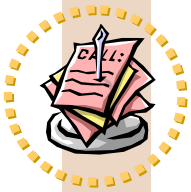
Design for Testability with DFT Compiler and TetraMax



黃信融

Hot Line: (03) 5773693 ext 885

Hot Mail: hotline@cic.org.tw

Outline

Day 1 – DFT Compiler

- Basic Concepts
- DFT Compiler Flow
- Basic DFT Techniques
- Advanced DFT Techniques
- Scan Path Insertion
- Memory Wrapper & AutoFix
- Other Skill

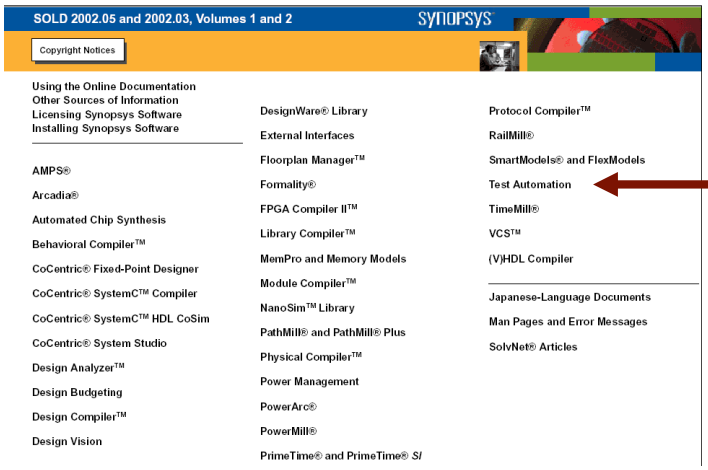
Day 2 – TetraMAX

- TetraMAX Overview
- Design and Test Flows
- STIL for DRC & ATPG
- TetraMAX Fault Classes & Fault list
- Fault Simulating Functional Patterns
- Modeling Memories
- Debugging Problems
- Something To Remember

Synopsys On-Line Documentation

- Invoke Synopsys On-Line Documentation (SOLD) by using this command:

```
%> acroread /usr/synopsys/sold/cur/top.pdf
```

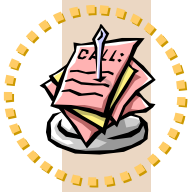


Test Automation

What is Design-for-Testability?

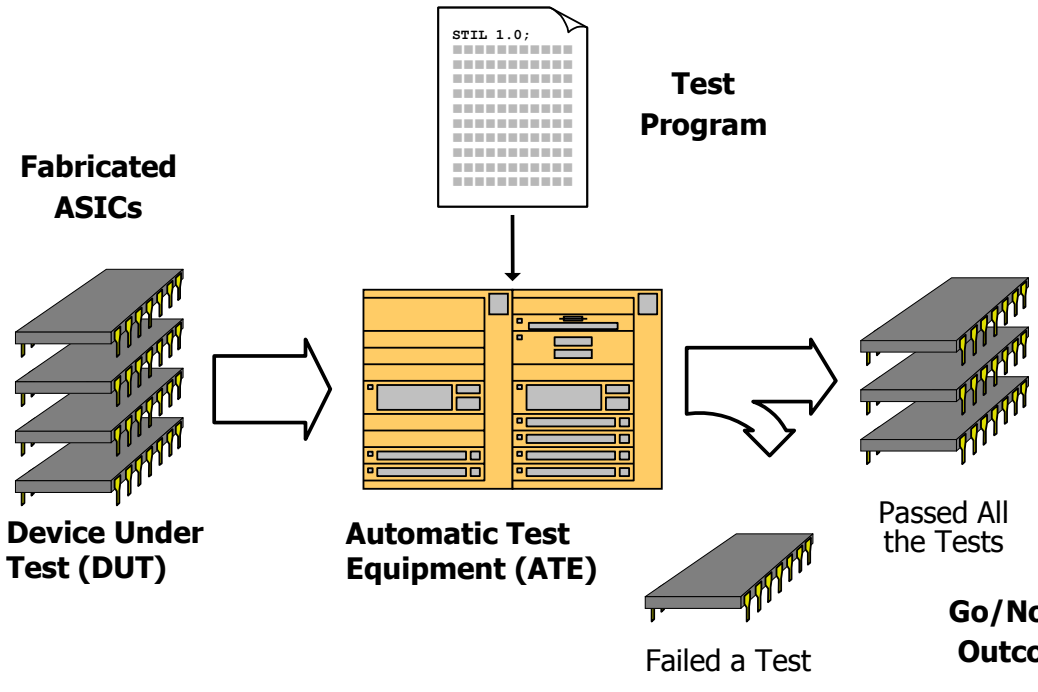
- Extra design effort invested in making an IC testable.
- Involves inserting or modifying logic, and adding pins.
- You can apply **structured** or **Ad hoc DFT** techniques.
- Structured:
  - Insert scan-chain
  - Memory BIST…
- Ad Hoc DFT:
  - Avoid asynchronous logic feedbacks.
  - Make flip-flops initializable.
  - Avoid gates with a large number of fan-in signals.
  - Provide test control for diffcult-to-control signals.

Design for Testability Using DFT Compiler

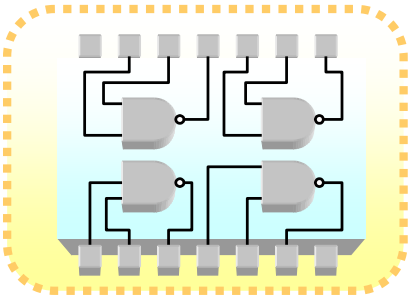


Basic Concepts

Introduction -- Modern IC Testing

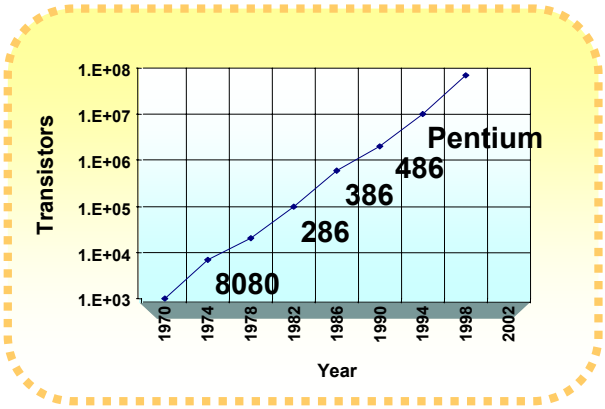


Introduction -- Pins/Gates



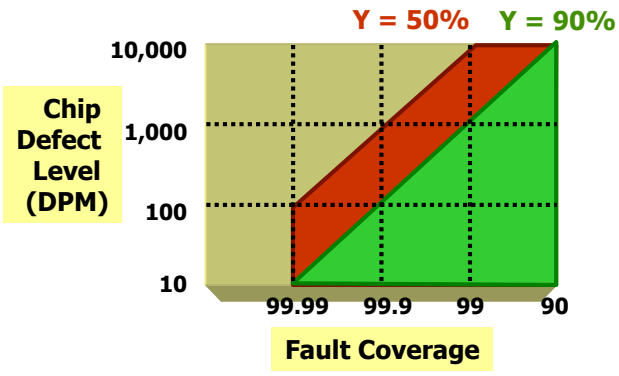
TTL logic allowed easy access to individual gates.

Pins / Gates << 0.001
Hard to access to a chip.



Introduction -- Defect Level & Fault Coverage

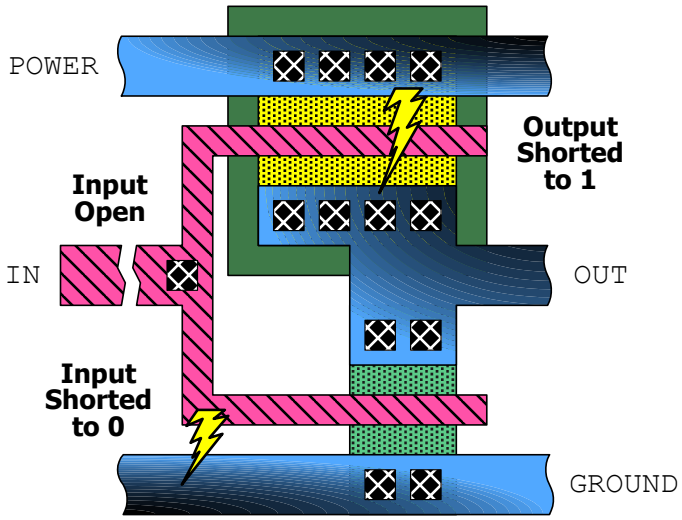
- Defect level (DL) is the fraction of devices that pass all the tests and are shipped but still contain some faults
  - DL = $1-Y^{(1-FC)}$
- Defect level is measured in terms of DPM (detects per million), and typical requirement is less than 200 DPM i.e. 0.02 %



For DL = 200 DPM

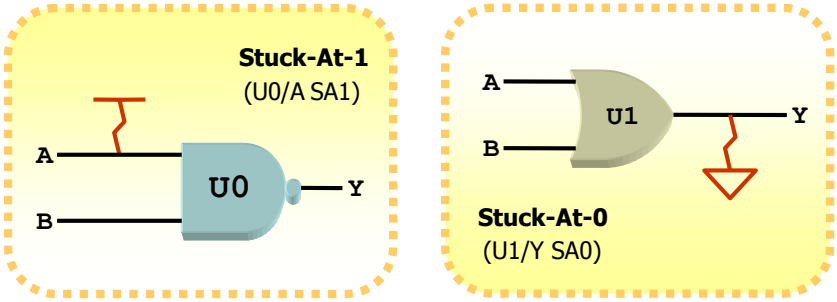
Y (%)	10	50	90	95	99
FC(%)	99.99	99.97	99.8	99.6	98

Introduction -- Physical Defect

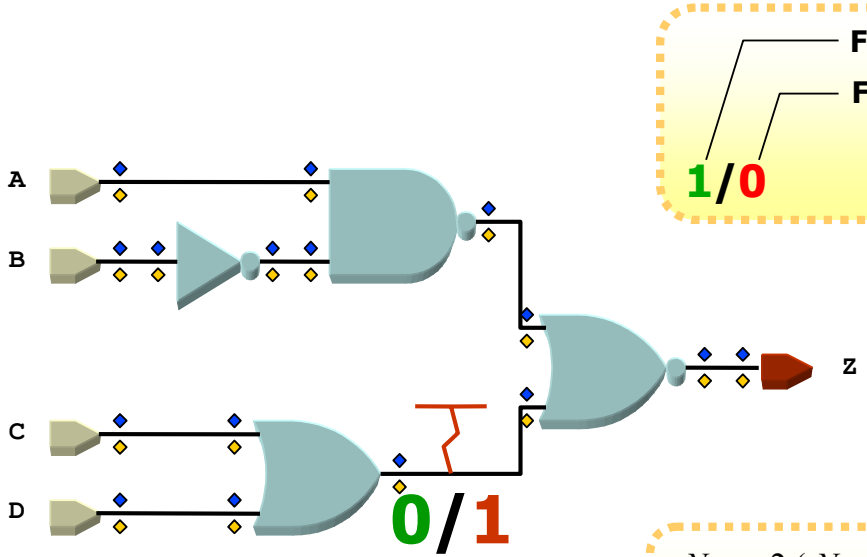


Introduction -- Stuck-At Faults

- Simple logical model is **independent of technology** details.
- It reduces the complexity of fault-detection algorithms.
- Applicable to **any** physical defect manifesting as a signal that is stuck at a fixed logic level.
- One stuck-at fault can model **more** than one kind of defect.



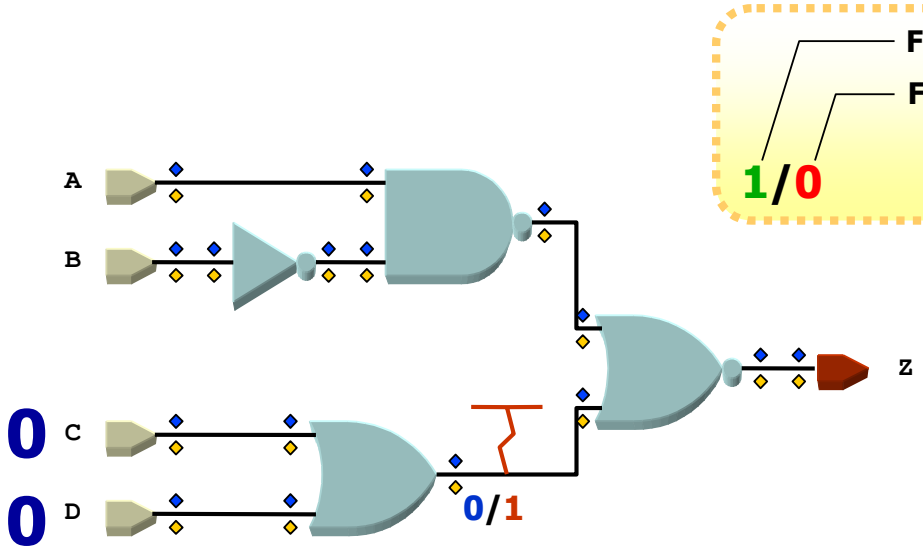
Stuck-At Faults Example@



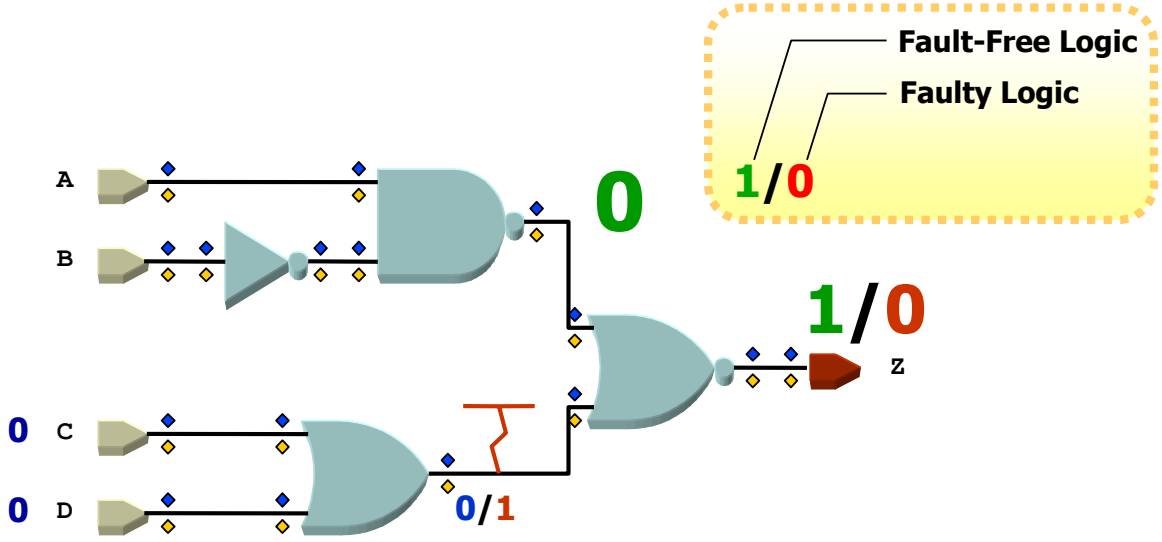
Total Faults

$$N_f = 2(N_{pins} + N_{ports}) = 2(11 + 5) = 32$$

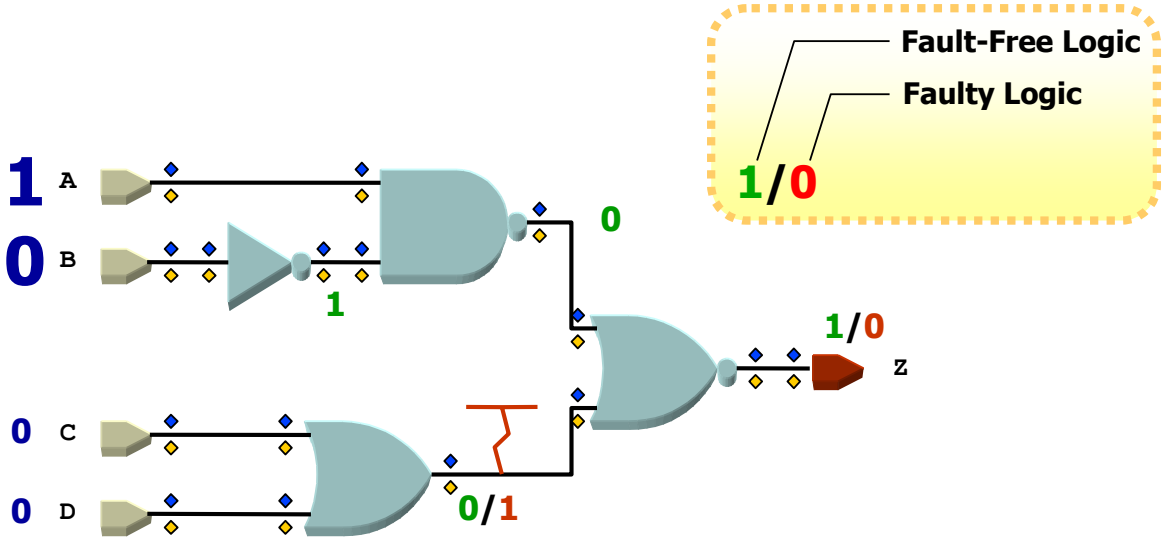
Stuck-At Faults Example@@



Stuck-At Faults Example@@@



Stuck-At Faults Example@@@@



Introduction -- D Algorithm

D Algorithm:

1. Target a specific stuck-at fault.
2. Drive fault site to **opposite** value.
3. Propagate error to primary output.
4. Record pattern; go to next fault.

- Generates patterns for one SA fault at a time.
- Involves decision-making at almost every step.
- Backtracking can be excessive for difficult-to-detect faults.
- Use an ATPG tool which relies on proprietary techniques to speed up and extend the basic D algorithm, by predicting the best paths, etc.

Introduction -- Test Pattern

Expected Output

Vector { All = 10010HLHH ; }

Input Stimulus

Test Program

```
STIL 1.0;
    •
Pattern "Burst" {
  Waveform Simple;
    •
  Vector {■■■■■■■■■}
  Vector {■■■■■■■■■}
  Vector {■■■■■■■■■}
  Vector {■■■■■■■■■}
  Vector {ALL=1000H;}
  Vector {■■■■■■■■■}
  Vector {■■■■■■■■■}
}
```

Detects the target SA0 fault.

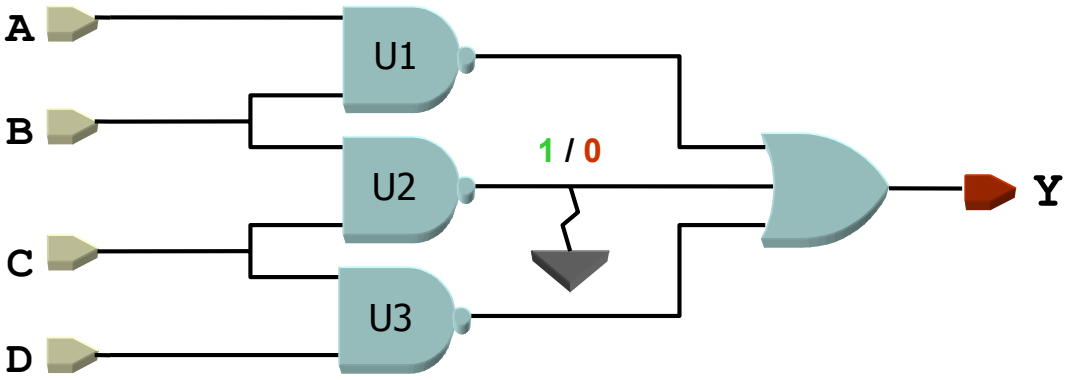
Introduction -- Fault Collapsing

- The gate behaves the same for any input SA0, SA1 or the output SA0, SA1.
- The faults { A SA0, Z SA0 }, { A SA1, Z SA1 } are thus all equivalent.
- Only Z SA0, Z SA1 or A SA0, A SA1 need be included in the fault universe.
- The faults after collapsing are included in the fault universe called **primary faults**, faults after collapsing are not included in the fault universe called **equivalent faults** of primary faults.

Introduction -- Single Stuck-At Fault Assumption

- ATPG tools assume at most **one stuck-at fault** is present on the chip under test.
- This assumption is made to **simplify** the analysis—detecting **multiple** faults would complicate ATPG.
- The SSF model disregards the possible presence of **any other fault** affecting the test for a target fault.
- SSF assumes no chance of another fault **masking** the target fault, making it impossible to detect.

Introduction -- Undetectable Fault



No pattern can be devised to detect fault **U2 / Z SA0**

Introduction -- Fault Coverage V.S Test Coverage

$$\textbf{Fault Coverage} = \frac{\text{number of detected faults}}{\text{total number of faults}} \leq 1$$

Default : 50%

$$\textbf{Test Coverage} = \frac{DT + (PT * posdet\_credit)}{\text{all faults} - (UD + (AU * au\_credit))}$$

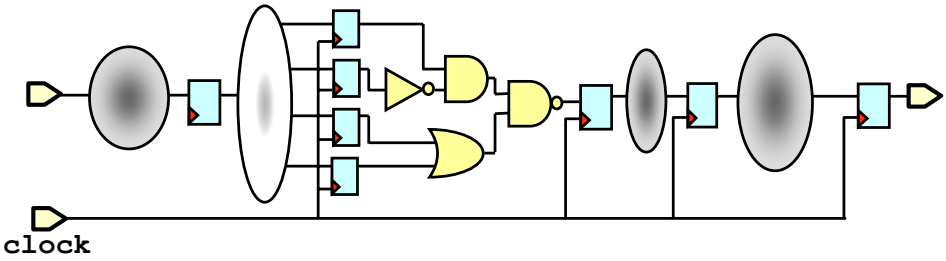
Default : 0%

Sequential Logic -- Harder to Test

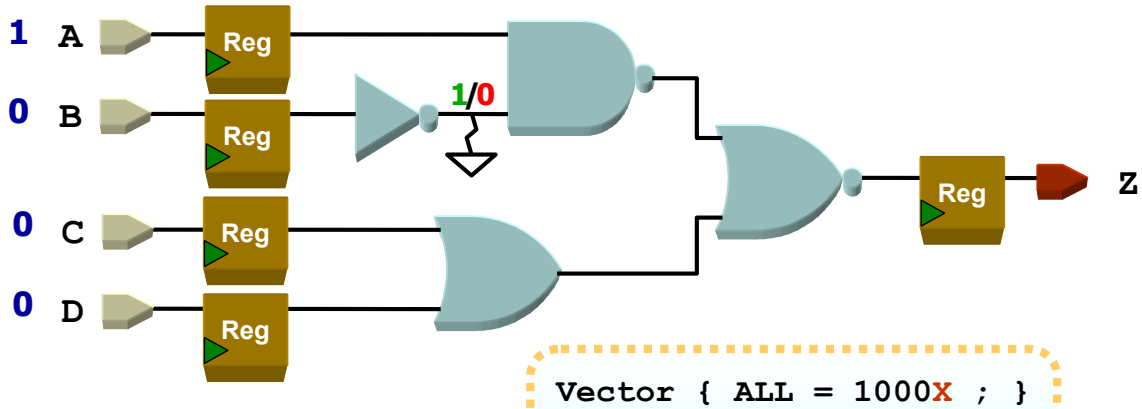
To detect a stuck-at fault in synchronous **sequential** logic, we can still use the familiar D algorithm, but it'll take….

- One or more clock cycles to **activate** the fault.
- One or more clock cycles to **propagate** the fault effect.

In general, we'll need a **sequence** of patterns to detect a fault!



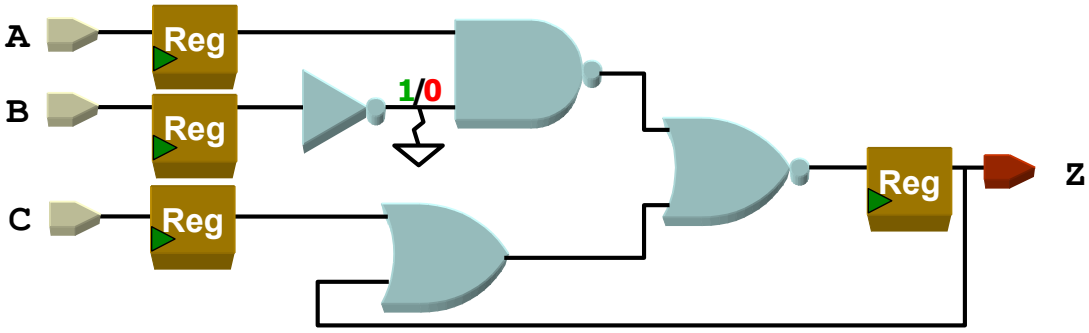
Sequential Logic -- Example 1




```
Vector { ALL = 1000X ; }
Vector { ALL = XXXXX ; }
Vector { ALL = XXXXH ; }
```

This takes three patterns.
In general, detecting a stuck-at fault insequential logic requires a sequence of one or more test patterns.

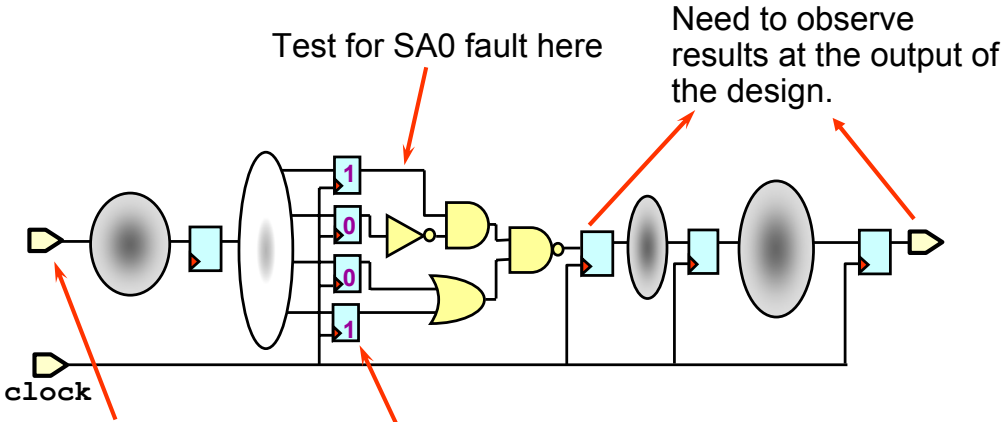
Sequential Logic -- Example 2 (feedback)




```
Vector { ALL = XX1X ; }
Vector { ALL = 100X ; }
Vector { ALL = XXXX ; }
Vector { ALL = XXX1 ; }
```

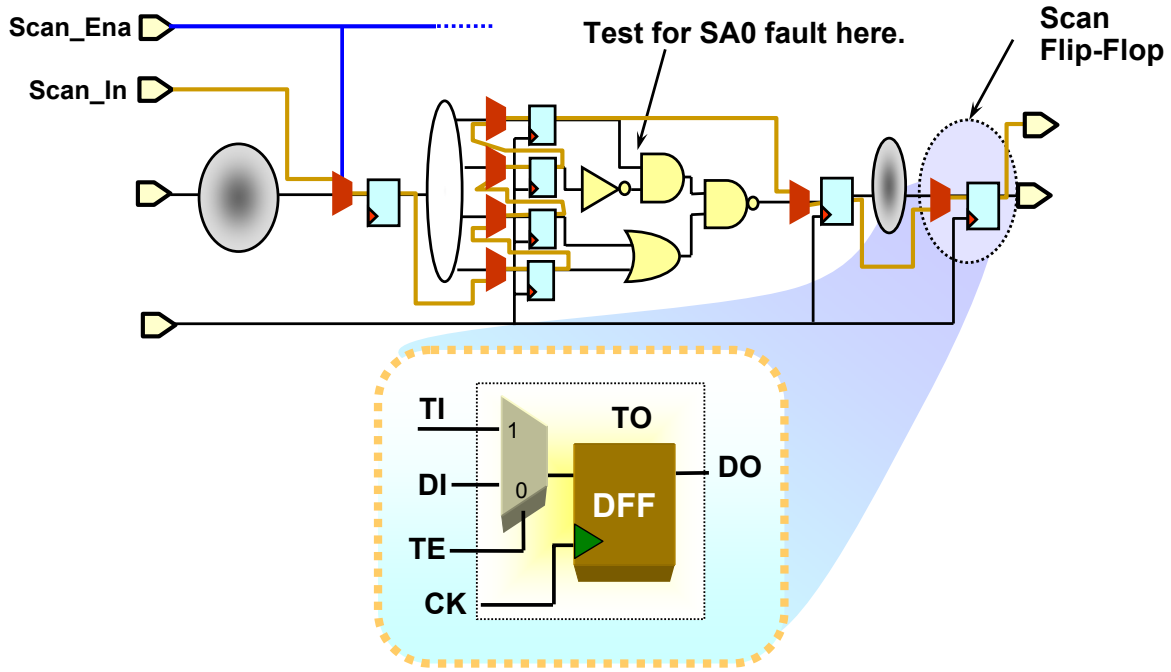
This takes four patterns.
Detecting faults in sequential logic can thus become very costly in terms of ATPG run time and program length.

Sequential Logic -- Test SOCs?

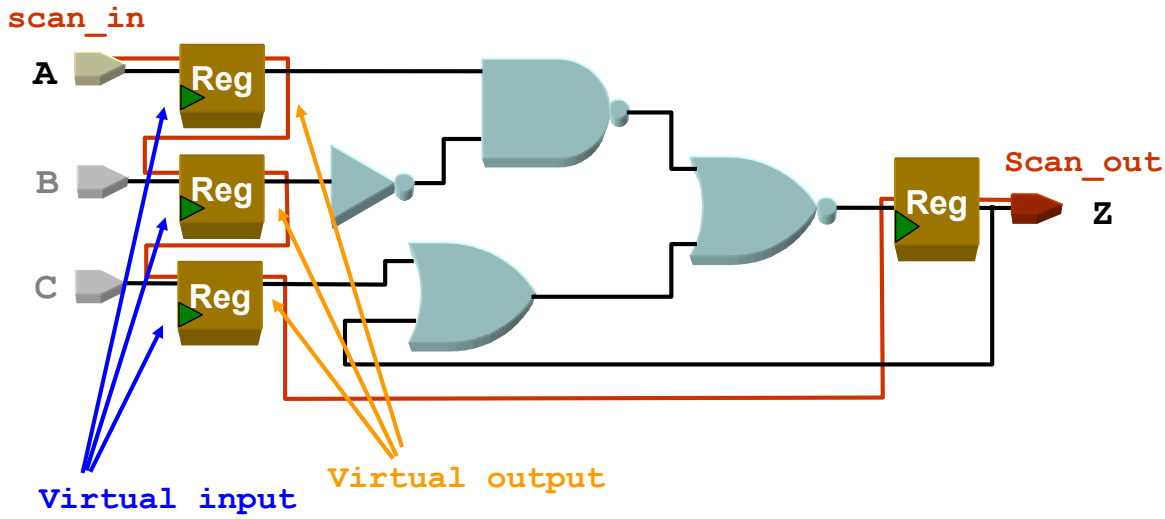


Utilize each available flip-flop, by pre-loading it with a bit of data to control logic gates upstream and/or observe logic gates downstream.

Sequential Logic -- Full-Scan



Sequential Logic -- Add scan-chain



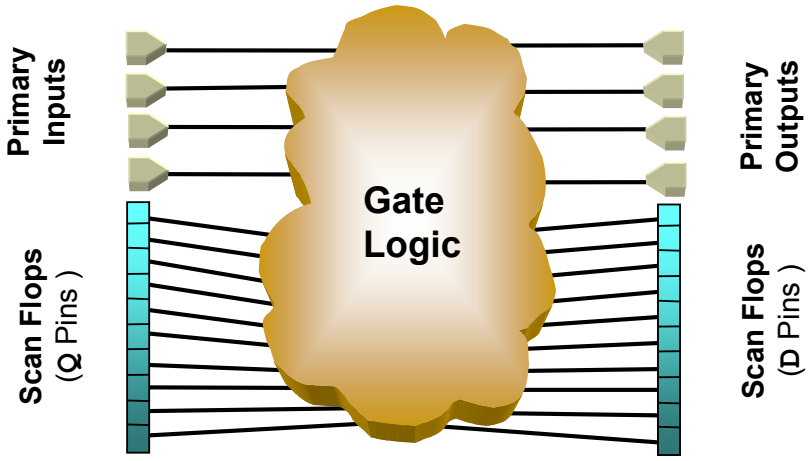
All flops scannable
Combinational ATPG is adequate to test all the gate logic.

Sequential Logic -- Scan Cell

- Scan flops initialize nodes within the design (controllability).
- Scan flops capture results from within the design (observability).
- Inserting a scan path involves replacing all flip-flops with their scannable equivalent flip-flops.

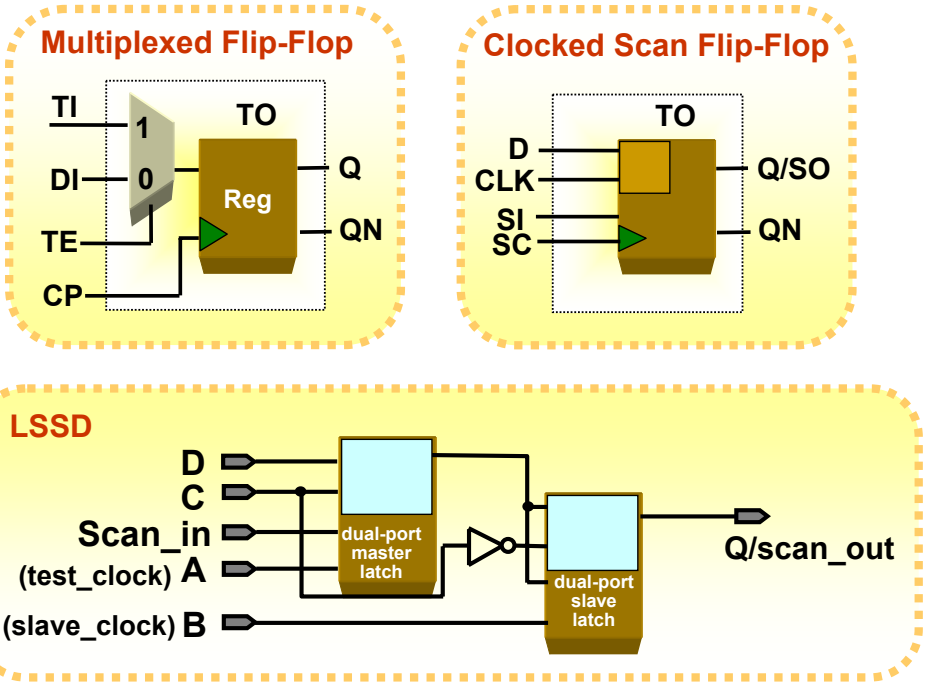
Larger area than non-scan registers;
Larger setup time requirement

Sequential Logic -- Virtual Input/Output



Flip-flops that can be **scanned**
→ **controllable** and **observable**.

Scan Cell Types



Scan Cell Types

	Timing	Area	I/O	Design Style
Multiplexed	↑	↑	Scan-enable	Edge-triggered
Clocked Scan	↑	↑	Test clock	Edge-triggered
Clocked LSSD	↑	↑	Test clock	Level-triggered

Sequential Logic -- Strengths & Weaknesses

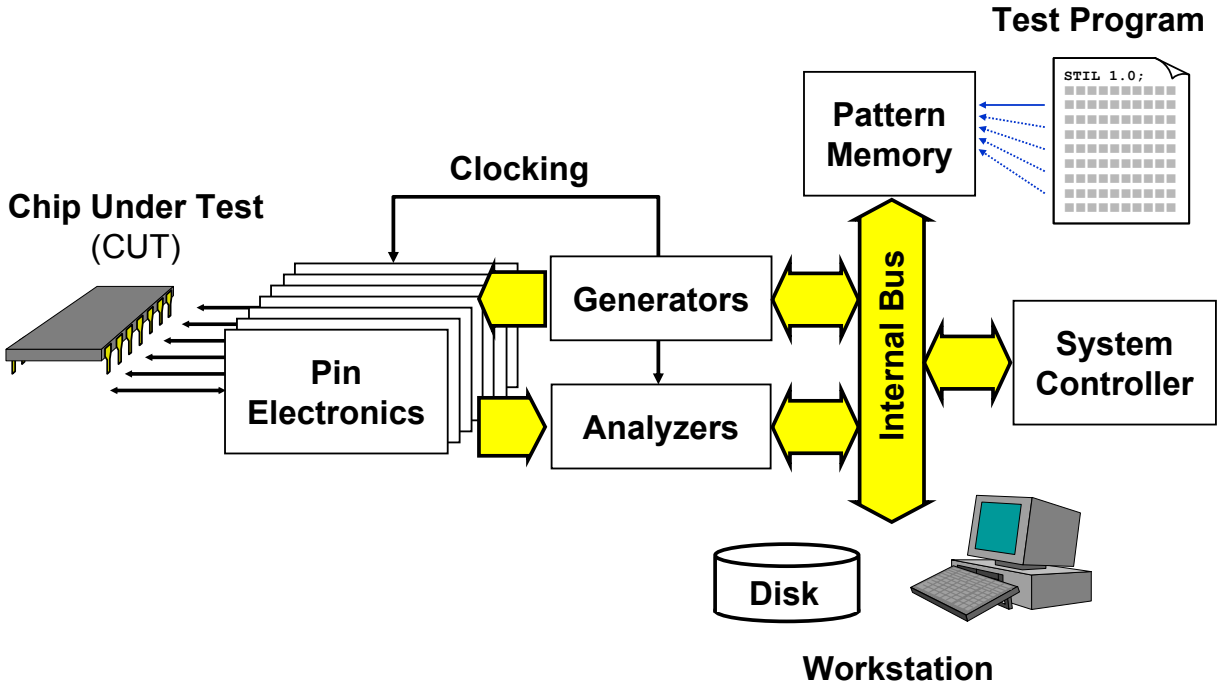
Strengths

- Scan paths allow use of combinational ATPG.
- Make 100% coverage possible in reasonable ATPG time and test-program length.

