VLSI / SOC Testing

Lecture 12

1. Combinational Test Set Compaction
   - Motivations: (1) test application time, (2) test data volume
   - Static compaction: compaction performed after ATPG is finished
   - Dynamic compaction: compaction performed along with ATPG
2. Dynamic Compaction:
   - When PODEM derives a vector v, v may still have many unspecified bits
   - fill the X's more intelligently to maximize detection of remaining faults $\longmapsto$ can use GAs here as well
3. Static Compaction: if test set not fully specified
   - can combine 2 or more vectors if they are *compatible*:
4. Static Compaction Overview
   - Idea: a fault may be detectable by a number of vectors in the test set, so we want to choose a smallest set of vectors from the original test set such that every originally detected fault is detected
   - Need: faultsim without fault-dropping $\longmapsto$ so a fault may be detected multiple times by different vectors
   - Construct: a dictionary mapping faults to vectors

5. Compaction Procedure

    - Identify essential vectors, which are the vectors by which some faults are detected exclusively
    - Since essential vectors must be included in the compacted test set, remove faults that are detected by these essential vectors first
    - For the remaining faults and vectors, find the best *cover* (subset) of vectors such that all faults in the detection dictionary are detected
    - GreedyCovering Procedure

      while compaction not finished
          sort vectors according to # remaining faults each detects
          pick v that detects the most of remaining faults

6. Dictionary can potentially be very large

    - To reduce dictionary size, one thing we could do is quickly identify the essential vectors and remove all faults they detect, then build the dictionary for the rest of the faults
    - To quickly identify essential vectors, simply perform faultsim with 2-det (drop fault after it's detected 2 times)
      $\longmapsto$ any fault that is detected only once is an *essential fault* and corresponding vector is an essential vector
    - If the remaining faults still large, do not build full dictionary, instead, build dictionary for N-detects (a fault can have at most N detection vectors)
      $\longmapsto$ This will result in a slightly suboptimal compaction

7. Reverse Simulation: A simple and cheap static compaction

    - No need to build dictionary
    - Original test set $\{v_1, v_2, ..., v_n\}$
    - a vector v_k is in the test set because it detects at least one fault missed by $v_1 \ldots v_{k-1}$
      $\longmapsto$ In other words, v_k detects some hard faults
    - by simulating the vectors in reverse order, perhaps the faults detected by some vectors in the beginning are detected by later vectors!

Example 1:

8. Test Vector Order Problem

   - Order the compacted sets such that vectors that detect most faults are placed early in the test set
   - If the entire test set will not be used, the first 80% vectors can still detect majority of faults

9. Sequential ATPG

   - A test sequence (of vectors) needed: problem much more complex than combinational ATPG
   - Vector order within sequence important, they determine the specific order of states visited
   - Iterative Logic Array (ILA) model used:

10. Deterministic ATPG

    - Excitation and propagation may each require several time-frames
    - Step 1: time-frame 0 excitation
    - Step 2: propagate fault effect till a PO, possibly in a later time-frame $\rightarrow$ more values needed at the FFs of time-frame 0, demanded by propagation
    - Step 3: justify the state at time frame 0

11. Problems and Issues in Sequential ATPG

    - State at time-frame 0 may be illegal/unreachable
    - If state is unjustifiable, need to backtrack and possibly find a new multi-frame propagation solution!
    - Must justify state from an all unknown state - sequence could be long

12. Ways to reduce ATPG costs

    - Minimize # state variables needed for state justification
    - Minimize # time-frames needed to justify
    - Detect illegal/unreachable states early to avoid future backtracks

    Example 2:

    Example 3:

13. Because defect/modeled fault is present in every time-frame, we now need 9-value algebra instead of just 5 values in combinational ckts

14. Approximation using 5-value algebra

    - use 5 values only, but must check after sequence is generated to make sure target fault is indeed detected

15. Some faults prevent the faulty machine from being initialized
    $\longmapsto$ to detect such faults, we need multiple observation strategy by exhausting all combinations of initial states

16. We can make better controllability/observability measures to help making better decisions in sequential ATPG

    - in addition to C0, C1, and O, also add Drivability, D(), of a signal
    - D(g) tells how easy/hard it is to drive a D or $\overline{D}$ from g w.r.t. controllability values

- it further differentiates observing a D from a $\overline{D}$

Example 4:

17. S-Graph of a sequential Circuit: a graph where vertices are FFs and directed edges between 2 vertices indicate a combinational path exists between them.

    Example 5:

18. Properties of S-Graph

    - If s-graph is acyclic (no cycles), then the faulty state is always initializable
    - Define: d_{seq} = sequential depth of the circuit = # FFs on the longest path in s-graph

- A test sequence for a detectable, non-FF fault in a cycle-free circuit has at most $d_{seq} + 1$ vectors.
- If s-graph contains cycles, the test sequence length is unbounded, since the sequential depth would be ∞

19. Sequential ATPG for acyclic circuits

    - If circuit is acyclic, then any fault can be tested within $d_{seq} + 1$ vectors
    - Unroll circuit $d_{seq} + 1$ time frames, and use combinational ATPG?
      $\longmapsto$ not so fast, fault is present in every time frame
    - In acyclic circuits, one can often lay out the FFs in a pipepline fashion, implicitly unrolling the circuit. In this case, perhaps each gate appears only once in the rolled-out circuit?
      $\longmapsto$ what if there exist multiple paths to a FF?

    Example 6:

VLSI / SOC Testing

Lecture 13

1. Testing Acyclic Sequential Circuits
   - Transform acyclic sequential circuit to a balanced combinational circuit $\longmapsto$ targeting some single faults may become multiple-faults
   - Apply only combinational ATPG to get test vectors
   - Transform vector to test sequence
2. Balanced Model (BM) of a circuit:
   - In a balanced circuit, all signal paths between any 2 nodes have the same number of FFs
   - Internally balanced: a balanced circuit where all node-pairs except those involving PIs are balanced
   - Strongly balanced: more strict than balanced, in which all reachable paths from an internal node to all PIs have the same sequential depth
3. Balanced Model Generation:
   - Step 1: Compute the weight of each PO to be the maximum sequential depth from any PI to the target PO
   - Step 2: For each PO, compute the weights of all gates within its cone, duplicate and split whenever necessary

Example 1:

4. Issues in balanced model:
   - Most sequential circuits are not balanced
   - Duplication-and-split may become expensive
   - Some faults become multiple-stuck-at-faults
   - Positive: combinational ATPG generally less expensive than sequential ATPG
5. Simulation-Based Sequential ATPG
   - Avoids backtracing and backtracking
   - Simplest: (weighted) random TG
6. CONTEST: 3 phases
   - phase 1: initialize fault-free circuit (no X's in FFs) by guided random walk
     $\longmapsto$ generate a set of vectors, pick the best, where
     $\longmapsto$ cost function = # FF's initialized
     $\longmapsto$ repeat till all FFs specified
     $\longmapsto$ Then, drop all detected faults by the initialization sequence

- phase 2: concurrent fault detection
  $\longmapsto$ let current state be s_f
  $\longmapsto$ generate a set of vectors, pick the best, where
  $\longmapsto$ cost function = # additional faults detected from s_f
  $\longmapsto$ repeat till remaining faults < threshold
- phase 3: single fault target
  $\longmapsto$ pick one fault at a time from remaining faults
  $\longmapsto$ generate a set of vectors, pick the best, where
  $\longmapsto$ cost function = $K\times$ activation + propagation
  $\longmapsto$ repeat till remaining faults < another threshold

