ECE 254 / CPS 225
Fault Tolerant and Testable Computing Systems

Testing and Design for Test

Copyright 2011 Daniel J. Sorin
Duke University

Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

A Good Reference and a Caveat

- M. Abramovici, M. Breuer, and A. Friedman. "Digital Systems Testing and Testable Design", IEEE Press, 1990.
- This book is the basis for much of the material in this section of the course, but many other textbooks on this subject exist.
- Caveat: ECE 254 is not a course on testing and testable systems. This course only has a small segment on these topics. If you're interested in pursuing this topic further, there are classes and research in ECE that you should explore.
  - See Prof. Chakrabarty (ECE 269)

Testing

- Testing: applying inputs to system and comparing outputs to expected outputs (for given inputs)
  - Goal: detect hard faults in system
  - Example: you drive a used Toyota Corolla before buying it
- (static, offline) Verification: checking the system design to make sure that it meets the specification
  - Goal: detect design flaws
  - Example: Toyota ensures that Corolla design will get 45 mpg
- Caveat: a verified design can be manufactured into a system that doesn't work correctly due to a flaw in manufacturing
  - This is why we need testing!
  - Example: Toyota makes 2% of Corollas that get only 30 mpg

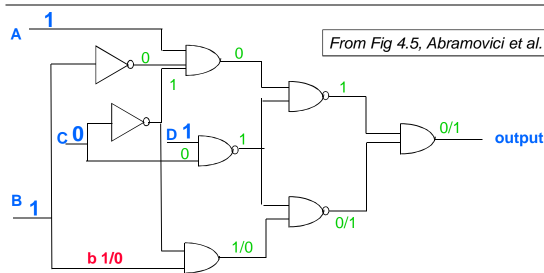
Aside on Dynamic Verification

- Normally, when we talk about verification, we are referring to static verification (as in previous slide)
- New terminology: dynamic verification = checking the behavior of a system as it is running
  - This is a form of error detection that operates at a higher level by checking higher level invariants of system behavior
  - E.g., DIVA is a dynamic verification scheme for microprocessor
  - Can detect operational faults
  - Can detect design faults
- Dynamic verification is known as online testing or concurrent testing in the testing research community

Testing Hardware and Software

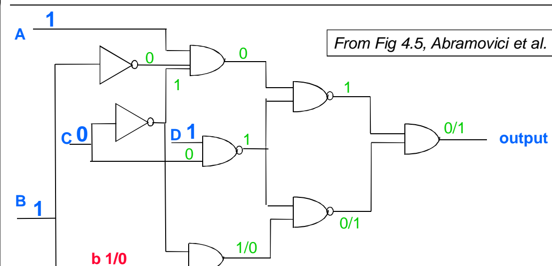
- Can test hardware and/or software
- Why test hardware?
  - Detect manufacturing faults
  - Can catch some design faults that were missed by verification
    - » Verification is not perfect for large complicated systems!
- Why test software?
  - Detect design faults (bugs) that were missed by verification
    - » Verification is very imperfect for real software
- For both, we apply inputs to the system and compare the outputs to what we expect for the given inputs
- For now, we focus on low-level hardware testing … and we start with combinational logic only

Testing Example



- Test vector ABCD=1101 detects fault b stuck-at-0
  - Question for later: do you see any other tests that detect this fault?
    - » Hint: does value of D matter?

Fault Activation



- This test (ABCD=1101) activates the fault b stuck-at-0
  - Generates an error at the site of the fault
  - Line justification problem: choosing a set of input values that puts a desired value on a given line in the circuit

Error Propagation

From Fig 4.5, Abramovici et al.

- This test propagates the error to a primary output
  - Makes all the lines along at least one path from fault site to output have different values in the presence of the fault

Sensitization (yes, this is a word)

From Fig 4.5, Abramovici et al.

- During the test, all lines whose values change in the presence of the fault are sensitized
  - A path of sensitized lines is a sensitized path

Detectability

- We want to choose inputs that test for all faults
  - Goal: smallest set of test vectors that covers all detectable faults
  - Do not necessarily need 2^N vectors for N-input circuit
- Some faults are undetectable
  - Due to redundancy in circuit, the faults are masked
  - So then why care about them? (Hint: single fault assumption)
- Simple example: TMR
  - How could you test for a fault in one replica?
  - Would have to add circuitry to be able to test them individually

Why Undetectable Faults Are Bad

From Fig 4.5, Abramovici et al.