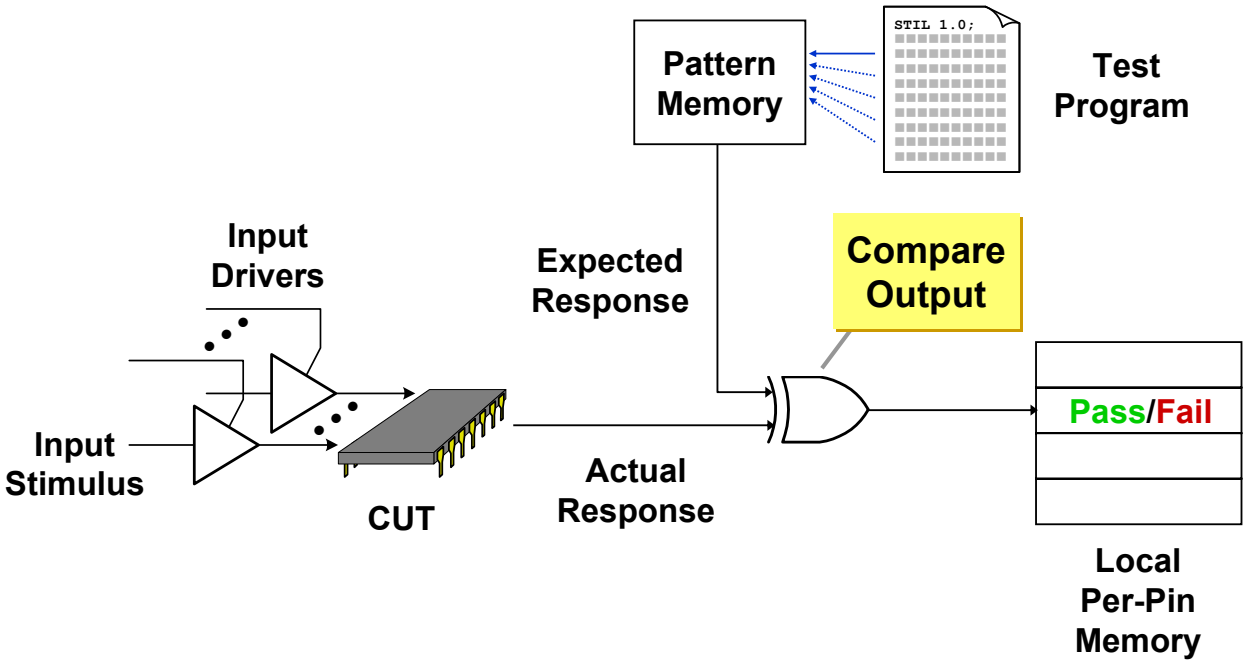
Weaknesses

- Scan paths do impact timing, area, and power goals.
- Designer must account for scan in each block from the very first compile.
- For million-gate designs, back-end re-optimization to fix scan timing is tedious and time-consuming.

About ATE -- Inside the ATE



About ATE -- Fault is Detected

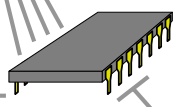


About ATE -- Test Mode

Apply Input:
The primary input is **driven** by the pin electronics to a logic **0** or **1**, regardless of prior value, for the current tester cycle. Also called **force**.

Compare Output:
The **actual** primary output value is measured, and compared against the **expected** value (logic **L** or **H**) from the test program. A mismatch is reported as a pattern failure.

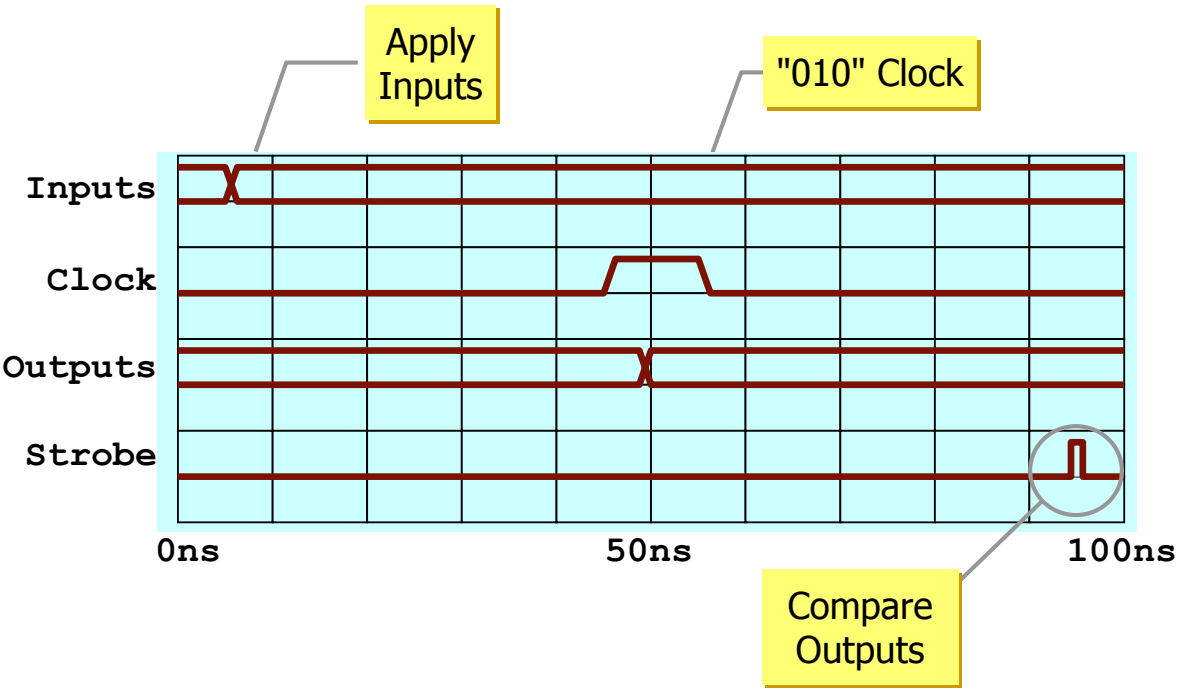
Test modes are applied to individual pins, vector by vector.



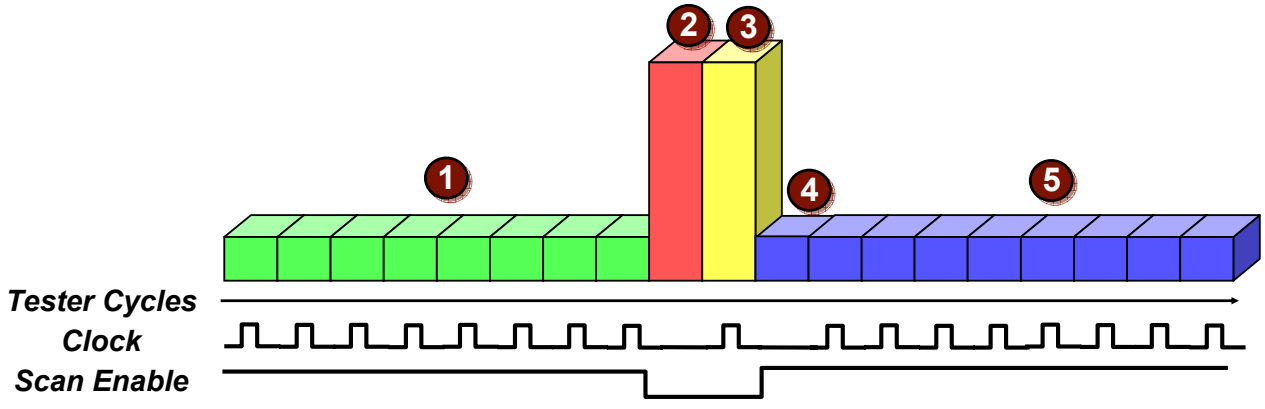
Inhibit Drive:
The primary input is left **undriven** by the pin electronics, whose driver is put into tristate mode. This effectively isolates the pin from the tester.

Mask Output:
The primary output is ignored, and **not compared**, for the current cycle.

About ATE -- Basic ATE Cycle

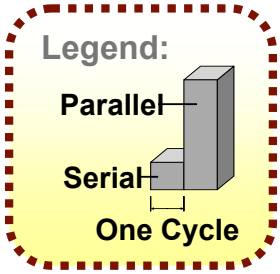


About ATE -- Scan Pattern



Five Phases

① Scan-In
② Parallel Measure
③ Parallel Capture
④ First Scan-Out
⑤ Scan-Out



About ATE -- Each Phase of Scan

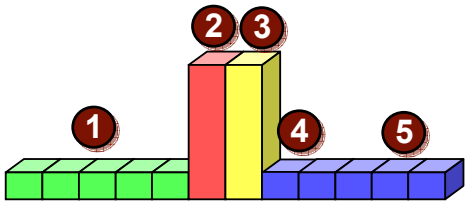
1 Scan-In Phase:
In each cycle, the next scan bit is applied serially to **SI**. On the clock edge, it is shifted in to the scan chain. Meanwhile, parallel outputs are masked.

2 Parallel Measure:
PIs are applied early in the cycle. The clock remains inactive. The CUT is now in a known state. POs are measured late in the cycle.

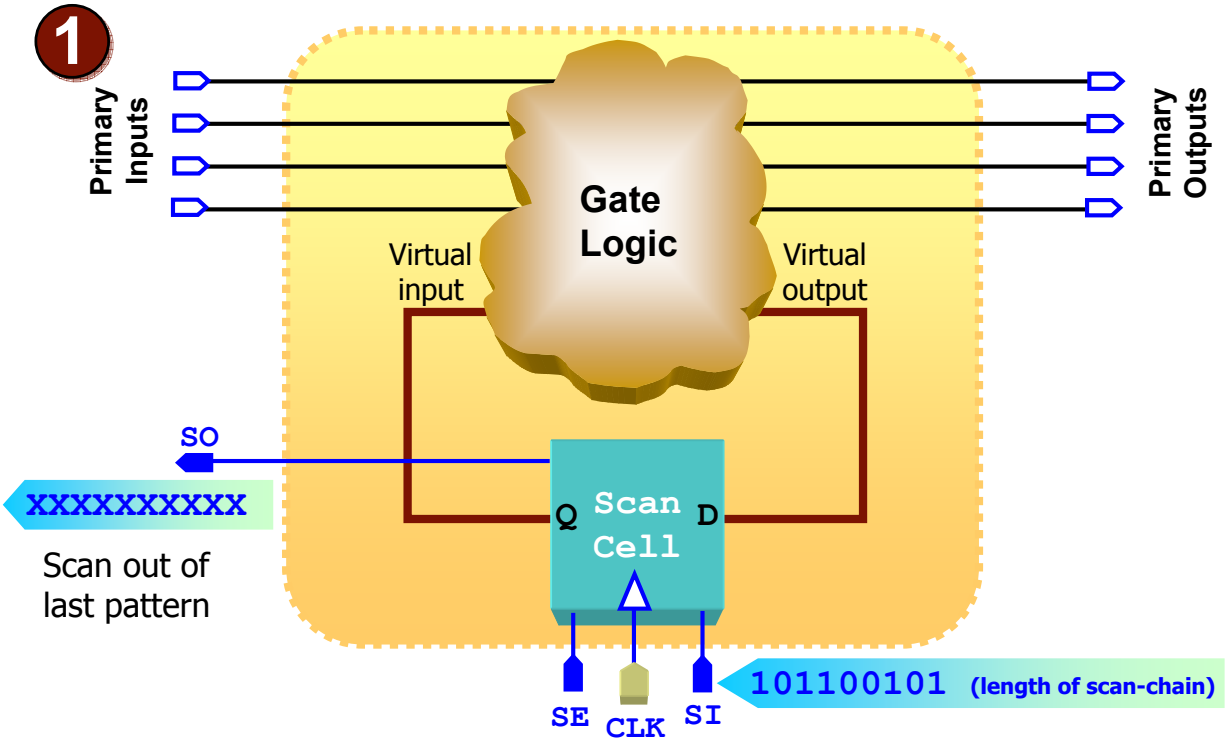
3 Parallel Capture:
The clock is pulsed once. This captures virtual PO data in the scan chain. The CUT is left in a don't-care state. Captured bits are ready for scan out.

4 First Scan-Out:
With no clock, the **SO** port is strobed, measuring the first scanned-out bit.

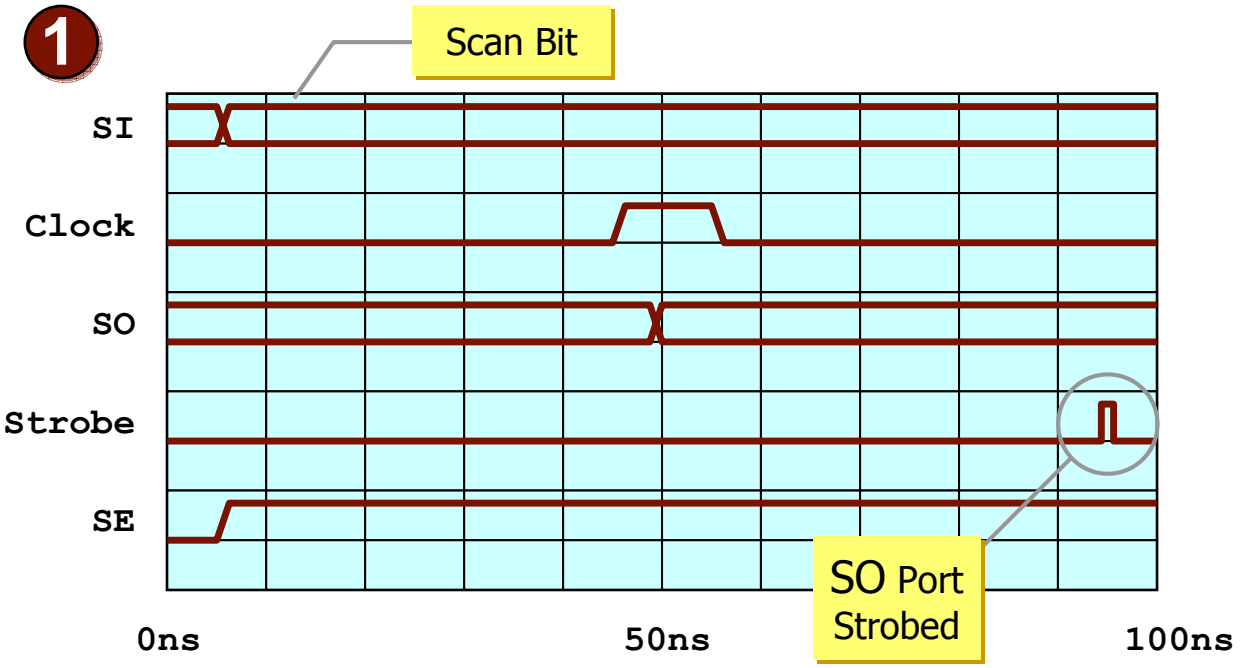
5 Scan-Out Phase:
In each cycle, the next captured bit is scanned out and measured at **SO**.



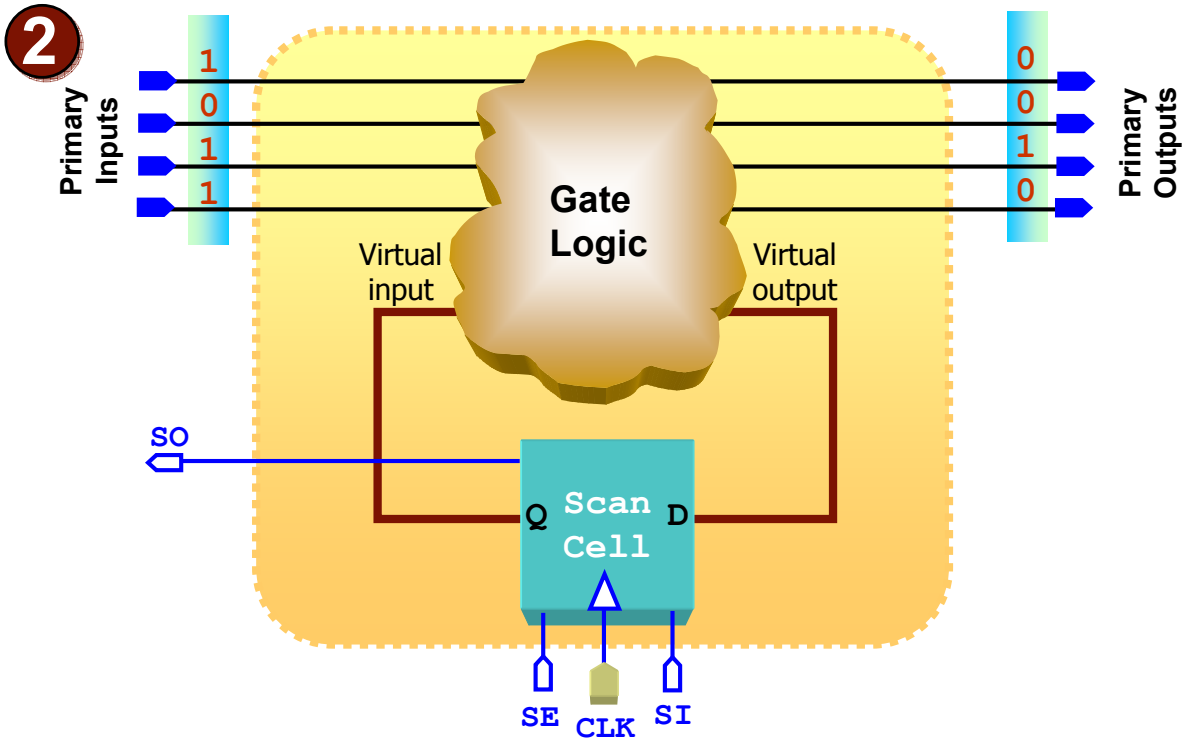
Scan Phase -- Scan-Shift Cycle



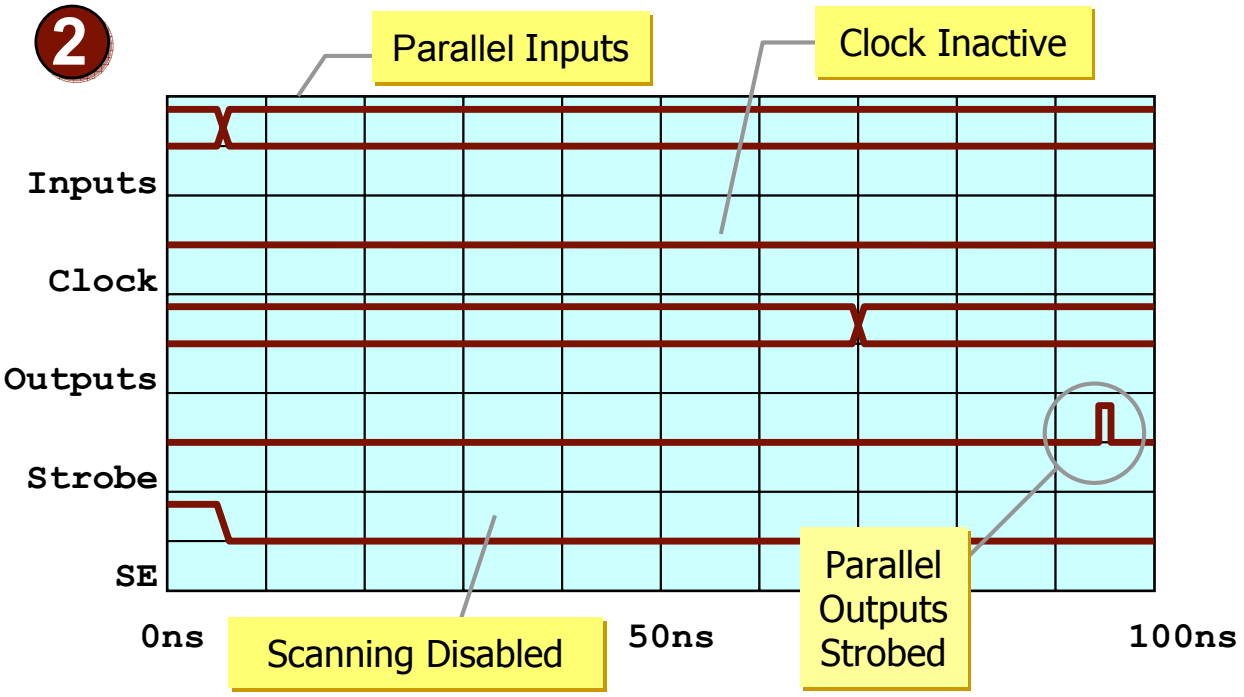
Scan Phase -- Scan-Shift Cycle@@



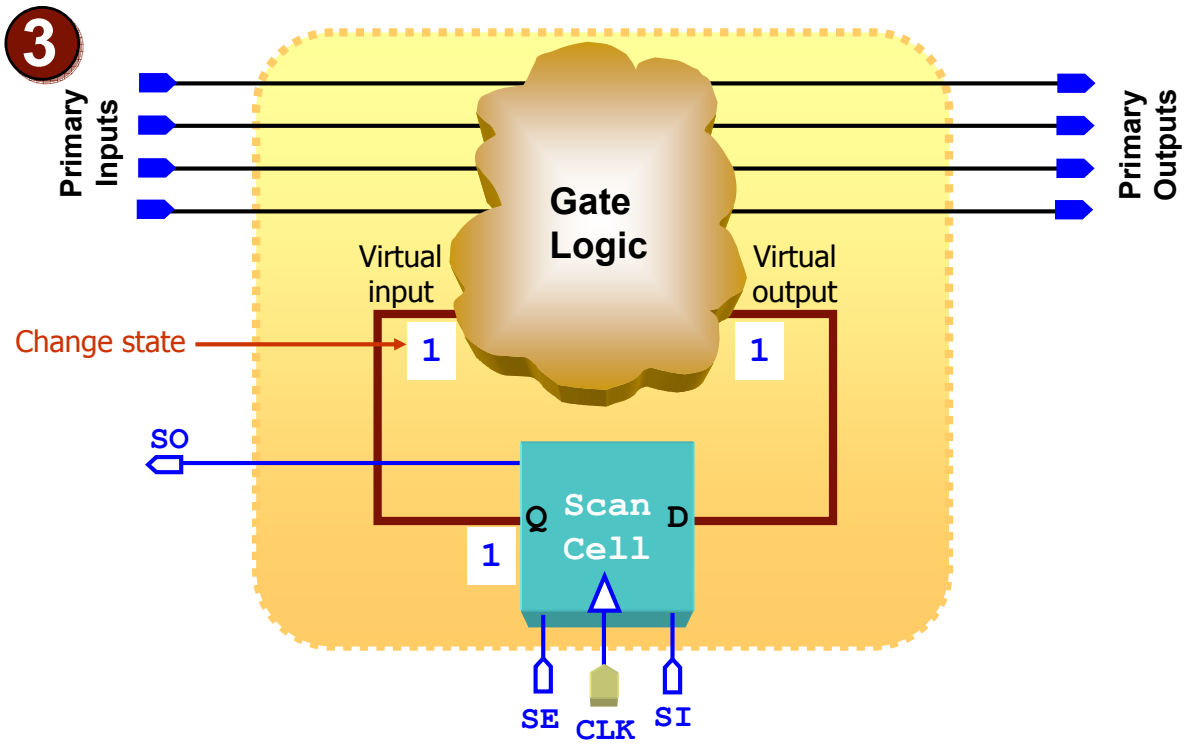
Scan Phase -- Parallel Measure Cycle@



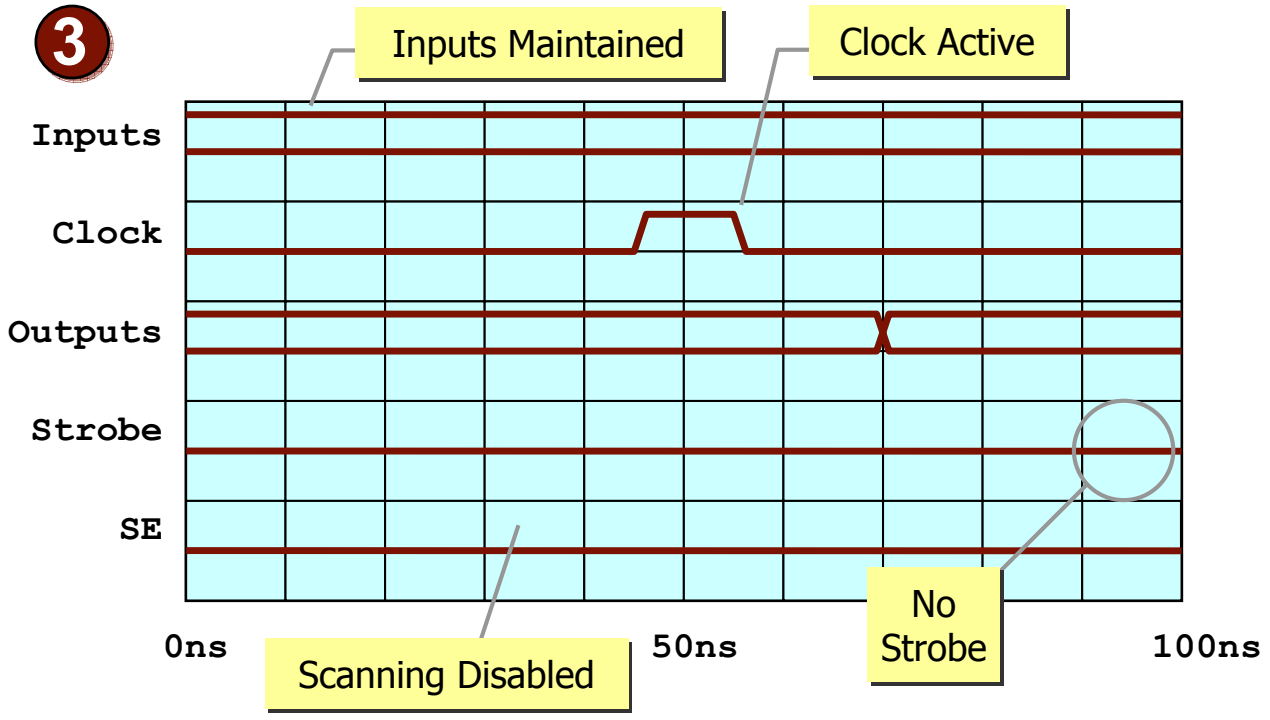
Scan Phase -- Parallel Measure Cycle@@



Scan Phase -- Parallel Capture Cycle@



Scan Phase -- Parallel Capture Cycle@@



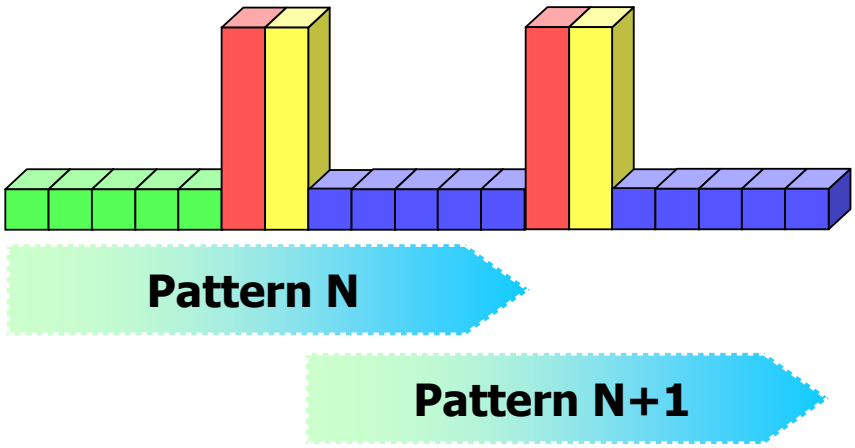
Summary of Scan Shift

- Overlapping Scan-Out of Last Pattern
  - CUT is enabled for scan mode all during scan shift.
  - First Scan-Out phase strobes SO in a no-clock cycle.
  - Remaining scan-out bits are then strobed, one per cycle.
  - Data represents virtual POs captured during last pattern.
- Scanning-In of Next Pattern
  - Multiple scan chains share SE, and are loaded in concert.
  - Scan bits are applied serially to the respective SI inputs.
  - On each clock edge, a scan bit is shifted into each chain.

Summary of Parallel Cycles

- Parallel Measure
  - The CUT is in normal mode, disabled for scan.
  - All scan chains previously loaded with scan data.
  - Applying parallel inputs places CUT in a known state.
  - After settling time, parallel outputs are measured.
- Parallel Capture
  - Still in normal mode, the CUT is clocked just once.
  - Virtual POs are thus captured in the scan flip-flops.
  - Captured data is now ready for First Scan-Out phase.

Pattern Overlap

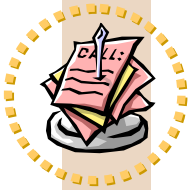


- Scanning out of previous pattern overlaps scanning in of next for all but first and last patterns in the test program.

ATE-Friendly Design

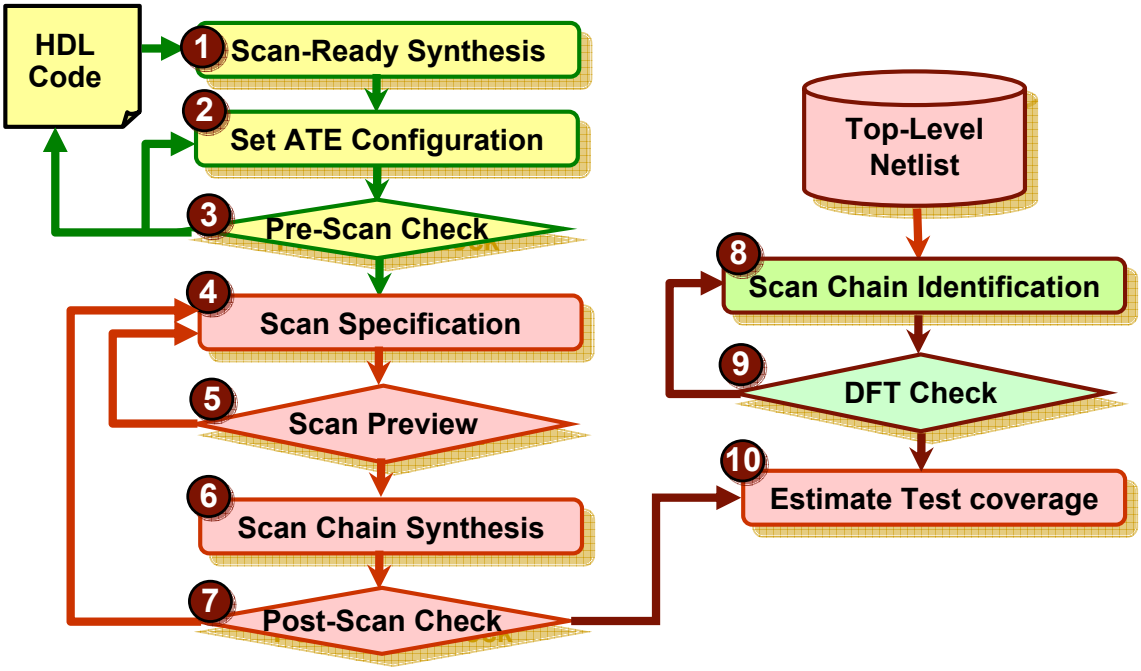
- Chip Clocking:
  - Clock all logic using ATE clocks wherever possible.
  - Minimize usage of internally-derived clock signals.
- Chip Reset:
  - Make DUT easy to reset - no complex initialization.
  - Prefer a plain asynchronous reset at a primary input.
- Scan Design:
  - Use a dedicated chip-wide SE input not shared with data.
  - To minimize package pins, share SI and SO for each path.
  - Break up a very long scan path into several shorter paths.
  - Avoid timing problems by simulating entire test program.

Design-for-Testability Using DFT Compiler

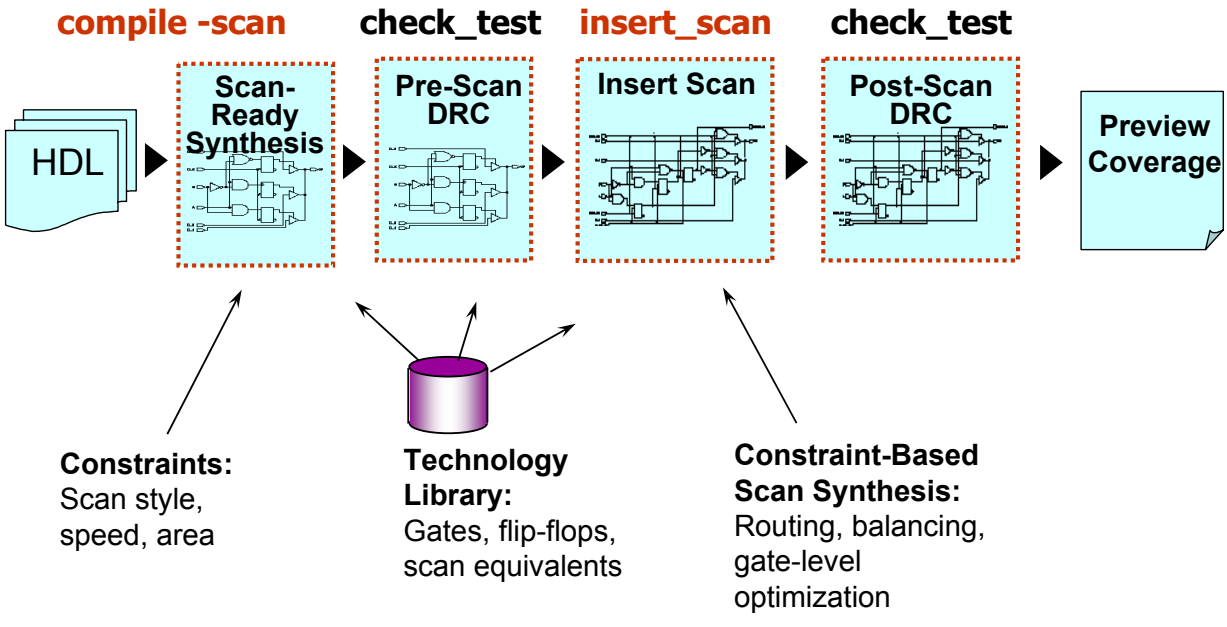


DFT Compiler Flow

Scan Synthesis Flow



Overview of DFT Compiler Flow



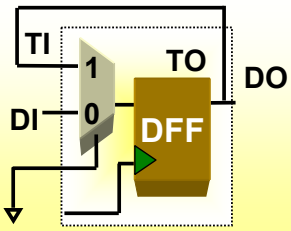
Specify Default Values

Specify this default in synopsys_dc.setup.

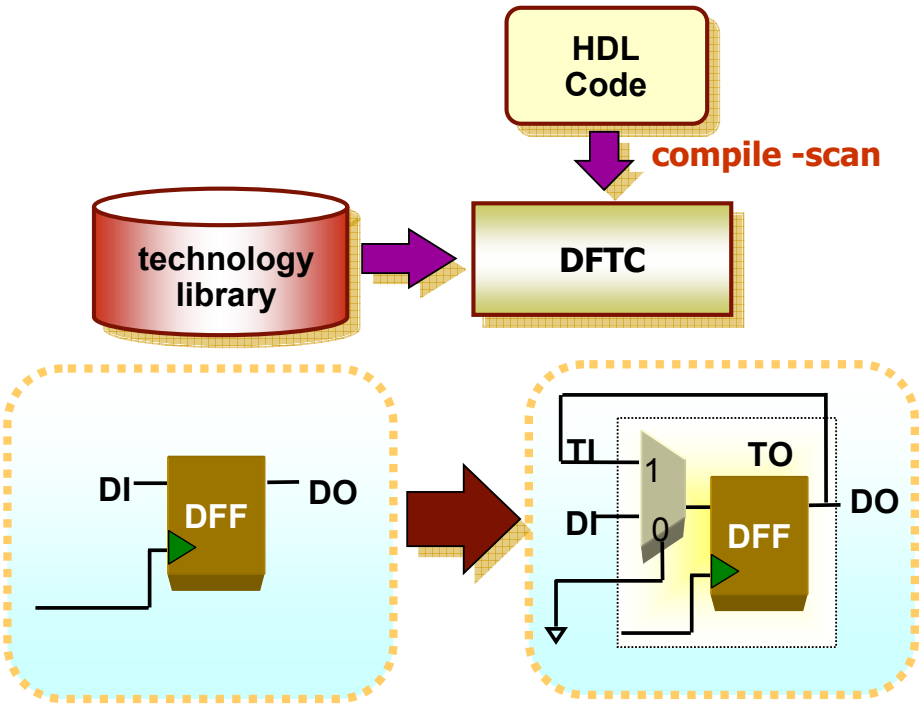
```
set test_default_scan_style multiplexed_flip_flop
set test_default_scan_style clocked_scan
set test_default_scan_style lssd
```

Your choice of style must be supported by vendor library.

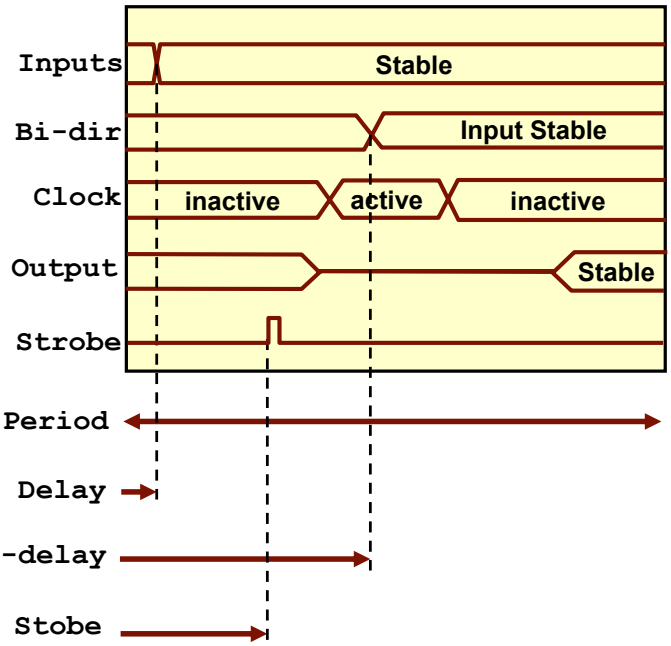
The scan style tells DFTC what type of scan-equivalent flip-flop to use in synthesizing the logic.



1 Scan-Ready Synthesis



2 Set ATE Configuration@




```
set test_default_period 100
set test_default_delay 5
set test_default_bidir_delay 55
set test_default_strobe 40
set test_default_strobe_width 0
```

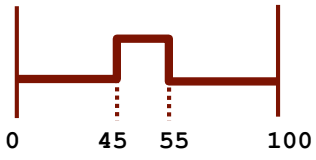
Specify this default in synopsys_dc.setup.

