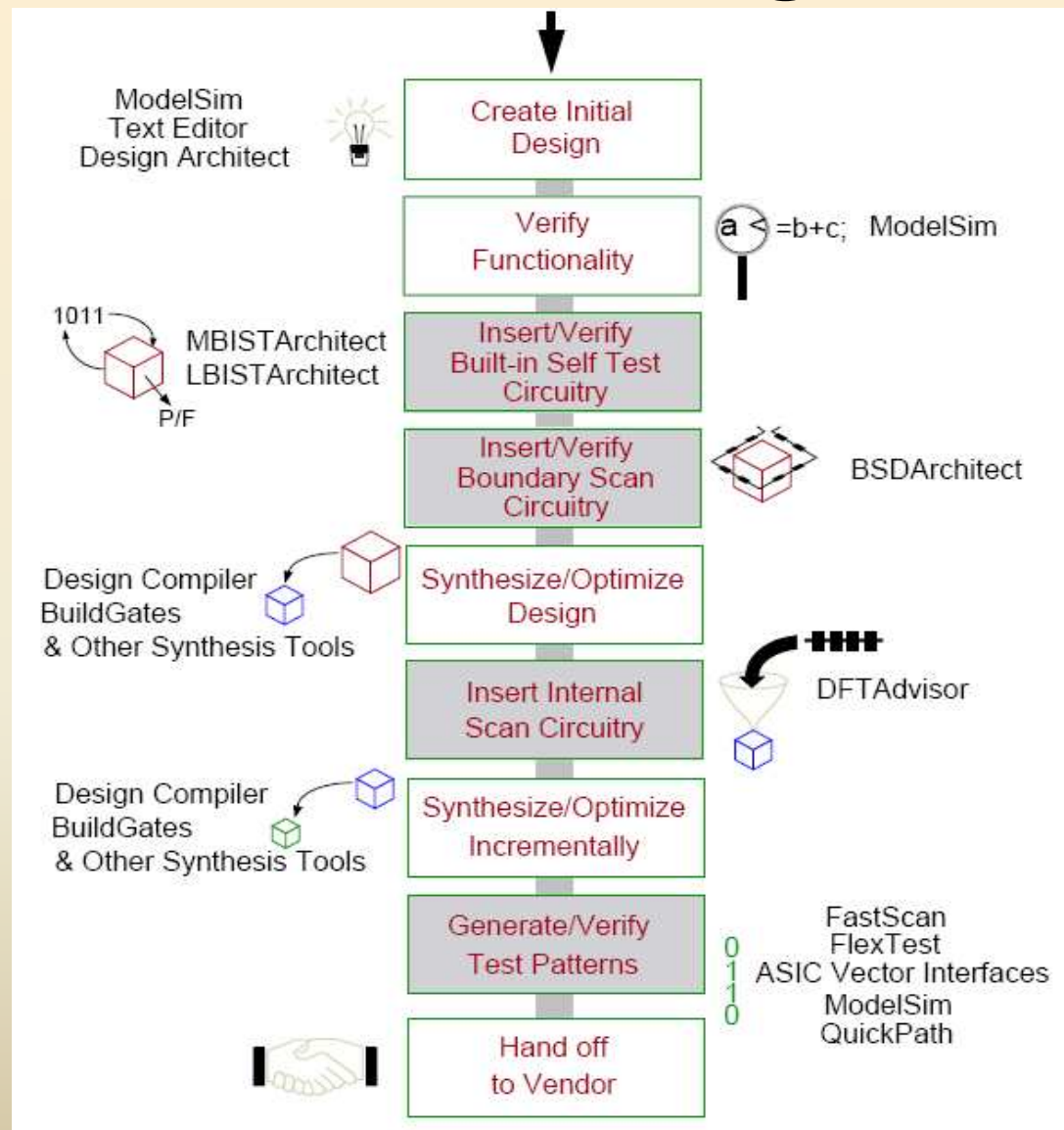
| Phase | Patterns | Activity |
|-------|----------|-----------------------|
| 0 | 1—256 | No active TP controls |
| 1 | 257—512 | 1st TP control active |
| 2 | 513—768 | 2nd TP control active |
| 3 | 769—1024 | 3rd TP control active |