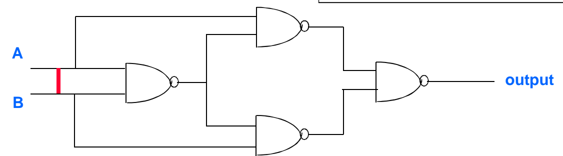
- Test vector ABCD=1101 does NOT detect b stuck-at-0 if undetectable fault a stuck-at-1 is present

Functional Equivalence of Faults

- **Two faults are functionally equivalent if the output is the same in the presence of each**
- **Simple example: NOT gate**
  - **Input stuck-at-0 is equivalent to output stuck-at-1**

Functional Equivalence Example

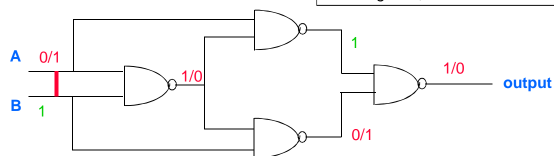
From Fig 4.11, Abramovici et al.



- **Bridging fault between A and B is functionally equivalent to output stuck-at-0**

Functional Equivalence Example

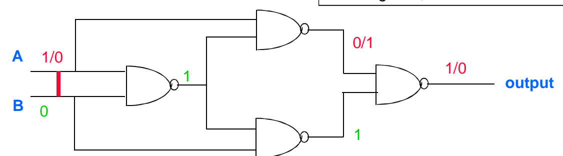
From Fig 4.11, Abramovici et al.



- **Bridging fault between A and B is functionally equivalent to output stuck-at-0**
  - **Consider case where A would've been 0 but now stuck at 1**

Functional Equivalence Example

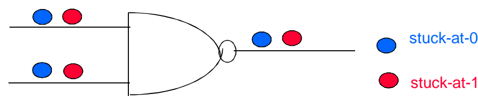
From Fig 4.11, Abramovici et al.



- **Bridging fault between A and B is functionally equivalent to output stuck-at-0**
  - **Consider case where A would've been 1 but now stuck at 0**

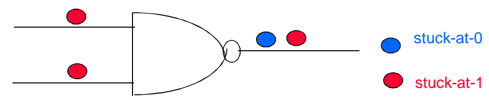
Fault Collapsing

- If a set of faults is functionally equivalent, we only need to use one test to detect any single one of them
  - Reducing the set of faults to test for by using equivalence classes is called fault collapsing
- Note that a test that detects functionally equivalent faults cannot diagnose which fault is present
  - To test for fault location requires distinguishable faults
- Simple example: 2-input NAND gate



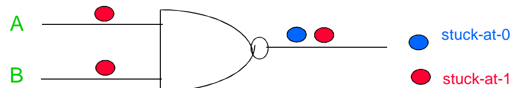
Fault Collapsing

- If a set of faults is functionally equivalent, we only need to use one test to detect any single one of them
  - Reducing the set of faults to test for by using equivalence classes is called fault collapsing
- Note that a test that detects functionally equivalent faults cannot diagnose which fault is present
  - To test for fault location requires distinguishable faults
- Simple example: 2-input NAND
  - Input stuck-at-0 is equivalent to output stuck-at-1 → collapse!