7. GA-Test:

   - Similar to CONTEST, except now use GA to optimize/find best vector
   - No single target fault phase (only phases 1 and 2)
   - Instead of adding 1 vector at a time, individuals consist of sequences
   - GA individual:

8. Define: *distinguishing sequence:* a sequence, when applied, can distinguish 2 or more states by producing different output sequences

9. DIGATE

   - Power of distinguishing sequences: if it can distinguish a large set of states from another, it is very powerful
   - Dist. seqs learned by the ATPG at run-time. For each FF, store up to 5 dist. seqs.
   - Application to ATPG: when a fault is activated to one or more FFs, propagation of the fault effect is similar to distinguishing the faulty machine from the fault-free machine

10. Distinguishing Sequence Learning

11. DIGATE Notes

    - Fault-free and faulty circuits not exactly identical (but very similar), thus distinguishing sequence may not always work
    - Instead of targeting a group of faults, target single faults
    - For each fault, first activate it to a FF, then seed dist. seq. learned in the GA population to help propagate the fault effect to a PO
    - Dist. seqs useless if target fault is not activated

12. Targeting fault activation

    - Divide fault activation into 2 phases: (1) single time frame, (2) state justification
    - Single time frame is similar to combinational GA-based ATPG, with a relaxation step to make some FFs don't-cares
    - State justification: justify the required FFs

Example 2: State Relaxation

13. State Justification of the relaxed state
    - Define: set/reset sequences: a sequence that sets/resets a particular FF from an all-unknown state
    - Use set/reset sequences to guide GA to justify the required relaxed state

14. Problems with set/reset sequences
    - set/reset sequences sets/resets the specified FF, and at the same time, may set/reset other FFs as well
      $\longmapsto$ conflicts in trying to justify a combination of FFs simultaneously

15. Pseudo-register justification
    - Instead of set/reset sequences for individual FFs, use sequences that sets a group of FFs to a specific value

16. STRATEGATE: state-traversal based

    - treat entire state instead of partitioning the state
    - find similar states visited before, and use those sequences to justify the desired target state
    - data structures:

17. Genetic engineering partial solutions to get the target state

18. Logic-Simulation Based Sequential ATPG

    - Observation: effective traversal of circuit state space helps to detect more faults
    - Thus, using only logic simulation, traverse as many states as possible
    - problem: fault coverage plateaus early. Addition of arbitrary new states may not be helpful.
      $\longmapsto$ need to differentiate new states

19. Differentiation of new states

VLSI / SOC Testing

Lecture 14

1. Sequential Circuit Test Set Compaction
   - Motivation: reduce test application time and test data volume circuit
   - Problem: unlike combinational test set compaction, removing a vector in $T = \{t_0, t_1, ..., t_n\}$, the sequence is disrupted
2. Terminology: S_k/S_k^f = the fault-free/faulty state at time frame k for fault f
3. Compaction by insertion
   - If $S_k/S_k^f \equiv S_m/S_m^f$, where $m > k$, then what would inserting the subsequence starting from m at k cause?
4. Compaction by omission
   - Associate each vector with a flag omitted

     initialize omitted[i] = 0 ∀ i
     for i = 0 to n
       set omitted[i] = 1;
       faultsim vectors in T whose omitted flag = 0;
       if FC the same or higher, vector i is omitted;
       else reset omitted[i] = 0;
   - This loop can be repeated, and with a different order (i.e., n downto 0)

5. Compaction by selection
   - for each detected fault f, compute the subsequence that can detect f
   - then, compute the compact set using covering algorithm (note, subsequences can overlap)

6. The 3 previous techniques computationally expensive
7. Compaction by subsequence removal
   - Motivation: test sets generally traverse through a small set of states, thus many states are repeated
     $\longmapsto$ Can we remove subsequences that start and end on the same state?
8. Inert subsequence removal: an inert subsequence is one where start and end states are the same, and there are no faults detected within the subsequence

   Example 1:

9. Compaction by state-recurrence subsequence removal

   - a state-recurrence subsequence is one that start and end on the same state, but some faults may be detected within
   - Algorithm:

   step 1a: compute state recurrence subsequences for test set T
   step 1b: compute excitation and detection points for each fault
   step 1c: identify all faults, F_{sr} detected within such subsequences
   step 2: for all faults in F_{sr}, perform faultsim without fault-dropping
   step 3: analysis phase: if in a state-rec subsequence, all faults detected within are also detected elsewhere, and no other faults excited within and propagated beyond the subsequence, then this subsequence may be removed

10. Combining inert and state-recurrence subseq removal

    - First remove inert subsequences
    - Next, perform rec. subseq removal

11. Relaxed subseq. removal

    - Motivation: not every state variable needed to reach the next state

12. State relaxation for test set T

Example 2:

13. Reducing cost of compaction (can be applied to any compaction algorithm)

    - Fault partitioning: compact w.r.t. only hard-faults, since easy faults may be detected by vectors and sequences that detect hard faults
      $\longmapsto$ only 10% faults need to be targeted

14. Compaction by vector restoration

    - step 1: compute detection time for each fault (with fault dropping)
      $\longmapsto$ each fault has one unique detection point
    - step 2: for each fault (starting with the last detected fault), restore it
    - procedure for restoring a fault involves repetitive fault simulation

      i = detTime for fault f
      $T_f = [t_i]$
      repeat
          if (faultsim(T_f) detects f) $\rightarrow$ done
          else $T_f = [t_{i-1}, T_f]$
      until f detected

    - worst case: no vector is omitted (generally this is not the case)

    Example 3:

15. Using compaction for helping ATPG

    - Motivation: compaction eliminates unnecessary vectors, thereby leaving only good ones
      $\longmapsto$ inherent information embedded within compacted test set

16. Simple approach: extract weights from compacted test set

17. Observations for the simple approach

    - Most weights between 0.4 and 0.6

18. Approach #2: vector copying and holding

    - compacted vectors are there for a reason. Copy them exploits spatial locality
    - holding vectors exploits temporal locality. Observation tells us that holding useful vectors several clocks helps traverse state space deeper

19. Approach #3: correlation-based

    - instead of computing weights, compute spatial and temporal correlation within the compacted test set
    - generate additional vectors using these correlation

20. Approach #4: Spectrum-based

    - Analyze spectrum of compacted test set
    - Since compaction removes unwanted/unneeded vectors, it *filters* the noise out
    - What frequency components does each PI bit-stream have in the compacted test set?
    - Issue: how to analyze spectrum?

VLSI / SOC Testing

Lecture 15

1. Untestable Fault Identification
    - Motivation: ATPG spends lots of time targeting untestable faults
    - Want: quickly identify combinational and sequential untestable faults
    - Fault-dependent approach: analyze per fault
    - Fault-independent approach: based on circuit structure, determine which faults are untestable
2. FIRE:
    - Key concept: identify faults that require conflicting values on a signal in circuit to be testable
    - Algorithm:
        - compute S_0 = all faults that require line $a = 0$ for detection
        - compute S_1 = all faults that require line $a = 1$ for detection
        - any fault in $S_0 \cap S_1$ are untestable

Example 1:

Example 1 (continued):

3. Effectiveness of FIRE
   - Size of S_0 and S_1 critical
   - S_0 and S_1 depend on the number of implications
   - WANT: as many implications as possible
4. Extended backward implications
   - if z implies an unjustified gate g, then we can learn more on what z can imply on nodes preceding g

Example 2: (Review of backward implication

Example 3:

5. Constant nodes:

6. Extending FIRE to sequential circuits

   - In general, FIRE needs a set of conflicting value assignments $\longmapsto$ illegal/unreachabl states are conflicts

   Example 4:

7. Computing explicit illegal states can be expensive

   - incorporate sequentiality into the implication graph with edge weights
   - sequential conflicts learned!