Boundary Scan

Smith Text: Chapter 14.2

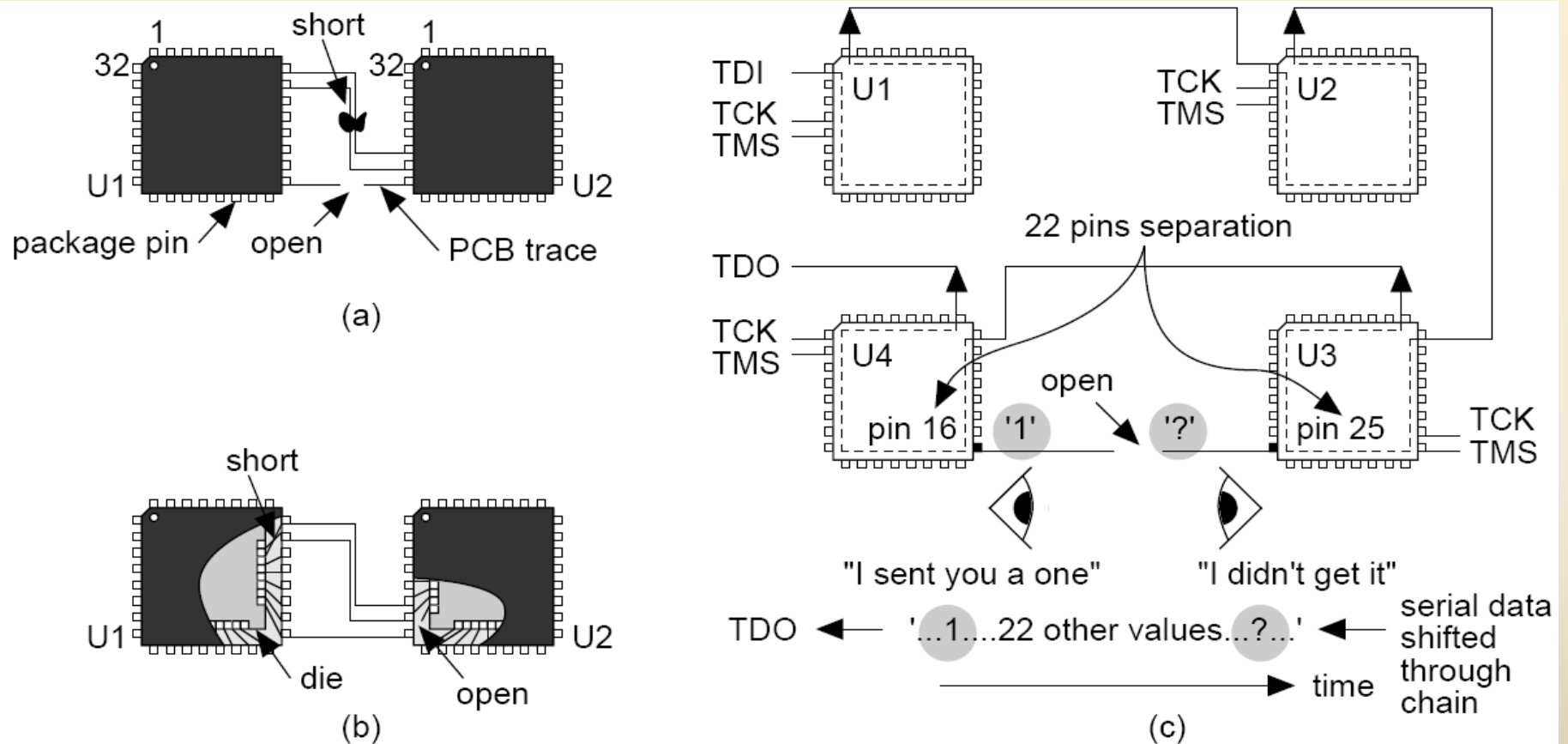
Top-down test design flow



Boundary-Scan Test

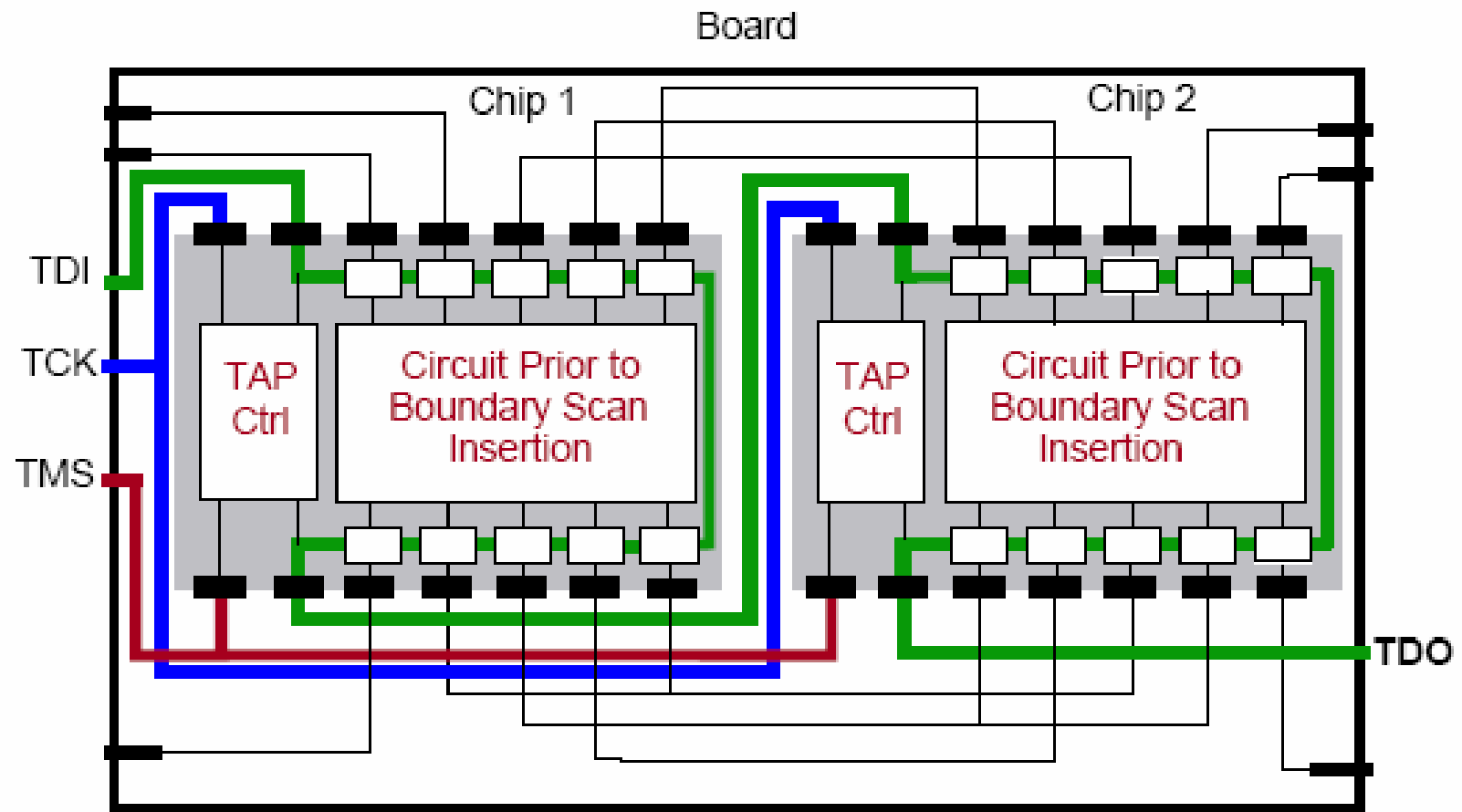
- JTAG (Joint Test Action Group) test standard became IEEE Standard 1149.1 “Test Port and Boundary-Scan Architecture”
- Allows boards to be tested via 4 wires:
 - TDI (test data input)
 - TDO (test data output)
 - TCK (test clock)
 - TMS (test mode select)
 - TRST (test reset) is optional
- Test data supplied serially via TDI & results checked via TDO, under control of TMS/TCK