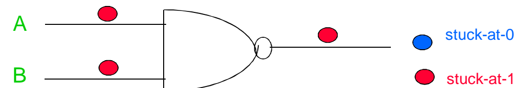
Fault Dominance

- A fault f dominates a fault g if any test that detects g also detects f (on the same output)
  - We don't have to test for the dominating fault f, since any test for g will test for f
- Back to our simple NAND example (after collapsing equivalent faults)



Fault Dominance

- A fault f dominates a fault g if any test that detects g also detects f (on the same output)
  - We don't have to test for the dominating fault f, since any test for g will test for f
- Back to our simple NAND example (after collapsing equivalent faults)
  - Test input vector to detect input B stuck-at-1 is AB=10, which also detects if the output is stuck-at-0. Thus, output stuck-at-0 dominates B stuck-at-1 and we don't need to test for it.
  - This reduction is called dominance fault collapsing



Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

Test Generation for Single Stuck-Ats

- Types of test generation (TG)
  - Random
  - Deterministic
    - » Manual
    - » Automatic
  - Hybrids of random and deterministic
- Goals
  - Quick, low-complexity process for generating faults
  - High fault coverage
  - Small number of test vectors

Random Test Generation

- Randomly produce test vectors
- Advantages
  - Easy and fast generation of test vectors
  - Can easily detect lots of faults with random tests
- Disadvantages
  - May take a huge set of test vectors to cover the hard-to-detect faults (i.e., faults that are only activated by a small number of possible tests)

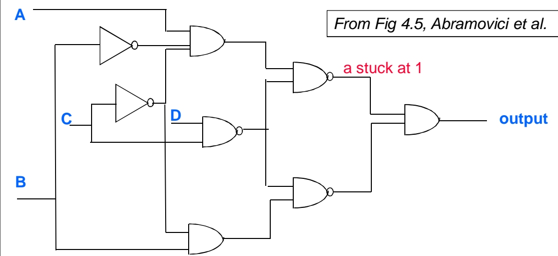
Deterministic Test Generation

- Can develop test vectors to target specific faults
  - Particularly useful for hard-to-detect faults
  - Manual test generation is not feasible for large circuits
- Automatic test generation (ATG) uses algorithms to devise tests that:
  - Activate each fault
  - Propagate the resulting error to an output
- After generating a set of tests for every fault, then ATG collapses the test set as much as possible
  - Functional equivalence
  - Fault dominance

Hybrid Testing

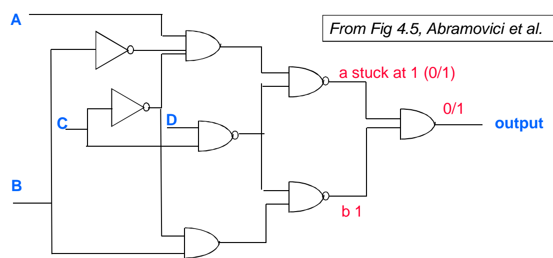
- **Goal: achieve best of both worlds**
  - **Random: low cost, quick generation of test vectors**
  - **Deterministic: smaller test vector sets**
- **Use random TG to quickly cover most faults**
- **Use deterministic ATG to cover the hard-to-detect faults**

Example of Activation and Propagation



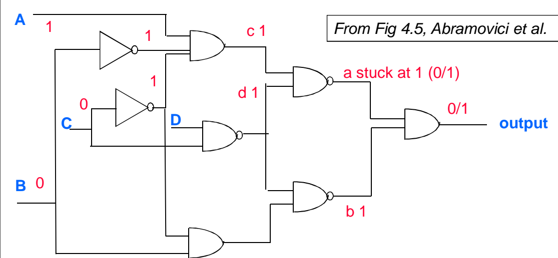
- **To test for a stuck-at-1, we must activate it and propagate it**

Example of Activation and Propagation



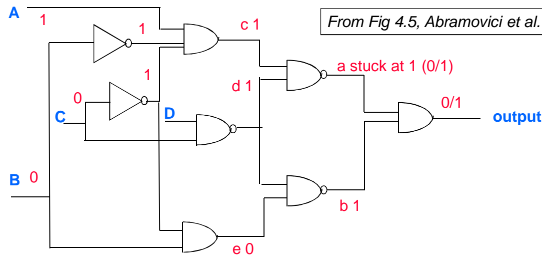
- **To propagate it, must set b to 1**
  - **Otherwise, the output would always be 0**
  - **Output is now equal to a (i.e., output = 0/1)**