2 Set ATE Configuration@@ -- Create Clock

Default Test Clock:

```
period      test_default_period
1stedge     0.45*test_default_period
2ndedge     0.55*test_default_period
```

```
create_test_clock -p 100 -w [list 45 55] clk_RTZ
```



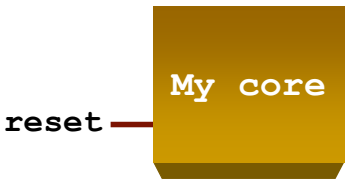
3 Pre-Scan Check@

- Check gate-level scan design rule **before** scan chain synthesis.
- Looks at four categories of testability issues:
  - Modeling problems, such as lack of a scan equivalent.
  - Topological problems, like unclocked feedback loops.
  - Protocol inference, such as test clocks and test holds.
  - Protocol simulation, to verify proper scanning of bits.

```
dc_shell> check_test
...basic checks...
...checking combinational feedback loops...
...inferring test protocol...
Inferred system/test clock port CLK (45.0,55.0).
...simulating parallel vector...simulating serial scan-in…
Information: The first scan-in cycle does not shift in data.(TEST-301)
Warning: Cell U1 (FD1S) is not scan controllable. (TEST-302)
Information: Because it clocks in an unknown value from pin TI.(TEST-512)
Information: Because port SI is unknown. (TEST-514)
Information: As a result, 3 other cells are not scan controllable.(TEST-502)
Information: Test design rule checking completed. (TEST-123)
```


3 Pre-Scan Check@@

- As we'll see later on fixing DFT violations, it's often necessary to:
  - Inform DRC that this port is programmed at a fixed value by the ATE all during the test session.



  - Add a top-level input port to put the IC in "test mode."

```
set_test_hold 1 TM
read_init_protocol
read_test_protocol
check_scan or check_test
```

4 Scan Specification@

- set_scan_configuration

```
-methodology          -chain_count          -disable
-style                -clock_mixing         -internal_tristate
-existing_scan        -internal_clocks      -bidi_mode
                      -add_lockup           -dedicated_scan_ports
                      -replace              -hierarchical_isolation
                      -rebalance
                      -route
```

```
set_scan_configuration -chain_count 4 \
                       -clock_mixing no_mix
```


4 Scan Specification@@

- set_scan_path
- set_scan_signal
- set_scan_element
- set_scan_segment
- set_scan_transparent

5 Scan Preview

- Scan Preview
  - Check your scan specification for consistency
  - Complete your scan specification
  - Report scan architecture
  - Using "-script" to generate a script file which specifies scan architecture

```
preview_scan -show
preview_scan -script
```

Text Report

Reusable Script

6 Scan Chain Synthesis

- Scan-Chain Insertion Algorithm:

1. Targets the previewed scan-path architecture.
2. Performs any remaining scan replacements.
3. Adds disabling/enabling logic to tristate buses.
4. Conditions the directionality of bidirectional ports.
5. Wires the scan flops into the specified chains.
6. Optimizes the logic, minimizing constraint violations.

```
insert_scan  -map_effort medium
```

7 Post-Scan Check@

- Why run check_scan again?
  - Confirm there are no new DFT problems.
  - Verify the scan chains synthesized operates properly.
  - Create an ATPG-ready database.

```
check_scan
```


7 Post-Scan Check@@

```
report_test -scan_path
```

```
************************************
Report: test  -scan_path
Design: ADES
************************************
(*) indicates change of polarity
Complete scan chain (SI --> SO)
containing 5 cells:
  SI              ->
  sh1_reg         ->
  sh2_reg         ->
  sh3_reg         ->
  sh4_reg         ->
  zr_reg          ->
  SO
```

8 Scan Chain Identification@

- When to use?
  - Import an existing scan design in non-db netlist format. (e.g., EDIF, VHDL, Verilog)
  - Using `reset_design` after scan chain synthesis. (it will remove the test attributes from the design)

```
set_scan_configuration -existing_scan true
```


8 Scan Chain Identification@@

- Example:

```
set_scan_configuration -existing_scan true
set_scan_configuration -bidi_mode input
set_signal_type test_scan_in [list SI1 SI2]
set_signal_type test_scan_out [list SO1 SO2]
set_signal_type test_scan_enable SE

check_test
report_test -scan_path
```

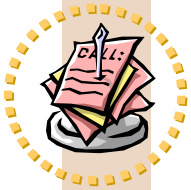
10 Estimate Test coverage

- Use the DFTC ATPG command:
  estimate_test_coverage will call TetraMAX for fault estimate.

```
estimate_test_coverage
```

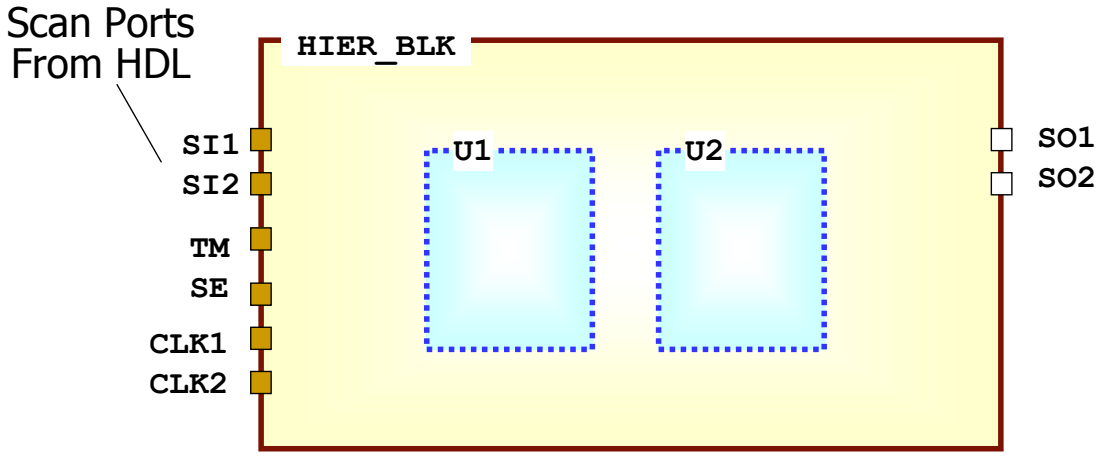
```
                    Pattern Summary Report

          Uncollapsed Stuck Fault Summary Report
  -----------------------------------------------
  fault class                     code   #faults
  ------------------------------  ----  ---------
  Detected                         DT       3084
  Possibly detected                PT          0
  Undetectable                     UD         12
  ATPG untestable                  AU          0
  Not detected                     ND          0
  -----------------------------------------------
  total faults                              3096
  test coverage                           100.00%
  -----------------------------------------------
Information: The test coverage above may be inferior
             than the real test coverage with customized
               protocol and test simulation library.
```


DFT Compiler – Lab 1



Test Synthesis Flow

Block-Level Example@



You are responsible for HIER_BLK, a **subdesign** on an ASIC.
It has its own hierarchical structure, with two clock domains.
Insert two separate scan chains, and use a test-mode port.
Preview the fault coverage on the post-scan design.

Block-Level Example@@ TCL script

```
  current_design HIER_BLK
@ compile -scan
@ create_test_clock -p 200 -w [list 90 110] CLK1
@ create_test_clock -p 200 -w [list 120 130] CLK2
  set_test_hold 1 TM
@ check_test
  set_scan_configuration -chain_count 2
  set_scan_configuration -bidi_mode input
  set_scan_configuration -clock_mixing no_mix
  set_scan_signal test_scan_in -port [list si1 si2]
  set_scan_signal test_scan_enable -port SE
  set_scan_signal test_scan_out -port [list so1 so2]
  preview_scan
@ insert_scan
@ check_test
  report_test -scan_path
```

ASIC-Level Example

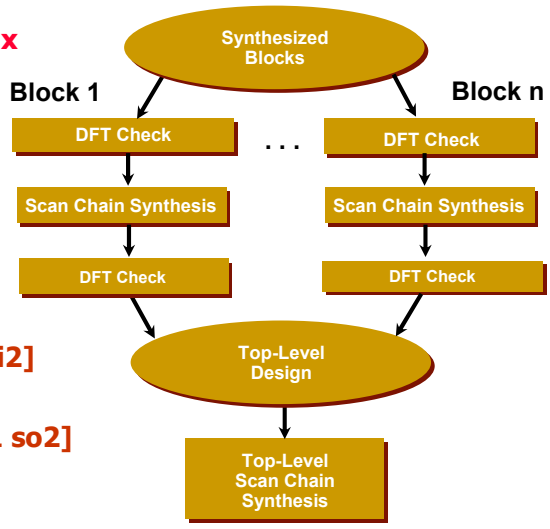
```
  current_design ASIC

@ create_test_clock -p 200 -w [list 90 110] CLK1
@ create_test_clock -p 200 -w [list 120 130] CLK2
  set_test_hold 1 TM

@ set_scan_configuration -existing_scan true
  set_scan_configuration -bidi_mode input

@ set_signal_type test_scan_in      [list SI1 SI2]
@ set_signal_type test_scan_enable SE
@ set_signal_type test_scan_out     [list SO1 SO2]

@ check_test
  report_test -scan_path

@ estimate_test_coverage
```


Bottom-Up Synthesis@

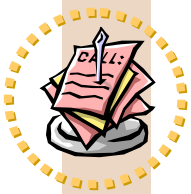
- Benefits of Bottom-Up Test Synthesis
  - Complete modules to satisfy design constraints.
  - Reduce scan synthesis runtime for large designs.
  - prevent DFT Compiler from renaming the sub-designs of the CORE.
- Tips of Bottom-Up Test Synthesis
  - Do **NOT** specify scan signals, mix clock domains.
  - Switch **OFF** the insertion of disable logic for multiply-driven nets.

Bottom-Up Synthesis@@

```
current_design A
compile –scan
check_test
set_scan_configuration –disable false
set_scan_configuration –clock_mixing no_mix
preview_scan
insert_scan
check_test
…
current_design TOP
check_test
set_scan_configuration –disable true
set_scan_signal test_scan_in –port [list si1 si2]
set_scan_signal test_scan_enable –port SE
set_scan_signal test_scan_out –port [list so1 so2]
preview_scan
insert_scan
```



Design-for-Testability Using DFT Compiler



Prevent DFT Violations

Test Design Rule Violation Summary

Total Violations: 12

Topology Violation

- 1 combinational feedback loop violation (TEST-117)

Scan In Violations

- 2 cells constant during scan violations (TEST-142)

Capture Violations

- 1 clock used a data violation (TEST-131)
- 4 illegal path to data pin violations (TEST-478)
- 4 cell does not capture violations (TEST-310)

Sequential Cell Summary

11 out of 45 sequential cells have violations

Sequential cells with violations

- 4 Cells have parallel capture violations
- 2 Cells have constant values
- 2 Cells have scan shift violations
- 3 Cells are black box

Sequential cells without violations

- 30 cells are valid scan cells
- 4 cells are transparent latches

How to Fix DFT Violations@

- Methods of correcting DRC violations in DFTC:
  1. Edit HDL code and resynthesize design with DFT logic.
  2. Use **AutoFix** DFT to insert bypass or injection logic.
  3. Use **ShadowLogic** DFT to insert ad-hoc test points.
- Or Ignore the problem if the reduction in FC is allowable.

How to Fix DFT Violations@@

- Add testability to the design early on, preferably in the synthesizable HDL code.
  - Avoid uncontrollable clocking.
  - Avoid uncontrollable asyn. set/reset.
  - Avoid uncontrollable three-state/bidir enables.
- Use AutoFix as a workaround after HDL code-freeze, or for unfamiliar legacy designs.

Scan Issue

- Design scenarios affecting risk-free scan include:

1. no clock pulse at scan flop due to gating of clock.
2. no clock pulse due to dividing down of the clock.
3. unexpected asynchronous reset asserted at flop.
4. hold-time problem due to short net or clock skew.

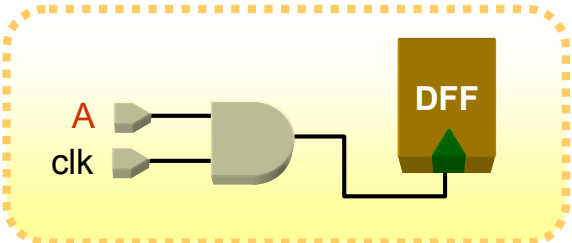
- The ATE must fully control the clock and asynchronous set and reset lines reaching all the scan- path flip-flops.

Gated Clocks

- DRC automatically infers clock signals by tracing back from flip-flop clock pins to PIs.
- To fix this violation, make the AND transparent:

```
set_test_hold 1 A
```

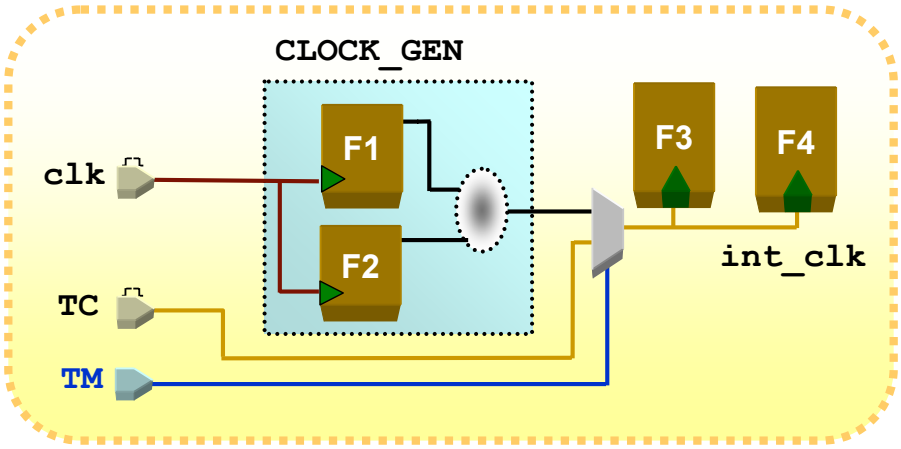
- If **A** is not a PI, add logic to inject a **1** during the test program, by means of a PI test-mode port.



Clock Generators Solution 1

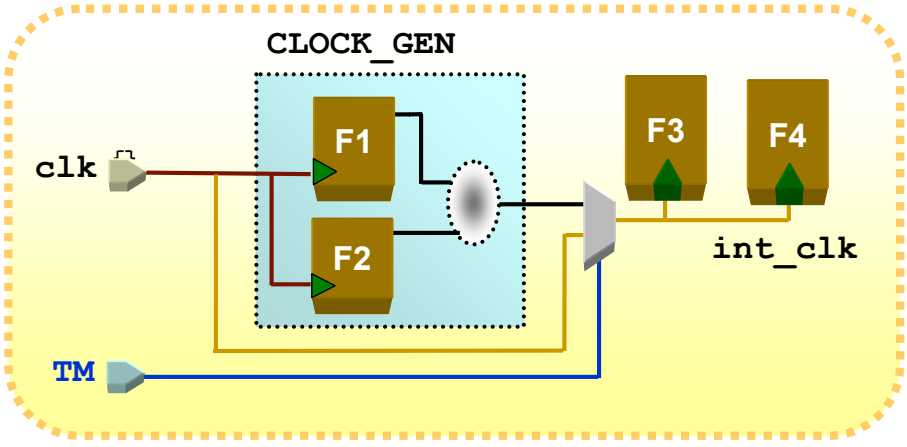
- Add a bypass MUX that selects an external clock during test.
- Hold the TM port high.

```
set_test_hold 1 TM
```



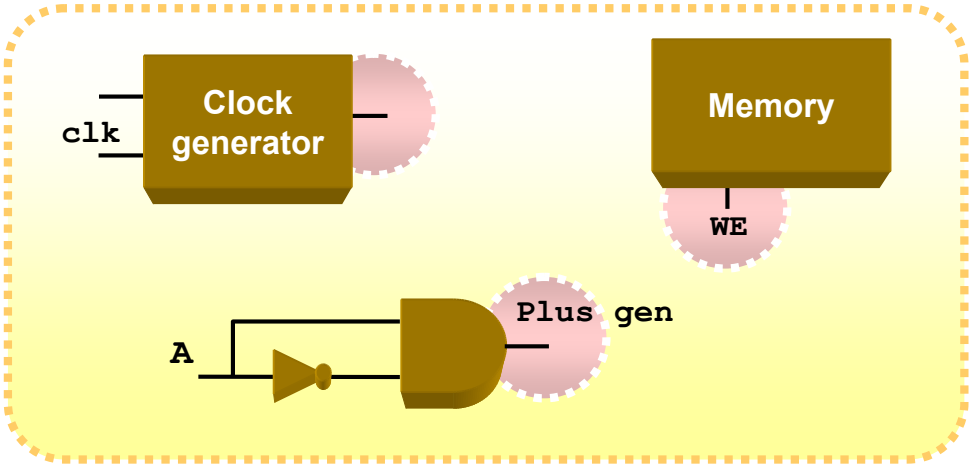
Clock Generators Solution 2

- This solution does not require an extra I/O pad, but may have skew problems!



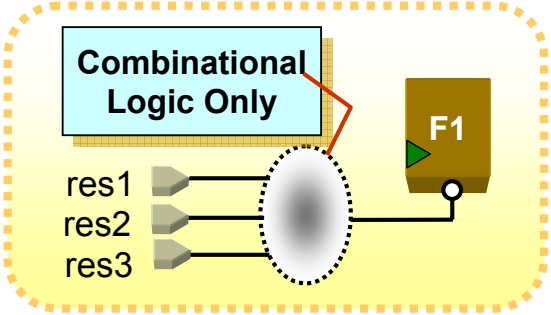
Related Clock Violations

- Apply similar solutions to these related situations:
- On-chip PLL (phase-locked loop) clock generators.
- Digital pulse generators: e.g.,(A & delayed~A).
- On-chip RAM with WRITE_EN or WRITE_CLK pins.



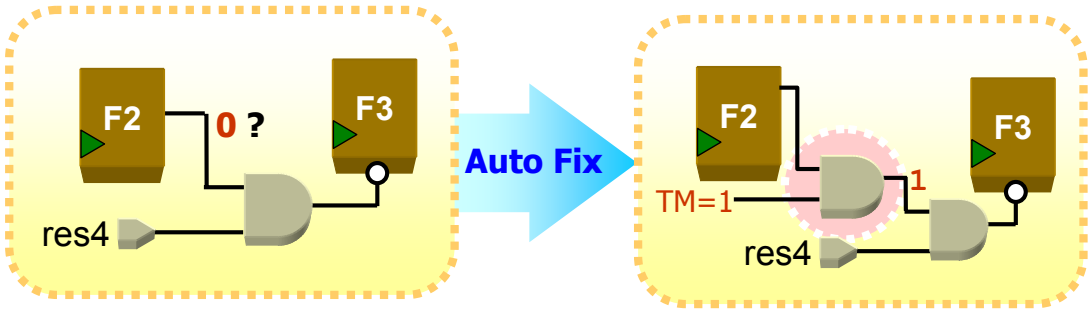
Controllable Asynchronous Reset 1

- Asynchronous Set/Reset is controlled by a combinational circuit block.
  - No DFT problem.
  - DC XP determine how to control input ports to ensure all FFs will not be Set/Reset.



Controllable Asynchronous Reset 2

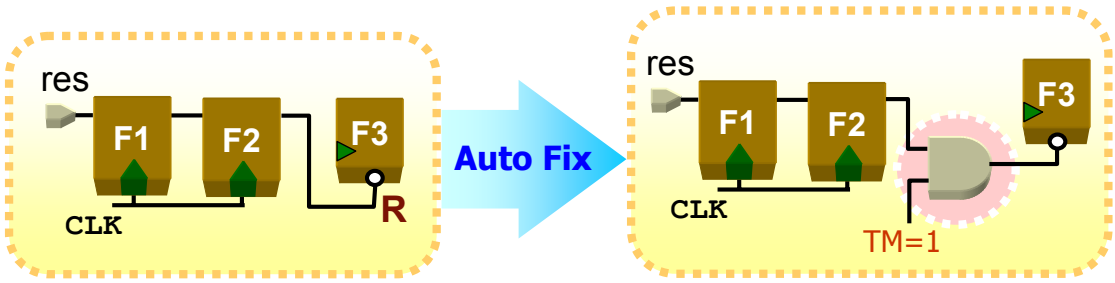
- Asynchronous Set/Reset is controlled by a sequential circuit block
  - Add control logic to prevent FFs from Set/Reset in test mode.



excludes F3 from any scan chain

Controllable Asynchronous Reset 3@

- Asynchronous Set/Reset is controlled by a shift register
  - Provide an initial test protocol to prevent FFs from Set/Reset in test mode.
  - Try Auto-Fix



Controllable Asynchronous Reset 3@@

- Set pin res as 1 and apply two clock pulse, then signal R will be 1 all the time (in test mode)
- This method does not change the circuit.

```
set_test_hold 1 res
check_test
write_test_protocol -format stil -out dut.spf

<EDIT dut.spf>

read_init_protocol -f stil dut.spf
check_test -verbose
```

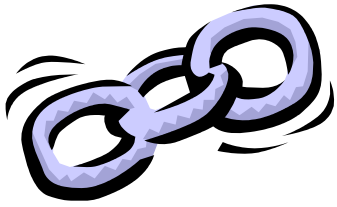

Controllable Asynchronous Reset 3@@@

- Edit test protocol file dut.spf

```
"test_setup" {
      W "_default_WFT_";
      V {  "clk"=0; }
      V {  "res"=1; "clk"=P; }
      V {  "res"=1; "clk"=P; }
      V {  "res"=1; "clk"=0; }
}
```

Initialization Sequences@

- No dedicated test mode primary input.
- Ex. 3-bit configuration register to generates an internal test mode signal if you initialize with the pattern “101”.
- During initialization the port CONF_ENABLE must be held '1' while the above pattern is serially applied to port CONF and the clock port CLK is pulsed.



Initialization Sequences@@

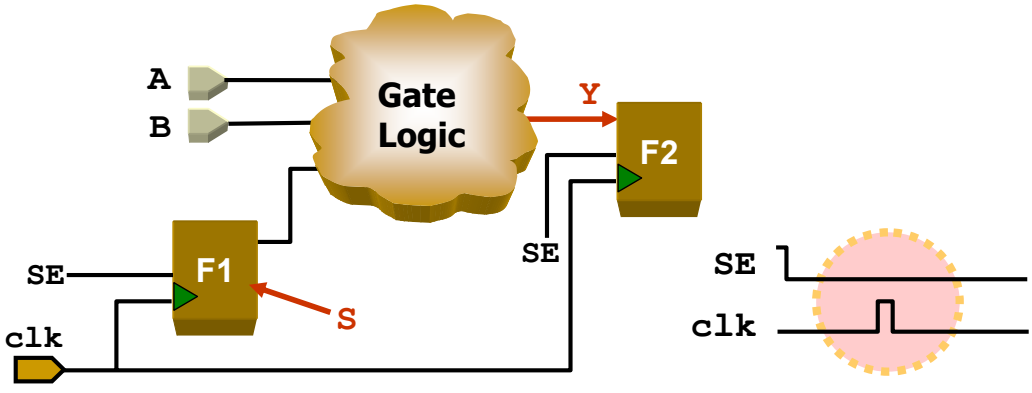
- set_test_hold 0 CONF_ENABLE
- Check test
- write_test_protocol –f stil –o no_scan.spf

```
"test_setup" {
      W "_default_WFT_";
      V {   "clk"=0; "CONF_ENABLE"=0; }
      V {   "CONF_ENABLE"=1; "CONF"=1; "clk"=P; }
      V {   "CONF_ENABLE"=1; "CONF"=0; "clk"=P; }
      V {   "CONF_ENABLE"=1; "CONF"=1; "clk"=P; }
      V {   "CONF_ENABLE"=0; "clk"=0; }
  }
```

What Causes Risky Capture?

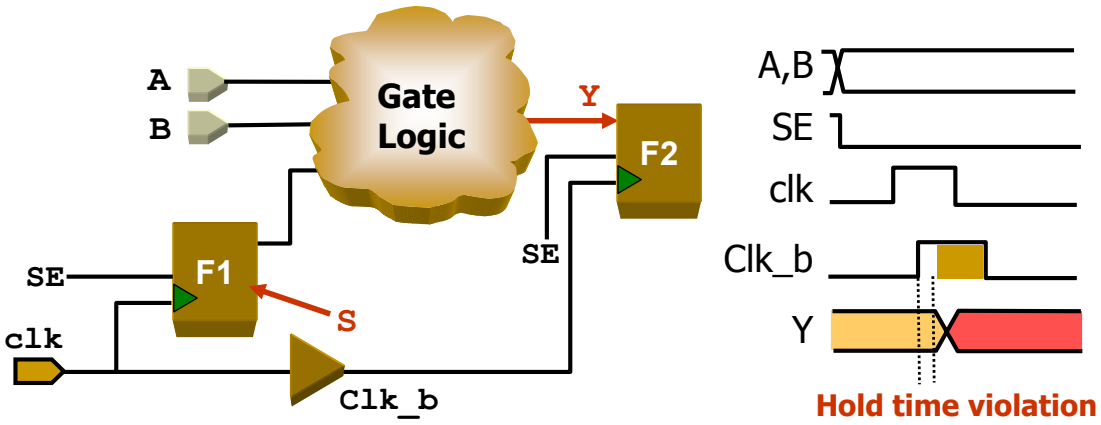
- Risky design practices for capture include:
  - logic that captures on both edges of clock.
  - clock also used as input to flop's logic cone.
  - asynchronous set or reset used in same way.
  - hold-time issue due to unexpected clock skew.

Risk-Free Capture Example



- Just prior to capture, scan flop F1 loaded with bit S.
- Intent of capture is to clock in the current state bit Y.
- Bit Y must depend only on scan bit S and ports A, B.

Capture-Problem

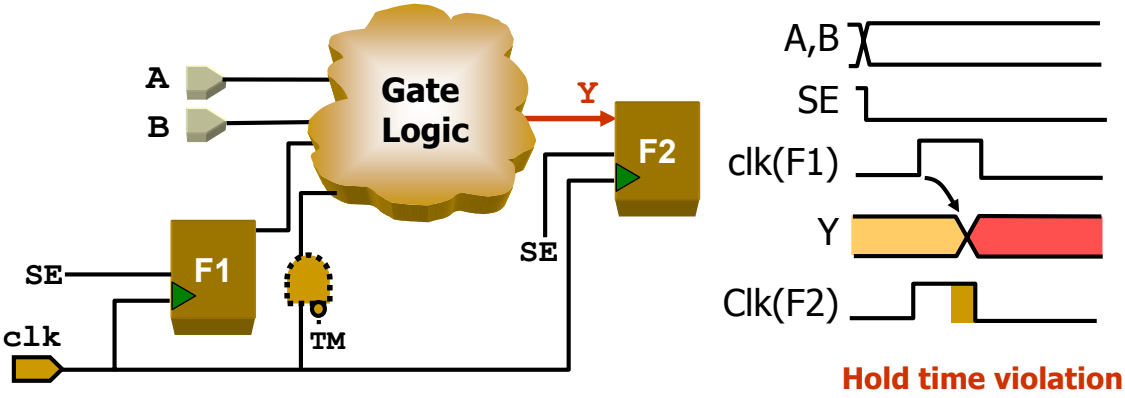


- For a logic cloud, output Y may change too soon.
  - (because F1 capture a new value)
- The timing diagram shows how F2 hold-time is violated.

Managing Clock-Skew Problems

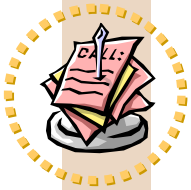
- To manage this class of capture problem within DFTC:
  - Use DC/PrimeTime to fix hold violations, post-layout.
  - Avoid false paths that cross between clock domains.
  - Allow at most one active clock per capture procedure.
  - These same guidelines apply to asynchronous resets.

Clock-as-Data Violation



- Edge of launch clock will affect cloud output.
- After delay-dependent interval (red), Y changes.
- If interval is too short, F2 hold-time is violated.

Design-for-Testability Using DFT Compiler



Advanced DFT Techniques

Buffering the Scan-Enable Network

- As a rule, the high-fanout scan-enable line should bebuffered.
- Buffering SE is critical for at-speed or transition-delay testing.

```
set_scan_signal test_scan_enable -port SE -hookup BUF/Z
```

- Instantiate a clock-tree buffer in the HDL description, or build a clock-tree for SE.
- Use set_dont_touch_network to keep DFTC from rebuffering.
- Route the SE tree to all major blocks.

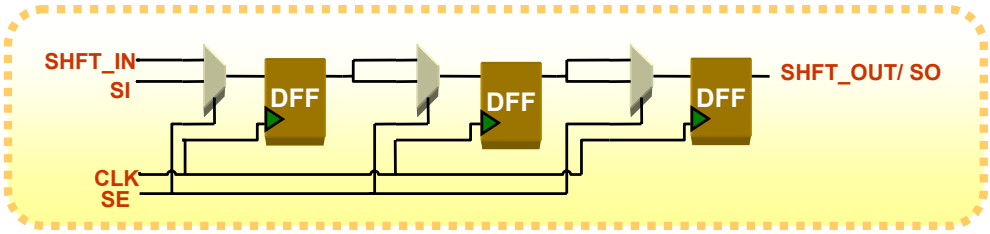
Shift Registers

They are no need to be replaced by Mux-FFs.

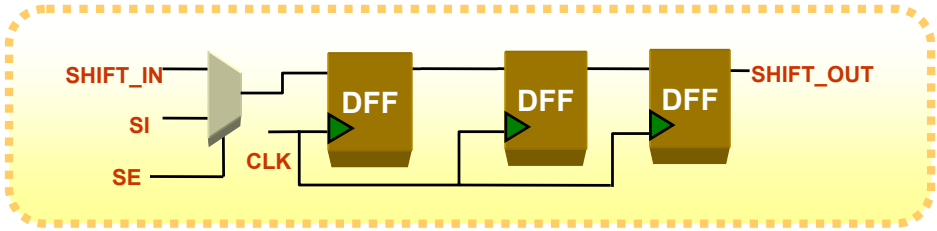
```
set_scan_segment shift -access [list test_scan_in\
  {dout_reg[1]/D} test_scan_out {dout_reg[7]/Q}]\
  -contains [list {dout_reg[1]} {dout_reg[2]}\
  {dout_reg[3]} {dout_reg[4]} {dout_reg[5]}\
  {dout_reg[6]} {dout_reg[7]}]

set_scan_element false [list {dout_reg[1]}\
  {dout_reg[2]} {dout_reg[3]} {dout_reg[4]}\
  {dout_reg[5]} {dout_reg[6]} {dout_reg[7]}]

preview_scan -show segments

insert_scan
```

Shift Registers@@

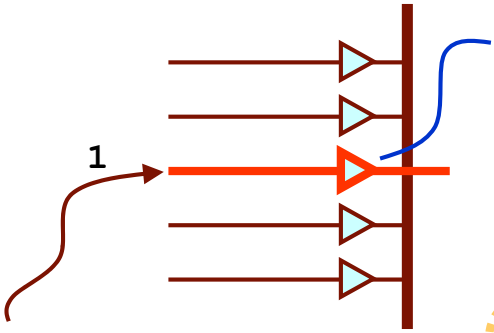
Before:



After:



Tristate@



Only one tristate driver can be active at a time on a given tristate bus.

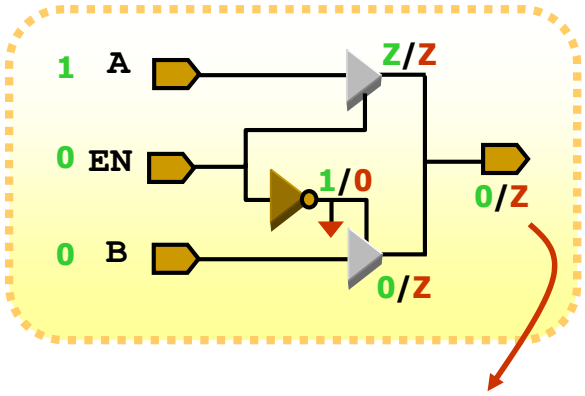
ATPG can only detect faults on an active tristate driver.

Unreachable driver:
Active driver which causes bus float or bus contention conditionis.

Fault Coverage ↓

Tristate@@

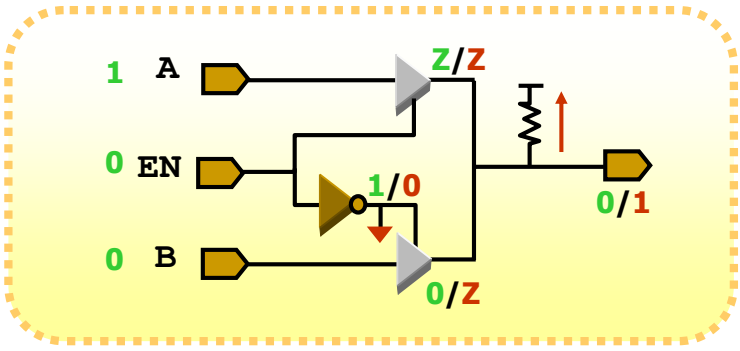
- Internal tristate nets have testability problems on the enable pins and their fanin.
- Avoid internal tristate nets when feasible.



Must strobe Z to detect

Tristate@@@ -- Pull-Up Resistors

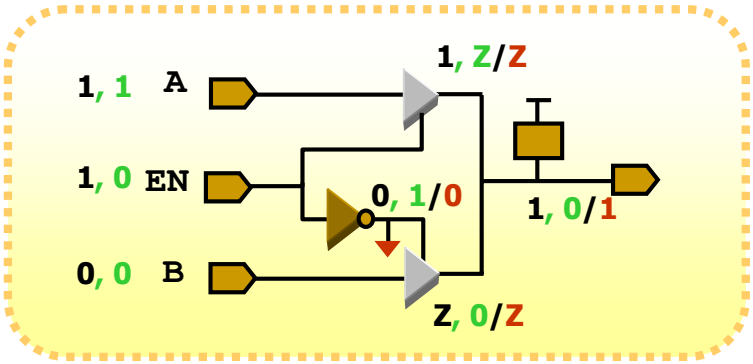
- If you connect a pull-up or pull-down resistor to the tristate net, the 'Z' turns into '1' or '0' which allows detection of all faults.



- Weak 1 overrides the floating output.

Tristate@@@@ -- Bus Keepers

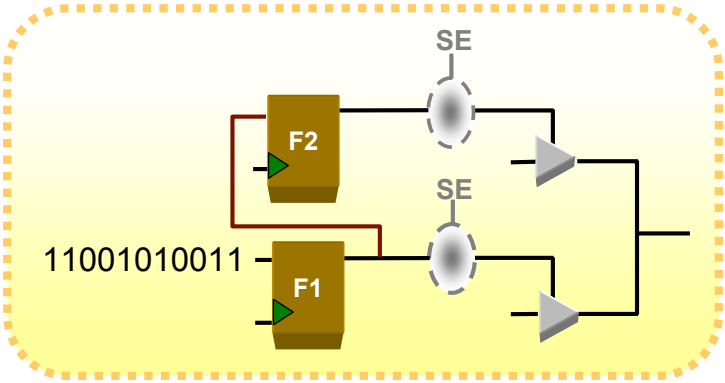
- Connecting a bus keeper cell to the tristate net will avoid float conditions, because it keeps the previously active value.



If we activate the target fault, the bus will float.
Thus we need to initialize the bus keeper first....

Tristate -- Scan Shifting@

- DFTC can synthesize disabling logic into the enable lines to guarantee that exactly one driver is active during scan shift.



Tristate -- Scan Shifting@@

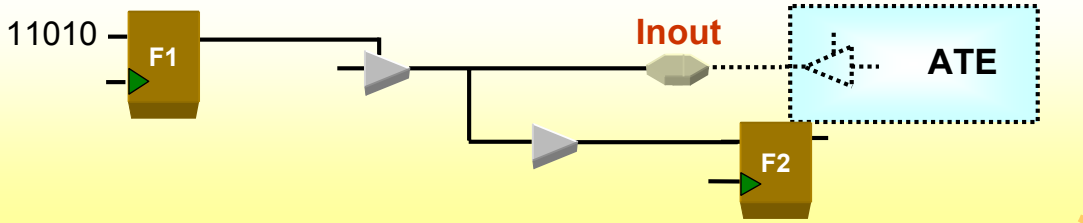
- In design TOP module:

Determines the disabling option during scan shift for all tristate nets that that disables all but one driver.

```
set_scan_configuration -disable true
set_scan_configuration -internal_tri enable_one
set_scan_configuration -external_tri disable_all
```

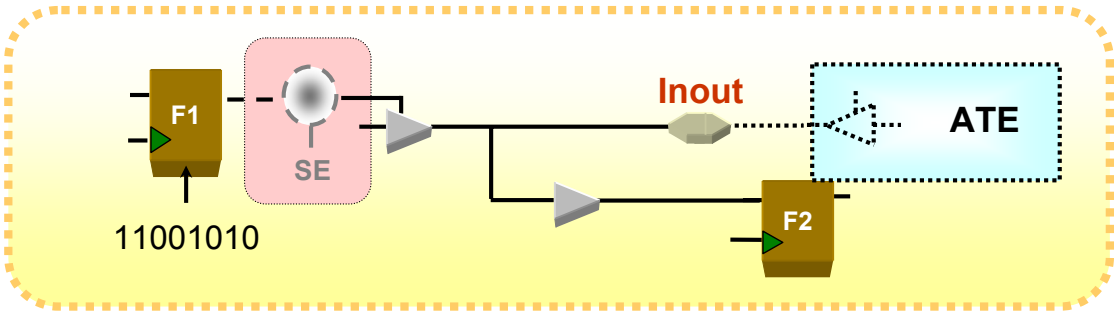
Determines the disabling option during scan shift for all tristate nets that drive output ports of a design.

Bidirectional Ports@



- If scan data is shifted through F1, the bidirectional port direction changes with every scan data change.
- This would create floating or contention states at the bidirectional ports during scan shift.

Bidirectional Ports@@



In TOP module:

```
set_scan_configuration -disable true
set_scan_configuration -bidi_mode input/output
```


Local Control of Tristates and Bidi

```
set_scan_tristate no_disabling -net TRI3
```

enable_one
disable_all

Net name

```
set_scan_bidi input -port [list BIDI3 BIDI5]
```

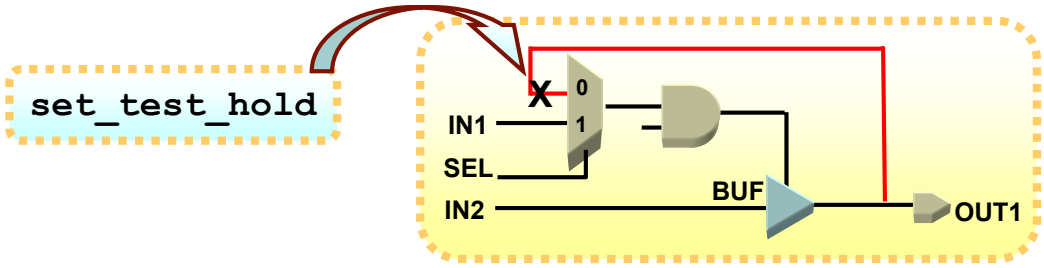
output
No_disabling

Port name

Local control overrides global configuration settings.

Combinational Loops

- If check_test detects a combinational loop, it breaks the loop at a certain pin it selects by inserting an 'X'.
- If the loop is logically broken by logic values, DFT Compiler will not break this loop again.

