COMSM0115 Design Verification:

Assertions and Assertion-Based Verification

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL
Department of COMPUTER SCIENCE

Last Time

- Checking
  - Monitors
  - Scoreboards
  - Reference Models

Assertions

Assertions

- An **assertion** is an if statement with an error condition that indicates that the condition in the if statement is false.
- Assertions have been used in SW design for a long time.
  - `assert()` function part of C `#include <assert.h>`
  - Used to detect `NULL` pointers, out-of-range data, ensure loop invariants, etc.
- Revolution through Foster & Bening's OVL for Verilog.
  - Clever way of encoding re-usable assertion library in Verilog. ☺
- Assertions have become very popular for Design Verification in recent years.
  - **Assertion-Based Verification** (also Assertion-Based Design).

Software Assertions

- Check that condition evaluates to TRUE exactly at time when `assert` statement is executed.
- Essentially a "zero-time test".
- Not sufficient for HW assertions!

HW Assertions

HW assertions:

- combinatorial (i.e. "zero-time") **conditions** that ensure functional correctness
  - must be valid at all times
    - "This buffer never overflows."
    - "This register always holds a single-digit value."
    - "The state machine is one hot."
    - "There are no x's on the bus when the data is valid."

and

- **temporal conditions**
  - to verify sequential functional behaviour over a period of time
    - "The grant signal must be asserted for a single clock cycle."
    - "A request must always be followed by a grant or an abort within 5 clock cycles."
  - Temporal assertion specification language facilitate specification.
    - System Verilog Assertions
    - PSL/Sugar

Classes of Assertions

Implementation assertions:

- **Specified by the designer.**
- encode designer's assumptions e.g. on interfaces
- state conditions of design misuse or design faults
  - detect buffer over/under flow
  - signal read & write at the same time

- Implementation assertions can detect discrepancies between design assumptions and implementation.
- But implementation assertions won't detect discrepancies between functional intent and design!

(Remember: Verification Independence!)

Classes of Assertions

Specification assertions:

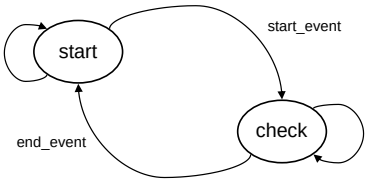
- **Specified by verification engineer.**
- encode expectations of the design based on understanding of functional intent
- provide a "functional error detection" mechanism
- supplement error detection performed by self-checking testbenches

- Often high-level white-box properties.
- Instead of using (implementing) a monitor and checker, in some cases writing a block-level assertion can be much simpler.

Simulation Assertions

- First introduced via OVL in Foster and Bening's book "Principles of Verifiable RTL Design".
  - library of predefined Verilog **Assertion Monitor** modules
    - `assert_always, assert_eventually, assert_never,`
    - `assert_even_parity, assert_no_overflow`, ...
- See `http://verificationlib.org` for free download.
- **Assertion Monitors** are instances of modules whose purpose is to verify that a certain condition holds true.
- Composed of condition, message and severity level.
  - condition is the (static/temporal) property to be verified.
- Demonstrates a clever and creative way of using Verilog to specify temporal expressions.

Sequential Assertion Monitor



- The start event initiates assertion validation process.
- Evaluation continues until the end event occurs.
  - Distinguish between time-bounded and event-bounded monitors.

Examples - I

- The traffic light on both sides of the crossing should never be green at the same time.

```
assert_never both_lights_are_green (clk, reset_n,
(major_rd_light=='GREEN && minor_rd_light=='GREEN));
```

- The minor road timer value should be less than the major road timer value.

```
assert_always minor_less_than_major(clk, reset_n,
(minor_timer_value < major_timer_value));
```


Examples - II

- If there are no more cars on the minor road and the traffic light is green for the minor road, then the traffic light should switch to yellow.
- I.e. the controller should always maximize the green time for the major road.

```
assert_time #(0,1) change_from_green_if_no_car
(clk, reset_n,
(minor_rd_light=='GREEN && !car_present),
    (minor_rd_light='YELLOW));
```

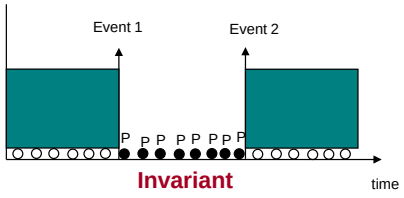
- start event: Event that triggers monitoring of the test expr.
- test expr: Expression to be verified at the positive edge of the clock.
- The test expr must evaluate to TRUE for 1 clock cycle after start event is asserted.