Example of Activation and Propagation



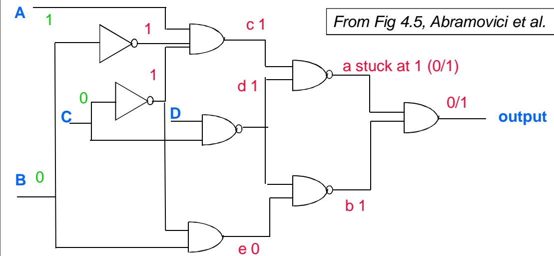
- **We must set a=0, and this implies that both of its inputs must be 1, which implies that A=1, B=0, and C=0**

Example of Activation and Propagation



- B=0 implies that e=0, which implies that b=1 (whew!)
- Note that C=0 implies that d=1 (also good)

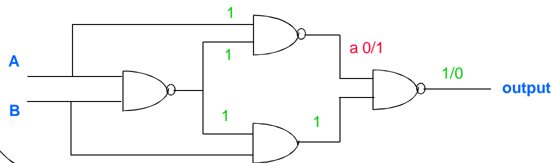
Example of Activation and Propagation



- So now we've determined that our test is ABCD=100x
  - The value of D doesn't matter for this test

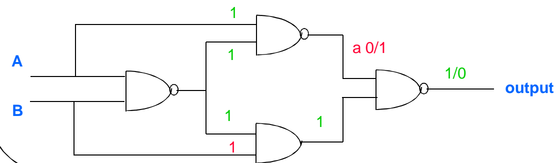
Implication

- We lucked out in the previous example --- we never implied a value on a line that was incompatible with another implicated value for that line
- If that happened, we'd have to try to find another test
- Example below: a stuck-at-1
  - Implicated values shown in green … so far, so good



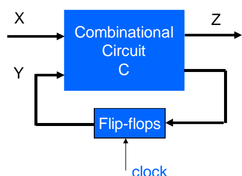
Implication

- We lucked out in the previous example --- we never implied a value on a line that was incompatible with another implicated value for that line
- If that happened, we'd have to try to find another test
- Example below: a stuck-at-1
  - We now have that A=1 and B=1, but A nand B = 1 → backtrack!



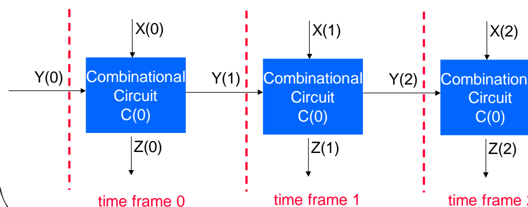
ATG for Sequential Circuits

- So far, we've only looked at combinational logic
- How can we deal with sequential logic?
- One option: make it look more like combinational!
  - Make each cycle a new frame of a combinational circuit



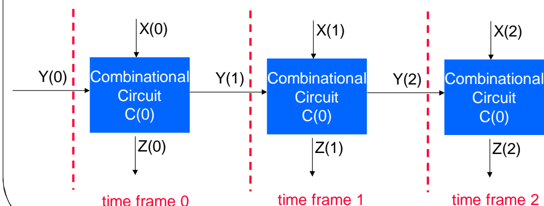
ATG for Sequential Circuits

- So far, we've only looked at combinational logic
- How can we deal with sequential logic?
- One option: make it look more like combinational!
  - Make each cycle a new frame of a combinational circuit



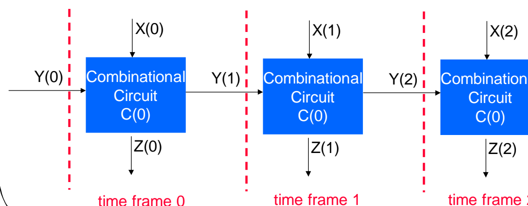
ATG for Sequential Circuits

- Now, we want to be able to propagate an error to an output Z(n), where smaller n is better (quicker!)
- The test sequence is X(0), X(1), … X(n)
  - Remember that each X(i) is a test vector of inputs (at cycle i)



ATG for Sequential Circuits

- Using time frames, we can create a combinational circuit and then just use ATG for combo circuits
- Catch: have to choose when to propagate fault, i.e., must choose n for Z(n). If impossible, try larger n



Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

Functional Testing

- Testing a system at a higher level of abstraction
  - If a system works at higher level, then you don't have to worry about the lower level
  - The user only sees the high level abstraction, not transistors
- For example, instead of testing for every possible stuck-at in a Pentium4, test to see if it boots Windows
  - If it boots Windows, you might believe that it is OK
- Other examples?
- Functional testing can be done at many levels
  - E.g., you could probably consider stuck-at testing to be functional if you were only worried about logic gate functionality

Why Use Anything BUT Functional Testing?