Use of boundary scan to detect shorts/opens between ICs



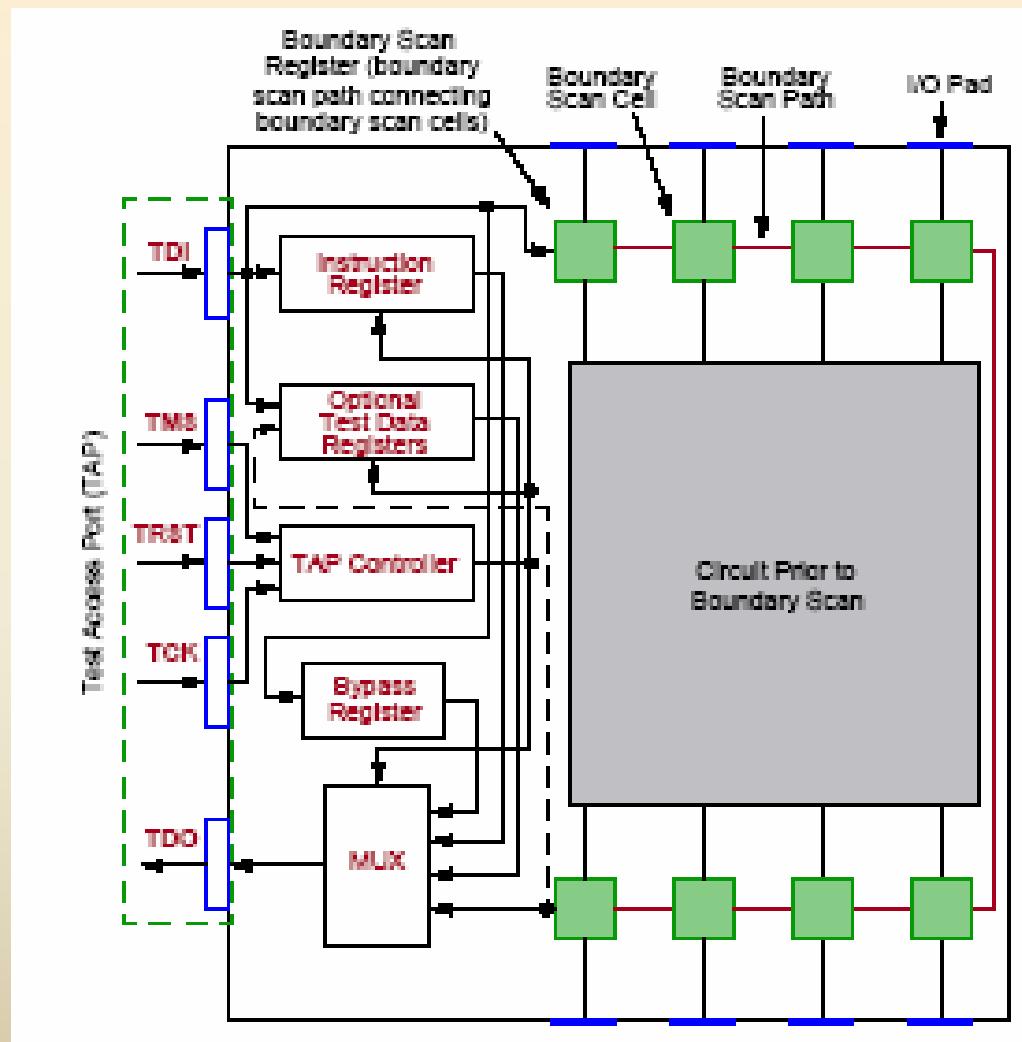
Smith text figure 14.1

JTAG/IEEE 1149.1 Boundary Scan Basic Structure



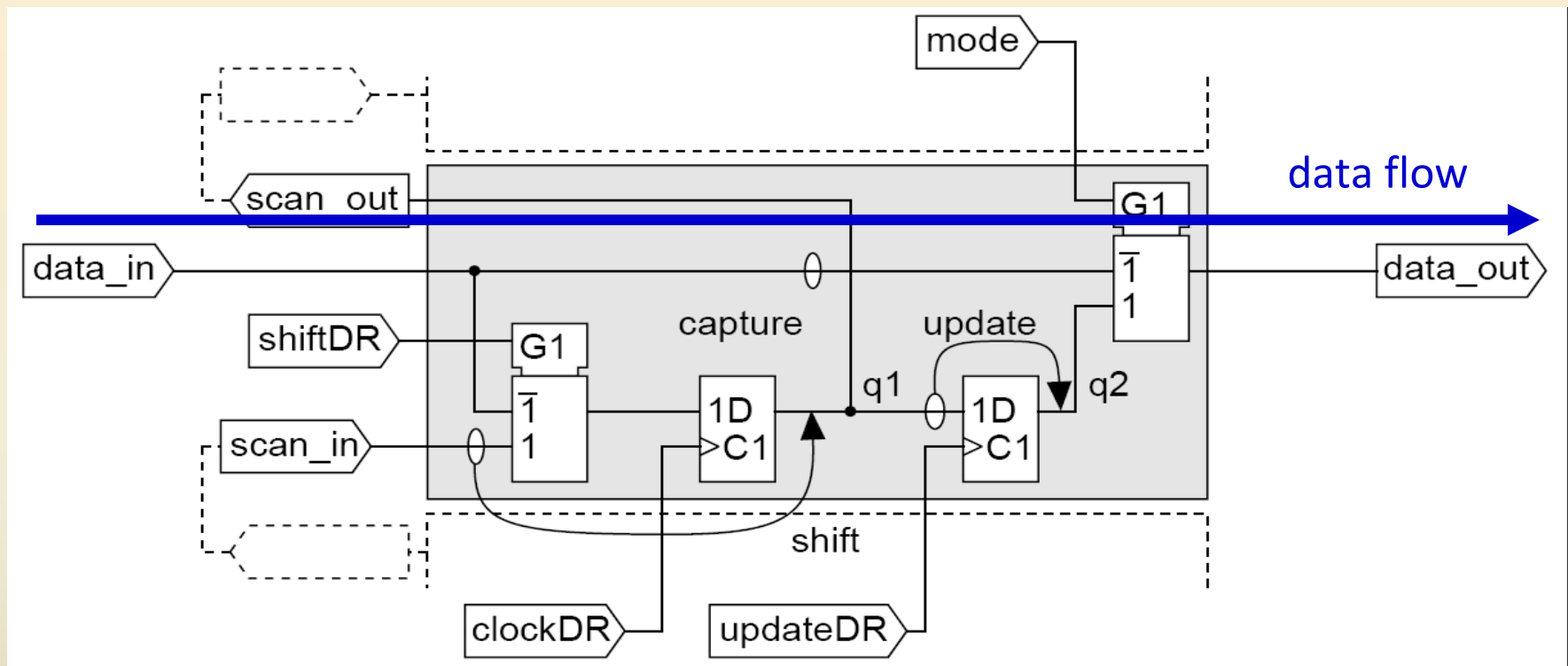
Source: Mentor Graphics "Boundary Scan Process Guide"

Chip-level boundary scan architecture



Source: Mentor Graphics "Boundary Scan Process Guide"

Data register (boundary) cell



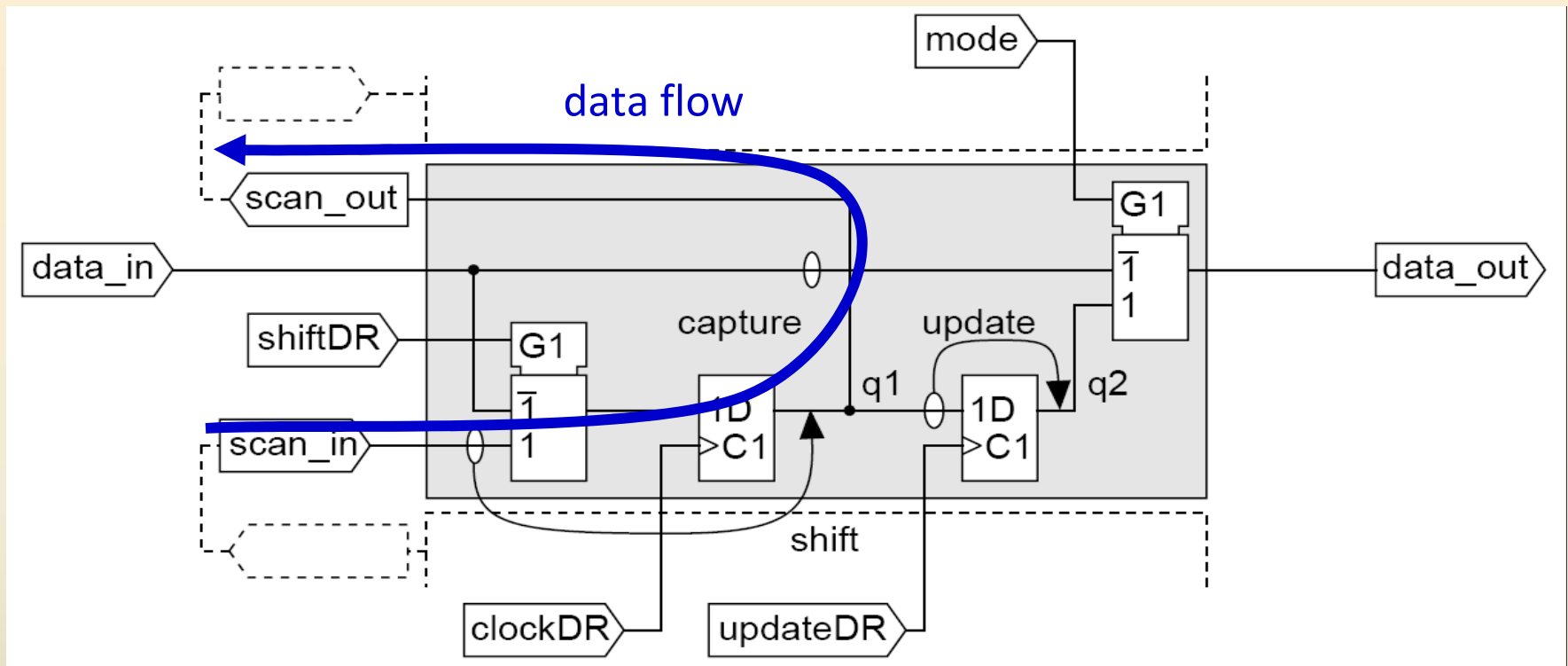
Normal mode: data_in to data_out (**mode=0**)

* Chip input pin: data_in from board, data_out to chip

* Chip output pin: data_in from chip, data_out to board

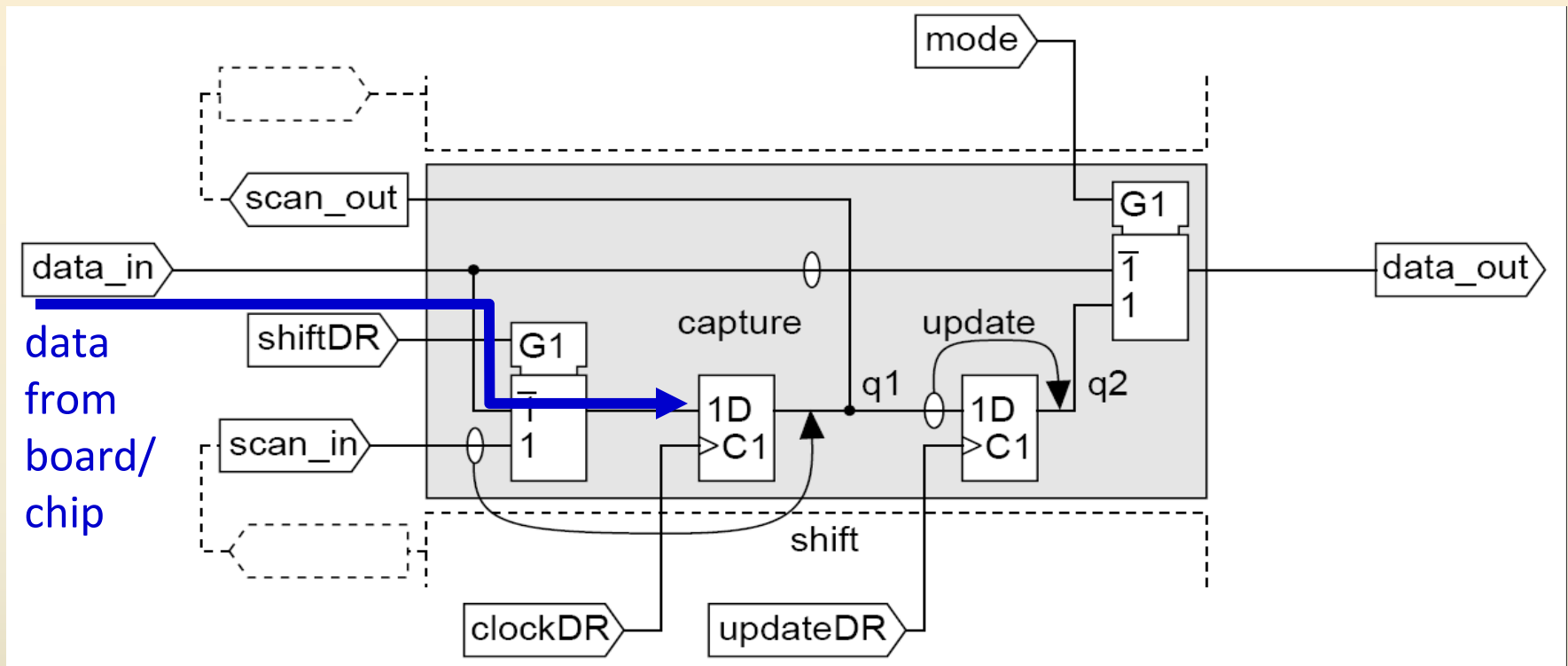
Also used in "Bypass" mode

Data register (boundary) cell



Scan mode: scan_in to capture FF, capture FF to scan_out
shiftDR=1 & clockDR pulse
TDI drives first scan_in signal in chain
Last scan_out in chain drives TDO

