Design Verification

Lecture 20 - Simulation-Based Verification II

1. For designs described at higher levels of abstraction (VHDL, behavioral, RTL, C, C++, etc.)
   - we would like to generate a set of smart input patterns based on it
   - similar to software-testing approaches
2. Textual-based
   - statement coverage: exercise of each statement in the code
   - branch coverage: exercise of each branch in the code
   - observation-enhanced coverage
   - Key: obtain high coverage

   Example 1

3. Mutation-Based
   - mutant: a version of code with single design error, such as variable substitution, arithmetic operator substitution, etc.
   - Key: generate input patterns that *kill* all mutants

   Example 2

4. Flow-Based

   - Control-data flow graph construction needed for a given design
   - branch coverage
   - path coverage: exercise of each path in the code
     $\longmapsto$ number of paths can be exponential
   - dataflow coverage: based on define-use paths

   Example 3

   Example 4

5. Domain-Based
   - domain: an input subspace such that a path is followed
   - Key: at domain boundaries, we're more likely to cause an error (corner cases)

   Example 5

6. Tag-Based to enhance observability
   - tag: placed at a location to indicate possibility of incorrect value
   - software observability via inspection at memory contents; hardware offers less observability
   - simulation approach similar to the 2-phase logic fault-simulation, on the corresponding data-flow graph
     $\longmapsto$ phase 1: error-free simulation and possible modification of code to enable propagation of tags in phase 2
     $\longmapsto$ phase 2: inject tags and propagate them
   - tag propagation: evaluate on the corresponding data-flow graph for code
   - coverage = % tags observed for a given test set

   Example 6

7. Key issues
   - how to simulate symbolic tags efficiently on a given vector set
   - propagation of tags dependent on data values of other variables
   - when tag is on a conditional variable, direction of branch may change
   - high quality test set implies high tag propagation coverage

8. Phase 1: Graph representation and code modification: Flow Graph G(V, E, L) consists of
   - vertices $v \in V$ correspond to variables in HDL. Include an additional sink node O to indicate *output*
   - a directed edge $e \in E$ between (v_1, v_2) implies data dependence between the 2 vertices
   - label on each edge $l(e) \in L$ indicate one or more of the following:
     - statement/line number with which the dependence is associated
     - conditional expression that the dependence relies on
     - array indices
     - multiplier (positive or negative)

9. On a tag-free simulation, we can identify which edges are active (inactive), thus determining the flow
   $\longmapsto$ this will help us later perform tag propagation

   Example 6 (graph construction)

Example 7

10. Phase 1 continued: modification of code.
    - add new variables and extract statements out of conditional blocks
    - This is necessary for propagation of tags

Example 8: simple conditional

Example 9: nested conditionals

Example 10: loop

Example 11: loop and conditional

11. Phase 2: graph simulation: propagation of tags throughout the graph

    - step 1: tag-free simulate on a given vector. determine active and inactive edges in graph. An active edge indicates that a tag can propagate from the predecessor node to the successor node joined by the active edge
    - step 2: temporarily remove all inactive edges
    - step 3: inject a positive tag at output node O
    - step 4: backtrace from O in reverse order of simulation trace, determine all $v \in V$ that are reachable from O with corresponding computed signed tags
    - step 5: the line numbers corresponding to reachable edges are observable with the noted signed tag
    - step 6: repeat from step 3 using a negative tag at O

    Example 12