- If functional testing subsumes lower-level testing, why do we bother with low level testing for stuck-ats?
- Reason #1: Devising functional tests with good fault coverage isn't easy
- Reason #2: Fault location and diagnosis --- during fabrication, if you notice that a lot of faults are in the same place, you might be able to adjust the fabrication process to make that place less of a problem
- Any other reasons?

Validation in the Real World

"Validating the Pentium 4 Microprocessor," by Bob Bentley and Rand Gray (Intel), DSN 2001

Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

Software Testing

- Goal: stress-test the software in order to find bugs more quickly than would occur for typical inputs
- This is a job that many entry-level software engineers start with
  - Trying to break software forces engineer to better understand it
- Software testing tends to be almost entirely functional testing

When Do We Stop Testing?

- Comprehensive software testing is impossible problem for almost all real software
  - Must try to find as many bugs as possible, while realizing that it's impossible to find all of them
- So when do we stop testing?
  - When rate of bug detection decreases below threshold
- Threshold depends on software
  - People rely more on Microsoft Windows than they do on a small utility program → Microsoft ***should*** test more thoroughly
- Key idea: if you're still finding lots of bugs, it's likely that there are still a bunch more that you haven't found yet

Software Testing Techniques

- Test software in parallel
  - Get many people to test it for you → release a beta test version of cool software and people will test it for free
    - » This is how I helped to debug a simulator called Simics
- Provide software with worst-case or unusual inputs
  - Input a number when prompted for a string
  - Input a character when asked for a floating point number
  - When asked how many entries you'll need in a database, give it a huge number and see if that breaks it (e.g., because the database stupidly tries to allocate all that memory immediately)
  - When configuring a simulator, give it unusual parameters
  - Etc.

Testing Non-deterministic Software

- **Non-determinism**: if a program won't behave the same exact way each time it is run with the same inputs
- With non-determinism, it is very hard to reproduce a bug, so a test that finds it once may not be able to find it again
  - You may not be able to tell the programmers what to fix

Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

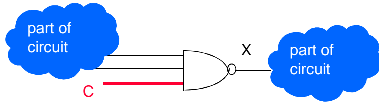
Design for Test (DFT)

- How do we design a system so that it is has good **testability**?
- We must improve the system's:
  - **Controllability**: ability to control the value of lines in the circuit
    - » 1-controllability: ability to set line to 1
    - » 0-controllability: ability to set line to 0
  - **Observability**: ability to observe the values of lines in the circuit
- We're going to cover two classes of approaches
  - Ad hoc schemes
  - Systematic schemes that use scan registers

Ad Hoc Scheme 1: Test Points

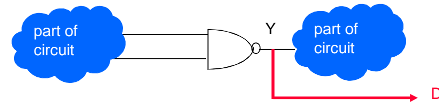
- We can add test points to the circuit
  - Control points
  - Observation points
- Control points are added inputs that we use to improve controllability
- Observation points are added outputs that we use to improve observability

Control Point Example



- **Adding new input C enables us to better control line X**
  - **Can easily set X to 1 (by setting C to 0)**

Observation Point Example



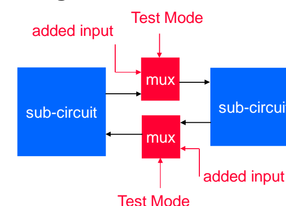
- **Adding new output D enables us to easily observe line Y**

Ad Hoc Scheme 2: Initialization

- **For sequential circuits, it is important to be able to get the circuit into a known state**
  - **It's tough to test something that's in an unknown state**

Ad Hoc Scheme 3: Partitioning

- **For large circuits or circuits with many states, it is easier to test them if we break them up into smaller sub-circuits**
  - **As functions of circuit size, testing time and complexity are greater than linear**
- **In general, we can partition circuits by adding multiplexor logic and new controllability inputs:**