   Example 5:

8. Managing the size of implication graph important
   - Number of nodes always a constant: $2 \times n$
   - Number of edges can be exponential
   - Need to periodically trim the graph by
     - remove transitive edges (transitive reduction)
     - eliminate equivalent nodes
9. Transitive reduction
   - remove transitive edges to
     - (1) reduce memory/storage requirements for all edges, and
     - (2) reduce the cost of DFS since fewer edges left

   Example 6:

10. Eliminate equivalent nodes
    - identify strongly-connected components (SCCs): nodes in a SCC have paths between every pair

11. Use representative nodes of SCCs only
    - approx. 50% nodes removed
    - many associated edges also removed
12. Using implications for fault-dependent untestable fault identification
    - Associate what necessary values are needed for each fault
    - Recall that in FIRE, S_0 and S_1 are computed as sets of faults untestable when a signal is equal to 0 or 1, respectively
    - Thus, for every fault in S_0, it must require the given signal to equal to 1 to be testable; conversely for faults in S_1
    - At the end of FIRE, every fault would have a list of necessary values $\longmapsto$ check if the list is *consistent*, ie. if the implications for every value in list conflict with each other or not
    - we need not store such lists of necessary values for every fault, only for the faults undetected by random vectors and those missed by FIRE to save storage
13. Algorithm:

For every signal s
  compute S_0
    add $s = 1$ to the lists of every fault in S_0
  compute S_1
    add $s = 0$ to the lists of every fault in S_1

For every undetected fault f
  L_f = list of necessary values for f
  imply each necessary value in L_f
  if conflict occurs
    f is untestable, go to next fault
  else if f can't be excited (implication on fault site = stuck value)
    f is untestable, go to next fault
  else if f can't be propagated (prop path blocked)
    f is untestable, go to next fault

Example 7:

14. In sequential circuits, a fault is present in every time-frame
    $\longmapsto$ thus if a fault is untestable, no vector sequence can detect the multiple fault
    $\longmapsto$ if a fault is combinationally untestable, it is also sequentially untestable

15. Single fault theorem

    - Motivation: using combinational algorithms to identify sequentially untestable faults
    - For each fault f, if it is testable, there must exist a time-frame TF_i during which it is first excited
    - the state for TF_i must be reachable
    - fault f in all time frames less than TF_i are not excited, thus the fault-free value = faulty value

16. Theorem: a target fault that is untestable in $C(n)$ is also untestable in the sequential circuit

    - There does not exist a sequence that can take the circuit from an all-unknown state to one such that the fault could be excited and propagated to at least one FF or PO

VLSI / SOC Testing

Lecture 16

1. Testing circuits with clock gating

2. Testing circuits with multiple clocks

3. Testing Bridging Faults
   - Recall that a bridge is formed between a pair of signal lines
   - Types of bridges:
     - AND-bridge:
     - OR-bridge:
     - a-dom:
     - b-dom:
     - etc.

Example 1:

Example 2:

4. Modeling bridging fault as a single stuck-at fault:

5. Feedback bridges

   - forward and back signals can affect one another

   Example 3:

6. Issues on BF testing

   - AND, OR, a-dom, etc. simple models
   - Recall the Byzantine problem:

7. Bridges involving dynamic CMOS gates

   - bridge can cause the discharge time to increase or decrease
   - thus, the testing speed critical, may need slow testing to catch defect!

   Example 4:

8. Testing Delay-related Defects

   - took 20 years to go from 100KHz to 100 MHz; but took only 6 years to go from 100MHz to 1 GHz, and 18 months from 1 to 2 GHz
   - Deep submicron (DSM) effects now make delay harder to test, in which delay can be caused by
     - noise: coupling/cross-talk
     - process variations: change in device parameters
     - thermal induced: heat generated when holes and electrons recombine; instantaneous switching causes local hotspots (many tens of degree hotter than elsewhere). Can't reach this temperature during testing, thus insufficiently stressing the circuit.
     - power-induced: some nodes place a higher demand on power-grid, causing lower VDD locally, thus slowing switches
     - performance-induced: lower V_t (threshold voltage) make chip more susceptible to noise
   - Thus path delays much harder to predict and estimate
   - At-speed test is needed. Structural path-oriented tests may cause yield loss (detection of non-functional delay-path). Such faults won't hurt circuit performance.
     $\longmapsto$ just need to exercise functional paths. (non-functional paths may be helpful for diagnosis)

9. Issues related to delay-testing

   - With DSM effects, testing for only critical paths becomes insufficient. Perhaps to pick the paths that are most likely to cause a delay problem based on statistical methods? But how?
   - Shrinking transistor sizes $\rightarrow$ more transistors need to be interconnected, resulting in longer wires.
     $\longmapsto$ On-chip interconnect becoming the major performance limiter.
   - Longer wires have higher resistance, thus to reduce resistance, wires are grown taller (not wider). But taller wires have more side-wall capacitance, causing more coupling/cross-talk problems.
     $\longmapsto$ Intro of copper wires instead of aluminum - one-time relief only.

- Scan-based design: load state, launch and capture. But this can potentially cause a large current surges.
- Clock skew: clock is distributed across the chip - so every FF may see the clock arriving at a slightly different time. A shorter path to a FF that clock arrives first may cause problems.
  Further, if a delay-defect is on the clock, delivery of clock to (a set) FF may be further skewed.

10. Transition fault model
    - Slow-to-rise (STR) and Slow-to-fall (STF) faults
    - Aim: capture spot (lump) delay defects on a signal in the combinational path
    - a slow-to-rise (fall) signal may cause the transition to arrive late at the FF
    - Need: 2 vectors to test for a transition.
      - vector 1: initialization vector
      - vector 2: test vector (launch transition and propagate to an observation point)

    Example 5:

11. Testing methods

    - Skewed Load: N-bit vector is loaded by shifting in the first N-1 bits, where N is the scan chain length. The last shift clock is used to launch the transition, followed by a quick capture.
      $\longmapsto$ only one vector is stored for each transition pattern in tester scan memory; the first vector is a shifted version of the stored vector.
    - Broadside testing (also called functional justification): a vector is scanned in and the functional clock pulsed to create the transition and subsequently capture the response.
      $\longmapsto$ only one vector is stored in tester per test; the second vector is derived from the first by pulsing the functional clock.
    - Enhanced-scan: two vectors (V1, V2) are stored in the tester memory. The first scan shift loads V1. It is then applied to the circuit under test to initialize it. Next, V2 is loaded, followed by an apply and a capture of the response. During shifting in of V2 it is assumed that the initialization of V1 is not destroyed.