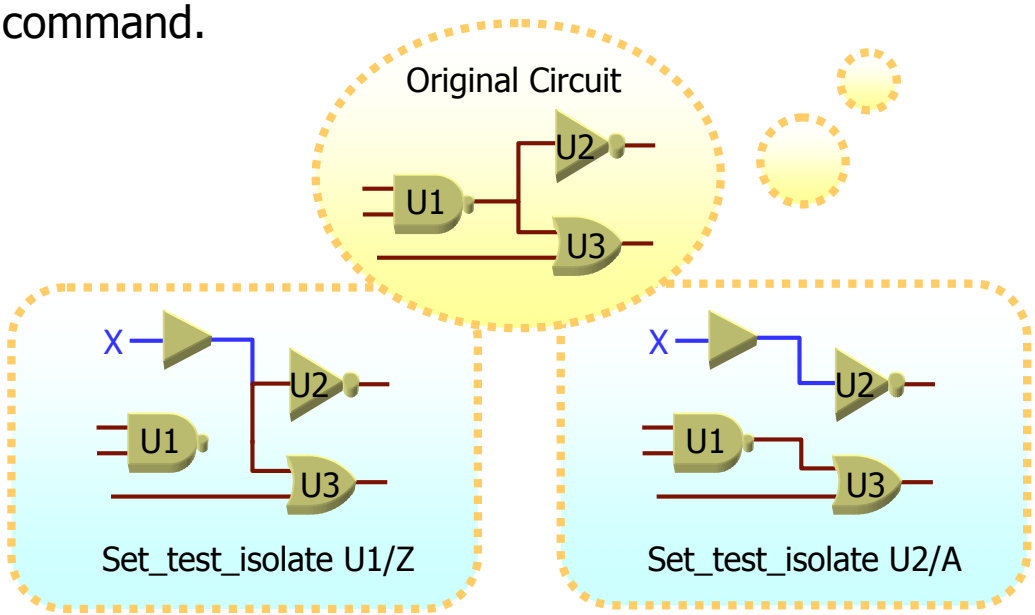


- You can override the pin selected by:

```
set_test_isolate user_selected_pin_name
```

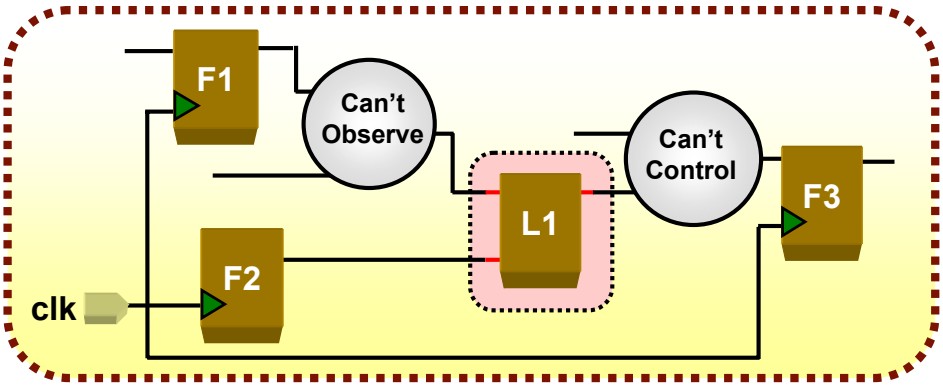

Set_test_isolate

- If a pin or a cell has an unknown value during test: Use set_test_isolate before you run the check_scan command.



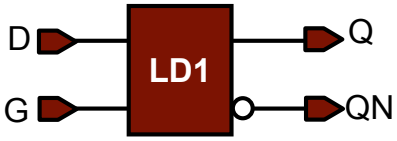
Latches Mixed in Flip-Flop Design

- For flip-flop based scan styles, there are scannable equivalents only for flip-flops, not for latches.
- Latches cannot be made scannable; they remain sequential elements within combinational logic.
- Latch cells will not be replaced. Otherwise, latch cells have no other effect on scan insertion.

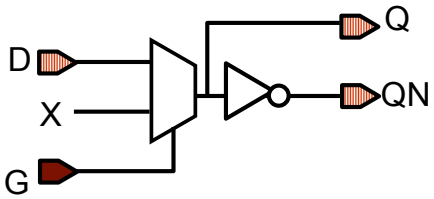


Select a Latch Model

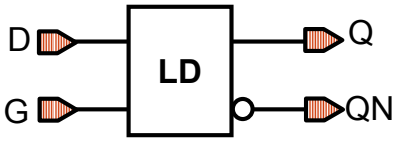
Black Box Latch Model (Default for Combo ATPG)



Combinational Latch Model (need set_scan_transparent)



Sequential Latch Model (Sequential ATPG only)



Testable
Untestable

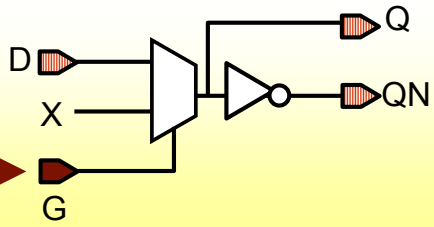
Use Combinational Latch Model

- For full scan designs, using combinational latch model is recommended.

```
set_scan_transparent true <object> -existing
```

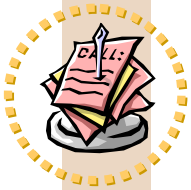
- No design modification is required.

Enable pin and its fan-in cones will not be testable



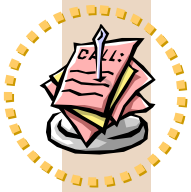
- This model is valid only if the latch enable is **not** connected to a test clock.

DFT Compiler – Lab 2



Bidirectional Ports

Design-for-Testability Using DFT Compiler



Scan Path Insertion

Key of Scan Insertion

- Chain counts, scan ports, clock domains, ...
- Minimize constraint violations:
  - Disable 3 states:

exactly one driver is active during scan shift

```
set_scan_configuration -disable true
```

  - Set bi-directional ports:

```
set_scan_configuration -bidi_mode input
```

to be either input or output mode during scan shift

- Create proper test clock
- Balance scan chains
- Predict scan result by **preview_scan**

Setting the Effort Level

Low Level:
Remaps added scan logic only.
Works only along critical paths.

Medium Level (Default):
Extends remapping, if needed, to surrounding non-scan logic.

```
insert_scan  -map_effort low|medium|high
```

High Level:
Extends remapping, if needed, to surrounding non-scan logic.
Applies sequential remapping, as needed, to flops on critical path.
Reduces area by downsizing cells off critical path, etc.

Switching Off insert_scan Optimization

- Logic optimization is a powerful feature of **insert_scan**. But sometimes you need to turn it off, to freeze an implementation.
- The settings below insert scan with no optimization whatsoever:

```
set test_dont_fix_constraint_violations true
set test_disable_find_best_scan_out true
insert_scan -map_effort low \
            -ignore_compile_design_rules
```

Type of Scan Clock

no_mix (recommend)
- Default. DC XP will generates one scan chain per clock domain.

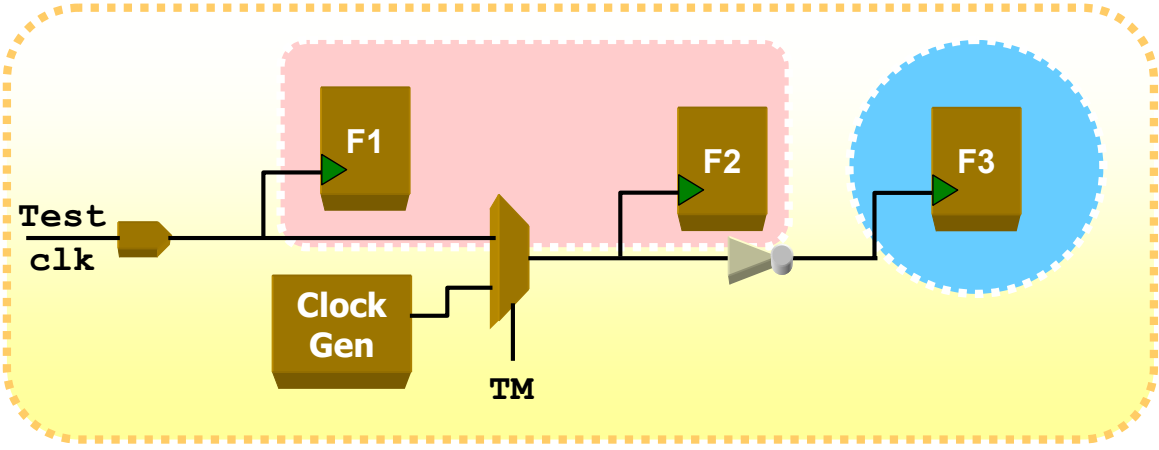
mix_clocks_not_edges (ricky)
- Allow any arbitrary mixing of clocks but not edge within a scan chain.

```
set_scan_configuration -clock_mixing _____
```

mix_clocks (risky)
- Allow any arbitrary mixing of clocks within a scan chain.
- Generate equal length scan chains.

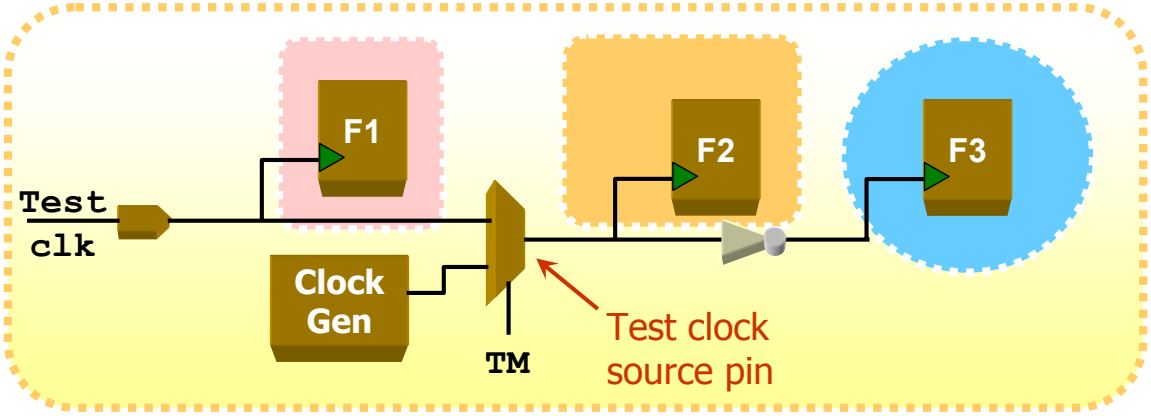
mix_edge (recommend)
- Allow mixing of clock edges within a scan chain.
- Default chain count equals the no. of clocks.

Clocking on Both Edges



- Flip-flops use both edges of the test-clock CLK
- The check_dft algorithm sees two distinct clock domains.
- By default, DFTC inserts a separate scan path for flip-flop F3

An Internal Clock Domain



- Clock gating or MUXing does not by itself create a domain.
- You must apply configuration option -internal_clocks
- Then network has an optimal three test-clock domains.

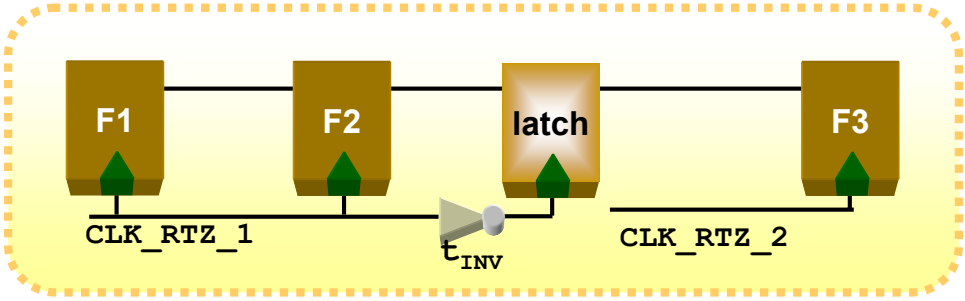
```
set_scan_configuration -internal_clocks true
```

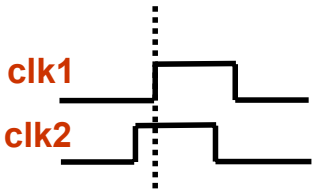

Lockup Latch

```
set_scan_configuration –add_lockup false
```

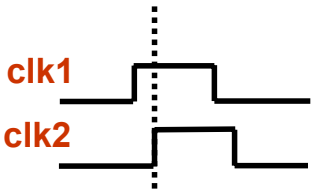
- DFT Compiler orders the FFs within a chain by clock domain.
- DFT Compiler inserts lockup latch between adjacent scan FFs if they are triggered by different clocks and the test clock waveforms are the same.

Why Add Lockup Latch ?



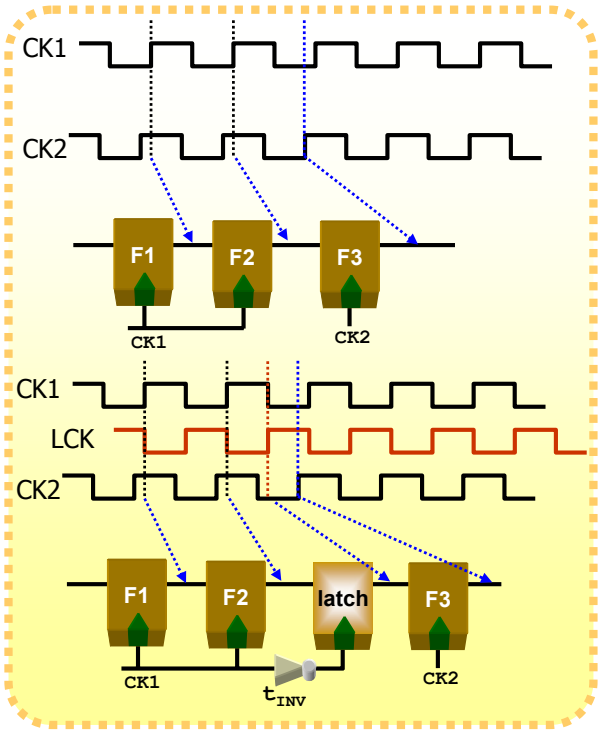


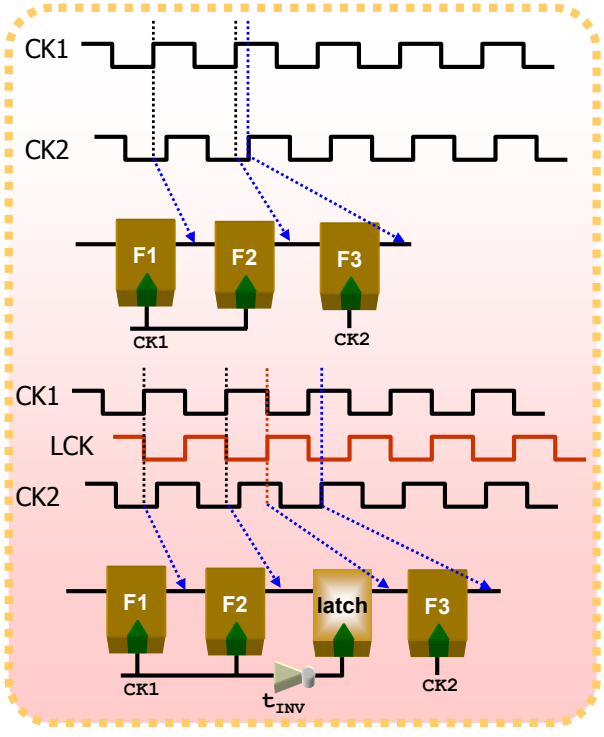
OK!



Big Problem !!
Rearrange clock domain or
add lockup latch

How Lockup Latch Works ?





Custom Scan Path@

```
set_scan_configuration -chain_count N
```

- DFT Compiler will spilt longest chains into smaller chains.
- The scan chains probably will not have equal length.

"Don't-Lengthen" Option:
Prevents later insertion from prepending elements to path

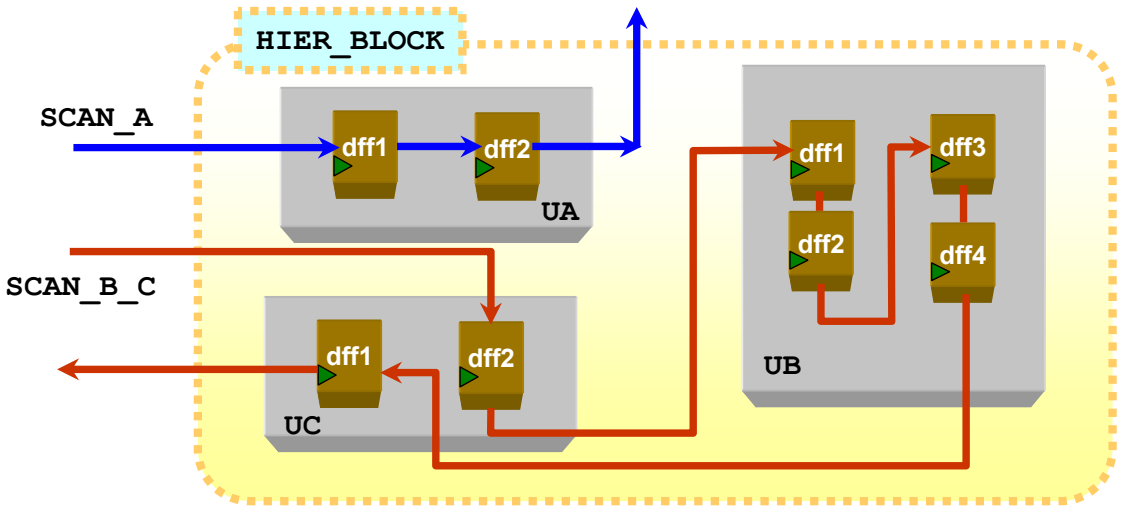
```
set_scan_path CHAIN_4 [list U1 U3 U2] -complete true
```

String Name:
Uniquely labels this path.
Used later to assign ports.

Ordered List:
List of cell names: flip-flops, scan links, segments, or blocks..

Custom Scan Path@@

```
set_scan_path SCAN_A [list UA/dff*] -complete true
set_scan_path SCAN_B_C [list UB UC/dff1]
insert_scan
```



Custom Scan Path@@@

- Specify the scan chain order directly.
- Modify from existing chain.

```
redirect scan.scr { preview_scan -script }
<edit scan.scr>
include scan.scr
preview_scan -show all
```


Specify Scan Enable

- For scan insertion: designate scan ports.

Access Type:
Scan-in, scan-out, etc.

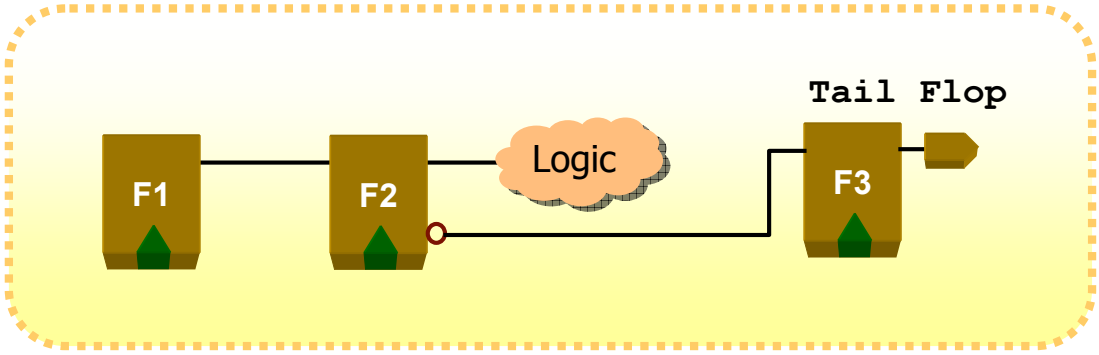
Port List:
SE, SI, SO

```
set_scan_signal test_scan_enable  -port SE \
                                  -hookup BUF/Z \
                                  -sense non_inverted

insert_scan
```

Port Properties:
Port name, buffer pin, polarity.

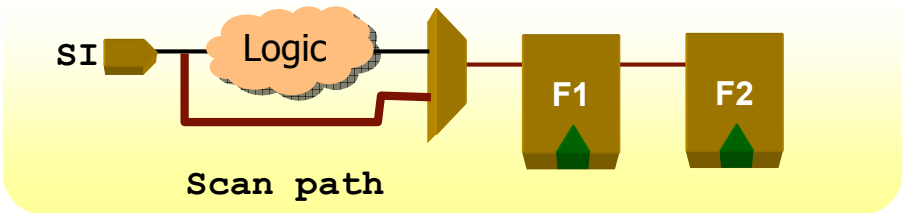
Default Scan-Out Sharing

- DFT Compiler will find a FF directly driving an output port, and move it to the end of the scan chain.

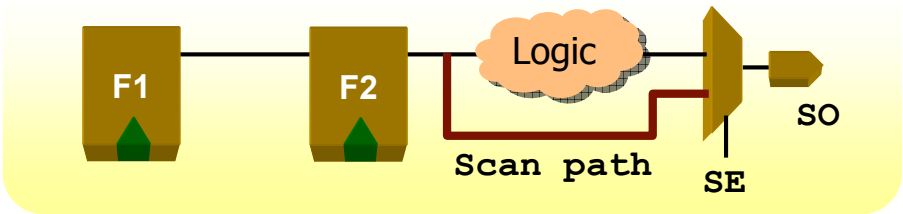


Sharing a Scan-In Scan-Out Port

```
set_scan_signal test_scan_in -port SI
```




```
set_scan_signal test_scan_out -port SO
```



Specify Scan Chain Signals

- For design rule checking: read in ASCII netlist.

```
set_scan_configuration -existing_scan true
set_scan_configuration -bidi_mode input
…
set_scan_signal test_scan_in SI -index 1
set_scan_signal test_scan_in SO -index 1

report_test -scan_path
```

Access Type: Scan-in, scan-out, etc.

Port List: SE, SI, SO

Specifies an index positive integer

Scan Chain Port

- DFT Compiler will automatically generate all ports needed to implement the scan chains if you don't specify them.
- Each scan chain requires a separate scan in and scan out port.
- Scan enable signal can be shared among all scan chains.
- You can use existing functional pins for scan in and scan out ports.
- Always use a dedicated pin for scan enable.

Guidelines for Scan Ports

- Include the scan enable (and also test mode) port in your top-level design.
  - This eliminates having to adjust your top-level testbench to match the design port list after scan chain synthesis.
- Set design rule constraints on the scan enable port.
  - This ensures that the scan enable net is being buffered during scan synthesis.
- Don't specify module functional ports for scan signals if you use bottom-up methodology.

Guidelines for Bottom-Up Methodology

- **DO NOT** insert 3-states disable logic in each module, **DO** it at top level.

```
set_scan_configuration -disable false
```

- Describe the correct disable logic in your HDL code and always let disable set to **false**.
- Specify clock no_mix for each module.

Remove Scan Specification

- Remove scan specifications made using **set_scan_configuration**, **set_scan_path**, **set_scan_segment**, and **set_scan_signal** commands:

```
remove_scan_specification [-all]
[-chain chain_name] [-configuration]
[-segment segment_name] [-signal port_name]
```

- Caution!
  reset_design command removes all constraints and attributes from a design, not just test related constrains and attributes

File Output

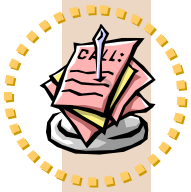
- Create Synthesized Design File

```
write -hierarchy -format verilog -output ore_syn.v
```

- Create STIL Protocol File (SPF)

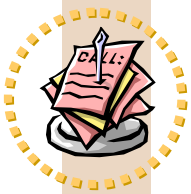
```
write_test_protocol -format stil -out core.spf
```

DFT Compiler – Lab 3



Scan Insertion Specifics

Design-for-Testability Using DFT Compiler



Memory Wrapper & AutoFix

AutoFix and Shadow LogicDFT@

- By default, the AutoFix and Shadow LogicDFT utilities are disabled.
- To use AutoFix, you enable the utility and specify the scope of the design on which it will apply.

New in 2003.06

```
set_dft_configuration -autofix
set_dft_configuration -shadow_wrapper
set_dft_configuration -autofix -shadow_wrapper
replaces:
set_dft_configuration -order {autofix,wrapper}
```

```
report_test -dft
remove_dft_configuration
```


AutoFix and Shadow LogicDFT@@

- To use AutoFix and Shadow LogicDFT, we need to change our command from scan to dft.

```
compile -scan
check_scan
preview_scan
insert_scan
report_test
set_scan_configuration
set_scan_signal
```

```
compile -scan
check_dft
preview_dft
insert_dft
report_test -dft
set_dft_configuration
set_dft_signal
```

AutoFix

- Commands for specifying the behavior of AutoFix:

1.specifies the types of design rule violations to fix:
- uncontrollable clock inputs
- uncontrollable asynchronous
- preset/clear inputs, or both

2. exclude a specified list of leaf-level cells or hierarchical cells from being fixed by AutoFix.

```
1.set_autofix_configuration
2.set_autofix_element
3.set_autofix_clock
```

3. specify the clock that is connected to the test points on the designated parts of the design.

AutoFix Script

```
read_file -f verilog TOP.v
current_design TOP
create_test_clock -w [list 45 55] clock
compile -scan
set_dft_configuration -autofix
set_dft_signal test_mode -port test_mode
set_test_hold 1 test_mode
set_autofix_clock CLKA UA
set_autofix_clock CLKB UB
set_autofix_element UC -clock false -async true
preview_dft
insert_dft
check_dft
```

Testing RAM/ROM

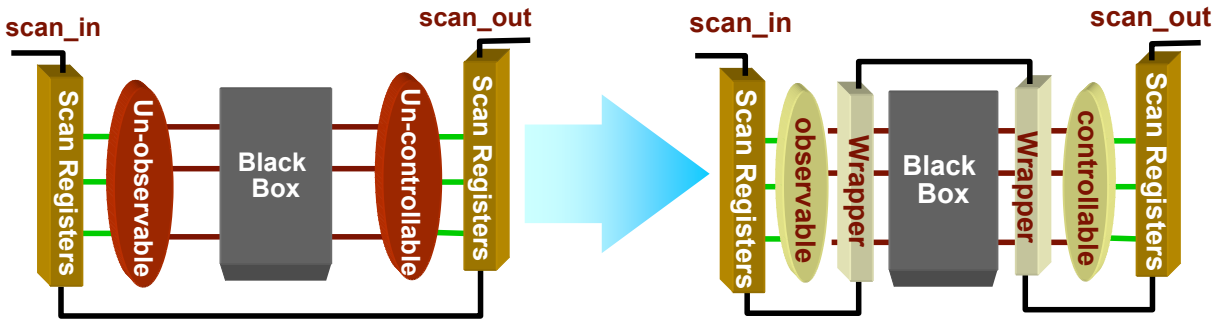
1. Testing RAM/ROM for internal faults
2. Applying a black-box model
3. Inserting test points
4. Writing functional models (will discussed in TetraMAX)

1 Testing RAM/ROM For Internal Faults

- The transistor-level fabric of a RAM/ROM can't be tested by gate-level stuck-at-fault techniques.
- Use direct pin access or memory BIST.
- Direct pin access:
  - Need more extra pins
  - Routing congestion
  - Impact on performance
- BIST:
  - Add fewer BIST logic
  - Reduce routing

3 Inserting Test Points

- The testability of shadow logic can be improved by automatic insertion of DFT control-and-observe test points around a black-box RAM/ROM.



Inserting Test Points Flow

- The commands you can use to specify shadow wrapper logic include the following:

```
1.set_wrapper_element
2.set_port_configuration
3.remove_wrapper_element
4.remove_port_configuration
```

1.specifies what cell instances in the design are to be given a test wrapper.

2. identifies the read and write inputs, clock input, and three-state outputs of the shadow block that is receiving the test wrapper.

RAM Wrapper Example@

```
compile -scan
...
create_test_clock -w [list 45 55] ck
set_dft_configuration -shadow_wrapper
set_port_configuration -cell RAM -port "q" -read [list "OEN" 1] -clock ck
set_port_configuration -cell RAM -clock ck
set_port_configuration -cell RAM -port "a" -write [list "WEN" 0] -clock ck
set_port_configuration -cell RAM -port "d" -write [list "WEN" 0] -clock ck
set_wrapper_element RAM_0 -type shadow
preview_dft
insert_dft
compile -incr  // for dangling nets (see SOLV-IT! Test-461)
check_dft
...
```

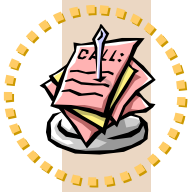

RAM Wrapper Example@@

```
preview_dft
```

```
************ Test Point Plan Report ************
Total number of test points   : 399
Number of Autofix test points: 0
Number of Wrapper test points: 399
Number of test modes          : 1
Number of test point enables  : 0
Number of data sources        : 128
Number of data sinks          : 271
************************************************
```

This summary shows that DFT compiler is going to add 399 scan flip flops to create a shadow wrapper around the RAM.

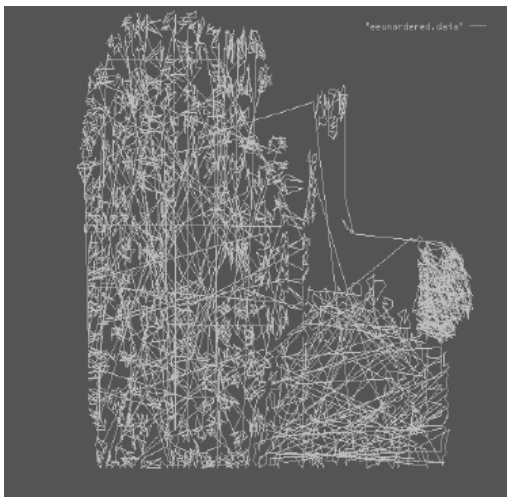
Design for Testability Using DFT Compiler



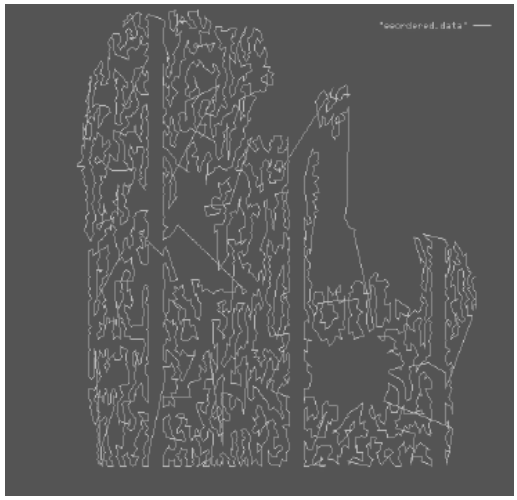
Other Skill

Routing Congestion Gone

- Scan path reorder in P&R tools.



Default Scan Ordering



Placement-Based

Scan Path Reorder Flow @

A design with scan chain

check_test

Set protocol

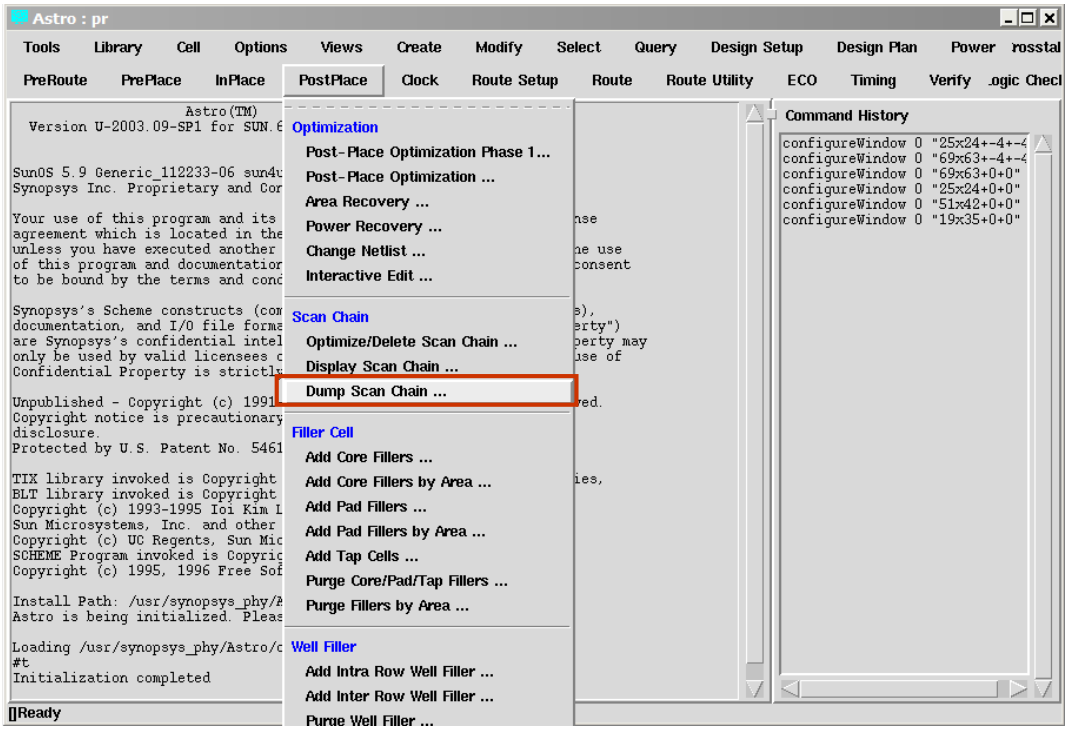
set_scan_configuration -prtool avant / cadence

Write files

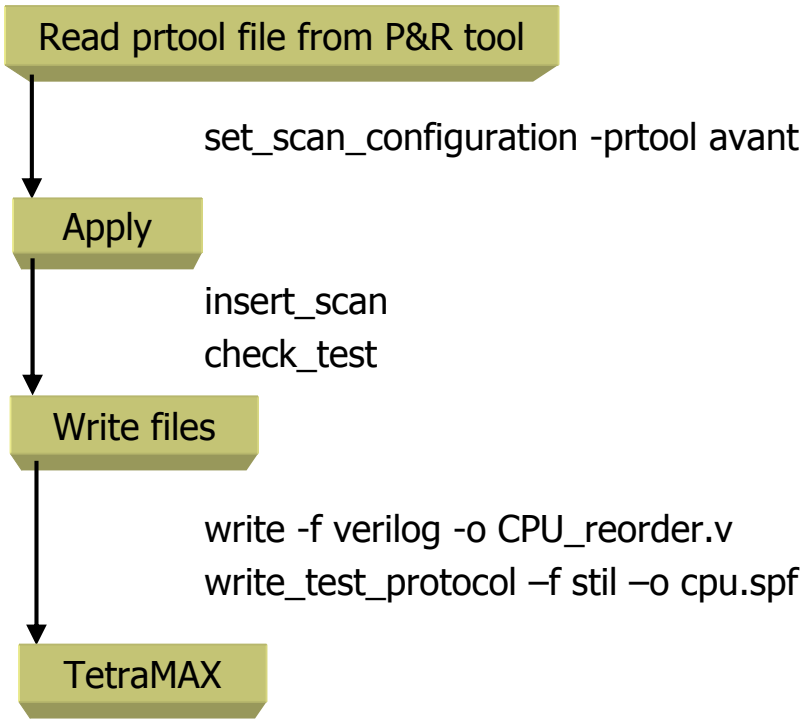
write_layout_scan -out CPU.def
write -f verilog -o CPU -initial.v
write -format db -o CPU.db

P & R Tool

Scan Path Reorder Flow @@



Scan Path Reorder Flow @@@



Test Data Volume Reduction

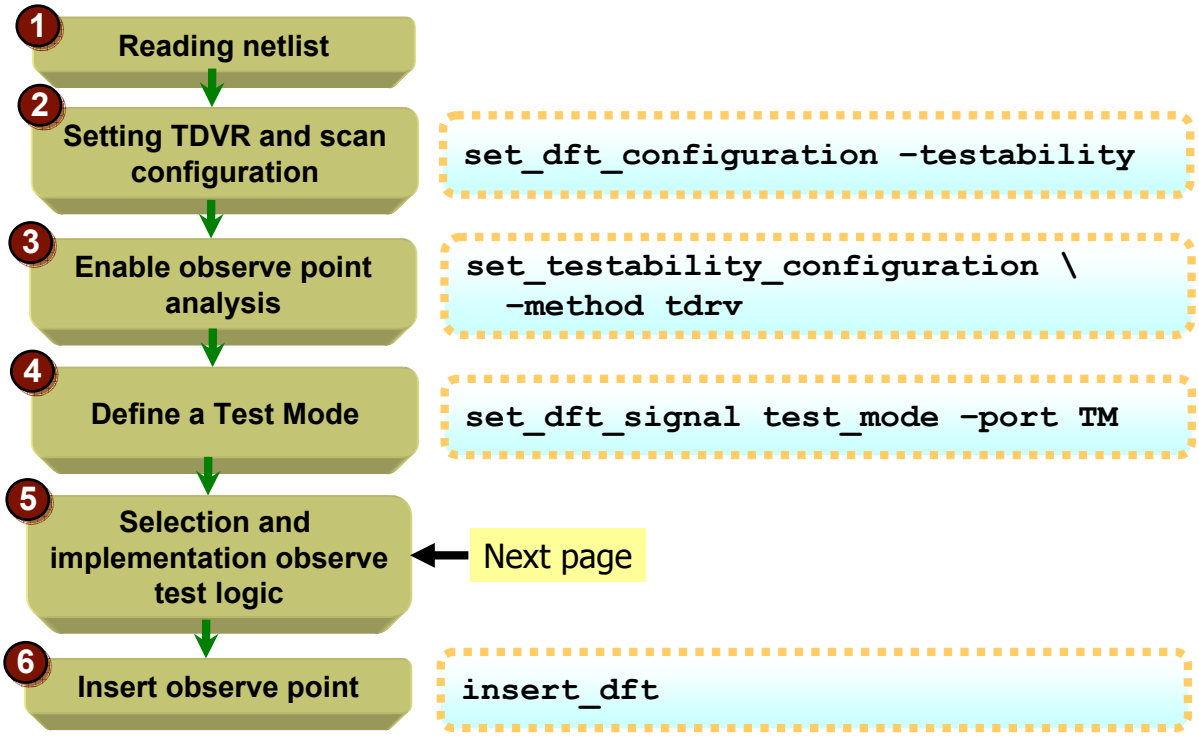
New in 2003.06

- The cost of test is increasing with increasing complexity of design.
- The cost of test is directly proportional to scan data volume.
- TDVR provides reduction in test data volume without compromising test coverage.

TDVR Technique

- **Test data volume reduction** is achieved by analyzing observe points in the design.
- **An observe flip flop** inserted at the observe point can observe a large number of **hard-to-detect** faults which results in a significant reduction in pattern count.
- DFTC supports both bottom-up and top-down observe analysis. For SoC designs, bottom up observe analysis is the preferred method.