Data register (boundary) cell



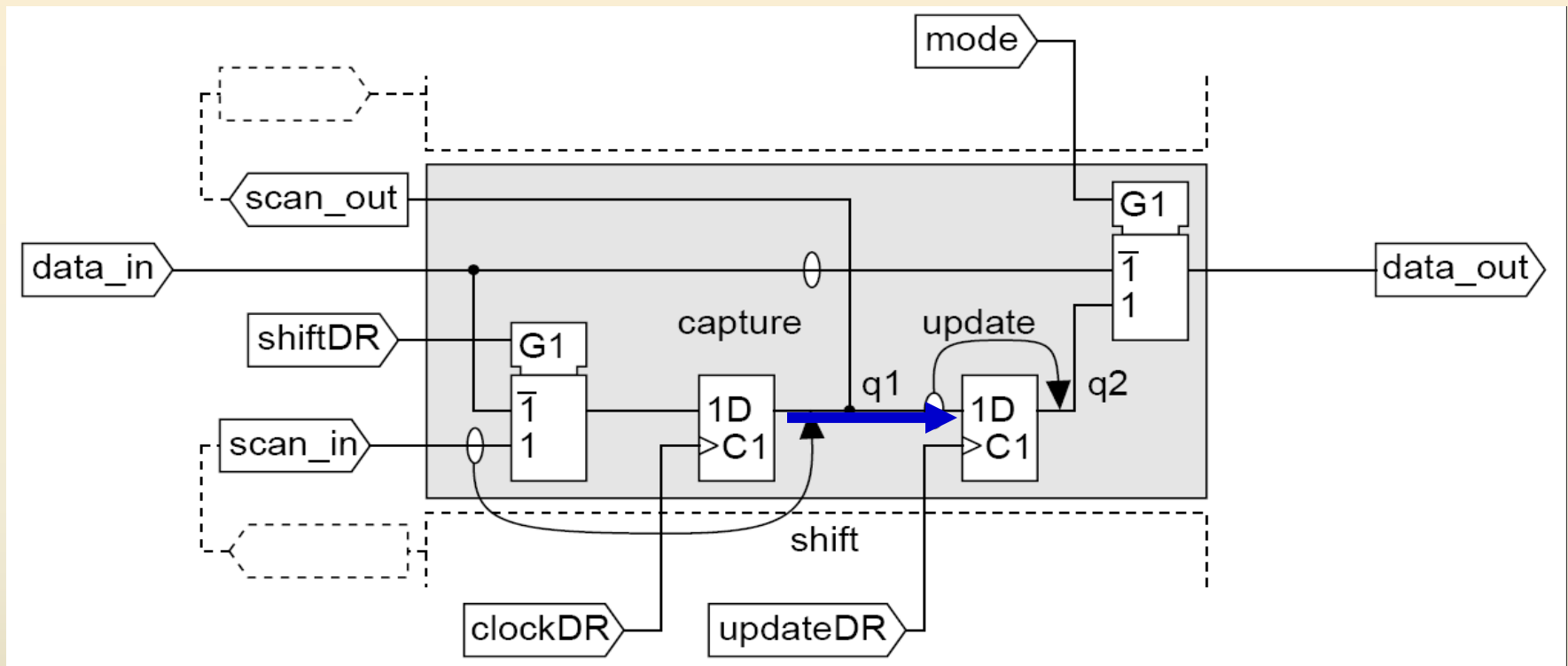
Capture mode: data in captured in “capture FF”