Terminology

- **Event:**
  - Boolean expression which evaluates to TRUE.
- **Assertion:**
  - **Claim** about an event or sequence of events.
- **Property:**
  - **Expected behaviour**, something verified.
- **Constraint:**
  - Bounded behaviour, legal stimulus.
- **Static (Invariant):**
  - Event TRUE for all time.
- **Temporal (Liveness):**
  - A time relationship of events, whose correct sequence must be true.

[Credits: Foster. OVL. Hewlett-Packard.]

Invariant



- Assertion P is checked after Event 1 occurs, and continues to be checked until Event 2.

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

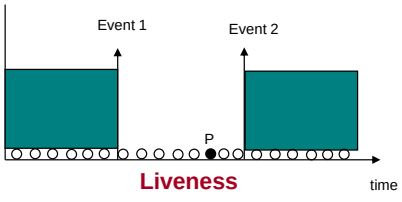
Property Types: Safety

- Safety: Nothing bad ever happens
  - FIFO never overflows
  - The system never allows more than one process to use a shared device simultaneously
  - Requests are always answered within 5 cycles
- These properties can be falsified by a finite simulation run.

Property Types: Liveness

- Liveness: Something good will eventually happen
  - The system eventually terminates
  - Every request is eventually answered
- In theory, liveness properties can only be falsified by an infinite run.
  - Practically, we can assume that the "graceful end-of-test" represents infinite time
    - If the good thing did not happen after this period, we assume that it will never happen, and thus the property is falsified

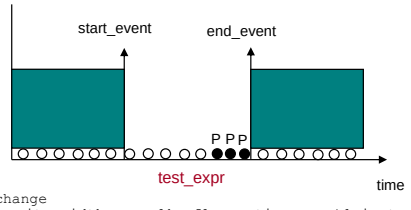
Liveness



- Assertion P must **eventually** be valid after the first event trigger occurs and before the second event trigger occurs.

[Credits: Bening & Foster. Principles of Verifiable RTL Design. Kluwer 2001.]

Example: `assert_change` (time bounded)




```
assert_change
   [#(severity,width,num_clks,flag,options,msg)] inst_name
   (clk,reset_n,start_event,test_expr)
```

- Use **`assert_change`** in circuits to ensure that after a specified initial event a particular variable or expression will change.
  - After a request an acknowledge will occur within a specified number of clock cycles.
  - Verification of state changes in an FSM

assert_change

- It is not possible for a car on the minor road to wait forever.

```
assert_change #(0,2,32) car_should_not_wait_forever
(clk, reset_n,
(major_rd_timer && major_rd_light=='GREEN &&
car_present),
major_rd_light);
```

- After a specified initial event, a particular variable/expression will change.
  - start event: Event that triggers monitoring of the test expr.
  - test expr: Expression to be verified at the positive edge of the clock.
  - In this example, timeout is set for 32 clocks, i.e. test expr has 32 clock cycles to change its value before an error is triggered after start event is asserted.
  - Maximum length of minor road red light is 32 clock cycles.

How Assertions work in Simulation

Observability Problem:

- If design assumption is violated during simulation, then design fails to operate according to original intent.

BUT:

- Symptoms of low-level bugs are often not easy to observe/detect.
  - Activating a faulty statement does not mean the bug will propagate to an observable output.

ABV: During simulation a design's **assertion monitor** activates.

- **The assertion** immediately fires when it is violated and in the area of the design where it occurs.
- Debugging and fixing an assertion failure is much more efficient than tracing back the cause of a corrupted packet.