TDVR Flow



set_testability_configuration

- set_testability_configuration

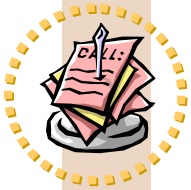
```
-method tdrv
[-max_observe_point [integer] ]
[-max_observe_logic_area [integer] ]
[-power_saving_on <true/false> ]
```

- Preview scan and observe test point logic

```
previev_dft -test_points all
```

```
Run autofix before TDVR
```


DFT Compiler – Lab 4

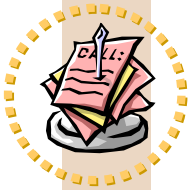


Initialization Sequences & AutoFix

Day 1 Review

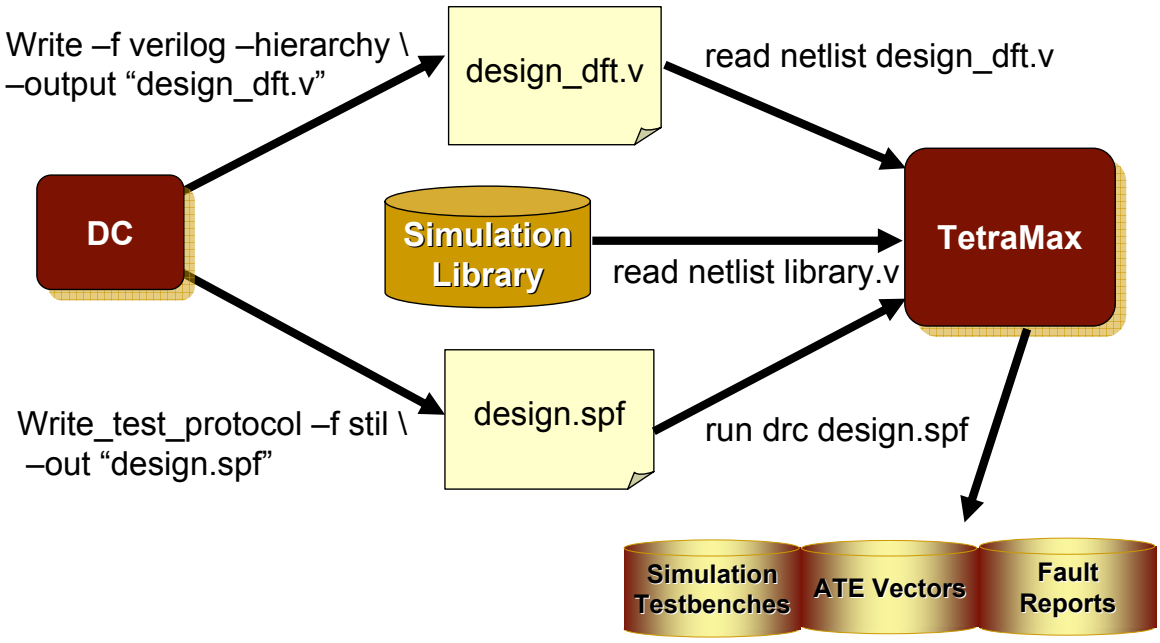
- Basic Concepts
- DFT Compiler Flow
- Basic DFT Techniques
- Advanced DFT Techniques
- Scan Path Insertion
- Inserting Scan Chains With insert_dft
- Lab1 ~ Lab4

Design-for-Testability ATPG with TetraMAX

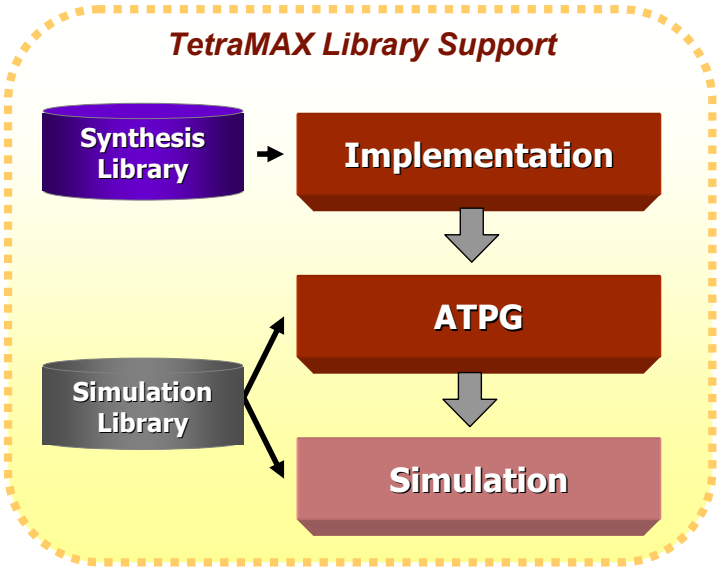


TetraMAX Overview

DFT compiler to TetraMax

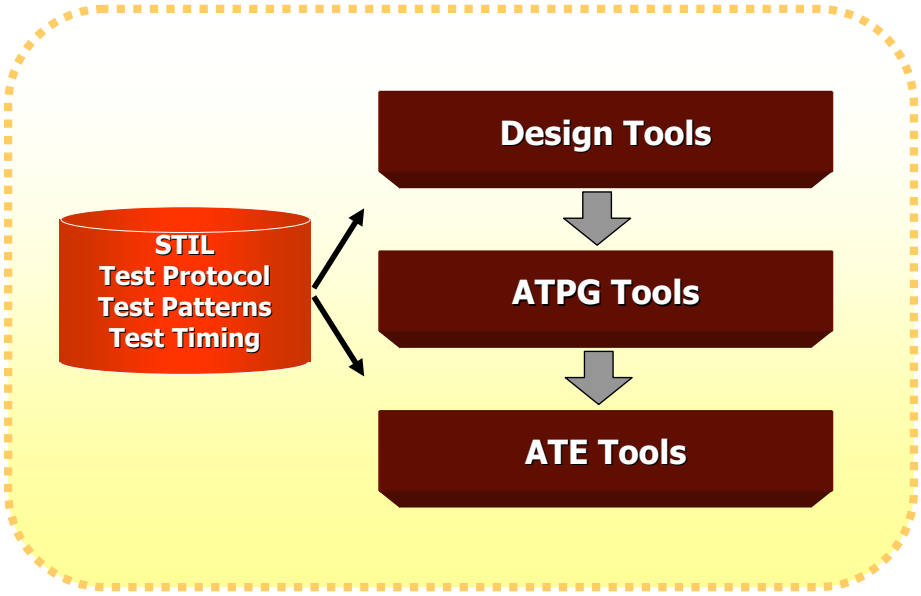


Innovative Technology Library Support



Uses existing ASIC vendor functional simulation libraries (Verilog or VHDL)

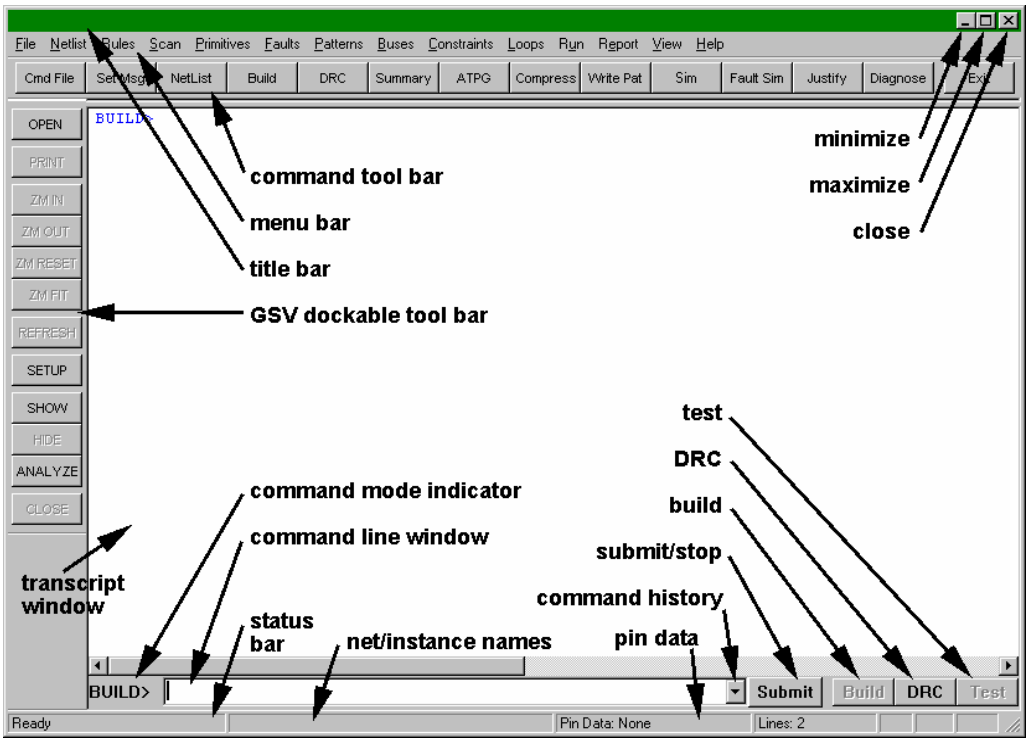
Support STIL (input and output)



Other Feature

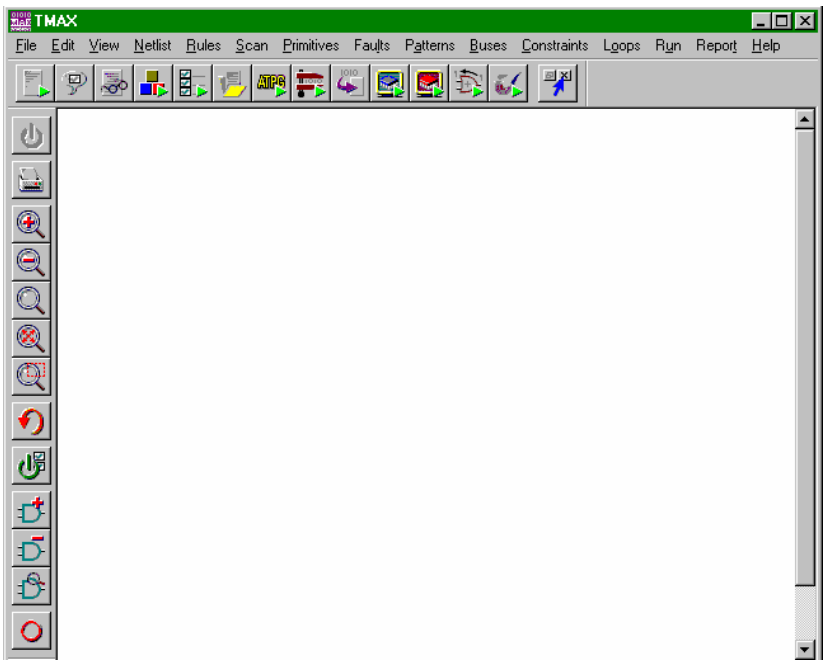
- Support RAM and ROM.
- Support IDDQ , transition pattern generation.
- Support Verilog, EDIF, and VHDL netlist input and pattern input/output.
- Provide static and dynamic pattern compression techniques.
- Provide built-in GZIP compression function for reading and writing (designs, libraries, protocols, patterns, and fault lists).

TetraMax GUI

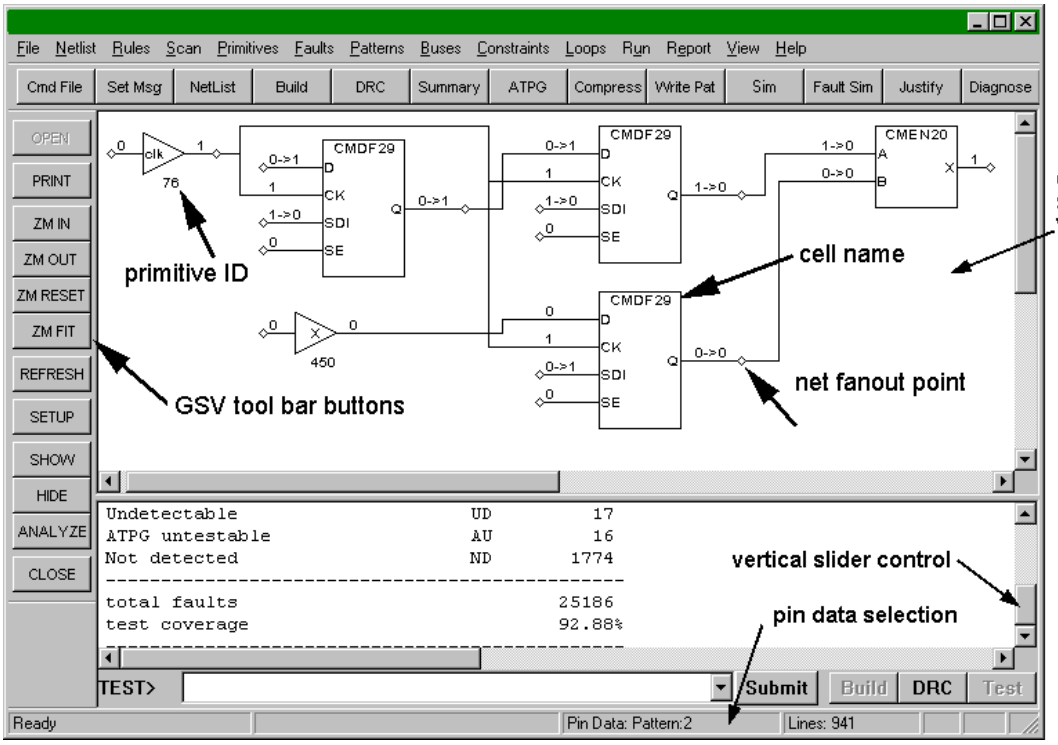


Alternative ICON Look

Edit -> Environment -> GUI -> Use toolbars composed of bitmaps



The GSV Tool Bar and Window



Command Input and Script Files

- Command input may be specified in a number of ways:
  - menus, GUI buttons, or dialog boxes
  - typed at the command input line
  - read from command files:

```
BUILD> source filename
```

- By default, TetraMAX aborts a script file when a command returns an error
- To continue executing scripts, use:

```
BUILD> set_command noabort
```

Command Redirection

> to replace a file or create the file if it does not exist.

>> to append to a file, or create if the file does not exist.

>? to create a file that does not already exist, if the file exists an error will be reported.

Examples:

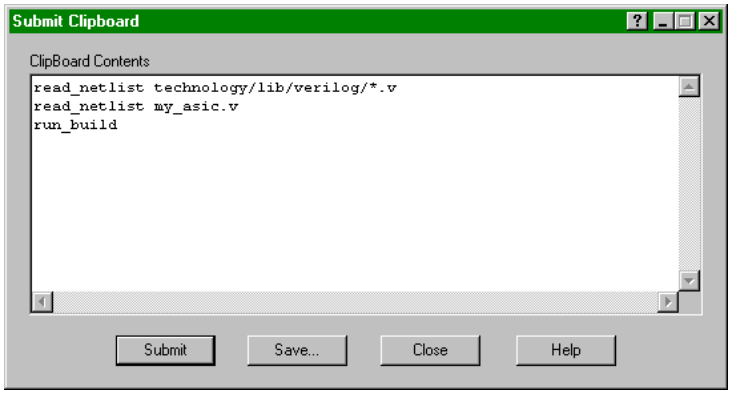
```
report_settings > current_settings.txt
report_clocks >> current_settings.txt
```


Hot key

- Within the ATPG tool window the Microsoft Windows emulation environment exists so cut/paste is done with command pop-up menus (or CTRL-C and CTRL-V).
- Other Hot key can be found in GUI menu.

Multi-Line Input in the GUI

- Multiple line input can be performed by bringing up the "clipboard" (type CTRL-M) and then pasting a range of command lines.



- Multiple commands may also be entered by separating each command by a semicolon and white space.

```
BUILD> build -force; read net mylib.v; run build
```


Command Aliases

```
alias
```

read_verilog	same as read_netlist
read_edif	same as read netlist
read_vhdl	same as read netlist
add_set	same as add clock
add_reset	same as add clock
add_set_reset	same as add clock
report_alias	same as alias
check_test	same as run drc
history	same as report command -history
include	same as source
link	same as run_build_model
create_test_clock	same as add clock
create_test_patterns	same as run atpg -auto
read_init_protocol	same as set drc
read_test_protocol	same as run drc
set_test_hold	same as add pi constraint
write_test	same as write patterns

```
BUILD> alias  readnet  read_netlist -delete
BUILD> alias  exit   exit -force
```

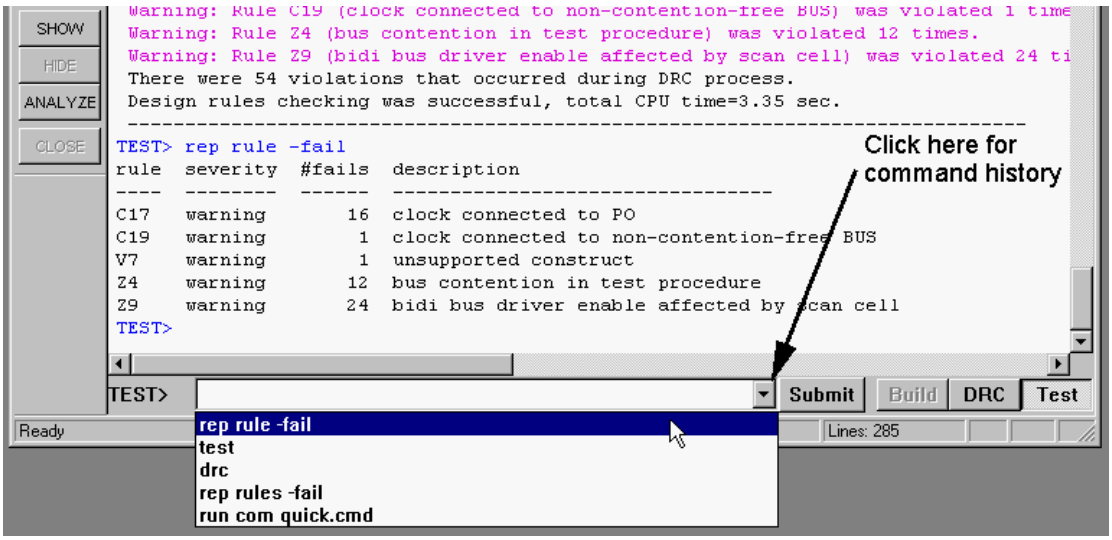
Comments

- Blank lines and "//" comments will appear in the transcript window.
- "#" comments will produce a blank line in the transcript without showing any text.
- There is no multi-line comment capability : "/* */"
- A limited set of shell commands are supported which can be performed directly from the command line:
  - cd, pwd, ls, cat, cp, mv, rm, mkdir, clear

These commands are implemented internally so they do not match exactly their UNIX equivalent commands

Command History

- Lists all commands entered through a dialog, or menu, or typed at the command input.



Log Files and Message Control

- Write log message to file:

```
set message log logfile [-replace | -append]
```

- Turn off displaying log message to transcript:

```
set message [nodisplay | display]
```

- Stop the comments appearing in the transcript:

```
set message [-transcript_comments |
             -notranscript_comments]
```

- Select message level:

```
set message [-level expert | standard ]
```


Help command

```
BUILD> help add
```

```
Add Atpg Constraints          Add Atpg Gates
Add CEll Constraints          Add CLocks
Add Equivalent Nofaults       Add Faults
Add Net Connections           Add Nofaults
Add PI Constraints            Add PI Equivalences
Add PO Masks
```

```
BUILD> help read netlist
```

```
REAd NEtlist <file_name> [-Format <Edif |
VErilog | VHdl>][-Sensitive | -INSensitive]
[-Delete] [-Noabort] [-Verbose]
```

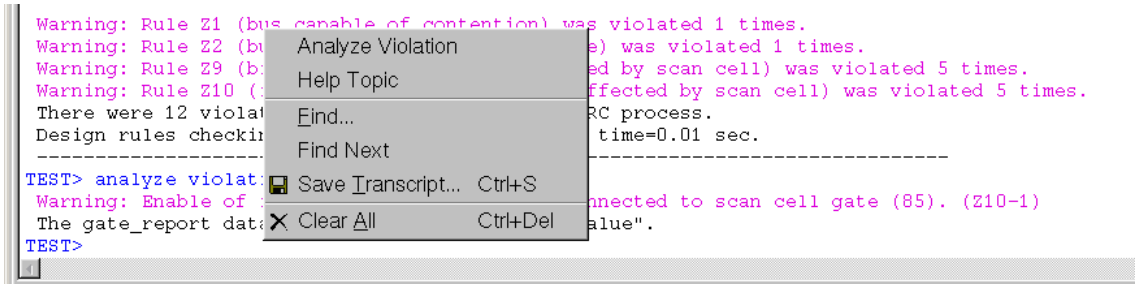
MAN for command reference

- Entering "**man**" and a command name, a message ID, or a DFT rule ID or violation ID will open up the **on-line help** to the reference page for that topic.

```
BUILD> man getting_started  // a topic
BUILD> man add clock        // a command
TEST> man report faults     // a command
TEST> man z4-6              // a violation
TEST> man m69               // a message ID
```

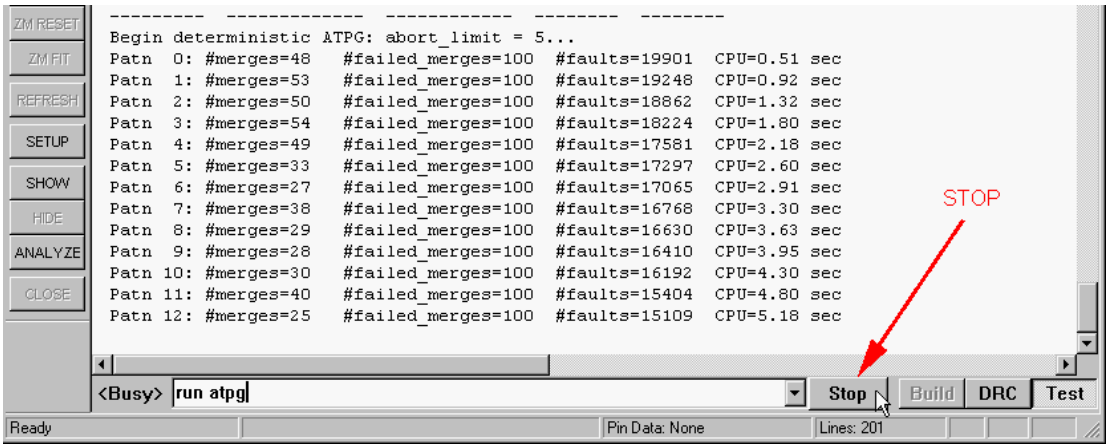

Hypertext Links to On-Line Help

- Messages in the transcript are hypertext linked to the on-line help manual and generally appear in an alternate color.
- Using **Right Mouse Button** will open on-line help to the page explaining the message.



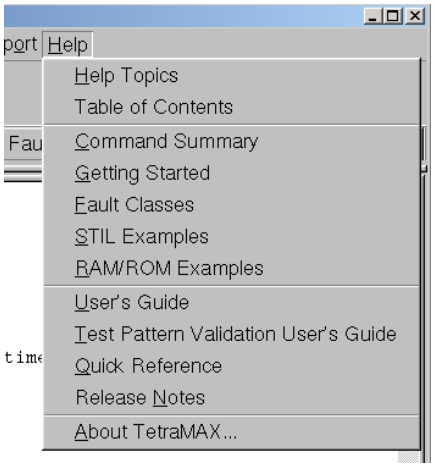
Stop Process

- "Submit" button changes to "**Stop**" while performing operation. You can abort this operation by this "Stop" button.
- **CTRL-C** and **CTRL-Break** do the same job.

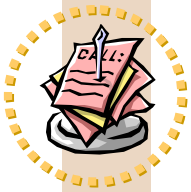


Accessing the User's Guide

- From the HELP menu the TetraMAX User's Guide can be reviewed. This invokes the Adobe Acrobat viewer.
- Also available are on-line versions of Release Notes as well as direct access to useful help topics: Getting Started, RAM/ROM examples, etc.

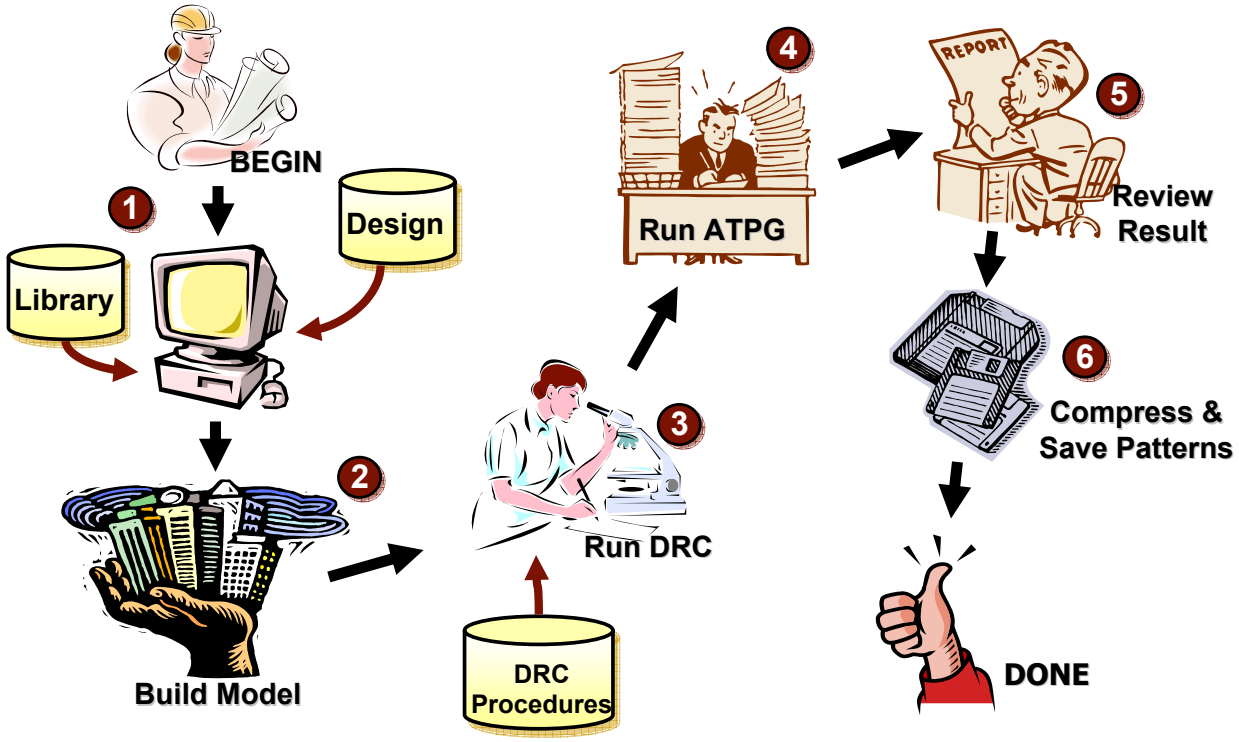


Design-for-Testability ATPG with TetraMAX



TetraMAX Baseline Flow

The TetraMAX Flow



TetraMAX ATPG Command Modes

- BUILD mode:
  - Initial mode
  - Read in design, libraries, models
  - Construct ATPG simulation model in preparation for DRCs
- DRC mode:
  - Testability Design Rule Checks (DRCs) are performed
  - Successful conclusion of DRCs sets mode to "TEST"
- TEST mode:
  - ATPG, Fault Simulation, Fault Diagnosis are performed
  - Test program files, simulation testbenches, etc. written out

Design Rule Checks

- Checked during read netlist, run build_model:
  - N rules (netlist) and B rules (build_model)
- Checked during run drc:
  - S rules (scan chain or shift)
  - C rules (clocks or capture)
  - Z rules (internal tristate buses and bidi-directional pins)
  - X rules (combinational feedback loops)
  - V rules (vector statements in the SPF)
- Other rule categories:
  - P rules (Path delay checks for Delay Test ATPG)

1 Read Design@

- Support Verilog, EDIF, and VHDL netlist input or mixture of above.
- Netlists may be either flat or hierarchical.
- Netlists may exist as a single file or multiple files.
- Netlist compression (none, or proprietary, or GZIP) is auto-detected.
- Duplication of module while reading netlist, it will keep the last module encountered.

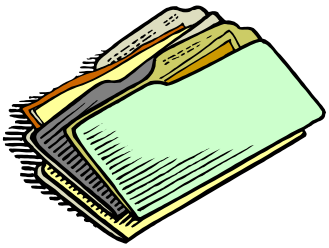
1 Read Design@@

- Use NETLIST button or read netlist command:

```
BUILD > read netlist mydesign.v
```

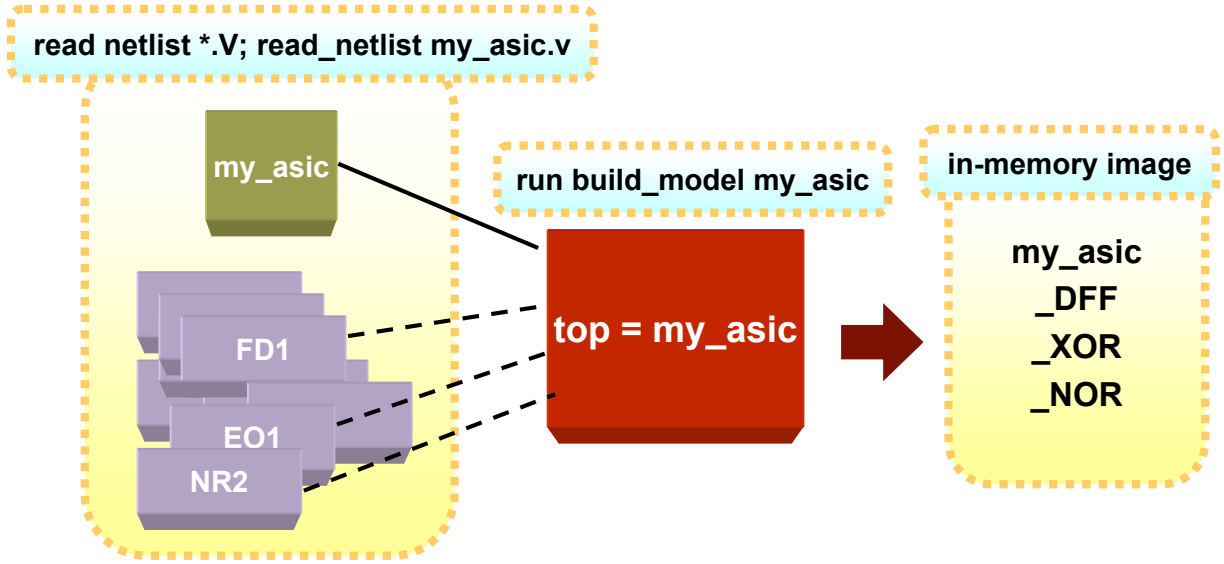
- Wildcards '*' and '?' are support:

```
BUILD > read netlist design/*/??design*.v
```

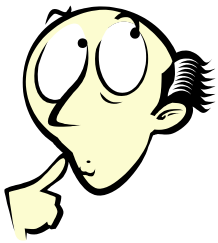


2 Build Model

- Build the top-level module in memory for ATPG algorithm.



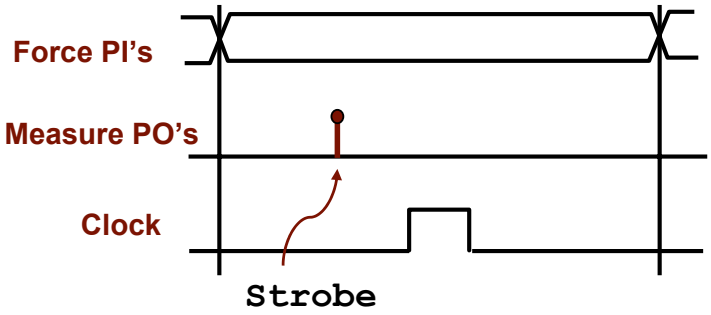
Some TetraMAX Terms to Remember



- PI = Primary Input
- PO = Primary Output
- Constraint = a restriction to be honored when generating ATPG patterns.
- Gate ID = unique integer assigned to each distinct ATPG primitive in the flattened design.
- CLOCK = **any** signal that affects the stored state of a sequential device including asynchronous sets and resets as well as RAM write controls.

ATE Tester Cycles

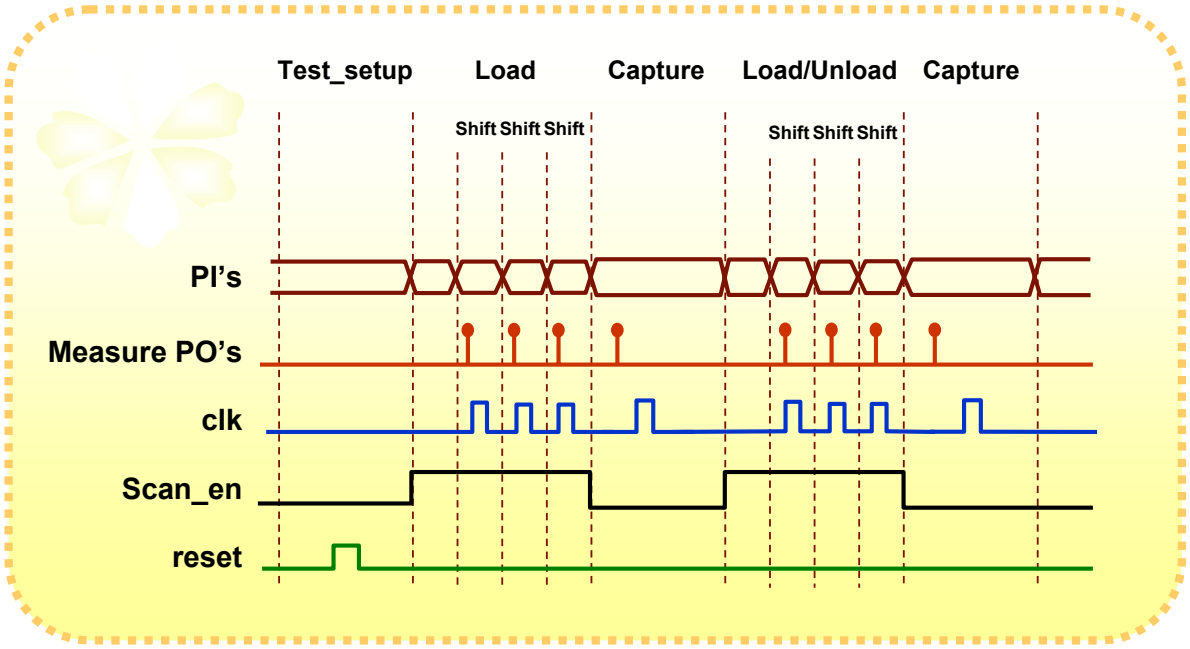
- At the MICRO level, each individual ATE tester cycle consists of:
  1. force PI's
  2. measure PO's
  3. optionally pulse a clock or asynchronous set/reset



Stobe timing is different from DFT compiler.
We can set this in DFT compiler:

```
test_default_strobe = 35
```


Typical Event Flow



Event Order in Terms of SPF Procedures

- At the MACRO level the fundamental event order is:
  1. Initialize the design (apply test_setup procedure).
  2. Force Virtual Inputs by shifting scan data using a shift clock:
     a. load_unload procedure to enable scan shifting.
     b. shift procedure applied "N" times, where N is longest chain.
     c. return to "normal" functional mode.
  3. Force Primary Inputs (PI's).
  4. Measure Primary Outputs (PO's).
  5. Apply a "capture" clock to capture virtual outputs.
  6. Measure Virtual Outputs by shifting out the scan data.

Summary of Procedures in the SPF

- Required
  - `load_unload {…}`
  - `shift {…}`
- Required but defaults inferred
  - `capture`
  - `capture_XXX`
- Conditionally Required
  - `test_setup {…}`
  - `master_observe {…}`
  - `shadow_observe {…}`

- The load_unload procedure and Shift statements are required.
- The capture procedures are required but have a default 3 cycle implementation if not explicitly defined.
- The other procedures may be required depending upon the individual design and style of scan cells used.

Defining PI constraints

- To instruct TetraMAX to hold a port constant during ATPG pattern generation define a PI constraint:

```
add pi constraint <held_value> <port_name>
```

- Constraints can also be defined in the SPF file.
- Constraints of Z can only be defined for bidirectional or tri-statable outputs.

Review the existing constraints

- To review the existing list of constraints use:

```
report pi constraints
```

- To review a list of input, output, bidi, or all ports use

```
report primitives [- PIS | -POS | -PIOs | -PORts]
```

Defining PI Equivalents

- When two ports should always be driven to the same or complimentary values by ATPG patterns you must define a PI (Primary Input) equivalence relationship:

  two ports to keep the same:

```
add pi equiv ENA_1 ENA_2
```

  two ports to keep complementary (differential inputs):

```
add pi equiv POS -inv NEG
```

- ATPG must honor this relationship when generating patterns but you may momentarily violate it within the SPF procedures you create.

Defining Clocks

- Defining clocks may be done manually using:

```
add clock <off_state> <port_name>...
BUILD> add clock 0 clk1 clk2
BUILD> add clock 1 /SLAVE_CK
```

- Clocks may be defined within the STIL protocol file.
- To check on the list of defined clocks defined either manually or by a STIL protocol file referenced by a run drc command use:

```
report clocks
```

Defining Asynchronous Set/Reset

- TetraMAX requires that all ports which **affect the stored state of sequential devices** be defined as "clocks".
- This includes traditional clocks as well as the asynchronous set and reset controls of flip-flops and latches and the read/write controls of RAMS.

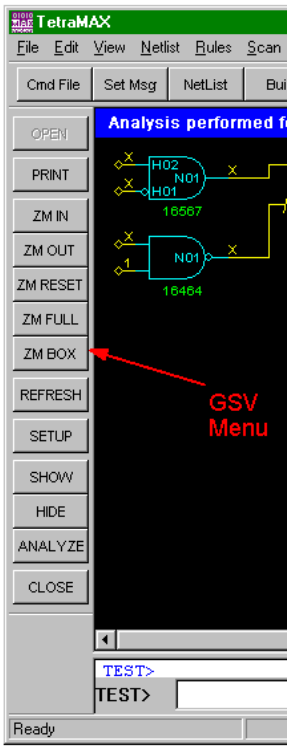
```
add clock <off_state> <port_name>...
DRC> add clock 1 RESET_B
```


3 Perform DFT Rules Check

```
run drc SPF_file
```

- The scan chains are checked for working order during shift mode.
- Clocks and asyn. Set/Reset ports are checked to see that they are controlled **only** by PIs.
- Clocks/Sets/Resets are checked for off state while switching from normal to scan shift and back again.
- Multi-driver nets are checked for contention.

3 Analyze Rule Violations@



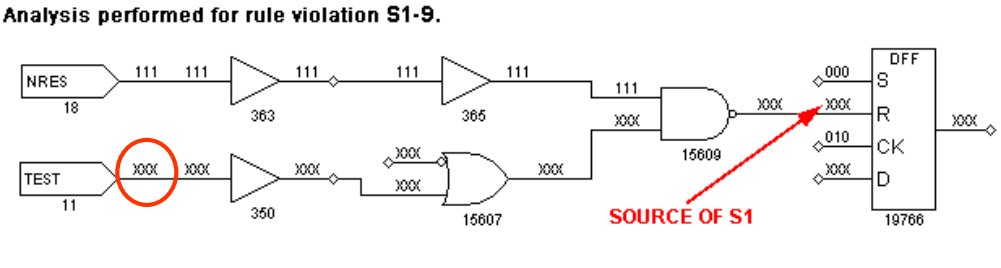
- During the DRC analysis you may get a rules violations such as:

```
Error: Chain "c3" blocked at DFF
gate /my_asic/per/reg3 (19766)
after tracing 3 cells. (S1-9)
```

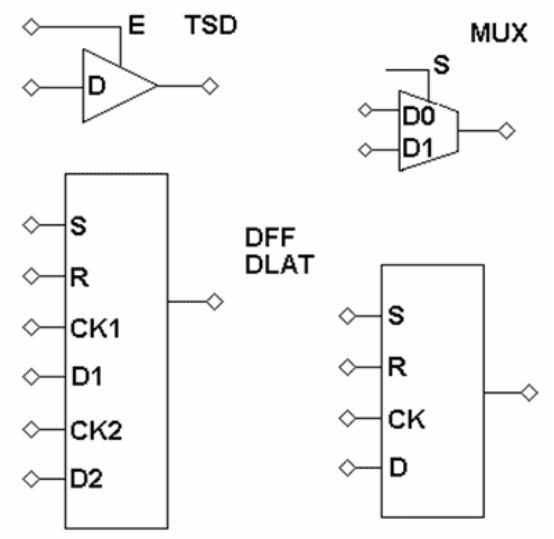
- Use the ANALYZE button, select violation S1-9, and the GSV window will open to display the problem graphically as a schematic segment.

3 Analyze Rule Violations@@

- Tracking back you see this X state comes from the TEST pin.
- Was TEST pin defined as a constrained pin? Was it initialized in load_unload?
- Edit SPF and run DRC again.

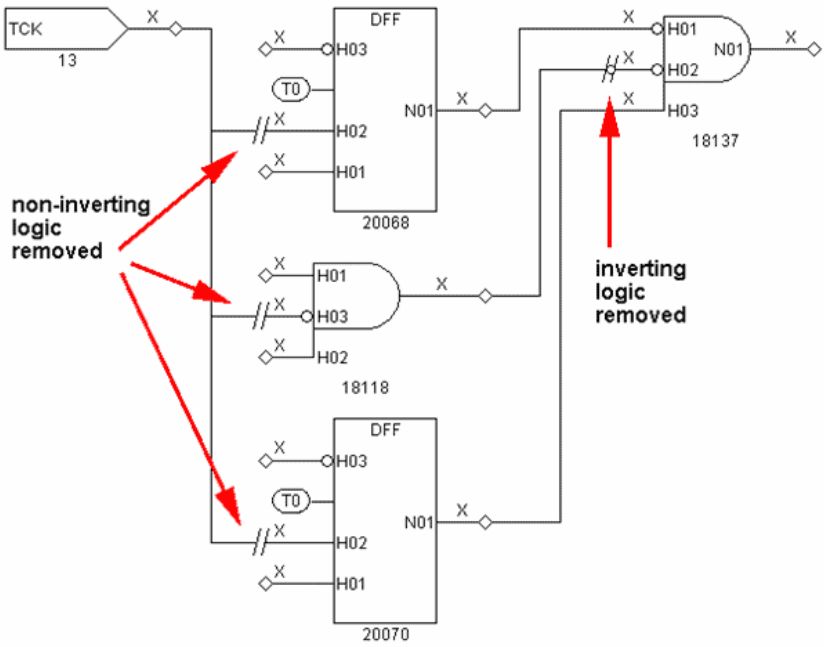


Understanding GSV@



- In the Graphical Schematic Viewer the names of pins are not generally shown, however they are always presented in a constant order from top to bottom:
  - TSD - active high Enable then Data.
  - MUX - select, D0 input..
  - DFF - asyn set, reset, clk, data.

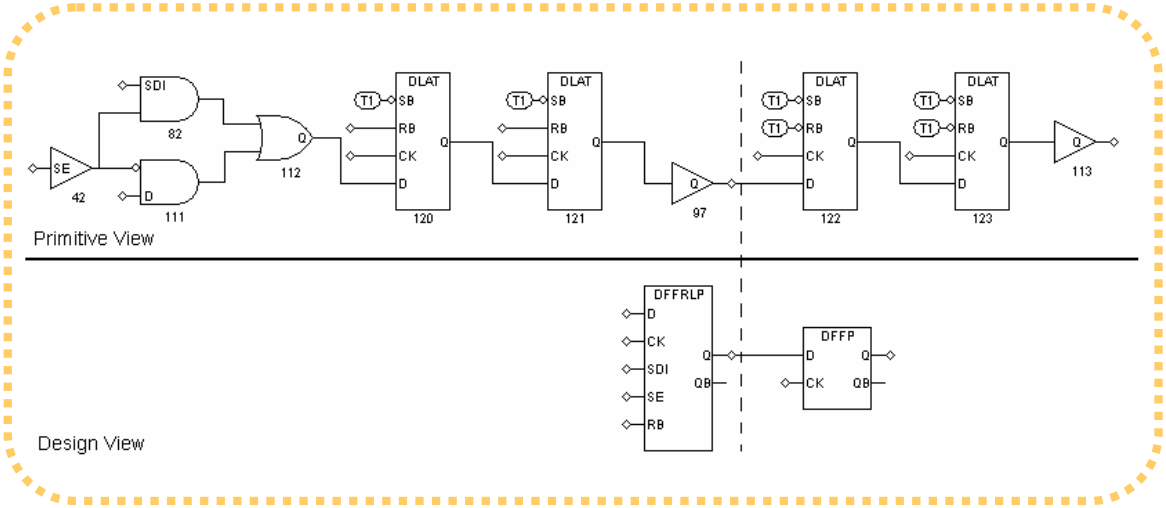
Understanding GSV@@



- Under the SETUP button on the GSV menu is a selection for hiding buffers and inverters.
- Hidden logic is shown with '**//**'
- Hidden logic with an inversion is shown with '**/o/**'

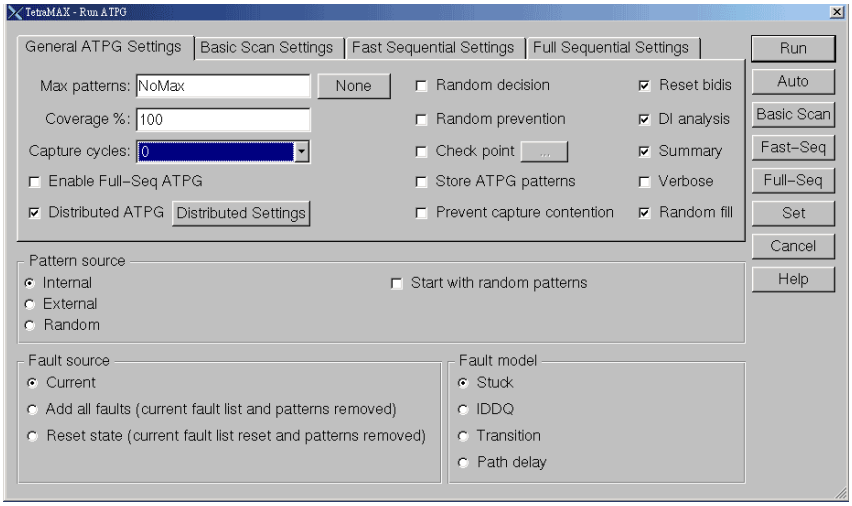
Understanding GSV@@@

- Under the SETUP button for the GSV menu is a control on whether to draw the schematic using ATPG primitives or Design (Vendor cell) level.



4 Generating ATPG Patterns@

- Adjust ATPG Effort:
  - Abort limit
  - Capture cycles
  - Learn depth
  - CPU seconds/fault
  - CPU seconds total
  - Pattern limit
  - Coverage limit
  - Clock choices
  - Merge effort
  - Fault model



ATPG Effort: Abort Limit

```
set atpg -abort N
```

- Default N is 10, reasonable range 5-1000.
- During the ATPG algorithm certain assumptions are made and paths sensitized in an attempt to control and observe fault locations. If these assumptions are proved false or lead to blocks then the algorithm backtracks and remakes the decision until it hits the "abort" limit.

ATPG Effort: Capture Cycles

```
set atpg -capture_cycles <n>
```

- Suggest initial settings is 4. Range: 0-10.
- This controls the sequential ATPG algorithm's depth and defaults to zero (off).
- This indicates the max. number of capture_XXX procedures to be applied between the scan chain load and unload.

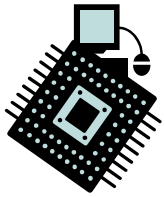
ATPG Effort: Learn Depth

```
set atpg -learn N
```

- Default is 0, range: 0-5.
- This controls dynamic learning that can occur during ATPG.
- Use of this option may result in excessive CPU time for high values of learn.

ATPG Effort: CPU Limit

```
set atpg -time max_sec_per_fault [max_sec_per_run]
```



- Use 0 to turn off this limit.
- Use this option to limit the number of:
  - CPU seconds spent on a particular fault.
  - CPU seconds spent on the current "run atpg" command.

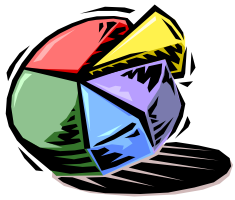
ATPG Effort: Pattern Limit

```
set atpg -patterns N
```

- Use 0 to disable this limit. Default 0.
- This option enables the ATPG process will stop when the specified number of patterns is achieved or fault detection is completed.

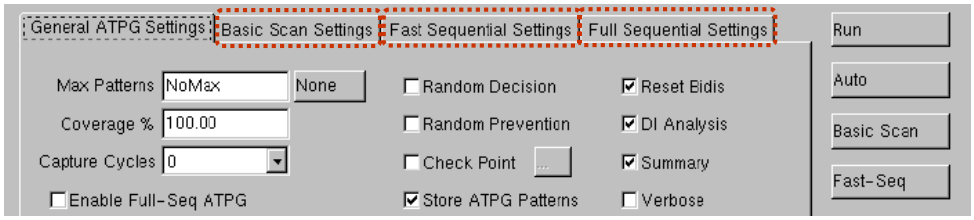
ATPG Effort: Coverage Limit

```
set atpg -coverage NN.N
```



- Default is 100, usable range 1.0-100.0
- This control indicates a stopping point when a certain test coverage is exceeded.
- The limit is checked on 32 pattern generation boundaries so the actual test coverage is usually slightly higher than requested.

3 ATPG Engines



- Basic-scan:
  - For full-scan logic
  - Highest performance
- Fast-sequential:
  - For near-full-scan designs with memories
- Full-sequential:
  - For designs with DFT violations

4 Generating ATPG Patterns

- Generate ATPG Patterns
  - After completing DFT rules checking successfully:

```
TEST > add_faults -all
TEST > run_atpg -auto
```

  - The transcript will indicate the progress of pattern counts, faults detected and remaining, and fault coverage.

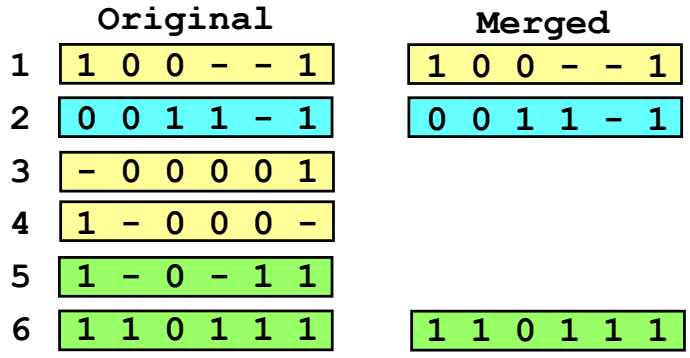
5 Review Results

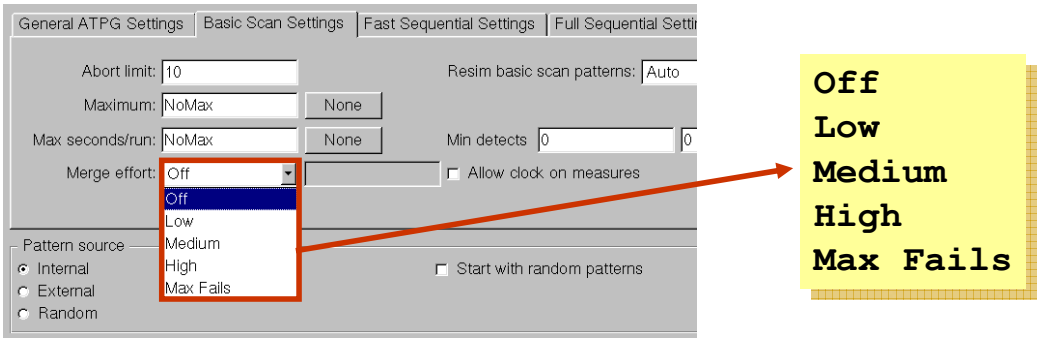
```
TEST > report_summaries
 Uncollapsed Fault Summary Report
-----------------------------------------------
fault class                     code   #faults
------------------------------  ----  ---------
Detected                         DT      83391
Possibly detected                PT         54
Undetectable                     UD       1085
ATPG untestable                  AU       3742
Not detected                     ND        136
-----------------------------------------------
total faults                             88408
test coverage                            95.53%
-----------------------------------------------
            Pattern Summary Report
-----------------------------------------------
#internal patterns                         750
-----------------------------------------------
```


6 Pattern Compression

- TetraMAX provides two different methods for pattern compression:
  - DYNAMIC - performed during ATPG pattern generation by using the merge option to the set atpg command prior to generating patterns.
  - STATIC - performed after patterns are generated using the run pattern_compression command.

6 Pattern Compression - DYNAMIC





6 Pattern Compression - STATIC

fault

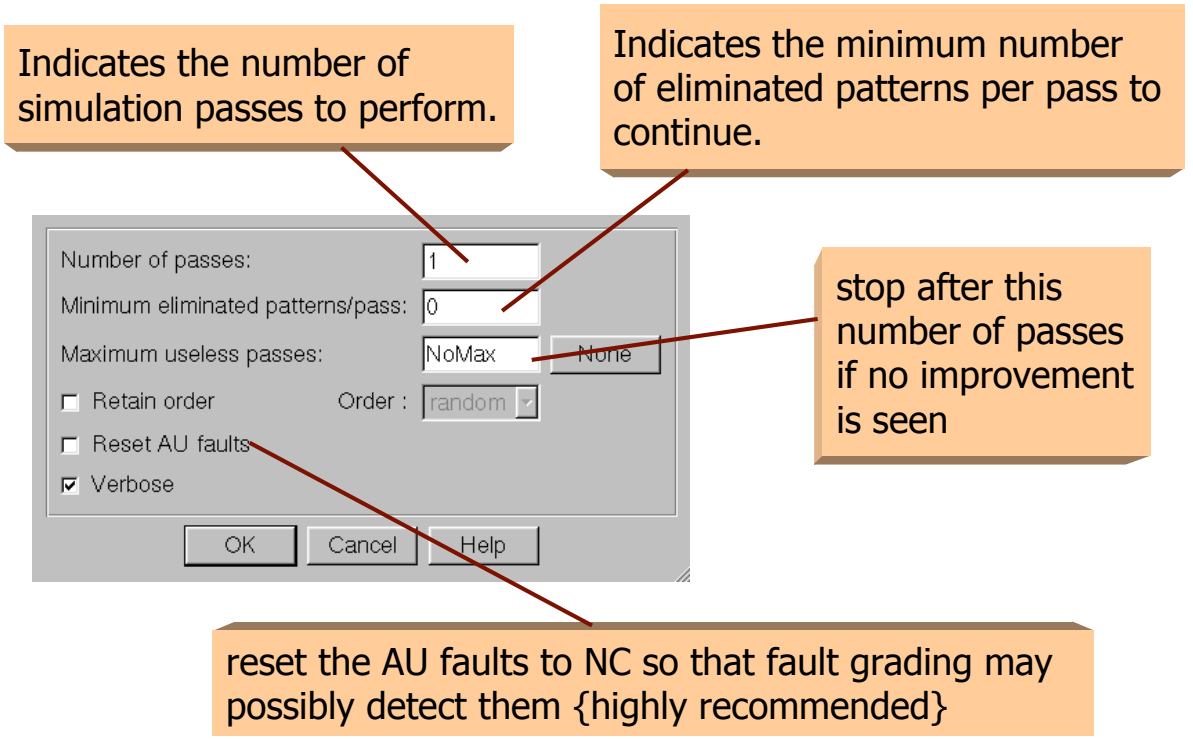
pattern

	1	2	3	4	5	6	7	8	9	10
1	x			x		x				
2		x		x	x					
3			x			x	x			
4				x				x		
5							x	x	x	
6	x	x		x	x					x

Pass 1. Reverse Order Compression:
6, 5, 4, 3 Remove Patterns 1 and 2

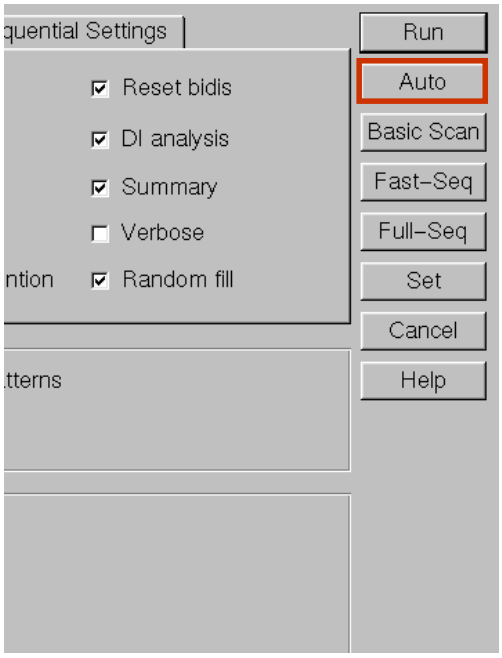
Pass 2. Random Order Compression:
5, 3, 6 Remove Pattern 4

6 Pattern Compression - STATIC@@

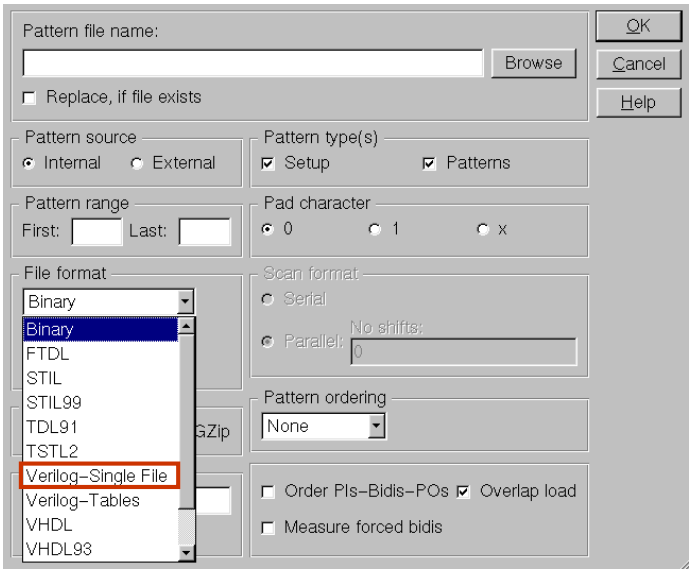


"AUTO" mode to RUN ATPG

- Add faults if none are present.
- Perform "quickie" run to identify AU faults. No patterns are stored, decision = random.
- Reset state.
- Run ATPG with high merge effort, store patterns.
- Run one pass of static compression.
- Report fault, test coverage, and pattern results.
- Report CPU time.



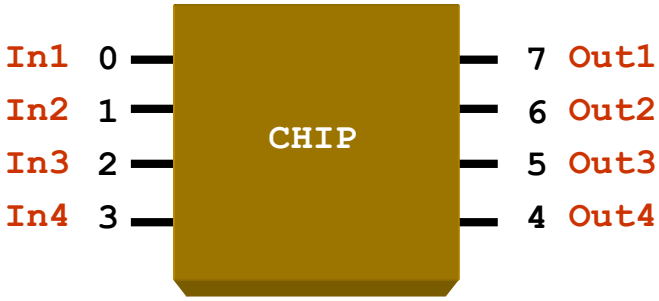
6 Saving ATPG Patterns

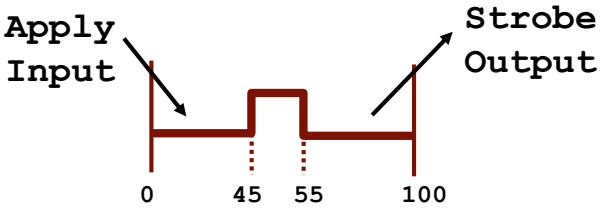


- Saving patterns presents many choices:
  - STIL, Verilog, WGL, VHDL, or proprietary Binary outputs
  - Compression choices: none, GZIP, or proprietary
  - Serial vs. Parallel Form (Verilog)
  - Compaction or not (STIL)
  - all or a selected range of patterns
  - ATPG patterns, Chain Test Patterns, Setup, or combo's

Pattern For CIC IMS Flow@

- Saving patterns with Verilog single-file, serial.
- Determine the order of final chip I/O.
- Edit that patterns file with text edit.





Pattern For CIC IMS Flow@@

- Edit Pattern.v

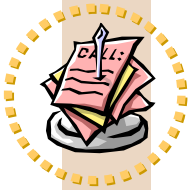
```
`timescale 1ns/10ps
......

//Dump the vector for tester
always @(posedge CLK)
 begin
  #0 $fwrite(file1,"%b%b%b%b", top.In1, top.In2, top.In3, top.In4);
 end
always @(negedge CLK)
 begin
  #20 $fwrite(file1," %b%b%b%b\n", top.Out1, top.Out2, top.Out3, top.Out4);
 end
```

```
1111 1010
1010 1Z00
XX10 ZZ10
....
```

File 1

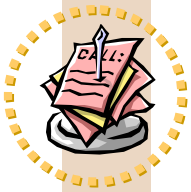
Use this file for IMS setup file → IC test course

TetraMAX – Lab 5



Continue from DFT Compiler

Design-for-Testability ATPG with TetraMAX



STIL Constructs for DRC and ATPG

How does TetraMAX perform DRC?

- TetraMAX learns information about your design…
  - read netlist, run build_model
  - It can perform the B and N rule checks
- You answer these questions for TetraMAX…
  - What are your Scan Chain Pins?
  - How do you enable scan for your design?
  - What is the test setup sequence for your design?
  - What are your clocks?
  - Which clocks are used for shift and capture?
  - What is your the timing for all your device pins?
  - And more questions
- You provide these answers via a **STIL Procedures File** or by using TetraMAX Quick STIL commands

STIL Procedure File

- **S**tandard **T**est **I**nterface **L**anguage IEEE 1450
- The STIL Procedure File (SPF) is used to provide the following information:
  - scan chain definition
  - which pins act as “clocks”
  - timing definitions for pins, clocks, and measures
  - constrained ports and equivalence relationships
  - ‘test_setup’ macro (test setup sequence)
  - ‘load_unload’ procedure (scan enable sequence)
  - ‘shift’ procedure (scan shift sequence)
- The STIL file may also contain pattern data.

Creating the STIL Procedure File

- At any time, a template SPF file can be created with the command.

```
write_drc SPF_filename [-replace]
```

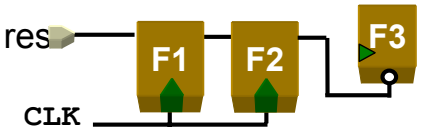
- SPF related information can be increased in TetraMAX by:

```
add clock <off_state> <port_name>
add pi constraint <held_value> <port_name>
add pi equiv <port_name> [-invert] <port_name>
```

Example: Simple STIL Procedure File

```
STIL;
ScanStructures {…}
Procedures {
  load_unload {
    V {…}
    Shift {…}
  }
}

MacroDefs {
  test_setup {…}
}
```

- STIL header
- ScanStructures block
- Procedures block
  - load_unload procedure
  - Shift statement
- test_setup macro is optional, but may be needed to initialize a particular design for test mode.



Defining Scan Chains

- Scan chains are defined in the ScanStructures block.

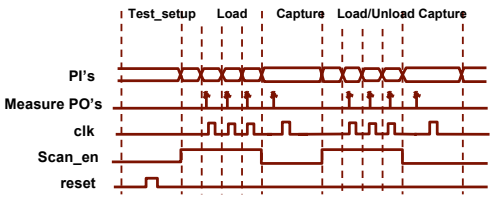
```
ScanStructures {
   ScanChain "c1"      { ScanIn SDI2; ScanOut SDO2; }
   ScanChain "name2"   { ScanIn SDI1; ScanOut D1; }
   ScanChain "JTAG"    { ScanIn TDI; ScanOut TDO; }
   ScanChain "chain4" { ScanIn "IRQ[4]"; ScanOut XYZ;}
}
```

Attention: uppercase/lowercase

Enabling Scan Mode

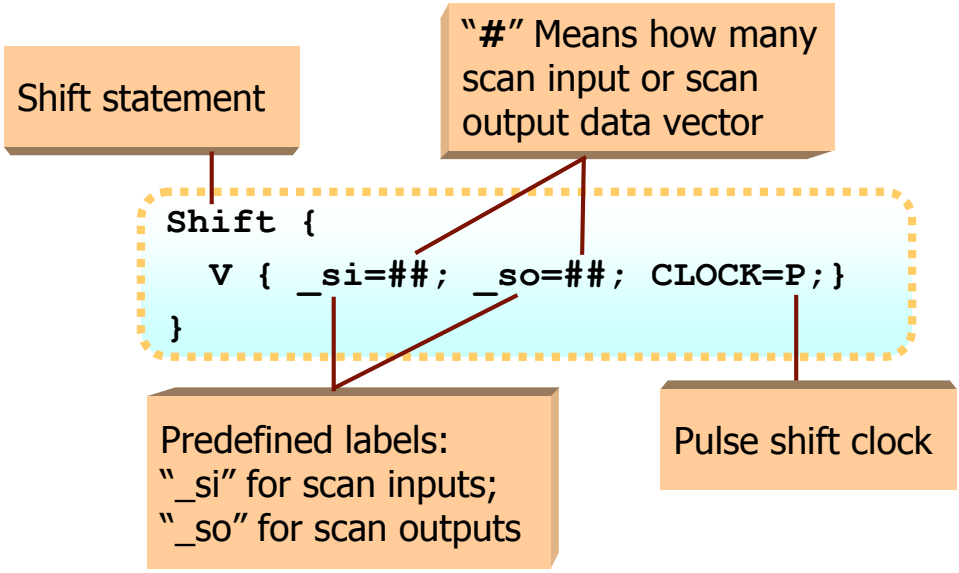
```
Procedures {
  "load_unload" {
    V {
      // clocks & resets off
      CLOCK = 0; RESETB = 1;
      // enable scan
      SCAN_EN = 1;
    }
    Shift {
      V { _si=##; _so=##;
          CLOCK=P; }
    } // end shift
  } // end load_unload
} // end procedures
```

There is usually one top level port such as "SCAN_EN" which will enable the shifting of scan chains. The port should be set appropriately inside the load_unload procedure.



Checking the Shift Statement

- The Shift statement defines how to shift the scan chains by one bit position.



Repeating Strings

- When you have long strings you may use the repeat syntax:

```
\r<count> string_to_repeat<space>
```

```
Shift {
     V { _si=########;
         _so=########; CLOCK=P;}
}
```

```
Shift {
     V { _si=\r8 # ; _so=\r8 # ; CLOCK=P;}
}
```


load_unload Procedure

```
Procedures {
  "load_unload" {
    V {CLK=0; RSTB=1; SE=1;}
    V {OE=0; bidi=ZZZZ;}
    V {bidi=1111;}
    Shift {
      V { "_si"= # ; "_so"= # ; CLK=P; }
    } // end shift
    V {bidi=ZZZZ;}
  } //end load_unload
} // end procedures
```

Default Clock Capture Procedures

```
capture_CLK {
  "forcePI":    V { "_pi"=#; }
  "measurePO": V { "_po"=#; }
  "pulse":      V { "CLK"=P; }
}
capture_RESETB {
  "forcePI":    V { "_pi"=#; }
  "measurePO": V { "_po"=#; }
  "pulse":      V { "RESETB"=P; }
}
capture {
  "forcePI":    V { "_pi"=#; }
  "measurePO": V { "_po"=#; }
}
```


Defining PI Constraints

Force

```
capture_CLK {
  F { TEST_MODE = 1; }
  V { _pi=## ; _po=###;  CLK=P; }
}
capture_RSTB {
  F { TEST_MODE = 1; }
  V { _pi=## ; _po=###;  RSTB=P; }
}
capture {
  F { TEST_MODE = 1; }
  V { _pi=## ; _po=###; }
}
```

Defining PI Equivalences

```
"load_unload" {
   E "CLKP" \m "CLKN";
   V { CLKP = 0; RSTB = 1; SE=1; }
   Shift {
     V { _si=#; _so=#; CLKP=P; CLKN=P; }
   }
}
"capture_CLKP" {
   E "CLKP" \m "CLKN";
   V { "_pi"=\r12 # ; "_po"=\r8 #; CLKP=P; CLKN=P; }
}
```

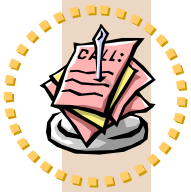
Means CLKP inv CLKN

Controlling Test Mode

- Into ATPG test mode usually involves holding a specific top level port to a constant value. →Defining a PI constraint on the port.
  - Initializing the port to it's constrained value in the 'load_unload' and other procedures.
  - Adding the 'test_setup' macro procedure to the end of the SPF file.

```
MacroDefs {
  "test_setup" {
   V {TEST_MODE = 1; PLL_TEST_MODE = 0; PLL_RESETB = 1;}
   V {PLL_RESETB = 0; }
   V {PLL_RESETB = 1; }
  }
}
```

Design-for-Testability
ATPG with TetraMAX



TetraMAX Fault Classes & Fault list

5 Fault Categories

- During ATPG pattern generation and functional fault simulation TetraMAX classifies faults into one of 11 fault classes grouped into 5 major categories:
  - DT = Detected
  - PT = Possibly Detected
  - UD = Undetected
  - AU = ATPG Untestable
  - ND = Not Detected
- TetraMAX defaults are to report summaries only the 5 major categories.

11 Fault Classes

- DT: detected
  - DS: detected by simulation
  - DI: detected by implication
- PT: possibly detected
  - AP: ATPG untestable, possibly detected
  - NP: not analyzed, possibly detected
- UD: undetectable
  - UU: undetectable unused
  - UT: undetectable tied
  - UB: undetectable blocked
  - UR: undetectable redundant
- AU: ATPG untestable
  - AN: ATPG untestable, not detected
- ND: not detected
  - NC: not controlled
  - NO: not observed

DT - Detected

- DS - Detected by Simulation:
  - Hard detected by a pattern which simulates to different, 1 or 0 observed value for the "bad" machine as compared to the "good" machine.
- DI - Detected by Implication:
  - Certain faults that are detected from an implication analysis:
    - pins in the scan chain data path.
    - shift clock distributions to scan cells.
    - set/reset distributions to scan cells.

PT - Possibly Detected

- AP - Atpg Untestable Possibly Detected:
  - An analysis has proven that this fault can not be detected under current ATPG methods {but this does not rule out non-ATPG methods} -or- An analysis was inconclusive or aborted.
  - The bad machine simulation produces an 'X' while the good machine simulation produces 1 or 0.
  - At least one pattern was generated and retained .
- NP - Not analyzed - Possible detected:
  - A fault in this class indicates that it was never hard detected and there was at least one occurrence of a measure where the fault effect was at an indeterminate state.
  - at least one pattern that caused the fault to be placed in this class will be retained.

UD - Undetectable Faults

- UU - undetectable unused :
  - Faults on unused outputs or unused circuitry.
- UT - undetectable tied :
  - a stuck-at-0 fault on a grounded pin.
- UB - undetectable blocked :
  - a fault that is blocked from observation due to tied logic.
- UR - undetectable redundant :
  - a fault that has no observable effect on output (redundant logic).

AU - ATPG Untestable

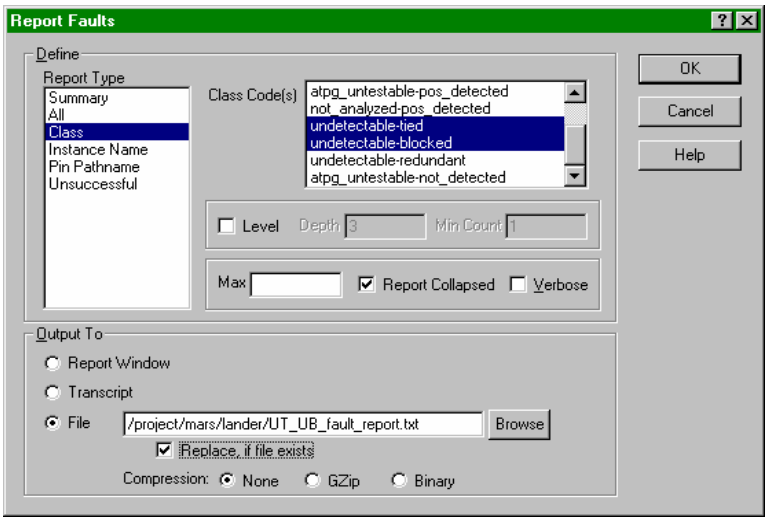
- AN - ATPG Untestable, not detected:
  - Faults which can not be hard detected under the current ATPG conditions nor can they be proved to be redundant (UR). This can occur for a number of reasons:
  - A constraint gets in the way
    - sequential patterns would be required to detect.
    - the fault can only be possibly detected.
    - detection would require sensing an internal Z state condition.

ND - Not Detected

- NC - Not Controlled:
  - A way in which to control the fault location to the value opposite the stuck-at value has not yet been found. The NC class is the default starting class for ATPG.
- NO - Not Observed:
  - The fault location could be controlled but a way to propagate the fault effect to an observe point, either a scan cell or output, has not yet been found.

Saving Fault Lists

- Saving fault lists to a file has many choices. You can save all faults or just specific classes; select collapsed or uncollapsed lists; save with or without compression, etc.



Collapsed V.S. Uncollapsed Faults

- TetraMAX processes a collapsed fault list but keeps track of both the collapsed and uncollapsed faults.
- Test Coverage can be reported using either collapsed or uncollapsed numbers (the default is uncollapsed):

```
set faults -report [-collapsed | -uncollapsed]
```

- Fault lists can be saved to files or reported in either the collapsed or uncollapsed form.

```
write faults filename -all [-collapsed | -uncollapsed]
```

Adding Faults

- S-A-1 and S-A-0 fault sites can be added using:
  - all locations (-all)
  - specific pin pathnames
  - specific instance names
  - module names

```
add fault <pin_path |inst_path |-module name |-all>
```

- Faults can also be defined by reading file.
  (Fault Interface File Format)

```
read faults filename
```


Defining Nofaults

- Fault locations can be inhibited by defining a “nofault” prior to adding faults
- Nofaults can be defined by:
  - pin pathname
  - primitive instance pathname
  - hierarchical instance pathname
  - module name

```
add nofault /sub_block_A/adder
add faults -all
```

- Nofaults can also be defined by reading file.

```
read nofaults filename
```

Fault Interface File Format

equiv to UR fault

```
Sa0 DI   /CLK
sa1 DI   /CLK
sa1 DI   /RSTB
sa0 DS   /RSTB
sa1 AN   /i22/mux2/A
sa1 UT   /i22/reg2/lat1/SB
sa0 UR   /i22/mux0/MUX2_UDP_1/A
sa0 --   /i22/mux0/A
sa0 DS   /i22/reg1/MX1/D
sa0 --   /i22/mux1/X
sa0 --   /i22/mux1/MUX2_UDP_1/Q
sa1 DI   /i22/reg2/r/CK
sa0 AP   /i22/out0/EN
sa1 AP   /i22/out0/EN
```


Random Fault Sampling

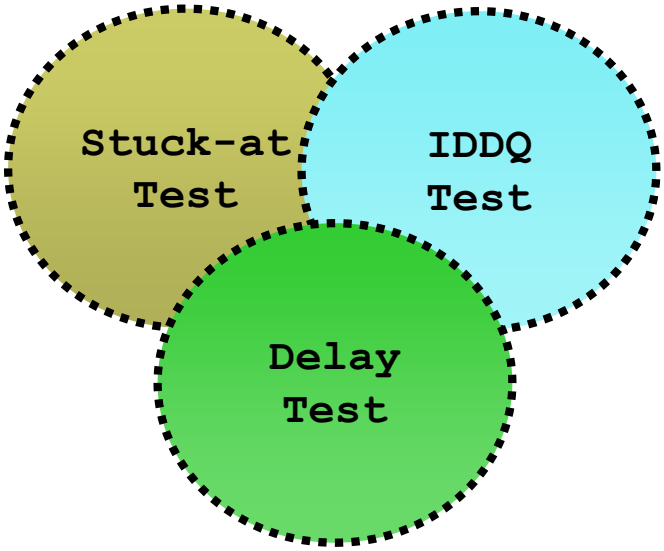
- Using a sample of faults rather than all possible faults can reduce the total runtime for large designs. You can create a random sample of faults using:

```
TEST> add faults /my_asic/block_B
TEST> remove faults -retain_sample 50
```

- EX: Retaining a 25% sample of faults in block_A and block_B and a 50% sample of faults in block_C.