12. Testing speed

    - High-speed testers extremely costly, testers usually slower than CUT
    - One solution is to add extra logic to circuit so that the speed of circuit in test mode becomes slower and comparable to tester speed
    - Alternative is to apply a vector k consecutive times
      $\longmapsto$ 1 ATE clock = k internal chip clock
      $\longmapsto$ output observed every k internal clocks

    Example 7

13. If ATE is fast enough, we may want to apply slow-fast-slow testing in non-scan or partial-scan sequential circuits
    - Apply vectors for initializing starting state at slow speed such that the circuit can be considered delay-fault free
    - Apply the activation time-frame at regular speed
    - Apply fault effect propagation also at slow speed
14. Path-delay fault
    - a more general delay model
    - Aim: capture defects due to process variation (delay for each gate increased by δ), cumulative delay of a combinational path exceeding the specified amount
    - Path starts at a PI/FF and ends at a PO/FF
    - a signal is an on-input of path if it is directly on the path; a signal is an off-input of the path if it is an input to a gate on path but is not an on-input
    - Exponential number of paths possible
      $\longmapsto$ instead of testing all paths, select only the longest $\sim$100 paths from circuit
15. Static sensitization
    - a path is static sensitizable if $\exists$ a vector such that all off-inputs in the path settle at non-controlling values
16. Single-path sensitization test
    - all off-inputs of a test pair must be non-controlling values
    - most stringent
    - very few paths are single-path sensitization testable
17. Non-robust PDF test
    - target path is the only faulty path.
    - condition 1: a transition is produced at path input and propagates to a path destination
    - condition 2: all off-path inputs assume non-controlling states only in vector 2

Example 8

18. Robust PDF test

- guarantees detection of PDF irrespective of delay faults in other paths
- condition 1: all on-path signals have different values for V1 and V2 (real events occurred on the path)
- condition 2: all off-path inputs assume non-controlling states in vector 2
- if on-path event is a transition from non-controlling to controlling value, all off-path inputs for this gate must have a steady non-controlling value in both V1 and V2, in order to ensure that delays in off-paths will not affect propagation of target path
- $\longmapsto$ a robust test is also a non-robust test

Example 9:

19. Path Counting

    - one-pass through the circuit - inexpensive
    - even when the number of paths could be 10^{20} or more

    Example 10:

20. Speed binning

    - if a chip fails at 1GHz, it may work at a lower frequency
    - Key: functionally, it may still be correct

21. More advanced concepts in path-delay faults
    - Validatable non-robust path delay fault
    - Functional (not static) sensitizable paths
    - Primitive faults that involve co-sensitization
    - False path identification

22. Validatable non-robust path delay fault
    - Non-robust test for a path assumes no other path delay faults exists $\longmapsto$ otherwise, this non-robust test may be invalidated
    - Must make sure all transitions to off-inputs come from paths that are not delayed

    Example 11:

23. Functional sensitizable path delay faults
    - is **not** statically sensitizable
    - more than one faulty path exists in circuit, usually are related paths that fanout and reconverge along points in circuit

24. More on functional sensitizable path delay faults

    - if at least one off-input is not late, then this path is not functionally sensitizable, and needs not be considered (cannot impact circuit)

    Example 12:

    Example 13:

25. Primitive Faults

    - robust, non-robust, and validatable non-robust are all primitive faults of cardinality 1.
    - for cardinality > 1, we need:
      - the multiple path is static sensitizable
      - no proper subset of the multiple path is static sensitizable
    - the single paths in such primitive faults are said to be *co-sensitized*

Example 14:

VLSI / SOC Testing

Lecture 17

1. Path delay fault simulation
   - Exponential number of paths
     $\longmapsto$ can't simulate per path, needs non-enumerative method
2. Path status graph
   - Vertices: PIs, POs, and internal nodes with fanout > 1

   Example 1:

3. Simulation Procedure
   - Mark edges in PSG as covered or not
   - will need to split/merge nodes if necessary

   Example 2:

4. Problem of false paths

   - Every redundant fault corresponds to one or more untestable path
   - Some false paths do not have a corresponding redundant fault

   Example 3:

5. Identification of false paths

   - Find conflicting value combinations

6. Find unsensitizable segments

   - all paths that contain such segments will be false
   - Define: prime segment: a functional unsensitizable segment, Q, where no proper sub-segment of Q is a functional unsensitizable segment
     $\longmapsto$ prime means smallest segment that is unsensitizable
   - Algorithm:
     - Step 1: find an unsensitizable sub-path, P, from a given PI
       this is done by concatenating edges until sub-path becomes unsensitizable
     - Step 2: going backwards in P, find the shortest unsensitizable segment
     - Step 3: Substitute the last leg of P with another edge and repeat steps 1 and 2
     - Step 4: repeat for all PIs and for all subpaths

   Example 4:

7. Testing for signal integrity
    - crosstalk between interconnects
    - transition on aggressor line causing a delay or glitch on another transition on victim line
    - needs: extract capacitive coupling between interconnect signals
    - ATPG: generate 2-pattern test that launches crosstalk behavior

    Example 5:

8. Functional Testing
    - Testing the circuit at a level/view other than netlist of basic gates
    - eg. FSM level, Register Transfer Level (RTL), Behavioral

9. FSM Testing
    - Given the FSM for the circuit, derive a test sequence that tests all states and transitions
    - Step 1: verify all n states exist in FSM
    - Step 2: test all transitions (verify correct transitional relationship)
    - Some terminologies:
        - Transfer sequence: takes the circuit to a desired state
        - Distinguishing sequence: a sequence that produces a unique output sequence for each starting state
          $\longmapsto$ by looking at the output sequence, we can determine the starting state prior to application of dist seq
    - Basic algorithm:

```
Apply an initialization sequence to a known state
Apply dist sequence to check for correct known state
for each state A in FSM
    compute a sequence from FSM to get to A
    apply dist seq to verify arriving at A
    transfer back to A
    for all transitions going out of A
        apply transition
        verify correct ending state with dist seq
        return to A (and test next transition out of A)
```

Example 6:

10. Effectiveness of FSM test sequence

    - if a s-a fault f is present and it is not untestable, then f must affect at least one transition in the FSM

11. Test sequences for FSM testing can be very long, if there are many states and transitions

    - Remedy: merge tests of different states together

12. FSM testing can be applied to testing regular structures

    - Key: test each cell in the regular structure exhaustively and simultaneously

    Example 7: (regular structure)

13. Define: a circuit is **C-testable** if it has an ILA structure and has a constant number of test vectors independent of the number of cells in the ILA

    Example 8:

14. The test vectors for regular structure can be obtained from the state diagram for the ILA representation!

    - exercise every transition
    - since the final cell is always observable, there is no need for distinguishing sequences!

    Example 9:

15. Theorem: if the state machine for the ILA is reduced and each state is repeatable, then the ILA is C-testable.

    - a state s_1 in transition $(s_1 \rightarrow s_2)$ is repeatable if $\exists$ a sequence T that takes the machine from s_2 back to s_1

16. All ILA's can be made C-testable with additional PIs

    Example 10

17. Universal test set: a test set that can detect all detectable faults regardless of the underlying implementation of the combinational function
$\longmapsto$ Want this universal test set to be as small as possible

18. A vector v_1 *covers* another vector v_2 if for every logic $1 \in v_2$, there is a logic 1 $\in v_1$.

19. Unate and binate: in a function z, if a variable x only appears in one polarity, then the function z is said to be unate in variable x, otherwise z is binate in x.

20. In order to compute universal test sets, we view the circuit as a truth table (can be constructed as a PLA), where inverters only appear at the PIs.

21. If the circuit has *no* unate variables (primary inputs), then the universal test set becomes the exhaustive test set!

22. true vectors and false vectors: an input vector that makes the function z evaluate to logic 1 is a *true* vector for z. Conversely for false vectors.

23. Monotone property: in a circuit with only AND and OR gates (no inverters), if $v_1 \supseteq v_2$, then $z(v_1) \supseteq z(v_2)$.