Observability can be increased using Assertion Based Verification & Coverage.

Simulation vs Formal Assertion Checking

- ... or dynamic (i.e. sim-based) verification vs static (i.e. formal) verification.
- Remember, sim can only show presence of bugs, never prove their absence!
- An assertion has never fired - what does this mean?
  - Does not necessarily mean that it can never be violated!
  - Unless sim is exhaustive..., which in practice it never will be.
  - It might not have fired because it was never evaluated.
- **Assertion coverage:** Measures how often an assertion condition has been evaluated.

Static Formal Verification (SFV)

- Exhaustive formal analysis often from a reset state of the design.
- In practice typically based on techniques such as model checking or symbolic simulation.
- Proven assertions can be guaranteed to be true under any scenario.
- Capacity limits: Fails for large designs or complex assertions.
- SFV primarily provides proofs.

Combining FV with Simulation - 1

Semi-formal Verification (Semi-FV)

- Guided simulation: Use formal analysis to produce input vectors for sim.
- Goal of formal analysis:
  - Guide sim to reach coverage holes.
  - Formal analysis chooses sim vectors.
- Not exhaustive.
- But remember:
  - Coverage metrics are only loosely related to bugs.
  - Guided sim might improve coverage but might not find bugs.

Combining FV with Simulation - 2

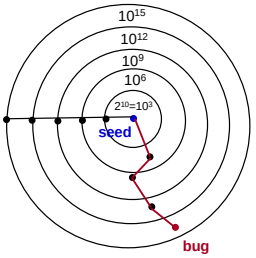
Dynamic Formal Verification (DFV)

- Simulation requires stimulus generation (test vectors).
- FV can use these test vectors as a starting point.
  - Rather than starting from the reset state.
- Method:
  - Capture internal status (seed) generated by driving test vector.
  - Try to find violations to assertions starting from each seed.
  - Perform exhaustive formal analysis for bounded number of cycles.
    - Based on bounded model checking (BMC) algorithm.
  - Provide counter example for any violations.
- **DFV bridges gap between formal verification and simulation.**
  - DFV is focused on finding counter examples (i.e. bugs).
  - DFV finds new behaviours that were not covered by simulation, and that cannot be reached by static FV.

Assertion-based Coverage Metrics

- **Assertion Coverage** (introduced earlier)
- (Assertion Density:)
  - Measures number of assertions of each type in each module.
- **Proof Radius:**
  - Measures the amount of verification (in cycles) that was achieved by formal analysis.
  - The larger the proof radius, the more thorough the verification.
  - Proof Radius of 200 cycles:
    - Means FV has exhaustively proven that no bugs can be triggered within 200 cycles of the initial state (of the analysis).
  - **In DFV:**
    - Measure of depth of the exhaustive search around each seed.
  - **Objective of DFV:** Analyze as many seeds as possible at shallow depth.
  - Hence, measure seeds per second per assertion at a small fixed proof radius (e.g. 5 cycles).

Dynamic Formal Verification



- Assumptions:
  - 10 inputs affect the assertion
  - 5-cycle proof radius

Deep Dynamic Formal Verification

- Advanced technique for finding counter examples.
  - Start from sim seed (like DFV).
  - Exhaustive search at a **large proof radius** until it either finds a counter example or reaches a user-defined time boundary.
  - (Proof radius 50..200 cycles.)
  - Based on Bounded Model Checking algorithm.
    - BMC is NP-complete.
    - All known algorithms are exponential in proof radius.
    - DDFV exploits optimizations specific to ABV and RTL design.
- Objective of DDFV:
  - Analyze a few seeds to the greatest depth possible.
  - Hence, measure proof radius achieved within large fixed time period (e.g. 1,000 seconds per assertion).

Summary: Benefits of using Assertions

- Capture and validate design assumptions and constraints.
- Can monitor/test internal points of design - **increase observability**.
- Simplify detection and diagnosis of bugs - occurrence of bug is constrained to assertion being checked.
  - Can reduce simulation debug time by as much as 50%.
  - Find more bugs faster!
- Properties can be (re-)used for both simulation-based and formal/semi-formal verification.