shiftDR=0 & clockDR pulse

data_in from board (extest) – chip input pin

data_in from chip (intest) – chip output pin

Data register (boundary) cell

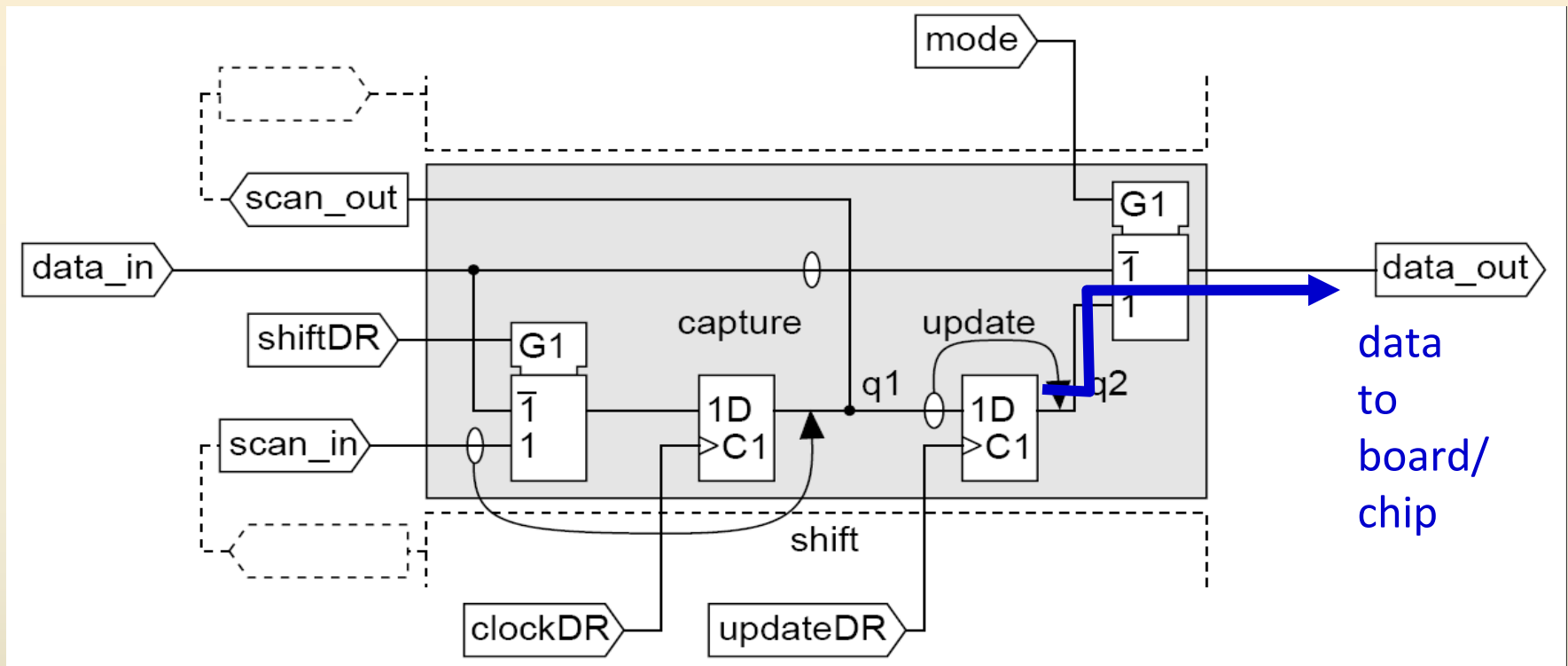


“Update Mode”: data from capture FF to update FF

updateDR=1

Save scan chain values in update FFs to apply to data_out later during EXTEST/INTTEST

Data register (boundary) cell



Drive mode: update FF to data_out

mode=1

data_out to board (extest) – chip output pin

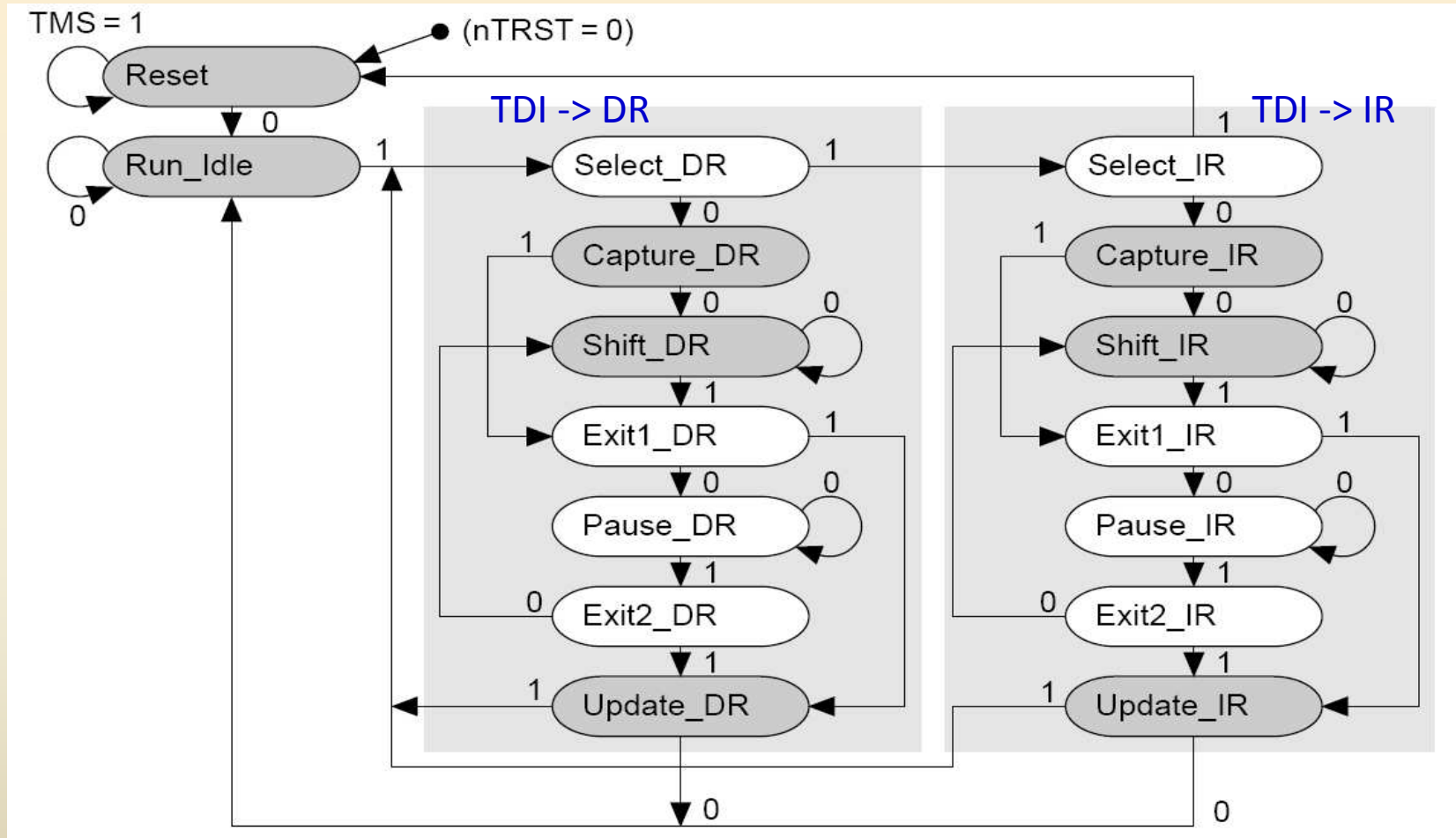
data_out to chip (intest) – chip input pin