24. In a fault circuit with fault f, if $v_1 \supseteq v_2$, then $z_f(v_1) \supseteq z_f(v_2)$. Why?

25. Given true vectors v_1 and v_2, if $v_1 \supseteq v_2$, and if v_1 detects fault f, then we know $z(v_1) = 1$ and $z_f(v_1) = 0$. Since $z_f(v_1) \supseteq z_f(v_2)$, $z_f(v_2)$ must be 0 as well. Thus, if $z_f(v_1) = 0$, $z_f(v_2)$ must be 0; converse is not true.
$\longmapsto$ THEREFORE, we prefer true vector v_2 over true vector v_1 because v_2 will detect all faults v_1 detects, and possibly more.

26. minimal true vector: a true vector that does not cover any other true vector.

27. Given false vectors v_1 and v_2, if $v_1 \supseteq v_2$, and if v_2 detects fault f, then we know $z(v_2) = 0$ and $z_f(v_2) = 1$. Since $z_f(v_1) \supseteq z_f(v_2)$, $z_f(v_1)$ must be 1 as well. Thus, if $z_f(v_2) = 1$, $z_f(v_1)$ must be 1; converse is not true.
$\longmapsto$ THEREFORE, we prefer false vector v_1 over false vector v_2 because v_1 will detect all faults v_2 detects, and possibly more.

28. maximal false vector: a false vector that is not covered by any other false vector.

29. Universal test set: the union of min true vectors and max false vectors

    Example 11

VLSI / SOC Testing

Lecture 18

1. Testing at the Register Transfer Level

   - Target: assemble RTL instructions to test for flow or data errors
   - Fault Model:
     - I_i/I_j: instruction I_i replaced with I_j
     - R_i/R_j: register R_i replaced with R_j
     - etc.
   - generate test *programs* that exercise the microarchitecture; both excitation and propagation are needed

   Example 1:

   Example 2:

2. Hierarchical Test Generation

   - Motivation: most ckts designed with a hierarchy
   - benefit: high-level view can see things that gate-level cannot, such as global constraints, etc.
   - vectors generated for blocks/modules of design first, then they are justified to derive the test set for the entire chip
   - problems: justification of module vectors difficult

   Example 3: (overview)

   Example 4: (vector justification)

3. Global control constraints extraction

   - identify controller, control signals, and control inputs for each module
   - traverse the high-level circuit and obtain legal control words for each module's control input signals
   - store all the legal control words - may optimize them by finding a minimal cover

4. Global data-path constraints extraction

   - simulate and obtain relationships between values among buses

   Example 5:

5. Hierarchical test generation

   - Call gate-level ATPG to get a vector for the module under test, such that the vector does not conflict with the extracted constraints
   - Call high-level value justification and propagation
     - equation-solving
     - simulation-based

6. Alternatively, for a given module, one can generate a *test environment*

   - a test environment is a solution of symbolic values for signals outside of the module under test
   - many test environments may suffice
   - PODEM-like algorithm to compute the test environment

   Example 6:

7. High-level metrics for ATPG

- module reachability and channel transparency
  $\longmapsto$ is module reachable from global inputs?
  $\longmapsto$ global path to individual modules
- while traversing upstream or downstream modules from MUT, probe their suitability/capability for providing the required values
  $\longmapsto$ traversal based on branch and bound
- transparent channel is one where an entire path from global PI exists that can provide the required value to MUT, as well as one that bridges the MUT output to a global PO
- may also identify bottlenecks in search, which is useful for DFT to alter designs at the high-level
- Issues: how to represent the reachability and transparency info; how to resolve conflicts

8. Another approach, abstract the necessary blocks that are needed to test the module under test

   - similar to program slicing: identify the variables and statements that are needed to test a specific block in VHDL code
   - resynthesize using the program slice → much smaller circuit to test
   - map the derived test sequence back to original design

   Example 7:

9. Behavioral-level ATPG

   - use concepts similar to software testing
   - metrics:
     - statement coverage: every statement in code exercised
     - condition coverage: every condition/branch exercised in both directions
     - path coverage: specific paths covered
   - key: test set that exercised 100% statement and condition coverage, covers a large number of paths can exercise large portions of the circuit
   - problem: observability not well addressed
     ⟼ remedy: observability-enhanced statement coverage: make sure that the effect of exercising statement propagates to a PO

   Example 8:

VLSI / SOC Testing

Lecture 19

1. Design for Testability

   - Goal: change circuit structure to achieve
     - easy to generate test vectors (manually or automatic)
     - small test set - shorter test application time and data volume
     - easy to compute fault-free response
     - easier for diagnosis and debug
   - Important: the modified circuit must retain original circuit functionality
   - general aims:
     - increase the controllability/observability of some signals - make them easier to control/observe
     - make justification of states easier
   - penalties:
     - area overhead: extra gates/pins/routing added
     - performance degradation: might slow down the circuit speed

   Example 1: Ad-hoc circuit partitioning

2. Where to partition circuit
   - around existing muxes
   - along sensitized paths (make long paths shorter)
   - along buses (separate around buses)
3. Testability Point Insertion
   - Increase control/observability of nodes in circuit
   - First identify nodes that are hard to control/observe
   - Addition of muxes (extra area)
   - Avoid placing on critical paths

   Example 2:

4. Scan Design for sequential circuits
   - Make FFs fully controllable/observable at a cost
   - scanned FFs connected in a chain
     - becomes a shift-register
     - can now force specific values into FFs
     - can also shift out values to be observed

5. Scan Cell Design

   - LSSD (Level-Sensitive Scan Design)
   - Mux-based scan design
   - Cost: area overhead in each scanned FF
   - Cost: performance overhead
   - Avoid scanning FFs on critical path

   Example 3:

6. Full-Scan Design: scan every FF

   - Converts sequential circuit into a combinational circuit, only combinational ATPG needed
   - Large test set application time and test data volume
     - every vector requires $n + 1$ cycles, where $n = \#$ FFs
     - every pattern has $n + m$ bits, where $m = \#$ PIs
     - expected responses (FFs + POs) for each pattern also need to be stored

7. Multiple Scan Chains Design

   - test application time now $\frac{n}{k}$, $k = \#$ chains
   - need more test pins for k chains
   - test data volume the same

8. Partial Scan Design: scan only a subset of FFs

   - scan enough FFs to achieve FC similar to full-scan
   - test application reduced

- test data volume reduced?
- but, circuit still sequential, need sequential ATPG
- validation of vectors more complicated
- Key issue: which subset of FFs to scan?

9. Connecting Scan FFs

   - order of connection critical to area

   Example 4:

10. Partial-Reset

    - add a separate partial-reset pin to a subset of FFs
      - makes state *jump* to a different state
      - helps to re-orient traversal

11. Direct Loading of FFs

    - instead of reseting or shifting in the value into a FF, directly load the desired value $\rightarrow$ test application time reduced
    - area overhead
    - at-speed testing possible $\rightarrow$ can capture delay defects

    Example 5:

12. Boundary scan
    - scan PIs and POs on a PC board
13. How to select FFs for scan/load/etc.?
    - testability-based
    - cycle-cutting based (structure-based)
    - ATPG-based
    - hybrid
14. Testability-Based
    - compute SCOAP measures for original circuit
    - select the FF with highest C0/C1/O
    - scan it, its C0=C1=O = 0
    - recompute SCOAP for circuit and repeat
    - stop when all FFs C0/C1/O are below a threshold or a maximum number of FFs have been scanned
    - Problems with this approach: (1) SCOAP only a metric, (2) FF selection a greedy approach, thus not optimal

    Example 6:

15. Structure-Based
    - Construct the S-graph for circuit (nodes = FFs, directed edge = combinational path in one time-frame between the 2 FFs)
    - Cycles within S-graph means that FF values may depend on one another
      $\longmapsto$ hard to control some FF with only PIs
      $\longmapsto$ hard to propagate fault effect from some FF to a PO
    - If break the cycle, sequential depth no longer ∞
      $\longmapsto$ recall testing of acyclic circuits

- Goal: break all cycles (excluding self-loops)
- Algorithm:
  - identify all cycles in S-graph excluding self-loops
  - select min # FFs to scan such that all cycles are broken
- Problems with this approach: self-loops ignored, they can still cause problems in testing

Example 7:

16. ATPG-Based

- Quick run of ATPG
- for all faults that were aborted
  $\longmapsto$ examine why (which state was hard to justify, etc.)
- From the set of aborted states
  $\longmapsto$ compute the minimal subset of FFs to scan
- Problems with this approach: quick run of ATPG can still be expensive

17. Random Access Scan (RAS)

    - arrange scan flip-flops in a two-dimensional array
    - need additional pins to address the specific scan cell
    - do not need to load every scan FF for each pattern
      $\longmapsto$ only need to load those FFs whose value change from the obtained output of the previous test pattern

18. Optimizations to RAS

    - instead of indexing both row and column of RAS, only address the column. The rows advance progressively. (Progressive RAS)
    - one can also re-arrange test vectors to minimize the number of loads necessary, thus reducing test application time, as well as test power
    - can also use intelligent ATPG to generate vectors that target RAS or PRAS.

VLSI / SOC Testing

Lecture 20

1. Hybrid techniques for partial scan
   - Combining testability-based, structure-based, ATPG-based, etc.
2. Combining structure-based and ATPG-based
   - obtain subset of valid/reachable states with logic sim of n random vectors
   - map states onto the cycles in S-graph
     $\longmapsto$ this gives a new metric of testability of FFs in the cycle
   - Define: density of encoding: for a group of n FFs:
     density =
   - Define: partial valid state: a subset of k FFs in circuit such that the state represented by this k-tuple is valid
   - testability of each FF in a cycle =
   - testability of each FF for all cycles it is involved =
     T(FF) =
     $\longmapsto$ larger T() means harder to test

   Example 1:

3. Once a FF is selected for scan, the T() values can change, since the partial valid states are now different

   Example 2:

4. Another way to combine structure-based + ATPG based
   - Goal-1: Should we break some FFs that are involved in only self-loops?
   - Goal-2: Should we break large cycles with more than one scan FF? breaking cycle into small segments
   - Example of circuits without cycles: counters
   - Instead of valid states, use aborted states
5. Testability-based + ATPG-based
   - aborted states from ATPG + testability measures
   - find the best scan FFs that can reach most # of aborted states
   - Define: ADP (aborted-fault detectability potential):

Example 3:

6. Algorithm:

   while not done
   perform a quick ATPG to gather some aborted states
   compute ADP(FF_i) $\forall$ unscanned FF_is
   pick a subset of best FFs for scan

7. Making aborted states easier to reach

   - Collect easy-to-reach states as aborted states
   - Want: make aborted states reachable from easy-to-reach states

   Example 4:

8. Algorithm:

   while not done
   collect easy-to-reach states
   quick run of ATPG to collect aborted states
   for each aborted state S
   if S was made to be reachable, how many other aborted states will also be reachable?
   Pick best aborted state and scan FFs such that it can be reached from an easy-to-reach state

9. High-Level DFT

   - Motivation: insert DFT early on in the design cycle (don't need to wait till gate-level is available)
   - Simple approach 1: avoid data-flow loops in RTL
     $\longmapsto$ break the loops either by (1) scanning, (2) add mux to direct load register, or (3) avoid sharing the same register
   - Simple approach 2: Compute density of registers in circuit
     $\longmapsto$ any register with low density means hard to control

   Example 5:

10. Behavioral modification to make circuit more testable

    - Define: locus of execution: a node within the control-data-flow graph is the locus for the system if it is currently executing
    - Define: K-controllability: a decision/branch within the CDFG is K-controllable if the direction of the branch can be controlled by the PI K clocks before the locus reaches that decision node
    - Define: non-controllable: a decision/branch not K-controllable, or K cannot be pre-determined
    - Goal: make every node K-controllable, with a small K

Example 6:

11. Nested Loops

    - may need multiple test pins inserted

12. Non-scan high-level DFT

    - Designs contain controller and data-path, interacting via control and status signals
    - Generally, data-path modules easy to test, but the control and status signals hard to control/observe
      $\longmapsto$ module alone is easy to test, but composing a test sequence that involves other controller and data-path modules is hard
    - add controllability/observability to these signals at the high level
      - via muxes, extra test pins, etc.
      - specifying particular values at these control/status signals become easier
    - Advantage over gate-level DFT:
      - no scan needed, thus can test at speed, and reduce test application time
      - at gate-level, many more signals to analyze, may place DFT on extra signals or miss some critical ones
    - Disadvantages: may incur higher area overhead if there are many control/status signals to add DFT to

13. From another angle, we'd like to add DFT such that the resulting ATPG problem becomes as easy as full-scan

    - Break all FF cycles at high-level: make sure registers that form cycles are broken with test pins
    - Circuit becomes like a pipelined one, without need of scan-shifting

    Example 7:

14. Once the registers do not form cycles, we can further analyze value reachability, to enhance controllability of registers not directly connected to PIs

15. To reduce area overhead, utilizable paths can be exploited

    Example 8:

VLSI / SOC Testing

Lecture 21

1. Compression Techniques
   - 100,000 vectors in ATE, storing all 100,000 output responses (plus FF state if full-scan) can require huge storage
   - just store a signature
2. Simplest signature: parity (even or odd) for each PO bit stream
   - Problem: aliasing

   Example 1:

3. One's counting signature: actual number of 1's in each PO stream

   Example 2:

4. Transition counting signature: 2 numbers for PO stream
   - count both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions
5. Linear Feedback Shift Register (LFSR)

6. MISR: Multiple-Input Signature Registers
    - instead of having an LFSR per PO, combine them
    - slightly greater aliasing probability, due to masking

   Example 3:

7. Built-In-Self-Test
    - Positives:
        - No ATEs
        - place TPG directly on chip
        - Avoid ATPG somewhat
        - Technology and fault-model independent
        - Useful for field test and diagnosis
        - At-speed testing
    - Negatives:
        - Fault Coverage sometimes low
        - Hard to determine exact fault coverage (due to aliasing)
        - Hard to diagnose

   Example 4:

8. On-chip TPG
   - Store vectors in ROM:
   - Exhaustive: use of a counter
   - Pseudo-exhaustive:
   - Pseudo-random:
   - Weighted-pseudo-random:
9. LFSR-based TPG
   - want to generate all $2^n - 1$ patterns
10. Weighted pseudo-random TPG
    - to allow different weights
    - add additional boolean logic to LFSR to create weighting

    Example 5:

11. Output Response Analyzer: similar to compression technique with MISR aliasing effects on FC:

12. BIST Architecture: STUMPS (Self-Testing Using MISR and Parallel SRSG

13. BIST Architecture: BILBO (Built-In Logic Block Observation)
    - convert registers in circuit to BILBO registers
    - test the circuit in a pipelined fashion

14. How about storing multiple signatures?
    - architecture similar to STUMPS
    - extreme compaction - possible to store a signature for every pattern
    - ATPG constraints: must make sure fault is detected on odd number of FFs in a particular group (shift vector)
    - eases diagnosis (increased diagnostic resolution) and allows for built-in self diagnosis