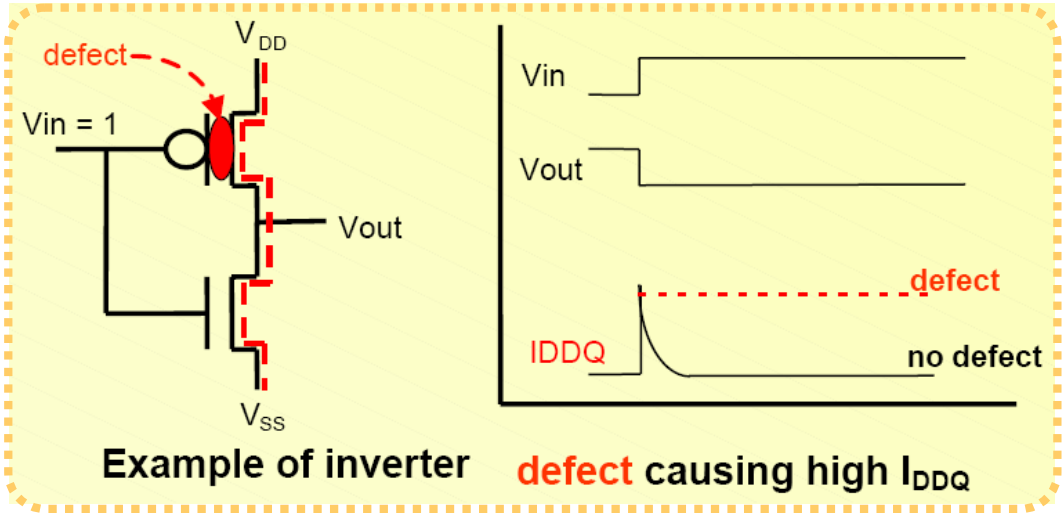
```
TEST> add faults /my_asic/block_A
TEST> add faults /my_asic/block_B
TEST> remove faults -retain_sample 50
TEST> add faults /my_asic/block_C
TEST> remove faults -retain_sample 50
```

100% coverage detect all defective parts?



High Stuck-At is Not Good Enough!

IDDQ Fault Model



Example of inverter defect causing high I_{DDQ}

- Applicable to static CMOS designs
- IDDQ current is observable at the tester power supply
- Only a few IDDQ test strobes yield high fault coverage
- Detects defects beyond stuck-at fault scan test

IDDQ Test Coverage versus IDDQ Strobes

- Iddq testing is slow compared to stuck-at fault testing:
  → More tester time = more mfg $$.
- Typical applications for Iddq tests only allow for **5 to 10 strobes**.
  →No significant increase in test coverage beyond 10 strobes (diminishing returns)

TetraMAX IDDQ ATPG

- Fault Model Selection

```
TEST> remove faults -all
TEST> set faults -model iddq
TEST> set contention nofloat | float
TEST> add faults -all
```

- Limit Number of IDDQ Patterns (Strobes)

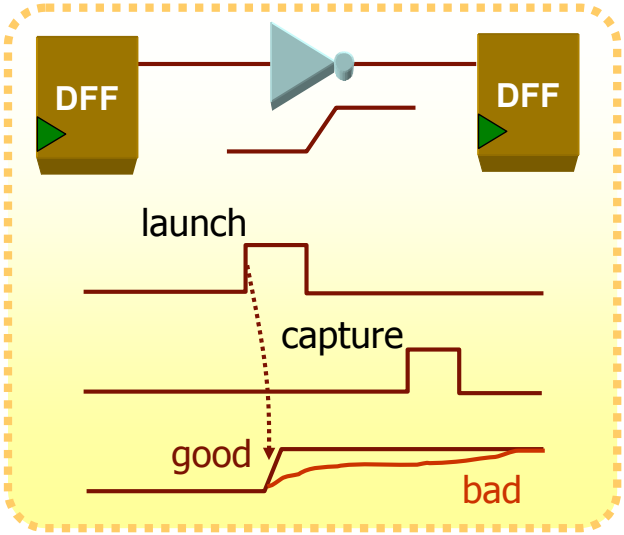
```
TEST> set atpg -patterns 20 -merge high
TEST> run atpg
```

Why Do Delay Testing?

- 20% of faults do not fail slow speed testing.
- Why Not Run Functional Vectors At Speed?
  - Manual Effort to Create and Select
  - Unpredictable Coverage (slow, iterative process to improve)
  - Costly high-speed, high bandwidth ATE required
  - Exercises many gates (but not necessarily detects timing faults)
  - Has realistic power consumption which may effect timing

Transition Fault Model

- Faults are associated with a library cell
- Each node has a slow-to-rise/slow-to-fall site
- Targets point defects

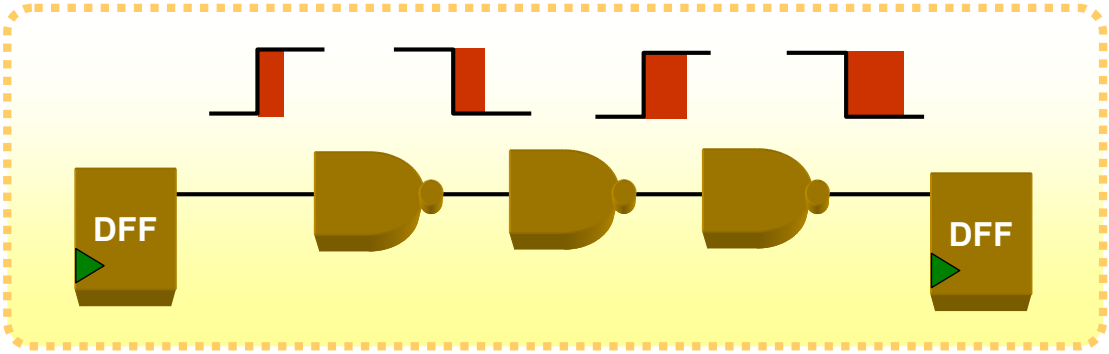


- Choose transition fault model for ATPG by:

```
TEST>set fault -model transition
TEST> add faults -all
```

Path Delay Testing

- Distributed Delay Defect with Path Delay Fault Model

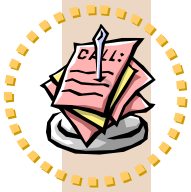


- Faults are associated with a path
- Targeted paths typically include "critical" paths (Static Timing Analysis using PrimeTime)
- Targets process-oriented defects

Tester Clock Frequency VS Fault Model

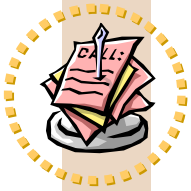
- 100 MHz – 1 GHz : Path Delay
- 1-100 MHz : Transition Delay
- 1 MHz : Stuck-at

TetraMAX – Lab 6



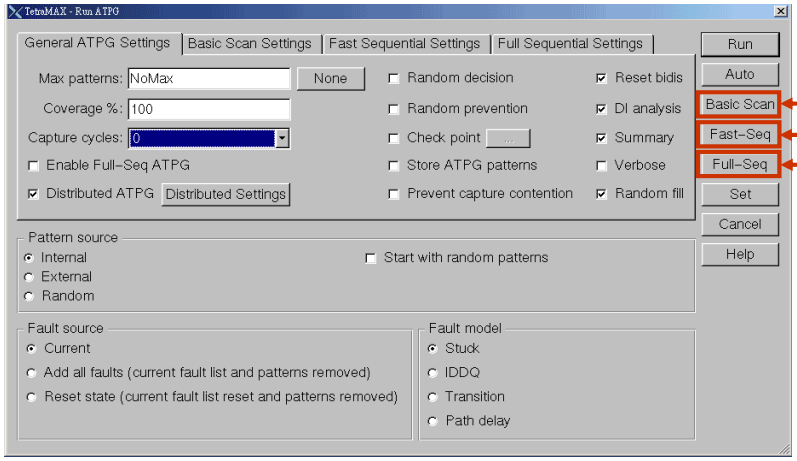
Getting start to the TetraMAX GUI

Design-for-Testability ATPG with TetraMAX



Controlling ATPG

ATPG Engines and Controls

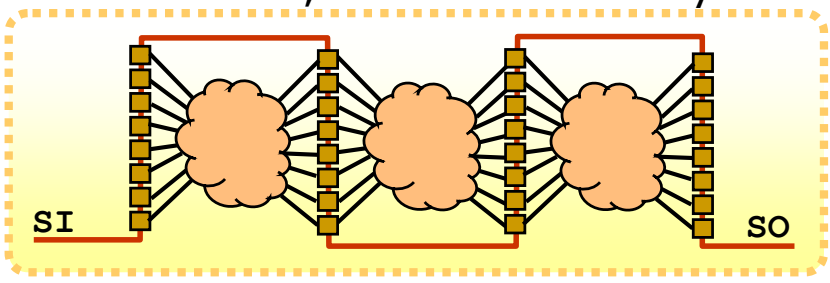


Basic-Scan: fast, good coverage for full-scan

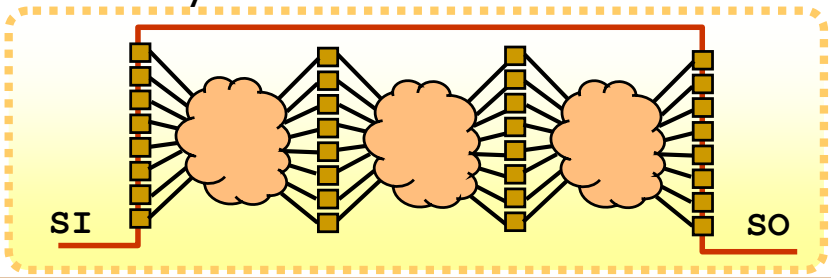
- **Fast-Sequential**: higher coverage for near-full scan designs; good for **shadow logic around memories**, limited non-scan
- **Full-Sequential**: most powerful engine supporting more complex designs

Basic Scan VS Fast Sequential

- Basic Scan: full scan, clocks controlled by PIs



- Fast SequentialiXlimited amount of non-scan (non-scan flip-flops, latches, bus keepers, RAMs), clocks still must be controlled by PIs



Enabling Fast-Sequential ATPG

- If you have AU faults remaining after Basic Scan ATPG, perhaps the Fast-Sequential engine can help
- To enable fast-sequential ATPG, specify the number of capture_cycles:

```
TEST> set atpg -capture_cycle <d>
TEST> run atpg
```

Where d is an integer between 2 and 10

How to Set capture_cycles

- In TEST mode, obtain the worse case setting from the report summaries → sequential_depths command.
- Only a few faults may have this worst case sequential depth; many faults might be detected with a lower setting:
- Start a small number such as

```
set atpg capture_cycles 2
```

- If you have embedded memories start with

```
set atpg capture_cycles 4
```

When Full Sequential Might Help You

- Clocks (ordinary clocks, resets/sets, memory writes) to non-scan elements not controlled from **Primary Inputs**.
- Sequential Detect **Depth** greater than **10**.
- Detection requires a **particular clocking** sequence (CLK1 before CLK2 in a two-phase approach, for example).

- Always run the Basic Scan engine first:
  - If you are going to use the iVauto switch, use it only with the Basic Scan engine enabled

Avoiding Patterns that Detect Too Few Faults

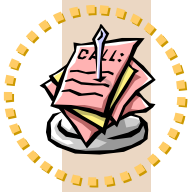
```
test coverage 99.25%
-------------------------------------------------
Pattern Summary Report
-------------------------------------------------
#internal patterns 453
#basic_scan patterns 205
#fast_sequential patterns 248
-------------------------------------------------
```

Each pattern can detect at least 10 faults

```
reset state ; set atpg -min_det 10 ; run atpg
```

```
test coverage 98.21%
-------------------------------------------------
Pattern Summary Report
-------------------------------------------------
#internal patterns 163
#basic_scan patterns 101
#fast_sequential patterns 62
```

Design-for-Testability
ATPG with TetraMAX



Fault Simulating Functional Patterns

Fault Grading of Functional Patterns

- TetraMAX has integrated, zero-delay fault simulator.
- Can write functional patterns to supplement ATPG.
- Recommended input format is STIL;
- IEEE P1364 E-VCD format is also supported.
- Accepts BSD Compiler functional wgl vectors.
- TetraMAX has the capability to fault grade functional patterns.
- Functional patterns are read into the ATPG tool using the command:

```
set_patterns external filename
```

Fault Grade then ATPG Script@

- Read in design netlist and library.
- Build design model.
- Remove PI constraints.
- Remove PI equivalent.

```
BUILD> read netlist *.v -d
BUILD> run build topmodule
DRC> remove PI constrain -all
DRC> remove PI equiv -all
```


Fault Grade then ATPG Script@@

- Set DRC nofile.
- Run DRC.
- If DRC violation has occurred and keeps us in DRC mode, change the severity of DRC rule down to warning.(*optional)

```
DRC> set drc -nofile
*DRC> set rule xxx warn
DRC> run drc
```

Fault Grade then ATPG Script@@@

- Set external pattern source.
- Perform and observe fault free simulation.
- Add fault list.
- Perform fault simulation.

```
TEST> set pattern external file.v
TEST> run sim -seq
TEST> add faults -all
TEST> run fault_sim -seq
```


Fault Grade then ATPG Script@@@@

- Get a summary report.
- Save the fault list that exists at the end of fault grade of the functional patterns.
- Return to DRC mode.
- Set internal pattern source.

```
TEST> rep sum
TEST> write faults fault.dat -all -uncollapsed
TEST> drc -force
DRC> set pattern internal
```

Fault Grade then ATPG Script@

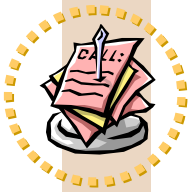
- Run DRC file.
- Read in the fault list that existed at the end of the functional pattern fault grade.
- Run ATPG.

```
DRC> run drc file.spf
TEST> rem faults -all
TEST> read faults fault.dat -retain
TEST> run atpg -auto
```


ATPG then Fault Grade Script

```
BUILD> run build top_mod_name
DRC> run drc atpg_related.spf
TEST> add faults -all
TEST> run atpg -auto
TEST> report summaries
TEST> write patterns pat.v -form verilog -replace
TEST> write fault pass1.flt -all -uncollapsed -rep
TEST> drc -force
DRC> remove pi constrain -all
DRC> remove clocks -all
DRC> set drc -nofile
DRC> test
TEST> set pattern external b010.vin
TEST> read faults pass1.flt -retain
TEST> run sim -sequential
TEST> run fault_sim -sequential
```

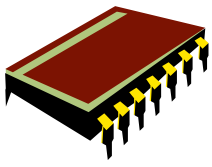
Design-for-Testability
ATPG with TetraMAX



Modeling Memories*

RAM/ROM Modeling*

- RAMs and ROMs are modeled using a simple Verilog behavioral description.
- Memories may have:
  - multiple read or write ports
  - common or separate address bus
  - common or separate data bus
  - edge or level sensitive read/write controls
  - one qualifier on the write control
  - read off state can hold or return data to 0/1/X/Z
  - asynchronous set and/or reset capability
  - memory initialization files
- A ROM is created by defining a RAM without a write port and is required to have an initialization file.



Basic Template*

```
module MY_ATPG_RAM ( read, write, data_in,
        data_out, read_addr write_addr );
  input  read, write;
  input [7:0] data_in;       // 8 bit data width
  input [3:0] read_addr;     // 16 words
  input [3:0] write_addr;    // 16 words
  output [7:0] data_out;     // 8 bit data width

  reg [7:0] data_out;        // output holding reg
  reg [7:0] memory [0:15] ;  // memory storage

  event WRITE_OP;    // event for write-thru

 ...memory port definitions...

endmodule
```

- The basic template for a RAM using limited syntax behavioral Verilog.
- The port list may vary as we define more complicated RAMS or ROMS with multiple ports, but the template is essentially the same.
- Note that the ATPG modeling of RAMS requires that bussed ports be used.

Edge Sensitive Write Ports*

```
always @(posedge write) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end

always @(posedge write) if (CS) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end

always @(negedge write) if (!CSB) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end

and U1 (wen, CS, en2, !en3b);
always @(posedge write) if (wen) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end
```

- Example #1: An edge sensitive port using rising edge of 'write'
- Example #2: An edge sensitive write port with a qualifier pin of CS.
- Example #3: A negative edge sensitive with an active low qualifier pin.
- Example #4: An edge sensitive write port with complex qualifier.

Level Sensitive Write Ports*

```
always @(write or write_addr or data_in)
 if (write) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end


and U1 (WEN, write, CS, en2, !en3b);
always @(WEN or write_addr or data_in)
if (WEN) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end
```

- Example #1: A level sensitive write port enabled by write=1. Changes on address or data with write=1 update the memory contents.
- Example #2: A level sensitive write port with a complex enable.

Edge Sensitive Read Ports*

```
always @(posedge read)
  data_out = memory[read_addr];


always @(negedge read) begin
  data_out = memory[read_addr];
end


and U1 (RCLK, read, CS, !en2b);
always @(posedge RCLK)
   data_out = memory[read_addr];
```

- Example #1: An edge sensitive read port controlled by the rising edge of 'read.
- Example #2: An edge sensitive read port controlled by the falling edge of 'read'.
- Example #3: An edge sensitive read port with a complex control.

Level Sensitive Read Ports*

```
always @(read or read_addr or WRITE_OP)
  if (read) data_out = memory[read_addr];

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'bzzzzzzzz;

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'b0;

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'b1111_1111;
```

- Example #1: A level sensitive read port enabled by read=1. Changes on address or data or a write operation update the memory read contents.
- Examples #2-#4: A level sensitive read port with read port off behavior of Z, zero, and one, respectively.

RAM Modeling: Output Enables*

```
output [7:0] data_out;
reg [7:0] data_out, data_reg;   /* add data_reg */

always @(read or read_addr or WRITE_OP)
  if (read) data_reg = memory[read_addr];

always @(OEB or data_reg)
  if (!OEB) data_out = data_reg;
  else data_out = 8'bZZZZZZZZ;
```

- Example #1: A level sensitive read port enabled by read=1 with a separate tri-state output enabled by OEB=0..

```
output [7:0] data_out;
reg [7:0] data_out, data_reg;   /* add data_reg */
and u1 (RCLK, read, CS);

always @(posedge RCLK)
  data_reg = memory[read_addr];

always @(OEB or data_reg)
  if (!OEB) data_out = data_reg;
  else data_out = 8'bZZZZZZZZ;
```

- Examples #2: An edge sensitive read port with separate tri-state output enabled by OEB=0.

RAM Completed Example*

```
module ATPG_RAM (CS, OE, read, write, data_in, data_out, addr);
  input  CS, OE, read, write; input read, write;
  input [7:0] data_in;           // 8 bit data width
  input [3:0] addr;   // 16 words
  output [7:0] data_out;         // module outputs
  output [7:0] data_reg;         // RAM outputs
  reg [7:0] data_out; // output holding register
  reg [7:0] memory [0:15] ;      // memory storage
  event WRITE_OP;
.
  and u1 (REN, !read, CS);       // form read enable
  and u2 (TSO, OE, CS);          // form tri-state out control
.
  always @(posedge write) if (CS) begin
    memory[addr] = data_in;
    #0; ->WRITE_OP;
  end
.
  always @(REN or addr or WRITE_OP)
    if (REN) data_reg = memory[addr]
.
  always @(TSO or data_reg)
    if (TSO) data_out = data_reg;
    else data_out = 8'bzzzzzzzz;
endmodule
```

A completed example with an edge controlled write port, a level sensitive read port, and a separate output enable.

ROM Modeling*

```
module MY_ROM ( oe, addr, data_out );
  input oe;              // output control
  input [3:0] addr;      // 16 words
  output [7:0] data_out;     // 8 bits per word
  reg [7:0] data_out;        // output holding reg
  reg [7:0] memory [0:15] ;  // memory storage

  always @(oe or addr)
    if (!oe) data_out = memory[addr];
    else data_out = 8'bZZZZZZZZ;

  initial $readmemh("rom_image.dat", memory);
end module
```

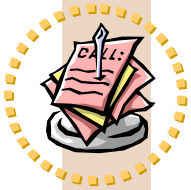
- A ROM is modeled identically to a RAM with these exceptions
  - it has no write ports
  - it requires an initialization file
- Here's a simple ROM with a tri-state output enable.

RAM/ROM data files*

```
0001
0002
0004
0008
0010 0020 0040 0080
0100 0200 0400 0800
1000 2000 4000 8000
// address = 0010
A001 c401 e404 700a
3816 1c2c 2e58 07b0
23e0 07c0 25e0 0b70
363c 2c1c 7c0e b006
8001 4002 2004 1008 8810 4421 2242 1184
0908 0650 0620 0950 1088
// skip loading next 3 addresses
@30
// address = 0030
fffe fffd fffb fff7 ffef ffdf ffbf ff7f
feff fdff fbff f7ff // end of line comment
// underscores for readability
ef_ff df_ff bf_ff 7f_ff
```

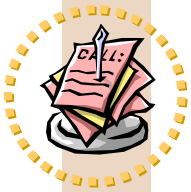
- ROM models require a memory initialization file and RAM models may also use them.
- The format is standard Verilog and supports some flexibilities in how data can be presented.

TetraMAX – Lab 7



RAM Modeling

Design-for-Testability
ATPG with TetraMAX



Debugging Problems

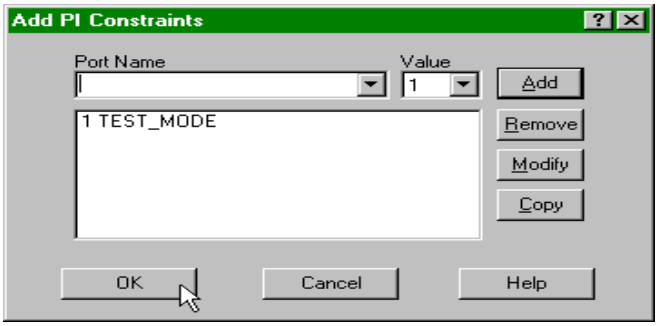
Defining Input Masks

- Input masks are accomplished by defining a PI constraint of X on an input.

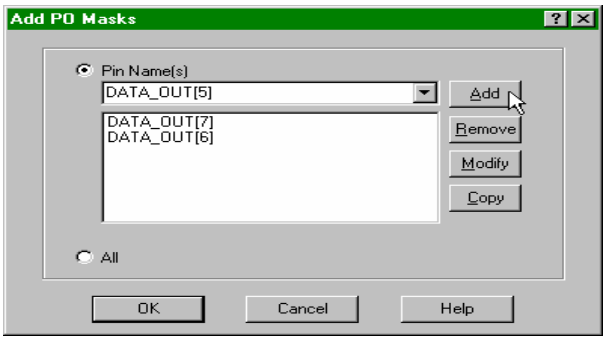
```
SETUP> add pi constraints X port_name
```

- To review existing PI masks or delete use:

```
SETUP> report pi constraints
SETUP> remove pi constraints [ port | -all ]
```



Defining Output Masks



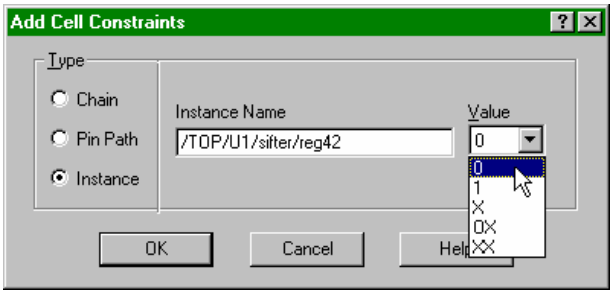
- When you do not want to allow the ATPG algorithm to observe faults through a specific output port.

```
SETUP> add po masks port_name
```

- To review existing PO masks or delete use:

```
SETUP> report po masks
SETUP> remove po masks [ port | -all ]
```


Scan Cell Masks and Constraints



- TetraMAX supports defining constraints on what to load into a specific scan cell.
- By defining an ADD CELL CONSTRAINT command with a value of 0, 1, or X.

  OX - the scan cell will always "**O**bserve **X**" regardless of the captured value.

  XX - the scan cell will always be "Loaded with **X** and Observed as **X**".

Internal Design Restrictions During ATPG

- Q: What do you do when you need to restrict internal points in the design to specific values?
- A: Define an ATPG Constraint.

- Q: What if the restriction is more complicated than just holding a single point to a constant value?
- A: Define a function using ATPG primitives and then reference that function name when defining the ATPG Constraint.

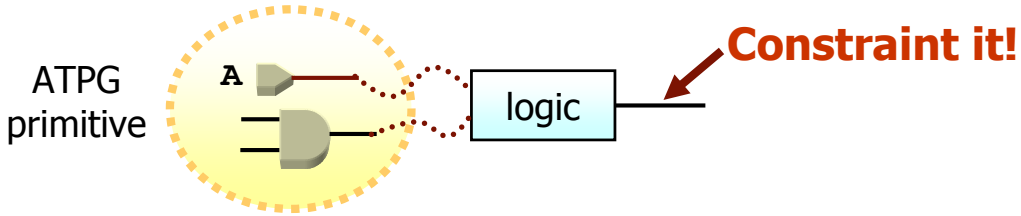
ATPG Constraints

- The constraint can be defined to be:
  - Constrained to 0, 1, or Z.
- The ATPG constraint can be placed on:
  - A pin pathname.
  - An instance output pin (specified by a numeric gate ID).
  - A specific pin on every instantiation of a specific module.
  - The output of a previously defined ATPG Primitive.

```
DRC> add atpg constraints my_cons_3 1 /CORE/ALU/DEMU/U266/H02
DRC> add atpg constraints my_cons_2 0 1911 -drc
DRC> add atpg constraints my_constraint1 1 my_atpg_gate1
```

ATPG Primitive@

- Creates a logic function which can then be used in an ATPG constraint command.

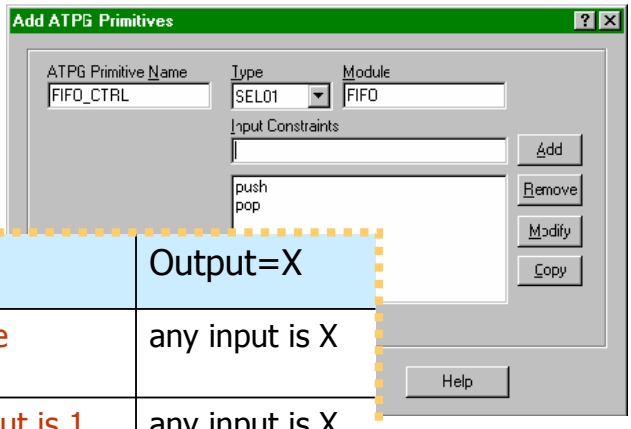


- This logic function is created by:
  - Designating a symbolic label for the ATPG primitive.
  - Specifying inputs from internal points or other ATPG primitives.

ATPG Primitive@@

```
DRC> add atpg primi FIFO_CTRL sel01 -module FIFO push pop
```

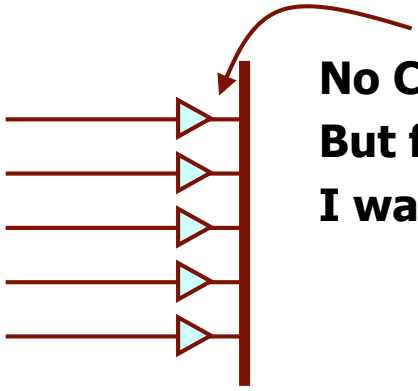
- Specifying one of the following logic functions:



	Output=0	Output=1	Output=X
EQUIV	any inputs are different	all inputs are equivalent	any input is X
SEL1	Other combination	only one input is 1	any input is X
SEL01	Other combination	all inputs are 0 or only one input is 1	any input is X

ATPG Primitive Example #1@

- In internal tri-state buses. ATPG will automatically generate patterns so that no contention exists, but can't avoid an "all floating" condition.



No Contention on the bus.
But floating condition ?
I want to avoid floating !

ATPG Primitive Example #1@@

- Define an ATPG Primitive which monitors the tri-state enable controls. Assuming all controls are active-high ON:

```
DRC> add atpg primitive ONE_ON sel1 \
       /top/moduleA/blkb/inst12/tsd/EN \
       /top/moduleA/blkb/inst33/tsd/EN \
       /top/moduleA/blkb/inst29/tsd/EN
```

- Then define an ATPG Constraint:

```
DRC> add atpg constraint 1 ONE_ON
```

ATPG Primitive Example #2

- In your design, there is a possible clock/data timing race unless the two input pins A, B are kept at complimentary values.
  In normal operating mode this is never a problem so the race condition did not need to been corrected, but with ATPG patterns it may cause a pattern simulation failure or tester failure.

- Define an ATPG Primitive and Constraint:

```
DRC> add atpg primitive FIXUP_GATE \
       equiv    /core/fifo_flush/sequencer/A \
                ~/core/fifo_flush/sequencer/B
DRC> add atpg constraints FIXUP_FUNCTION 1 FIXUP_GATE
```


Test Coverage Hierarchically

```
TEST> rep faults -all -level 4 256
```

Display the block has at least 256 faults.

Display 4 level hierarchy.

```
#faults  testcov  instance name (type)
-------  -------  ----------------------
88148    95.63%   /my_asic  (top_module)
86538    95.88%   /my_asic/MAIN  (MAIN)
  258     2.33%   /my_asic/MAIN/NTREE_1  (NandTree)
43666    99.58%   /my_asic/MAIN/CPU  (CPU_test_1)
25808    99.30%   /my_asic/MAIN/CPU/ARIT  (ALU_test_1)
17842    99.99%   /my_asic/MAIN/CPU/TP  (TOP_test_1)
38796    99.63%   /my_asic/MAIN/MUX  (4BY8MUX_test_1)
 9644    99.74%   /my_asic/MAIN/MUX/BAM_3  (WAMBAM_test_1)
  730   100.00%   /my_asic/MAIN/MUX/WOW_1  (WOW_test_1)
11208   100.00%   /my_asic/MAIN/MUX/AOK_1  (AOK_test_1)
```

Fault Classes Hierarchically

```
TEST> report faults -class AU -level 4 256
```

report faults in specific classes

```
#faults  testcov          instance name (type)
-------  -------          ----------------------
  22342   92.60% /my_asic  (top_module)
   2566   82.85% /my_asic/tvif  (tvif)
   2371   28.00% /my_asic/tvif/fpga2  (fpga2)
    788    5.35% /my_asic/tvif/fpga2/avge1  (avge)
   1583    3.28% /my_asic/tvif/fpga2/avge2  (yavge)
   5235    0.00% /my_asic/ramdac  (ramdac)
   5223    0.00% /my_asic/ramdac/ramdp  (ramdp)
  11093   66.54% /my_asic/video  (video)
  11093   66.24% /my_asic/video/vdp_ckey  (vdp_ckey)
  11029   60.00% /my_asic/video/vdp_ckey/vdp  (vdp)
    404   96.91% /my_asic/ge  (ge)
    264   93.94% /my_asic/ge/src_fifo  (src_fifo)
    811   95.35% /my_asic/vmi  (vmi)
    810   54.29% /my_asic/vmi/vclk_mux_v  (vclk_mux_v
```

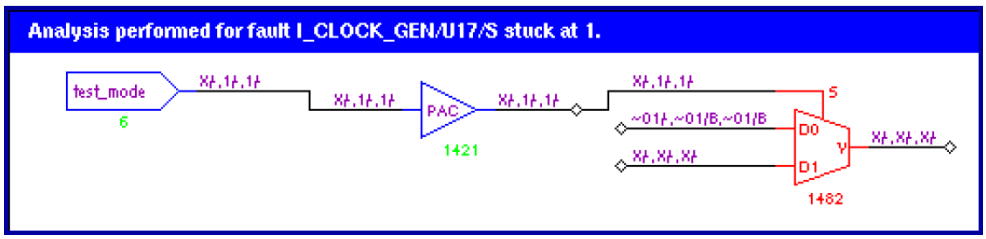

Detailing the Cause of AU Faults

```
TEST> analyze faults -class AU -verbose
```

```
Fault analysis summary: #analyzed=3293, #unexplained=60.
1528 faults are untestable due to constrain values.
sa1 AN I_CLOCK_GEN/I_PLL_SD/clk_4x
. . .
sa1 AN I_CLOCK_GEN/U17/S
1670 faults are untestable due to constrain value blockage.
sa0 AN I_CLOCK_GEN/U16/S0
sa1 AN I_CLOCK_GEN/U16/S0
. . .
sa1 AN I_ORCA_TOP/I_RISC_CORE/I_STACK_TOP/I3_STACK_MEM/U158/S
. . .
130 faults are connected from TIEX.
sa0 AN I_CLOCK_GEN/U16/S0
sa1 AN I_CLOCK_GEN/U16/S1
sa1 AN I_CLOCK_GEN/U16/S0
```

AU Fault Graphically

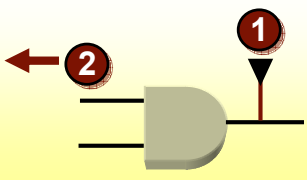
```
TEST> analyze faults display -stuck 1 I_CLOCK_GEN/U17/S
```



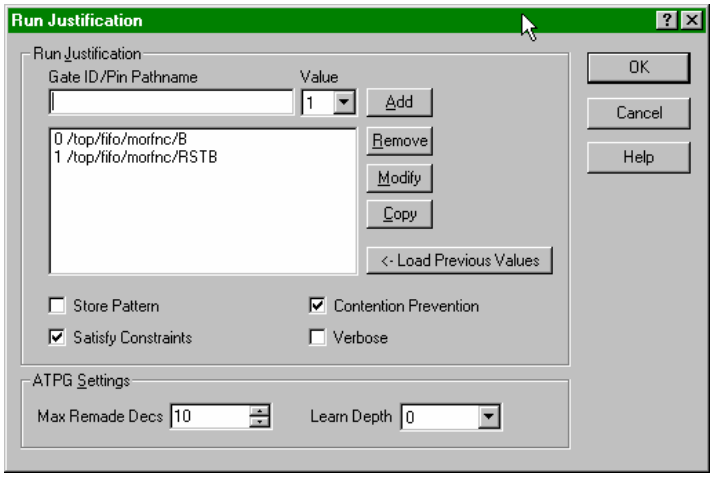

```
-------------------------------------------------------------------------
Fault analysis performed for I_CLOCK_GEN/U17/S stuck at 1 (input 0 of
MUX gate 1482).
Current fault classification = AN (atpg_untestable-not_detected).
-------------------------------------------------------------------------
Connection data: to=MASTER,DSLAVE,CLOCK from=CLOCK,TIEX
Fault site is constrained to value 1.
Source of constrained value is gate test_mode (6).
The gate_report data is now set to "constrain_value".
```


Using Run Justification

- Useful for understanding why fault locations can not be tested or are difficult to test.

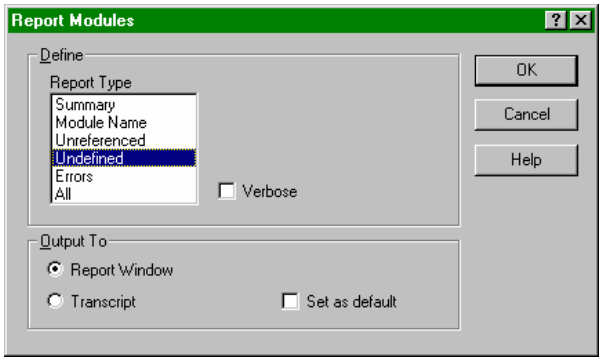


Set (1) is vcc, then trace back to all input pins to see if there exist the patterns to make it.




```
TEST> run justification -set /core/buf/Y 1 /core/alu/CI 0
 Successful justification: pattern values available in pattern 0.
```

Identifying Missing Modules



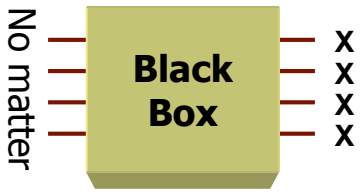
- You will receive a B5 violation if you attempt to build the design and there are missing module definitions.
- To identify all of the missing modules for the currently selected "top" use the command:

```
BUILD> rep modules -undefined
```

```
                                    pins
module name                    tot(  i/  o/ io)   inst   refs(def'd)   used
-------------------------------  ----------------   ----   -----------   ----
MUX21                            0(  0/  0/  0)      0    5340   (N)        0
ND2A                             0(  0/  0/  0)      0    1599   (N)        0
FD1SQ                            0(  0/  0/  0)      0    7849   (N)        0
FD2S_SYNC                        0(  0/  0/  0)      0    2945   (N)        0
AN2P                             0(  0/  0/  0)      0    1122   (N)        0
```


Creating Black Box Models@

- Black Box :
  - the inputs don't matter.
  - the outputs are all tied to X.



- Set Rule B5 to warning and run build_model command.
  → Undefined modules will be replaced with a black box model.

```
BUILD> set rule b5 warning
BUILD> run build
```

- To get a list of the modules which were black boxed.

```
DRC> report violations b5
```

Creating Black Box Models@@

- Synopsys recommends you explicitly define each module which is to be black boxed so that unintentionally missing modules are not black boxed.

```
Set BUIld [ -Black_box <module_name> ]
```

```
EX.
set build -reset_boxes
set build -black_box RAM32x8
set build -black_box PLL_BLOCK
set build -bl RAM32x8 -bl PLL_BLOCK
```

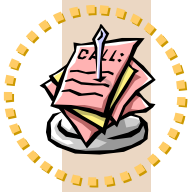

Creating Empty Box Modules

- An "empty box" is similar to a black box with the exception that its output pins are floating.

```
Set build [ -empty_box <module_name> ]
```

```
EX.
set build -reset_boxes   // clears black & empty box list
set build -empty_box BUS_SWITCH
set build -black RAM32x8 -empty IRQ_CORE
```

TetraMAX – Lab 8



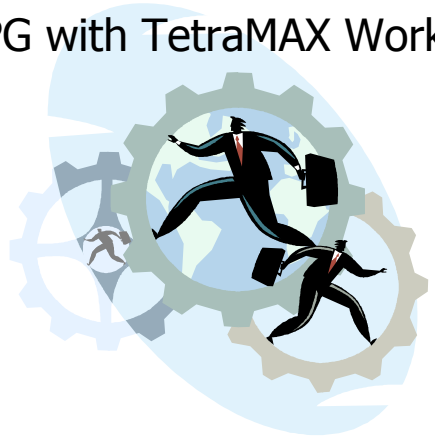
Involving bus contention and Z-states

Day 2 Review

- TetraMAX Overview
- Design and Test Flows
- STIL Constructs for DRC and ATPG
- TetraMAX Fault Classes & Fault list
- Fault Simulating Functional Patterns
- Modeling Memories*
- Debugging Problems
- Something To Remember

References

- DFT Compiler Test Design Rule Checking User Guide (Synopsys).
- DFT Compiler Scan Synthesis User Guide (Synopsys).
- TetraMAX User Guide (Synopsys).
- Training Material of DFT Compiler Workshop (Synopsys)
- Training Material of ATPG with TetraMAX Workshop (Synopsys)



DFT Design Kit
(Synopsys)

Lab materials

Attention!

In these labs:

Text: means the instructions you have to key in.
Text1 → Text2: means you can choose those instructions on GUI.
Icon: means instruction icons.

include file1.dc: means an instruction script. You can type "include file1.dc" instead of key in all the instructions in that step.