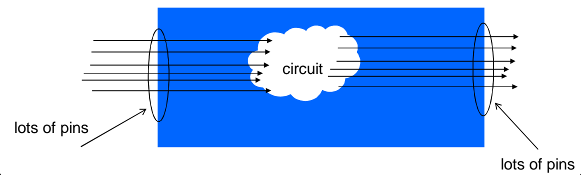


Ad Hoc Scheme 4: Avoiding Redundancy

- Recall that redundancy in circuits can lead to undetectable faults
- Two types of redundancy
  - Intentional: used for fault tolerance (good!)
  - Unintentional: most circuits have redundancy unless they've been optimized specifically to eliminate it (not good)
- We'd like to get rid of unintentional redundancy
- We'd like to keep intentional redundancy, but this means that we'll need to add test points to make sure that we can detect faults in the redundant logic

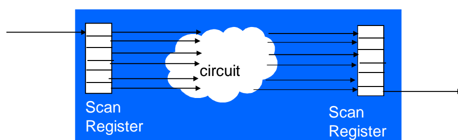
Systematic DFT: Scan Registers

- Test points are great, but they add lots of pins to a chip (both for inputs and outputs)
  - Pins are an expensive, scarce resource



Systematic DFT: Scan Registers

- One way to add test points without adding so many pins is to use scan registers (shift registers)
  - Sequentially scan test vector into scan register via a single pin, and then apply shift register contents to control points
  - Similarly, scan observation points into scan register and read out sequentially via a single pin



Scan Registers

- Advantage
  - Fewer pins!
- Disadvantages
  - Need scan hardware
  - Sequential → slower!!
- Commercial chips have scan registers
  - JTAG and IEEE standards for scan register design
- There are many, many different types of scan registers and scan design methodologies, but we will not cover these in this course

Outline

- Introduction and Terminology
- Test Generation for Single Stuck-At Faults
- Functional Testing
- Software Testing
- Design For Test (DFT)
- Built-In Self-Test (BIST)

Goals of BIST

- Want system to be able to test itself
- Why?
  - Generation of test vectors within chip eliminates pins used for testing → cheaper and higher bandwidth
  - Enables in-field testing (after deployment)
    - » If hard fault detected, can notify user and/or reconfigure system to handle the fault

Off-line vs. Online BIST

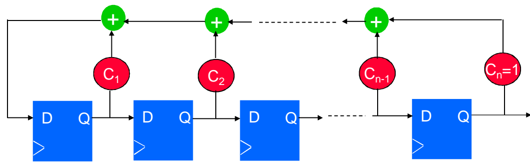
- Off-line
  - Testing while system is not performing its normal functions
- On-line or concurrent
  - Testing during operation
  - Also known as dynamic verification

A Paradox?

- If the system uses its own BIST hardware to test itself, how does it test this BIST hardware?
- Must be able to boot-strap based on assumption that a certain amount of the system (called the hardcore, for reasons I can't explain) is operational
  - Remember this concept from the Teramac paper??
- Examples of hardcore circuitry
  - Power and ground
  - Clock distribution
  - Test generation logic
- Probably need to do external testing to determine if hardcore is faulty

Hardware for Generating Pseudo-Random Tests

- **Linear feedback shift register (LFSR)**
  - Shift register with XOR-based feedback loop
  - C_i are feedback coefficients
  - C_i=1 implies that a connection exists
- **Based on initial state and choice of C_i, LFSR can generate long pseudo-random sequences (up to 2^n-1, where n is number of stages in LFSR)**



Flavors of BIST

- **Exhaustive**
  - Generate and apply all 2^N possible input combinations
- **Pseudo-random**
  - Similar to random ATG, but tests generated with LFSRs
- **Pseudo-exhaustive**
  - Partitions system and exhaustively tests these smaller sub-circuits

Outline

- **Introduction and Terminology**
- **Test Generation for Single Stuck-At Faults**
- **Functional Testing**
- **Software Testing**
- **Design For Test (DFT)**
- **Built-In Self-Test (BIST)**