Boundary-scan instructions

- EXTEST
 - external test of chip-chip connections
- SAMPLE/PRELOAD
 - sample values from input pads during capture
 - preload BSC update register during update
- BYPASS
 - scan data through 1-cell bypass register
 - other BSC's pass data_in to data_out

Load/decode in Instruction Register

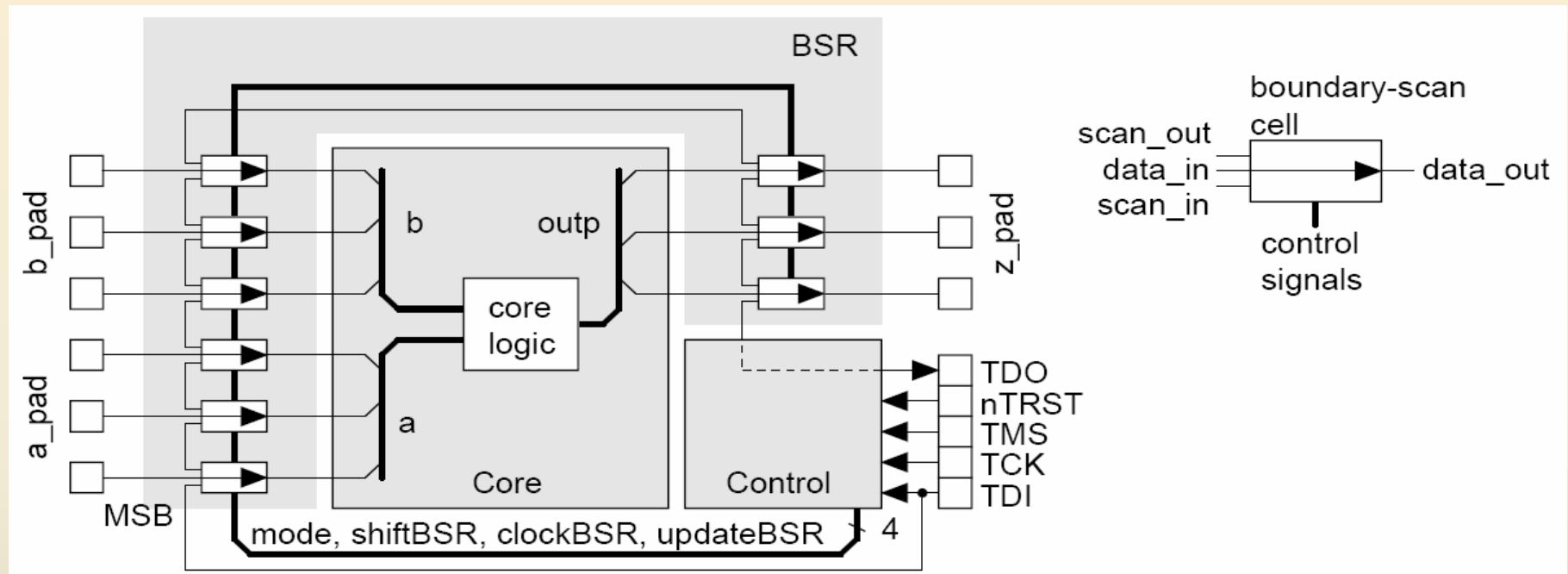
TAP controller state diagram



State changes controlled by TMS & TCK

Smith Text: Figure 14.7

Boundary-scan example



Smith Text: Figure 14.9

Boundary-scan tools

- Mentor Graphics “BSDArchitect”
 - synthesize boundary-scan circuits
 - insert boundary-scan circuits
 - generate boundary-scan test vectors
 - generate VHDL test bench
- BSDL
 - Boundary-Scan Description Language
 - Subset of VHDL - describes features of IEEE 1149.1
 - Use in test generation software