Before you start:

Login WS trainaxx (CIC)

username: **trainaxx**

password: **train0xx**

```
% \rm -rf *
% cp -r /cad2/lab/DFT/* .

% source -dft.cshrc
```

Contents

Lab 1

DFT Compiler –

Test Synthesis Flow

Learning Objective:

- Understand the baseline DFTC flow.
- Write DFTC scripts to automate the flow.

1. Go to the directory Lab_1 and invoke DFTC from there.

```
UNIX> cd lab1
UNIX> design_vision &
```

Part 1: Top-down basedline flow

2. Read in the FLOW1.db file.

```
read_file FLOW1.db
```

3. Perform Scan-Ready Synthesis.

```
current_design FLOW1
compile -scan
```

4. Create test clock and constrants. (we have a clock port “CLK”)
   ps. If there is a reset port in your design, remember to constraint it in inactive mode.

```
create_test_clock -period 100 -waveform [list 40 60] [find port "CLK"]
```

5. Preview the scan synthesis, if it is ok, then insert scan.

```
preview_scan -show all
insert_scan
```

6. Check scan rules after scan inserting and report the result.

```
check_test
report_test -scan_path
```

7. Run ATPG immediately after scan synthesis.You do not need to specify any additional scan information since the current_design for ATPG and scan synthesis are the same. What was the test coverage reported?

```
estimate_test_coverage
```

8. See the circuit, what pins have been added?

9. Remove all designs.

```
remove_design -designs
```

Part 2: Top-down scan insertion with scan ports sharing

10. Read in the FLOW1.db file.

```
DFTC> read_file FLOW2.db
```

11. Perform Scan-Ready Synthesis for design CORE.

Type: `include lab1_1.dc` *. It contains following 4 instructions.*

```
current_design CORE
set_scan_configuration -style multiplexed_flip_flop
set_scan_configuration -methodology full_scan
compile –scan
```

12. Perform scan synthesis at the top level, FLOW1.
    We want to share the following functional ports as scan ports:

    scan in: HRS

    scan out: SPEAKER OUT

    scan enable: TEST_SE

Type: `include lab1_2.dc` *. It contains following 6 instructions.*

```
current_design FLOW1
set_scan_configuration -chain_count 1
create_test_clock -period 100 -waveform [list 40 60] [find port "CLK"]
set_scan_signal test_scan_in -port "HRS"
set_scan_signal test_scan_out -port "SPEAKER_OUT"
set_scan_signal test_scan_enable -port "TEST_SE"
```

13. Preview the scan synthesis, if it is ok, then insert scan.

```
preview_scan -show all
insert_scan
```

14. Check scan rules after scan inserting and report the result.
    How many scan chains did you get and how many flip-flops are in each chain?

```
check_test
report_test -scan_path
```

15. Run ATPG immediately after scan synthesis.You do not need to specify any additional scan information since the current_design for ATPG and scan synthesis are the same.

What was the test coverage reported?

```
estimate_test_coverage
```

16. Remove all designs.

```
Remove_design –designs
```

Part 3: Core-level insertion, Top-level ATPG

17. Read in the FLOW2.db file.

```
DFTC> read_file FLOW2.db
```

18. Perform Scan-Ready Synthesis for design CORE.

Type: `include lab1_3.dc` *. It contains following 4 instructions.*

```
current_design CORE
set_scan_configuration -style multiplexed_flip_flop
set_scan_configuration -methodology full_scan
compile –scan
```

19. Perform scan synthesis for design CORE.

Type: `include lab1_4.dc` *. It contains following 9 instructions.*

```
set_scan_configuration -chain_count 1
create_test_clock -period 100 -waveform [list 40 60] [find port "CLK"]
set_scan_signal test_scan_in -port "HRS"
set_scan_signal test_scan_out -port "SPEAKER_OUT"
set_scan_signal test_scan_enable -port "TEST_SE"
preview_scan -show all
insert_scan
check_test
report_test -scan_path
```

The design should have one scan chain and the same scan ports as before. That is, the core-level design has ports with the same names as their corresponding top-level ports.

20. Change your current design to the top-level design FLOW2 and prepare the design for ATPG.

Type: `include lab1_5.dc` *. It contains following 6 instructions.*

```
current_design FLOW2
create_test_clock -period 100 -waveform [list 40 60] [find port "CLK"]
set_scan_configuration -existing_scan true
set_signal_type test_scan_in [find port "HRS"]
set_signal_type test_scan_out [find port "SPEAKER_OUT"]
set_signal_type test_scan_enable [find port "TEST_SE"]
```

21. Run check_test, and if it recognized your single scan chain, proceed with ATPG.

Type: `include lab1_6.dc` *. It contains following 4 instructions.*

```
check_test
report_test -scan_path
estimate_test_coverage
```

Lab 2

DFT Compiler – Bidirectionals

Learning Objective:

- The goal of this lab exercise is to show you how you can improve the testability for the logic around bidirectional I/Os.

Part 1: Top-down flow

1. Go to the directory Lab2 and invoke DFTC from there.

```
UNIX> cd lab2
UNIX> design_vision &
```

2. First, read the TEST DEFAULT VARIABLES setten in .synopsys_dc.setup

```
set test_default_period 100
set test_default_delay 0
set test_default_bidir_delay 0
set test_default_strobe 40
```

3. Assume your ASIC vendor requires a test period of **200 ns** and a **strobe time of 190 ns**. Override the defaults for these components of the Test Configuration.

```
set test_default_period 200
set test_default_strobe 190
```

4. Read in the BIDIR.db file.

```
read_file BIDIR.db
```

5. Execute check_test.
   Which testability problem is being reported?_____.

```
current_design BIDIR
check_test
```

   First look for the TEST-563 message. This is the scan shift issue mentioned in the background information.
   Analyze the reported **capture problem** (TEST-478) by examining the messages TEST-559 and TEST-473. (Do NOT change anything yet!!)

6. To ensure valid capture data, specify the value for the test_default_bidir_delay variable so the bidirectional port releases data after the active edge of the clock. Should the bidirectional delay value be set to occur before or **after your capture clocks**?

```
set test_default_bidir_delay 110
```

7. Repeat ATPG.

```
check_test
```

Which faults are still untested? _____.

8. Look at the TEST-563 message.
   Which mode does DFTC assume for the bidirectional ports during scan shift?
   Specify output mode by using the following command:.

```
set_scan_configuration -bidi_mode output
```

9. Re-run check_test. What happened?

```
check_test
```

10. Now run insert_scan.

```
insert_scan
check_test
```

11. Remove all designs.

```
remove_design -design
```

Part 2: Top-level handle bi-direction problem

12. Read in the BIDIR.db file Again. Let's try to run **insert_scan at the top-level** of a design with already existing scan at the core level in order to **fix problems that were not seen at the core level**.

```
read_file BIDIR.db
current_design BIDIR
```

13. Switch off the insertion of disabling logic for bidirectional ports.
    Then run insert_scan and check_test

```
set_scan_configuration -disable false
insert_scan
check_test
```

What is the reported problem now? _____.

How severe is the problem? _____.

```
set_scan_configuration -bidi_mode input
set_scan_configuration -disable true
```

In **core level**, switch off the insertion of disable logic for bidirectional ports.
Apply ➔ insert_scan

In **top level**, switch on the insertion of disable logic for bidirectional ports.
Apply ➔ insert_scan

14. Run insert_scan and check_test. Any problems now? _____.

```
insert_scan
check_test
```

Lab 3

DFT Compiler –

Scan Insertion Specifics

Learning Objective:

- The goal of this lab exercise is to understand scan configuration issues.

Part 1: Top-down Scan Insertion

1. Go to the directory lab3 and invoke DFTC from there. Read TOP_HIER.db

```
read_file TOP_HIER.db
current_design TOP_HIER
```

2. Execute check_test and preview_scan.

```
check_test
```

How many clock domains are there in the design? _____.

```
preview_scan -show [list scan_clocks cells]
```

How many scan chain? _____.

3. Now allow DFTC to mix clock edges within scan chains.

```
set_scan_configuration -clock_mixing mix_edges
preview_scan -show [list scan_clocks cells]
```

What is the length of each scan chain? _____.

4. To get equal length scan chains. → -clock_mixing mix_clocks

```
set_scan_configuration -clock_mixing mix_clocks
set_scan_configuration -chain_count 3
preview_scan -show [list scan_clocks cells]
```

5. Try NOT insert any lockup latches. Use preview_scan to verify.

```
set_scan_configuration -add_lockup false
preview_scan -show [list scan_clocks cells]
```

6. The block ADES contains a 4-bit shift register, which should be **declared as a scan segment to avoid scan replacement of the shift register cells**. This will save area.

```
set_scan_segment shift -access [list test_scan_in instA/sh1_reg/D \\
  test_scan_out instA/sh4_reg/Q] -contains [list "instA/sh*reg"]
set_scan_element false [find cell "sh*_reg"]
preview_scan -show all
```

7. (Option action)
   If you want to change the routing order within scan chains, but you do not like typing large lists of instances, use the "preview_scan -script" method.

```
preview_scan -script > scan.scr
```

then edit scan.scr

```
include scan.scr
preview_scan -show all
```

8. By default DFTC generates new ports for use as scan in/enable/out. There are, however, some exceptions regarding scan out ports. You will now explicitly define existing ports for use as scan in/enable/out. Please note that the scan enable port should always be a non-functional port.
   Defining Scan Ports:

   Type: `include lab3_1.dc` ***. It contains following 7 instructions.***

```
set_scan_signal test_scan_in  -port {b1[1]} -chain path1
set_scan_signal test_scan_out -port {d1[1]} -chain path1
set_scan_signal test_scan_in  -port {b1[2]} -chain path2
set_scan_signal test_scan_out -port {d1[2]} -chain path2
set_scan_signal test_scan_in  -port {b1[3]} -chain path3
set_scan_signal test_scan_out -port {d1[3]} -chain path3
set_scan_signal test_scan_enable -port test_se -hookup sebuf/Z
```

9. Verify your specification using preview_scan.

```
preview_scan -show all
```

10. Configuration of bidirectionals.

```
set_scan_configuration -bidi_mode input
check_test
```

11. Top-Down Scan insertion.

```
insert_scan
check_test
report_test -scan_path
estimate_test_coverage
```

12. Remove all designs.

```
remove_design -designs
```

Part 2: Bottom-up Scan Insertion

13. Use the following command sequence as a template for a script to perform scan synthesis for blocks **ADES, BDES, CDES** and **DDES**. Advanced option: use the **foreach** looping construct of dc_shell.

    Type: `include lab3_2.dc` ***. It contains following 7x4 instructions.***

```
read_file -format db ADES_GENERIC.db
current_design ADES
compile -scan
check_test
preview_scan -show all
insert_scan
check_test
report_test -scan_path
write -format db -hierarchy -output ADES.db
```

    How many block internal scan chains do you get for each of the four designs?

Design	Chain #	Scan Clock/Edge	Chain Length

14. Read Top level design and link design, the block .db files that you saved from the first step will be read automatically.

```
remove_design -designs
read_file TOP_ONLY.db
current_design TOP_ONLY
link
```

15. Defining Scan:

```
set_scan_configuration -bidi_mode input
check_test
preview_scan -show all
set_scan_configuration -chain_count 10
preview_scan -show all
```

16. Defining Scan Ports:

Type: `include lab3_3.dc` *. It contains following 10 instructions.*

```
set_scan_configuration -clock_mixing mix_clocks
set_scan_configuration -chain_count 3
set_scan_configuration -add_lockup false
set_scan_signal test_scan_in -port {b1[1]}
set_scan_signal test_scan_out -port {d1[1]}
set_scan_signal test_scan_in -port {b1[2]}
set_scan_signal test_scan_out -port {d1[2]}
set_scan_signal test_scan_in -port {b1[3]}
set_scan_signal test_scan_out -port {d1[3]}
set_scan_signal test_scan_enable -port test_se
```

17. Insert scan and check:

```
insert_scan
check_test
```

18. Subdesign ADES contains a shift register, which does not need to be scan replaced. In this step, you will repeat scan insertion with the shift register in block ADES declared as scan segment.

Type: `include lab3_4.dc` *. It contains following 16 instructions.*

```
remove_design ADES
read_file ADES_GENERIC.db
current_design ADES
set_scan_configuration -style multiplexed_flip_flop
compile -scan
set_scan_element false [find cell "sh*_reg"]
set_scan_segment shift -access [list test_scan_in sh1_reg/D test_scan_out \\
  sh4_reg/Q] -contains [list "sh*_reg"]
check_test
preview_scan -show all
insert_scan
check_test
report_test -scan_path
write -format db -hierarchy -output ADES.db
current_design TOP_ONLY
check_test
```

Lab 4

DFT Compiler –

Initialization Sequences & AutoFix

Learning Objective:

- The goal of this lab exercise is to get you familiar with test protocols and AutoFix.

Part 1: Initialization Sequences

1. Go to the directory lab4 and invoke DFTC from there. Read lab4.v

```
read_file -f verilog lab4.v
```

2. Compile circuits to be scan-ready netlist.

```
compile -scan
```

3. Check the testability of the current design. What wrong? _____.

```
check_test
```

4. Try to control asynchronous reset for the violation FFs by set "cdn" as 0 in test mode.

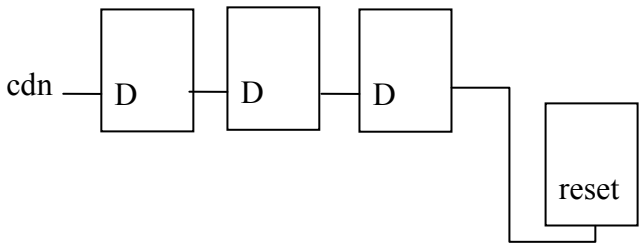
```
set_test_hold 0 cdn
check_test
```

5. Writeout test protocol file in SPF format, then edit this file.

```
write_test_protocol –format stil –out lab4_init.spf"
```

6. **Edit lab4_init.spf**.
   The third shift registers control the reset port of other register, we have to use 3 clock to stable the reset port.

```
"test_setup" {
   W "_default_WFT_";
   V {  "clk"=0; }
   V {  "cdn"=0; "clk"=P; }
   V {  "cdn"=0; "clk"=P; }
   V {  "cdn"=0; "clk"=P; }
   V {  "cdn"=0; "clk"=0; }
}
```



7. After save the test protocol file, read the modified test protocol file as initial test protocol,

and then check_test again. This time you can try to use -verbose option. What kind of information DFT compiler presents this time?.

```
remove_test_protocol
read_init_protocol -f stil lab4_init.spf
check_test -verbose
```

8. Set scan configuration. Use dedicated scan enable (scan_en), scan in and scan out (user define) and try to minimize chain count.

Type: `include lab4_1.dc` *. It contains following 4 instructions.*

```
set_scan_configuration -clock_mixing mix_edges
set_scan_signal test_scan_enable -port scan_en
set_scan_signal test_scan_in -port din
set_scan_signal test_scan_out -port {Dbus[7]}
```

9. Preview the scan chain routing. Is it all right?
   If OK, then go ahead and implement scan chain routing.

```
preview_scan -show all
insert_scan
```

10. Check test (check_test) to find any violations after scan insertion. Is it OK? Also Use the command (report_test) to see the scan path report.

```
check_test
report_test -scan_path
```

11. Estimate fault coverage.

```
check_test
estimate_test_coverage
```

12. Write out STIL procedure file (SPF) and synthesized netlist for TetraMAX.

Type: `include lab4_2.dc` *. It contains following 3 instructions.*

```
write -format verilog -hierarchy -output "lab4_dft.v"
write -format db -hierarchy -output "lab4_dft.db"
write_test_protocol -format stil -out "lab4.spf"
```

13. Remove all designs.

```
remove_design -designs
```

Part 2: Using AutoFix

14. Re-read lab4.v and compile circuits to be scan-ready netlist and check_test.

```
read_file -f verilog lab4.v
compile –scan
check_dft
```

15. Enable AutoFix and assign the test mode port, make sure the TM held on 1.

Type: `include lab4_3.dc` *. It contains following 4 instructions.*

```
set_dft_configuration -autofix
set_dft_signal test_mode -port TM
set_test_hold 1 TM
check_dft
```

16. Repeat step8 to 11, but change insert_scan to insert_dft, check_scan to check_dft.

Type: `include lab4_4.dc` *. It contains following 7 instructions.*

```
set_scan_signal test_scan_enable –port scan_en
preview_dft –show all
insert_dft
check_dft
report_test –scan_path
estimate_test_coverage
```

17. Write out STIL procedure file (SPF) and synthesized netlist for TetraMAX.

Type: `include lab4_5.dc` *. It contains following 3 instructions.*

```
write –format verilog –hierarchy –output "lab4_2_dft.v"
write –format db –hierarchy –output "lab4_2_dft.db"
write_test_protocol –format stil –out "lab4_2.spf"
```

18. Which one (Part1 or Part2) has the better fault coverage? _____.
    WHY?_____.

Lab 5

TetraMAX –

Continue from DFT Compiler

Learning Objective:

- Learn how to use TrtraMAX after we generated the STIL procedure file and synthesized netlist from DFT compiler.
- In this Lab, we type the command directly; we will introduce how to use the GUI in next Lab.

1. Go to the directory Lab4 and invoke TetraMAX from there.

```
tmax &
```

READ LIBRARY MODULES THEN THE DESIGN

2. Read in library and CUT (synthesized netlist generated in lab 4).

```
read netlist cb35os142.v
read netlist lab4_2_dft.v
```

CREATE THE FLATTENED IN-MEMORY IMAGE (Build)

3. Build the fault simulation model. Are there any error or warning messages? Does TetraMAX enter into DRC mode? If yes, you can go through the following procedures.

```
run build asyn
```

READ STIL PROTOCOL FILE

4. Run design rule checking using STIL procedure file generated in Lab4. warning messages? _____.

```
run drc lab4_2.spf
```

POPULATE THE DESIGN WITH FAULTS

5. Does TetraMAX enter into TEST mode? If yes, you can go ahead and run ATPG procedure.Add fault list.

```
Add faults -all
```

GENERATE ATPG PATTERNS

6. Run ATPG. You will obtain test coverage and pattern number. Do you want to compact them? How many patterns needed? What is the test coverage? What is the total fault count?

```
set pat internal
run atpg –auto
report_summaries
```

SAVING ATPG PATTERNS

7. Do you satisfy the current result? If yes, you can write the test patterns out now; if not, try to improve it.

```
write patterns lab4_atpg.v -internal -format verilog -serial -replace
```

Lab 6

TetraMAX –

Getting start to the TetraMAX GUI

Learning Objective:

- Getting start to the TetraMAX GUI.
- This Lab we will go through all the flow by using GUI.

1. Go to the directory Lab6 and invoke TetraMAX from there.

```
cd Lab6
tmax &
```

2. Use the SET MSG button to bring up the Set Messages dialog box. Select Log File and enter:

```
try1.log
Then OK the form.
```

READ LIBRARY MODULES THEN THE DESIGN

3. Use the NETLIST button to bring up the read netlist dialog box and then BROWSE to select the library modules file **lab6.lib** then OK. Select RUN to execute the read netlist command.
   Repeat to use the NETLIST button to read the design file **lab6.edif**

CREATE THE FLATTENED IN-MEMORY IMAGE (Build)

4. Use the BUILD button (at top) to bring up the **RUN BUILD MODEL** dialog box. You should see the default top module name listed as LAB6, if not, type in LAB6. Take the default options and select RUN to perform the build operation.
   Don't worry about the warning messages, these are not fatal.

READ DCXP PROTOCOL FILE

5. Use the DRC button to bring up the **RUN DRC** dialog. Then use the BROWSE button to select the DCXP Protocol file named: **lab6.tpf** and then click on RUN to execute the run drc command.

```
YOU SHOULD GET A C1 VIOLATION:
Clock PIs off did not force off clock input...
```

   You can open the on-line help by moving the mouse cursor over the red text and hold the CTRL key down and click the RIGHT MOUSE BUTTON (CTRL-RMB).

USE ANALYZE TO INVESTIGATE THE CAUSE OF THE DRC

6. Use the ANALYZE button on the left menu to bring up the **Analyze dialog box**. Make sure the Rules tab is shown, if not click on the Rules tab so that you see a list of DRC violations such as B6, B8, B13, etc.

7. Select the **C1** rule class and notice that the Rule Violation now says **C1-1**. Click on OK to exit the dialog.TetraMAX will draw a schematic to show the problem graphically.

What you should observe is a small circuit with a **DFFRLPH** device that has its active low asynchronous reset pin **RB** at an **X** value. This net attaches to the top level port **RST2**.
What happen to RST2 ?

INVESTIGATE THE CAUSE FOR THE VIOLATION

8. Review the list of constrained PIs to see whether the RST2 port is on the constrained PI's list. Use the Menu:

   Report → PI Constraints

   Was the RST2 port on the constrainted PI's list? Sometimes a C1 violation occurs because a port should be defined as a constrained PI and it is not.

9. **TetraMAX requires that asynchronous set and reset ports are defined as clocks.** Review the list of clocks to see if RST2 has been defined as a clock.

   DRC> report clocks OR report → clocks

   Was the RST2 port in the clock list? _____.

   In this case, the RST2 port is an asynchronous reset but has not been defined as a clock via the Test Compiler Protocol file. We will have to do that ourselves.

10. Use the Scan → Clocks → Add Clocks... menu to bring up the **Add Clocks dialog**. You can type the port name RST2 into the box under Port Name or click on the **DOWN ARROW**, that brings up a port list browser.

    After the RST2 appears in the Port Name field check that the **Off State is 0.** Use the ADD button then use the OK button to exit and define RST2 as a clock.

11. After we changed the constraintes, we need to run **DESIGN RULE CHECKS** again.

    Click on CLOSE near ANALYZE to close the GSV window.
    Click on the Test button located in the lower right corner of the window.
    When in DRC mode the Test button will re-run DRC checks and enter TEST mode if no errors are encountered.

12. Did the C1 violation go away? No, other reset pins needed to be set, too!

    Add RST1, RST3, RST4 to be clock with off state = 0.

13. Review the current list of defined clocks, and enter TEST mode.

    DRC> report clocks

    You should observe clocks: CLK, RST1, RST2, RST3, RST4

Try Click on the Test button to enter TEST mode again (re-run DRC).

POPULATE THE DESIGN WITH FAULTS

14. Use the Faults → Add Faults... menu to bring up the **Add Faults dialog box**. Select All. OK the form to exit. (You've just populated the design with faults to be detected.)

15. Use the SUMMARY button, take the default options, and select OK to exit and generate the report.

    This will display the inital fault statistics and you should observe that there are roughly 9400 total uncollapsed faults. current_design FLOW2

GENERATE ATPG PATTERNS

15. Use the ATPG button to bring up the **Run ATPG dialog box**. Use the defaults for the rest of the form and click RUN to start the ATPG pattern generation process.

    What was the test coverage achieved? _____.

    What is the pattern count? _____.

16. We can run static pattern compression by:

    TEST> run pattern_compress 1

    TEST> rep sum

    How many patterns exist? _____.

    This step can be repeat to get fewer patterns.

SAVING ATPG PATTERNS

17. Use the WRITE PAT button to bring up the **Write Patterns dialog box**.

    Enter **try1.pat** into the Pattern File Name box.

    Select **Replace**, if file exists so that a checkmark appears.

    Select File Format of **VERILOG** (Single File).

    Select Scan Format of **SERIAL**.

    Use **OK** to exit the form and save patterns.

SAVING FAULT LISTS

18. Use the Faults → Report Faults... menu to bring up the **Report Faults dialog box**.

    Set the Report Type to **All**.

Select Output To of FILE and specify **try1.faults** as the filename.

Check the box for Replace, if file exists.

Use OK to exit the form and write faults to the file.

TURN OFF MESSAGE LOGGING

19. Use the SET MSG button to bring up the **Set Messages dialog box**.

    Remove the **check mark** from Log File.

    Keep the check mark for Display.

    OK the form to exit.

    (This turns off logging to a file and closes the file, but continues to display command output in the transcript window.)

20. Use an editor to look at the try1.log file.

COMMAND HISTORY

21. Type in the commands:

    TEST> report command -history > batch.cmd

    TEST> rep com –his

    Batch.cmd can be reuse as command file.

Lab 7

TetraMAX –

RAM Modeling

Learning Objective:

- The goal of this lab exercise is to understand how to create an ATPG RAM model.

Part 1: Set RAM As Black Box

1. Go to the directory Lab7 and invoke TetraMAX from there.

```
cd Lab7
tmax &
```

2. Use NETLIST to read in the Verilog design **lab7.v**.

3. Run **build_model** for **ram_test**.
   What's wrong? An error due to an undefined module.

4. Figure out which other modules are missing.

```
BUILD> report module –undefined
```

5. Declare the missing RAM as a black box and try the build again:

```
BUILD> set build -black_box RAM64x8
BUILD> run build
```

6. Define a clock **clk** with offstate 0.

```
DRC> add clocks 0 clk
```

7. Run DRC's using lab7.spf.

```
DRC> run drc lab7.spf
```

8. Use ATPG → AUTO Be sure to enable the limited sequential pattern generation!

```
What is the test coverage? _____.
What is the total fault count? _____.
```

9. Write a COLLAPSED fault list as try1.faults.

```
TEST> write faults try1.faults -collapsed -all -replace
Write down the collapsed fault count: _____.
```

Part 2: Create an ATPG RAM model

10. Exit TetraMAX and re-invoke TetraMAX.

11. Create a RAM module in file ram.v with the following specifications:

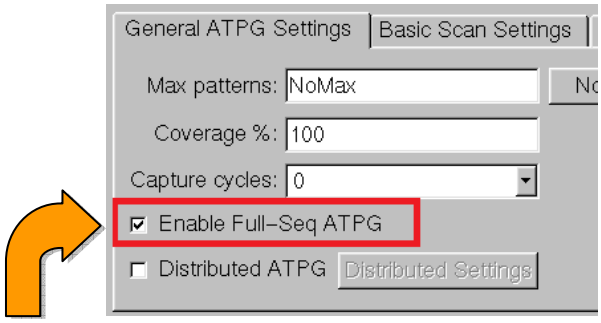
    write port #1:
    rising edge sensitive write clock w1,
    6 bit address bus a1
    8 bit data write bus d1
    write port #2:
    rising edge sensitive write clock w2,
    6 bit address bus a2
    8 bit data write bus d2
    single read port
    level sensitive, active LOW read enable r3,
    6 bit address bus a3
    8 bit data read bus d3
    HINT: The on-line reference has many RAM modeling examples that you can find by browsing the contents page or doing a keyword search. **File has been created in ram.v**

12. Use NETLIST to read in ram.v

13. Use NETLIST to read in lab7.v

14. Repeat step 6 – step 7.

    ```
    BUILD> run build
    DRC> add clocks 0 clk
    DRC> run drc lab7.spf
    ```



15. Use ATPG → AUTO **Be sure to enable the *sequential* pattern generation!**

    What is the test coverage? _____.
    What is the total fault count? _____.

16. Write a COLLAPSED fault list as try2.faults.

    ```
    TEST> write faults try2.faults -collapsed -all -replace
    Write down the collapsed fault count: _____.
    ```

17. Take a look at the memory information for this design:

    ```
    TEST> report memory -all
    TEST> report memory -all -verbose
    TEST> report prim /ram
    ```

Lab 8

TetraMAX –

Involving bus contention and Z-states

Learning Objective:

- Investigate Z2, Z9, or Z10 violations.
- Define ATPG gates.
- Define ATPG functions.

1. Go to the directory Lab8 and invoke TetraMAX from there.

```
cd Lab8
tmax &
```

2. Use NETLIST to read in the Verilog design **lab8.v**.

3. Use BUILD to build the design. The top level module name is **top**.

4. Define clocks for clk and sclk.

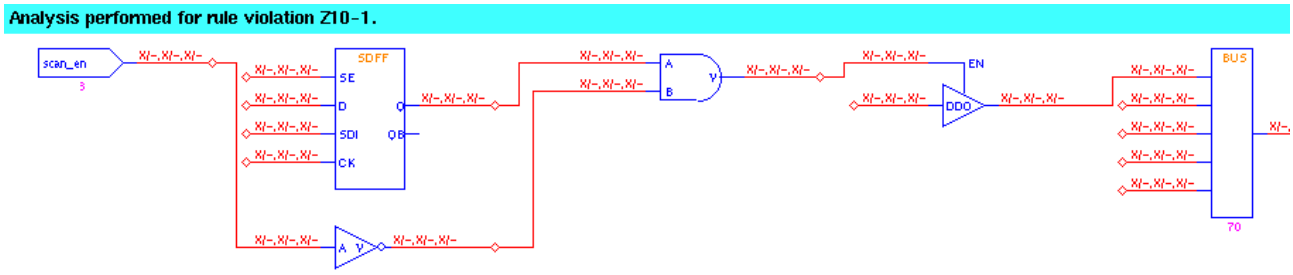
```
add clock 0 clk sclk
```

5. Use DRC to Run DRC's using file **lab8.spf**.

```
The DRC Summary Report should contain a number of DRC violations:
Rule Z1  (bus capable of contention)
Rule Z2  (bus capable of holding Z state)
Rule Z9  (bidi bus driver enable affected by scan cell)
Rule Z10  (internal bus driver enable affected by scan cell)
```

INVESTIGATE Z10 VIOLATIONS

6. Use the **ANALYZE** button to investigate a **Z10** violation.




```
To read the on-line reference on the DRC rule Z10 has these options:
A) You can CTRL-RMB over any Z10 text in the transcript.
B) You can type man z10.
C) You can use the Help menu to open up the on-line reference and then manually
navigate to the Z10 DRC rule.
```

7. Although the tri-state enable pin was connected with the output of scan flip-flop, but it was also controled by scan_en, so we can ignore this warnning.
Use the Rules → Set Rule Options... menu to bring up the **set rules dialog box**.

```
Specify Rule ID of z10.
   Select severity = ignore
```

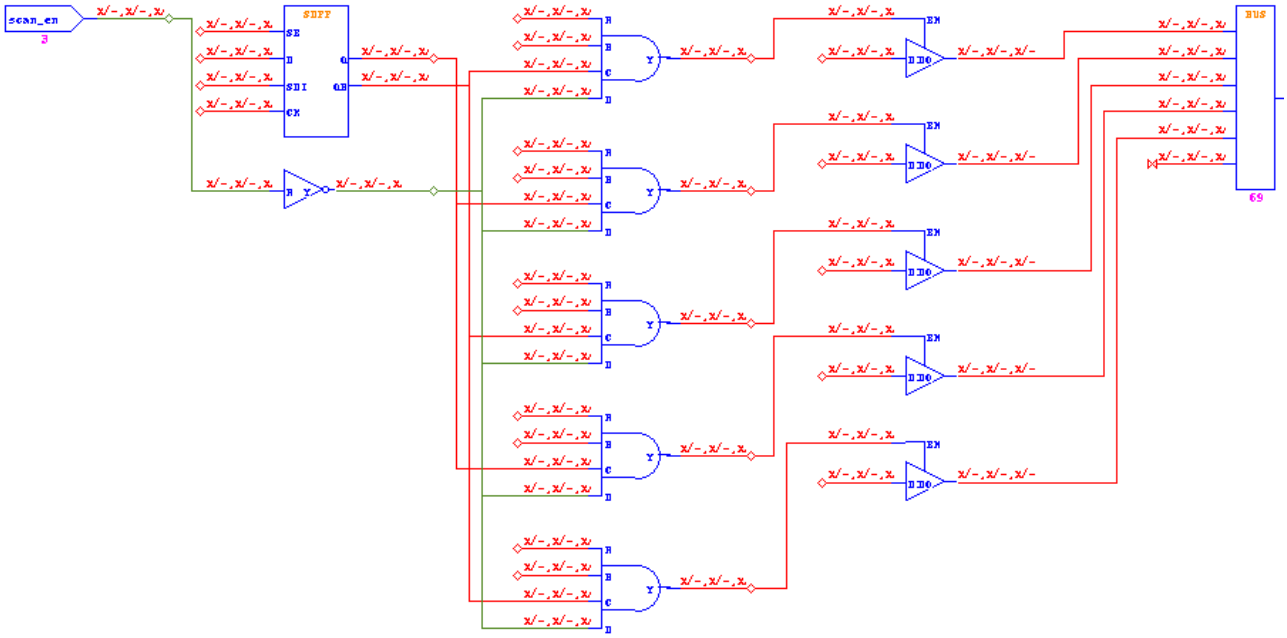
OK the form.

8. Use the TEST buttom located in the **lower right corner** of the windows to re-run DRC.

   → There are no more Z10 violations

INVESTIGATE Z9 VIOLATIONS

9. Use the **ANALYZE** button to investigate a **Z9** violation.



10. The inputs of the AND4 devices that form the **tri-state driver enables** you will find that one of the inputs comes from the **scan_en input.**
    → Set DRC rule Z9 to ignore and re-run DRC checks.

BUS SUMMARY and BUS ANALYSIS

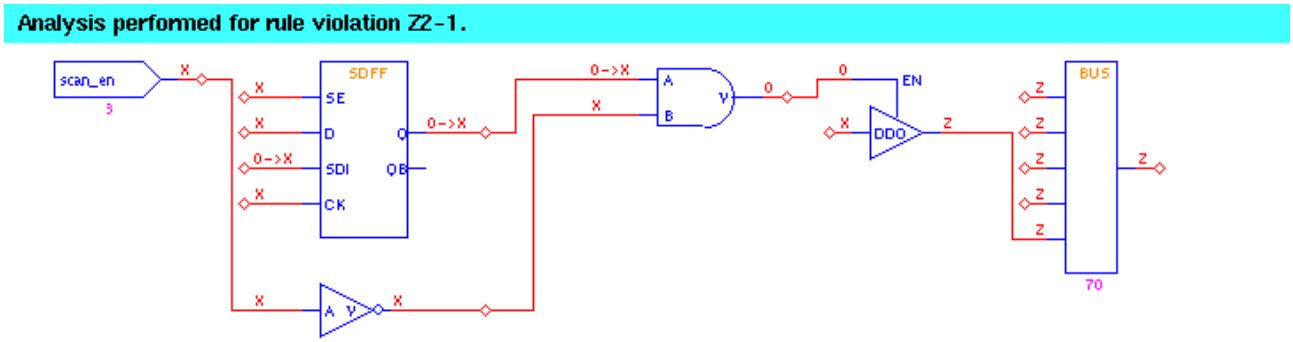
11. Obtain a summary of the bus analysis done during DRC. Use this command:

    ```
    report bus -summary
    ```

    Observe that there is one bus that has been classified as fail for contention checking, and one bus that has been classified as fail for Z-state. TetraMAX will automatically monitor the bus that has been classified as fail for contention and avoid patterns that cause contention. However, **TetraMAX *DOES NOT* automatically avoid patterns that have a floating internal net, or Z-state condition.**

INVESTIGATE Z2 VIOLATIONS

12. Z2 violations present to bus Z-state condition. We have to solve it. Use the ANALYZE to investigate a **Z2 violation**.



When TetraMAX finds a pattern that will result in the internal Z-state condition it places that pattern into the internal pattern simulation buffer as pattern 0. The simulation values from pattern 0 are displayed on the pins next to the primitives in the GSV.

On the far right of the schematic, you should observe a 5 input BUS primitive that has all inputs at a Z state simultaneously. To avoid patterns that produce an internal floating bus condition we must make use of the **ATPG PRIMITIVE** and **ATPG CONSTRAINT** capability.

DEFINE ATPG GATE

13. For each of the BUS inputs, track back to the tri-state driver for this input. Identify the pin pathname of the enable pin (EN pin) for all 5 drivers.

Use Edit → Environment → **Viewer dialog**. Check the "**Display Instance Name**".

The TSBUF pin name shows up on the schematic as EN. Typical pin pathnames are:

/mdn20/EN

/mdn21/EN

/mdn22/EN

/mdn23/EN

/mdn24/EN

14. Return to DRC mode. ATPG Gates must be defined prior to entering TEST mode.

Use the DRC buttom located in the lower right corner of the windows to return to DRC mode.

15. Use the Constraints → ATPG Primitives → Add ATPG Primitives... menu to bring up the **Add ATPG Primitives dialog box**.

Specify **mylabel** in the **ATPG Primitive Name field.** This is a symbolic label

that you create to refer to the ATPG primitive.

Select **Gate Type =** **SEL1**

Leave Module blank.

Specify the **first pin_pathname** in Input Constraints and **click** **ADD**.

Specify the **second pin pathaname** and **click** **ADD**.

Specify the **3rd, 4th, and 5th pin pathnames** in a similar fashion.

When all 5 pin pathnames exist in the window, click on **OK**.

Observe a message indicating an ATPG primitive has been added.

>>> If you have difficulty, run the command file: lab8a.atpg_gate.

16. You can review the **ATPG primitive** definition by entering the command:

```
rep atpg prim -all –verbose
```

NOTE: You will not see the same pin pathnames as you originally specified. TetraMAX translates references of INPUT pins into the corresponding OUTPUT pin which drives that input.

DEFINE ATPG FUNCTION

17. Now we have an **ATPG primitive** we can use it as part of an **ATPG constraint** definition.

    The **SEL1** function we defined earlier **produces a 1 on its output if exactly one input is 1** with the rest zero, otherwise it produces a 0.

    We hope there is always **one and only one** internal driver enabled
    → Define a constraint that keeps the output of the ATPG gate mylabel at a 1.
    → Floating net possibility is avoided.

18. Use the Constraints → ATPG Constraints → Add ATPG Constraints menu to bring up the **Add ATPG Constraints dialog box**.

    Specify **control1** in the Name field. This is a symbolic label that you create to refer to the ATPG constraint.

    Select **Value =** **1**

    Leave Module blank

    Specify the name of the ATPG Gate, **mylabel** in the Constraint Site.

    Enforce during = **ATPG**

    OK the form to create the constraint.

19. You can review the **ATPG constraints** definition by entering the command:

```
rep atpg const -all -verb
```

GENERATE PATTERNS WITH AND WITHOUT ATPG CONSTRAINTS

With the ATPG constraint defined its time to add faults and generate patterns but first we want to turn on the ability to get credit for Z's on outputs. Not all testers can detect Z's.

Assume that our tester can test for Z's so we want TetraMAX to give credit for detection of faults related to Z's on tri-state and bidirectional ports.

Use the **Buses → Set Bus Options...** menu to bring up the **Set Buses dialog box**.

Select **External Z =** **Z** {this is usually the default}.

OK the form.

20. Now we can go ahead and generate faults:

```
DRC> test
TEST> run atpg -auto
```

Observe the test coverage and number of AU faults.

21. What happens without the ATPG constraint in place?

```
TEST> reset state ; reset au faults
TEST> remove atpg constraint -all
TEST> run atpg -auto
```

You should observe that the test coverage is much greater. WHY???

CONCLUSION: Constraints serve a useful purpose but every constraint comes with a price and can potentially reduce achievable test coverage.