Assertion-Based Verification

ABV

Assertion-based Verification (ABV)

- Verification methodology that is heavily based on assertions
- Aspects of the spec, high-level design, and implementation intent are represented as a set of properties
- Assertions are placed in the DUV code to check that these properties hold

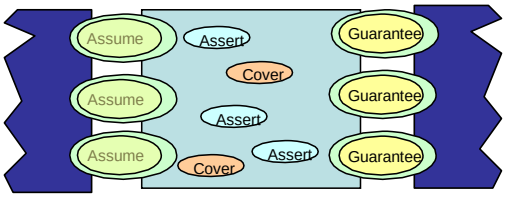
```
assert (not (a_active and b_active))
  report "Two units are active"
  severity error;
```

Property – a_active and b_active are not active together

Possible Use for Properties

- Assertions – claim about the behaviour of the DUV
- Assumptions on the behavior of the environment (neighboring units)
- Guarantees about the DUV behavior
- Restrictions – generic restriction on behaviors
  - For example, restriction on generated stimuli to focus on specific area or function in the DUV
- Coverage – properties that we want to see happen during the verification process

Using Properties for Verification



- Guaranteed behaviors should be a subset of assumed behaviors
  - Guarantee response within 5 cycles; assume response within 10 cycles is OK
  - Assume response within 5 cycles; guarantee response within 10 cycles is BAD

How to Specify Properties

- Plain English
  - The most natural way
  - May contain ambiguities and missing details
    - "Always A or B are true"
      - Can A and B be true together?
- HDL code
  - May be long and complex code
    - Difficult to write, debug, and maintain
    - OVL
- Dedicated property specification languages

Property Specification and Assertion Languages

- OVL (Open Verification Library)
  - Template library for assertions
  - Not really a language
  - Mostly simple parameterized assertions
- PSL (Property Specification Language)
  - Based on temporal logic
  - Originated in the property language Sugar developed by IBM
  - An IEEE standard
- SystemVerilog Assertions
  - Integrated part of the SystemVerilog language
  - Based on temporal language
  - Include DUV behaviour and coverage
- Temporal e
  - Integrated part of e (and Specman)
  - Based on temporal language

PSL, SystemVerilog Assertions and temporal e share many common concepts

ABV Methodology

- Use assertions as a method of documenting the exact intent of the specification, high-level design, and implementation
- Include assertions as part of the design review to ensure that the intent is correctly understood and implemented
- Write assertions when writing the RTL code
  - The benefits of adding assertions at later stage are much lower
- Assertions should be added whenever new functionality is added to the design to assert correctness
- Keep properties and sequences **simple**
  - Build complex assertions out of simple, short assertions/sequences

Write Assertions For

- Basic building blocks
  - Queues/FIFO's
    - For example, FIFO overflow and underflow conditions
  - State Machines
    - Invalid states and invalid transitions.
- Interfaces
  - These assertions can help defining the interface protocol, legal values and required sequences
- Internal functionality
  - Cache coherency/consistency policies
  - Mutual exclusion, absence of contentions
- End-to-end functionality

Use ABV with Caution

- It is easy to specify "how" properties with assertions
  - But it is much harder to specify "what" properties
  - ➔ Assertions should be accompanied by other checkers to check the data
- It is easy to write local properties with assertions
  - These properties do not capture the end-to-end behavior of the DUV
  - These properties are the first ones to break when the design changes
  - ➔ Make sure you also have end-to-end assertions or equivalent mechanisms to check end-to-end behavior
- Do not allow the designers to write all your assertions
  - Assertions are embedded in the DUV code, so it seems natural that the designers write them
  - But, designers have their own view of the design – local, with many details
  - ➔ Add your own assertions to the ones entered by the designer
    - If it is hard to embed them in the DUV code, use alternative methods
      - E.g. checkers in the verification environment, etc.