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 $\rightarrow$ 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
     $\longmapsto$ remedy: observability-enhanced statement coverage: make sure that the effect of exercising statement propagates to a PO

   Example 8: