**Outline**

- Motivation
- Issues in checking
  - When to check
  - What to check
- Checking technologies
  - Reference models
  - Scoreboards
  - Rule-based checking
    - Assertions
  - Assertion-based verification (ABV)
  - Property specification languages
  - Property Specification Language (PSL)

---

**The Yin-Yang of Verification**

- Driving and checking are the yin and yang of verification
  - We cannot find bugs without creating the failing conditions
  - We cannot find bugs without detecting the incorrect behavior

---

**Ideal Checking**

- In theory – detect deviation from expected behavior as soon as it happens and where it happens
  - No need to worry about “disappearing errors”
  - Easy to debug – the checker points to the bug
- This is not easy (even if we ignore many practical aspects) because in many cases we understand that something bad happened only in retrospect
  - Several “good” behaviors collide to create a bad behavior
- And what about the bugs we are not looking for
**“Good” Behavior Collision**

- At cycle 1000 fdiv F1, F2, F3 is dispatched to the M unit
  - It reaches stage M2 at cycle 1001
  - Its execution time is 60 cycles
- At cycles 1023 fld F1,100(G2) is dispatched to the S unit
  - It reaches stage S2 at cycle 1024
  - The data returns from the cache at cycle 1060
- At cycle 1061 the fdiv is ready to write
  - It moves to stage M3
- At cycle 1061 the fld is ready to write
  - It moves to stage S3
- Both instruction write to the same register together

**“Good” Behavior Collision**

- There are many possible causes for the problem
  - The lfd instruction should have waited in M3 until fdiv finishes and only then write
    - S3 and M3 identify and avoid the collision
  - The lfd instruction should have waited in M2 until fdiv finishes and only then move to M3
    - S2 and M2 identify and avoid the collision
  - The dispatcher should not dispatch lfd until it is sure that lfd would finish after fdiv

**Practical Aspects**

- Ground rules
  - Only black-box checking is allowed
- The cost of implementation and maintenance
  - Vs the cost of debugging
- The cost of mistakes
  - Misdetection – we failed to detect a bug that was exposed by the stimuli
  - False alarm – we mistakenly flagged a good behavior as bad
- Which is more expensive?

**When to Check?**

- Checking can be done at various stages of the verification job
  - During simulation
    - On-the-fly checking
  - At the end of simulation
    - End-of-test checking
  - After the verification job finishes
    - External checking
- Checking at each stage has its own advantages and disadvantages
On-the-fly Checking

- Checking is done while the simulation is running
- The DUV is continuously monitored to detect erroneous behavior

Advantages
- Detection can be as close as possible (in time and space) to the bug source
- Do not require large traces and external tools to do the checking

Disadvantages
- May slow down simulation
- Checking is limited to allowed time and space complexity
- Need to plan the checking in advance
- To a new checker, we need to rerun simulation

End-of-test Checking

- Checking is done at the end of simulation
- The checker checks the state of internal and external resources and makes sure that they are correct

Disadvantages
- Provides limited checking capabilities
  - Static look at the state of resources at the end of the test
  - High probability of masking bugs by rewriting to the resources
  - Hard to detect performance bugs
    - Correct things are happening, but not at the right time
    - Hard to correlate symptoms to bugs
    - Hard to debug

Advantages
- Simpler than other forms
  - May not require as deep understanding of the DUV
  - Reduces probability of false alarms
  - Caused by disappearing bad effects
External Checking

- Monitors keep internal resources values and behaviors in trace files
- Checking is done by an external program that examines these files

External Checking

- External checking separates the checking from the simulation
  - We can perform any check we want without rerunning the simulation
    - As long as the data is in the trace files
  - We can perform more complicated checks
    - Use longer history, process events out-of-order
  - We can combine information coming from different sources
    - For example, different verification environments
  - In theory, external checking has all the powers of on-the-fly checking plus end-of-test checking plus more
  - Trace size and amount of traced facilities is a practical limitation

What to Check

- There are five main sources of checkers
  - The inputs and outputs of the design (specification)
  - The architecture of the design
  - The microarchitecture of the design
  - The implementation of the design
  - The context of the design
- Note that the source of checkers and their implementation are two different issues

Coarser Classification – The What And The How

- There are five main sources of checkers
  - The inputs and outputs of the design (specification)
  - The architecture of the design
  - The microarchitecture of the design
  - The implementation of the design
  - The context of the design
- Note that the source of checkers and their implementation are two different issues
Checking the What

- Check the final outcome of a behavior
  - Data oriented
    - But not limited to data
  - Usually based on higher level of abstraction
    - Checking is less tight
    - Requires less familiarity with the DUV
    - Less false alarms, more misdetections
  - Low correlation between failure and bugs
    - Harder for debugging
    - Can find “unexpected” bugs

Checking the How

- Check how things are done internally
  - Control oriented
  - Usually at lower levels of abstraction
  - More false alarms, less misdetections
  - Tighter relations between failure and bugs

Stimuli Generation and Checking

- In general, checking should be isolated from the stimuli generation
  - Modularity – ability to replace the stimuli generator
  - Reusability – ability to use the checkers at higher level of the design hierarchy
- Exceptions
  - Self-checking tests
  - Golden vectors
  - The stimuli generation can assist checking by improving observability
    - Help transfer events from dark corners to the spotlight
Scoreboards

- Scoreboards are smart data structures that keep track of events in the DUV during simulation
- Usually, scoreboards are global
  - One scoreboard per verification environment
- Scoreboard are not checking mechanisms, but
  - The main purpose of using scoreboards is for checking
  - In practice, many checkers are implemented inside scoreboards
  - There are many typical checks that are done with scoreboards

Scoreboard Operation

- Sources of information to the scoreboard
  - Primarily, the inputs and outputs of the DUV
  - Internal events can also be used
- Types of checks done with scoreboard
  - Matching between inputs and outputs
    - Nothing is lost
    - Input with no matching output
    - Nothing is born
    - Output with no matching inputs
  - Data matching
  - Timing rules
  - Delay from input to output is within limits
  - Ordering rules
- Scoreboards are very useful in data flow designs
  - Fragmenter and calc2, but not processors

Side Note – Graceful End-of-test

- Checking that nothing is lost is very important
- If an input does not have a matching output, how can we distinguish between two cases
  - The input is lost or hopelessly stuck in the DUV
  - The DUV did not have enough time to handle the input
- Possible solution – Start a timer when a new input enters the DUV
  - If the timer expires, that input is lost or stuck
  - But, what if cannot bound the delay?
- Alternative (or complementary) solution – stop the inputs before the end of the test and let the design clean itself
  - Because there are no new inputs, things that are stuck inside have a chance to get free
Reference Models

- Reference model is an oracle that tells how the DUV should behave
  - Usually in the form of an alternative implementation
- It runs in parallel to the DUV, using the same inputs and provide the checking mechanisms with information about the expected behavior
  - Checking is done by comparing the expected behavior to the actual one
- Pure reference models can run independently of the DUV
  - But not all reference models are pure (example later)

Reference Model Operation

Reference Models

- Reference models have many uses
  - Checking
  - Aids for stimuli generation
  - “Smart” BFM
  - Vehicles for SW development
- What can we check with a reference model
  - In principal, anything
  - Depends on the level of details and accuracy of the reference model
    - And how much of its behavior we are willing to expose

Levels of Abstraction

- The level of abstraction in reference model dictates the type of information we can get out of it for checking
  - Functional accurate model can be used only to check correctness of data, usually at the end of the test or well defined points in time
    - Timing, order, and other checks need other means
  - Cycle accurate model can be used for checking all aspects of I/O behavior
  - Cycle accurate and latch accurate model can be used also for checking the internal state of the DUV
    - The book calls this type of model deep function reference model
Impure Reference Model

- Sometimes it is impossible (or very hard) for the reference model to duplicate significant decisions made by the DUV
- Possible solution: Use information from the DUV to assist the reference model

Rule-based Checking

- Checks that a set of rules holds in the DUV
- Essentially, all checking is rule-based
- Possible solution: Use information from the DUV to assist the reference model

Rule-based Checking

- Checks that a set of rules holds in the DUV
- Essentially, all checking is rule-based
- Rule-based checking usually refers to the last case

Rule-based Checking

- Checks that a set of rules holds in the DUV
- Essentially, all checking is rule-based
- If (not something) then error
- Something can be
  - Value of a register matches value in reference model
  - Data in a packet at the DUV output matches data in the input as stored in the scoreboard
  - response_out == 0 → data_out == 0
- Rule-based checking usually refers to the last case

Rule-based Checking

- Rules can come from many sources
  - All levels of the design process
    - Spec, high-level design, implementation
    - Behavior of neighboring units
  - Rules checking can be implemented in many places
    - External checking tools
    - Various places in the verification environment
    - Interface monitors
    - Scoreboards
    - End-of-test checkers
    - In the DUV itself
  - Rule-based checking that is embedded in the DUV code is called assertions

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 set of properties
- Assertions are place in the DUV code to check that these properties hold

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 set of properties
- Assertions are place 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
What Does The Property Mean

- Inside a sequential process the property is checked only when the process is active
  - \( A_{\text{active}} \) and \( b_{\text{active}} \) are never set together when \( \text{CLK} \) raises and \( \text{SINGLE\_MODE} = '1' \)

- Outside a sequential process the property is always checked
  - \( A_{\text{active}} \) and \( b_{\text{active}} \) are never set together

Possible Use for Properties

- Assertions – properties that we want to make sure are not violated
- 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

Property Types: Safety

- Safety: Nothing bad ever happens
  - \( \text{FIFO} \) never overflows
  - The system never allows more than one process to use a shared device simultaneously
  - Requests are always answered within 5 cycles

- This property 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

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

- **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

- **Temporal e**
  - Integrated part of e (and Specman)
  - Based on temporal language

- PSL, SystemVerilog Assertion 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 Cautious

- 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 seem 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 one entered by the designer
    - If it is hard to embed them in the DUV code, use alternative methods
      - Vunits, checkers in the verification environment, etc.