15. Delay-Fault BIST

    - Aim: launch transition at every PI and FF, and sensitize (longest possible) path for each transition to a PO or state
    - launch transitions from TPG: only 1-bit change at a time
    - use same ORA to capture signature

16. Low-Power BIST

    - random vectors generally create more activity than functional vectors
    - Aim: derive BIST vectors that create fewer transitions and still achieve similar fault coverage
      - lower power during scan shift
      - lower power during circuit evaluation/simulation

VLSI / SOC Testing

Lecture 22

1. Memory testing
   - defect may be in address decoder or cell array
   - since each cell of memory can have a different state, the # of vectors to test a memory is potentially very huge

2. Address decoder faults
   - no cell accessed for a given/specific address
   - a given cell can't be accessed by any address
   - a given address accesses multiple cells
   - a given cell can be accessed by multiple addresses

3. Cell-array faults
   - stuck-at: a bit cell stuck at a value
   - transition fault: a cell fails to make transition from 0 to 1 or vice versa
   - coupling fault: write in one cell affects a neighboring cell
     - idempotent coupling: neighboring cell affected the same way
     - inversion coupling: neighboring cell affected the opposite way
   - pattern-sensitive fault: can't write a specific value when neighboring cells have a particular pattern

   Example 1:

4. March Test Terminology

   - : stepping through memory array and tests all cells by ensuring that both the addressing and cells can retrieve/store 0 and 1
   - $\Uparrow_0^{n-1}$ $(r0, w1)$: read, compare if 0, then write 1 to all cells in ascending order. Complexity of $2n$. Can take care of forward coupling faults
   - $\Downarrow_0^{n-1}$ $(r0, w1)$: descending order, takes care of backward coupling faults

5. Basic March Test Algorithm

   $\Uparrow_0^{n-1}$ $(w0)$: initialize memory to all 0s
   $\Uparrow_0^{n-1}$ $(r0, w1)$: for $i = 0$ *to* $n - 1$
   read(i)
   compare with expected value 0
   write (i,1)

6. Sample march tests:

   - MATS: $\Uparrow (w0)$; $\Uparrow (r0, w1)$; $\Uparrow (r1)$: covers forward coupling and stuck-at faults
   - MATS+: $\Uparrow (w0)$; $\Uparrow (r0, w1)$; $\Downarrow (r1, w0)$: covers also backward coupling

7. Testing for inversely-coupling faults

8. ATPG for march tests

   - For a given fault, generate the corresponding march test
   - Problem: march test can be complex and requires long application time
   - In general, three steps are needed:
     - initialization

- excitation
- effect propagation via verifying/reading necessary cell values

Example 2: testing coupling fault

9. Viewing coupling faults by FSMs

   - apply FSM testing approaches to test memories

10. Testing Word-Oriented Memories

   - Intra-word coupling faults
   - Many possibilities exist
   - m-out-of-n codes (Hamming distance between any two code words is m) $\longmapsto$ m = n/2

Example 3:

11. Testing pattern sensitive faults

    - basic idea: initialize the base cell, then change the patterns around the base cell to see if content of base cell changes
    - Issues: how to minimize the number of patterns applied

12. Graph traversal

    - let each node in graph denote a specific pattern
    - an edge exists between two nodes if the patterns differ in only 1 bit
    - Hamiltonian path/sequence traverses the graph such that each node is visited exactly once
      $\longmapsto$ each pattern applied exactly once
    - Eulerian path/sequence traverses the graph such that each edge is traversed exactly once
      $\longmapsto$ each pattern may be applied multiple times, but all single bit transitions in the patterns exercised

    Example 4:

13. Dynamic faults: faults that require multiple, successive operations to sensitize. (eg. write followed by an immediate read flips the bit value)

14. Testing System-On-A-Chip

    - System consists of interconnecting blocks/cores
      $\longmapsto$ benefits: design reuse, test reuse, etc.
    - Test access mechanisms:
      - global test bus to deliver test patterns to each core
      - wrapper around each embedded core for test access purposes
      - self-test using embedded programmable cores as TPGs (software-based testing)
    - Issues
      - area overhead
      - test application time
      - power consumption during test

    Example 5:

    Example 6:

15. Area Overhead

    - Does TAM need to be available for every core? In other words, can a given core be testable without TAM?
    - Need: testability analysis method to evaluate embedded core's testability
    - Alternative: transparency mode in the predecessor core to allow transport of test vectors
      $\longmapsto$ must make sure costs due to transparency is less than conventional TAM

    Example 7:

16. Test Resource Partitioning and Test Scheduling

    - TAM bus width, wrapper, power, BIST, etc. are limited resources
    - even if power is not exceeded, it may not be possible to test two cores in parallel due to bus-width limitation
    - derive a schedule by which cores are to be tested
    - test cores in parallel to reduce test application time
    - must monitor power consumption
    - Need: constrained scheduler/optimizer

17. Test data compression

    - test data for all cores on an SOC can potentially be very large
    - unlike compaction, compression is to reduce the overall test data volume by compressing the data

- such as gzip, compress, etc.
- problems: gzip and compress generally don't work well on vectors
- need: other compression methods
- need: low-cost on-chip decompressor

18. Compression by Colomb code

    - based on (1) difference vector set and (2) run-lengths

    Example 8:

VLSI / SOC Testing

Lecture 23

1. Where defects are located
   - random defects are controlled by process integrity
   - systematic failures/defects due to poor design, layout need to be identified to improve future yield
2. What causes systematic failure
   - aggressive design/layout styles
   - high-traffic lines too narrow
   - high-traffic lines too close
   - hard-to-fab regions, where density of material changes drastically in a short distance

   Example 1:

3. Inductive failure analysis
   - given layout, identify nets/signals that are susceptible to bridges, opens, etc.
   - statistical assumption: defects caused by particles of varying sizes of known probability of occurrence
     $\longmapsto$ given weighted critical area for bridges, one can compute the likelihood of a bridge for every net.

   Example 2:

4. Material density

   - a virtual grid is placed on the design/layout, where the dimensions of the grid are larger than the feature size
   - density is defined as the fraction of the grid covered
   - density gradient: change in density from one grid to another
   - observation: if density gradient are small for a given grid in all directions (within ϵ), then this grid is on a relatively uniform region
   - conversely, if density gradients change abruptly, it is generally harder to control the fabrication of such regions

   Example 3:

5. Design for yield improvement

   - identify potential places where fabrication may be difficult
   - sweep through the virtual grid plane, one layer at a time, to identify high-risk regions
   - can weigh each grid, giving more weight to nets with heavier traffic load

6. Yield prediction

   - prior to routing, estimate statistically channel density by approximating number of wires within channel
   - estimate density gradient based on approximate routing

7. Diagnosing a failure
   - objective: find the source of defect that caused the failure
     $\longmapsto$ may result in a number of possible sources/locations
   - static approach: based on signatures of potential modeled faults stored in a dictionary
   - dynamic approach: incrementally simulate and analyze the circuit to identify failure site
8. Static, dictionary-based diagnosis
   - idea: record signatures of every fault in a dictionary
   - Don't need to store complete responses for all primary outputs, just record the responses of erroneous outputs
   - using the dictionary, inductively search for the potential defect
   - Issues:
     - dictionary built by fault simulation without fault dropping, thus the cost could be high
     - dictionary may be large for large circuits
     - dictionary built for given modeled faults, which may not fully represent the actual defect, thus the response of actual defect may not match exactly with those obtained by fault simulation

   Example 4:

Example 5:

9. Adaptive diagnosis
   - pick the best vector to apply based on results obtained with each step

   Example 6:

10. Because defect response may not exactly match the simulated faulty response
    - score each fault correspondingly
    - find the fault that most closely resembles the defect's response
11. Distinguishability of faults
    - For a given test set T, the faulty response for two non-equivalent faults may be identical $\longmapsto T$ cannot distinguish these 2 faults
    - Can add more vectors to distinguish all distinguishable fault pairs
12. Diagnostic test generation
    - objective: enrich the test set so that more faults can be distinguished
    - modeled as a constrained ATPG problem on a pair of faults: search for a vector that detects one but not the other
13. Issues in diagnosis
    - Diagnostic accuracy: was the actual defect included in the final list of candidate sites
    - Diagnostic speed: time taken to find the candidate fault sites
    - Diagnostic resolution: how many fault sites did the diagnosis report
      - high resolution:
      - low resolution:
14. Simulation-based diagnosis
    - idea: gradually filter the nets (not faults) that the defect may be linked to, based on circuit structure and simulation results
    - helpful guides: cone intersection, sensitization, back-propagation, etc.
15. Diagnosis by cone intersection
    - step 1: divide the outputs into correct outputs and faulty outputs
    - step 2: compute fanin cones starting from the faulty outputs
    - step 3: compute intersection of the faulty output cones
    - Key: any gate/signal not included in the intersection cannot be solely responsible for the defect

Example 7:

16. Diagnosis by sensitization

    - idea: if gate a is responsible for the failure, then there must exist a sensitizable path from a to some faulty output(s)
    - note: there may result in multiple candidate gates, and there might have existed multiple defects in circuit
    - Algorithm:

    for each erroneous vector v
        logic simulate with v
        for each signal a in circuit
            complement the value at a
            simulate the fanout cone of a due to this complementation
            if one or more erroneous output value is flipped, then v can
                sensitize a discrepancy from a

Example 8

17. Diagnosis by back propagation

    - similar to critical path tracing, it finds candidate signals by backtracing from erroneous outputs
    - backtracing is done by traversing through sensitizable paths

    Example 9

18. Diagnosis for sequential circuits

    - complexity an order of magnitude higher
    - dictionary can still be built
    - fault can propagate through a number of time-frames before it reaches a PO

19. Diagnosis for BIST structures

    - only one signature for applying 100,000+ BIST patterns
    - increasing resolution is essential

VLSI / SOC Testing

Lecture 24

1. Diagnostic test generation

   - Given a fault pair, generate a test that can detect one but not the other
   - Define: two faults α and β are distinguishable if $\exists$ a test t such that the output of fault $\alpha \neq$ the output of fault β by test t
   - Indistinguishability can be defined conversely. If two faults are indistinguishable, they are also functionally equivalent
   - To improve diagnostic test generation, it would be nice to determine if 2 faults are distinguishable quickly in advance

2. Functional equivalence of two faults

   - Recall that a dominator gate of gate g is a gate through which all paths from g to any PO must pass
   - A *common dominator gate* for gates g_1 and g_2 is one that both pass
   - Common dominator cone: starting from the common dominator gate and backtrace in the circuit, including g_1 and g_2, together with all gates that are sufficient to completely determine the functions of the common dominator gate

   Example 1:

3. Properties of dominator cones

   - If logic functions at the common dominator gate for faults α and β are identical when expressed in terms of the inputs of the common dominator cone, then faults α and β are functionally equivalent

- Even if the logic functions expressed at the inputs of cone are not identical, α may still be functionally equivalent to β if the inputs at the cone that distinguishes the faults cannot be justified $\longmapsto$ if α and β are different for tests $t_1,\ t_2, ...,\ t_m$ at the cone inputs, and none of $t_1,\ t_2, ...,\ t_m$ is justifiable from the PIs, then α and β are functionally equivalent
- Note that the PIs responsible for propagating the fault-effect from common dominator gate to a PO are not included in the dominator cone, since they are not needed to define the common dominator gate

4. Use of redundancy information

   - if faults α and β produce same fault-effect at the common dominator gate output for a given test t, and fault β is known to be redundant, then test t must not be justifiable at the PIs of the circuit

   Example 2:

5. Distinguishability of faults in sequential circuits

   - a fault α in sequential circuit is present in every time-frame in the ILA model of the circuit
     $\longmapsto$ denote this fault α_k
   - two faults α and β are indistinguishable if α_k and β_k are indistinguishable for any starting state of the ILA $\longmapsto$ if two faults are indistinguishable for $k = 1$, then they are combinationally equivalent

   Example 3:

6. What if the starting state for the ILA is illegal/unreachable?

   - Only need to consider valid states for circuits C_α and C_β
     $\longmapsto$ valid states = set of all reachable states
     $\longmapsto$ valid states for C_α may not be the same for C_β
   - If either circuit is unsynchronizable, we can consider a subset of states
     $\longmapsto$ this subset may contain some unreachable states
   - Define: $\text{RS}(\alpha,\ m)$ = set of states reachable when fault α is present within m cycles. $\text{RS}(\alpha,\ 0)$ = all possible states
     $\longmapsto \text{RS}(\alpha,\ i+1) \subseteq \text{RS}(\alpha,\ i)$

   Example 4:

7. Compaction of Fault Dictionaries

    - Given a circuit with f faults, o POs, and v vectors, a naive construction of the matrix-like fault dictionary would involve $f \times v \times o$ entries
    - Conventional compaction by avoiding storage of all faults or all PO values can result in loss of information
    - Is there a way to compact the dictionary without loss of info?

8. Compaction without loss of info is possible since:

    - the number of distinct fault effects generally less than 2°
      $\longmapsto$ don't need to store all PO values in each entry, rather, store a pointer to which of the n distinct fault-effect it is
      $\longmapsto$ if $n < o$, then the savings simply by this method would be $\frac{2^o}{2^n}$
    - Further, since a distinct fault effect may be shared by many faults at various test vector positions, they can all point to the same distinct fault effect
      $\longmapsto$ more savings here

    Example 5:

9. Diagnosing Transistor Stuck-open Faults

   - Do we want to build another dictionary (or other methods) for stuck-open faults, or can we use SSF techniques?
   - Want: diagnose stuck-open faults with known stuck-at diagnosis techniques
   - Review: stuck-open fault detected by a 2-vector pair.

   Example 6:

10. Diagnosis approach

    - After identifying the failing chips, first diagnose assuming the failure due to a stuck-at defect
    - Then, based on the diagnostic info on SSF, deduce which stuck-open faults could cause this
    - Need: simply build a table to match behavior

    Example 7:

11. A defect may not match any fault model *exactly*

    - Can we come up with a technique that captures the possible locations of of the defect without any given fault model?
    - Motivation: if a defect is active for test vector t, it must affect at least one signal in its vicinity. And the affected signal must have a propagation path to a PO.

    Example 8:

12. Region-based diagnosis

    - Any defect within the region must propagate a FE to at least one output of region for the detecting vector
    - Number of regions in the order of number of gates: each gate can be the center node for a region
    - Don't enumerate all possible fault-effects at the region outputs, since there can be many
      $\longmapsto$ Simply inject don't-cares (X) at the region outputs to rule out false candidate regions
    - Can perform diagnosis hierarchically, starting from large regions down to small regions

    Example 9:

13. For candidate regions where the defect may reside, focus on gates within these regions
    - May enumerate all fault-effects if number of region outputs few

    Example 10: