Design Verification

Lecture 22 - Architectural Verification

1. Goal: verification of the microarchitecture and instruction set architecture (ISA)
   - Control-flow errors:
     $\longmapsto$ program counter not updated correctly
     $\longmapsto$ delayed branch slot not executed correctly
   - Data-flow errors:
     $\longmapsto$ read-after-write (RAW), WAR, WAW errors
     $\longmapsto$ essentially, failure in interlock logic, forwarding/bypass logic, etc.
2. Simple approach:
   - control-flow: generate simple programs with branches
   - data-flow: generate simple programs with data hazards involving all possible def-use register combinations
   - problem: simulation-based. How can we guarantee 100% correctness?

Example 1

3. Another approach: model/view the design as interacting finite-state-machines. Failure to detect data hazard means FSM_1 and FSM_2 are in an illegal combination of states

   - let FSM_1 be the FSM for register fetch/renaming logic
   - let FSM_2 be the FSM for reservation station
   - let FSM_3 be the FSM for forwarding logic
   - a RAW hazard means **should not** fetch from register file immediately $\longmapsto$ must (1) detect hazard, (2) forward value via bypass logic when value is ready, (3) may need to insert bubbles into pipeline
   - RAW hazard detection failure:
     FSM_1: instruction fetches reg_i from register file and FSM_2: reservation station says reg_i waiting for result from function unit
   - Forwarding logic failure:
     FSM_1: instruction fetches reg_i from register file and FSM_3: forwarding logic ready to send value (corresponding to reg_i) from function unit
   - Can use ATPG or model checking to check for such illegal combinations

   Example 2

4. Some definitions
   - a RAW hazard interlock failure is defined as the following: instruction i is currently executing, will produce result r; instruction j is issued and expects value r; instruction j incorrectly reads in the stale value of r from register file.
   - similarly, a RAW forwarding failure occurs when the hazard is detected, but value of r is not forwarded
   - a RAW hazard exists when the ID FSM attempts to read r_k and the reservation station FSM has an entry whose state indicates the dest reg is r_k
5. A hazard detection failure occurs when both of the following are true:
   - producing instruction i has not yet produced result for r_k
   - consuming instruction j has been issued to reservation station and the state it enters indicates that the src operand with r_k is ready
6. A forwarding failure occurs when any of the following is true:
   - reservation station FSM never transitions out of the state where src operand r_k is not ready
   - a Functional Unit FSM just transitioned to a state where r_k is produced, and reservation station FSM currently at a state waiting for operand r_k, and does not transition to another state

7. Verification using self-consistency and symbolic simulation

   - difficulty in verification usually lies in verification of the complex modules that enhance performance such hazard detection, bypassing, etc.
   - thus, key to verify the system is by comparing system with and without performance enhancements (view unehanced system as *reference* system)
   - Since we do not have a system *without* performance enhancers, we **can** transform the input such that the enhancement modules become dormant
     $\longmapsto$ check if $S_{enhanced}(I) = S_{unenhanced}(\sigma(I))$, where I is the input used to test system
     $\longmapsto$ What should $\sigma()$ be?
   - If $S_{enhanced}(I) = S_{unenhanced}(\sigma(I))$, then we know the enhancement module works

   Exmaple 3:

8. In order to make sure all cases are held, we must enumerate all possible starting states and inputs
   $\longmapsto$ symbolic simulation can be helpful here
   $\longmapsto$ one-pass symbolic simulation and compare the output BDDs for enhanced and unenhanced systems

9. From ATPG perspective, two models possible:
   - iterative logic array (ILA): in each time frame, multiple instructions are concurrently being executed in a given pipeline (but in different pipeline stages). Decision points may be large for pipelined microprocessors
   - pipeframe model: an orthogonal view to ILA; pipeframes interact with one another via the controller.

10. Test generation algorithm:
    - step 1: path selection: select the paths in each pipeframe for the targeted error
    - step 2: value selection: place specific values onto the primary inputs for the paths selected
    - step 3: justification of control signals that can merge the pipeframes together
    - Note: can treat each pipeframe as an expanded combinational circuit

11. Design for Verifiability Perspectives

- Design using verified RTL modules
  - FSMs and simulations of each module readily available
  - we now only need to verify interacting components, making sure their transactions are observed
- Observer and Checker Modules
  - embed into the design an observer module that monitors the circuit's behavior
    $\longmapsto$ capture the verification objectives $\longmapsto$ annotates transactions that were monitored
  - also embed a checker module that checks the annotated outputs produced by the monitor
  - if verification done by simulation: simply allow the checker to monitor the observer output to see if any objectives violated
  - if verification done formally: verify properties of the observer and checker, where the properties are those the observer was built to monitor