COMS30026 Design Verification

Checking

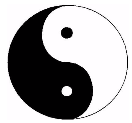
Kerstin Eder

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

University of BRISTOL

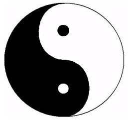
Department of COMPUTER SCIENCE

Checking: Outline

- Motivation 
- Issues in checking
  - When to check
  - What to check
- Checking technologies
  - Reference models
  - Scoreboards
  - Rule-based checking

https://clipartix.com/detective-clipart/

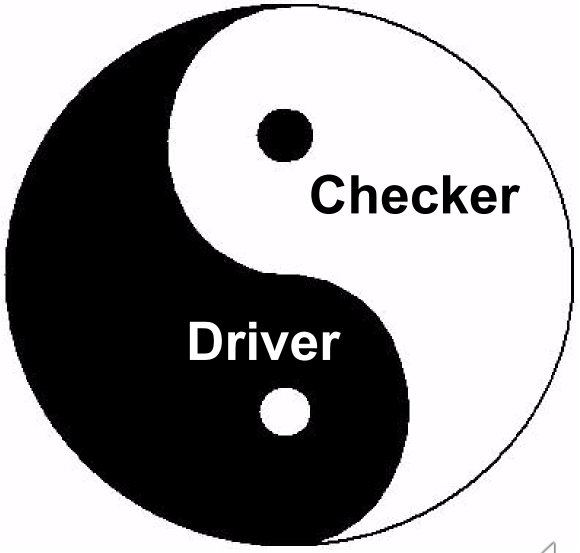
Checking: Outline

- Motivation 
- Issues in checking
  - When to check
  - What to check
- Checking technologies
  - Reference models
  - Scoreboards
  - Rule-based checking
  - Assertion-based verification (ABV) (later)

https://clipartix.com/detective-clipart/

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.**



The Importance of Driving and **Checking**

- Drivers activate the bug.
- The observable effects of the bug then need to propagate to a checker.
- A **checker** needs to be in place to detect the incorrect behaviour.

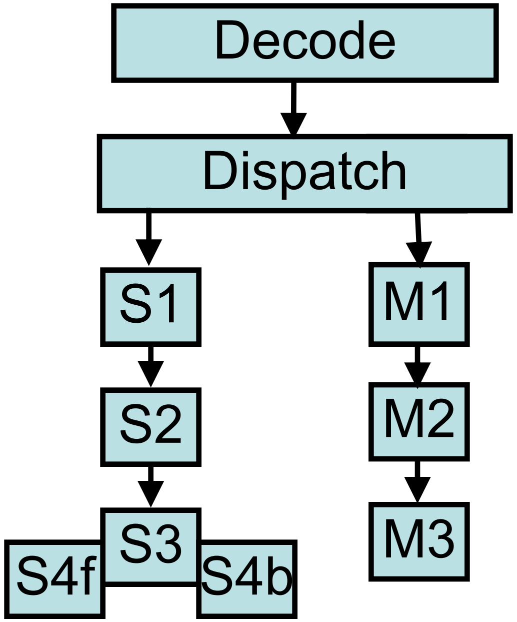
All three are needed to find bugs!

Ideal Checking

- In theory, we wish to detect deviations from the expected behavior as soon as these happen and, ideally, where they happen
  - Easy to debug: the checker points to the bug
  - No need to worry about “disappearing errors”
- In reality, this is not as 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

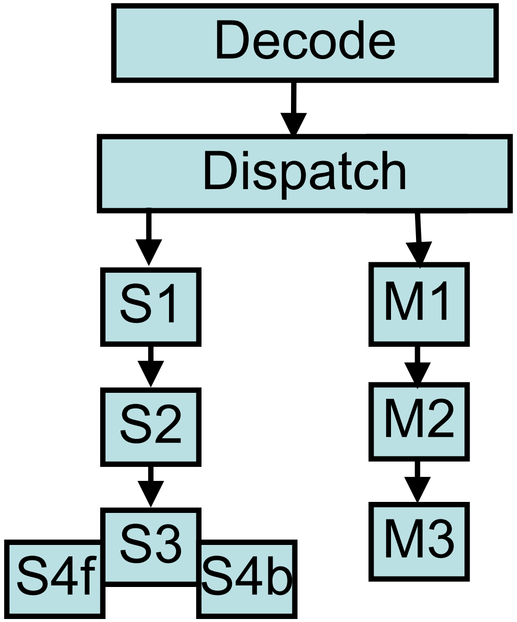
And, what about the bugs we are not looking for?

“Good” Behavior Collision



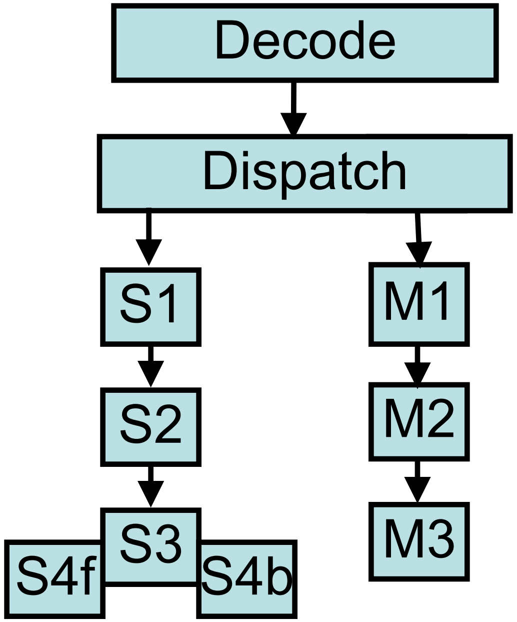
“Good” Behavior Collision

- 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



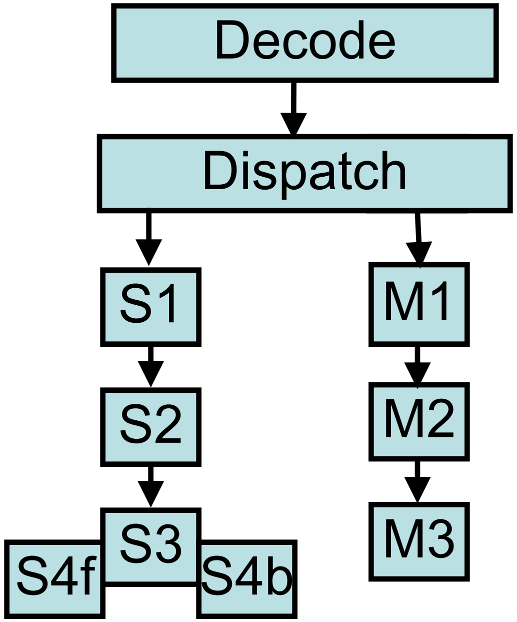
“Good” Behavior Collision

- 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 cycle 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



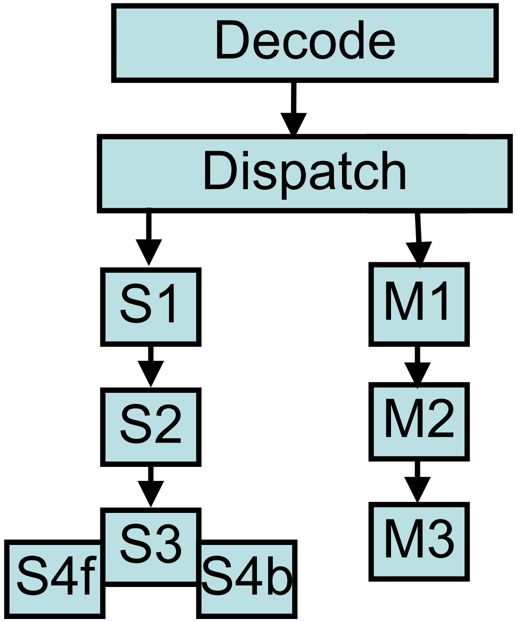
"Good" Behavior Collision

- 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 cycle 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



"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 cycle 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, e.g.
  - bugs in the detection of the data dependency
  - bugs in the logic that stalls execution
  - ....

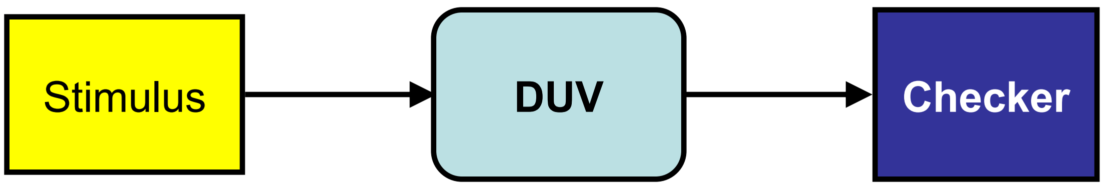
When to check

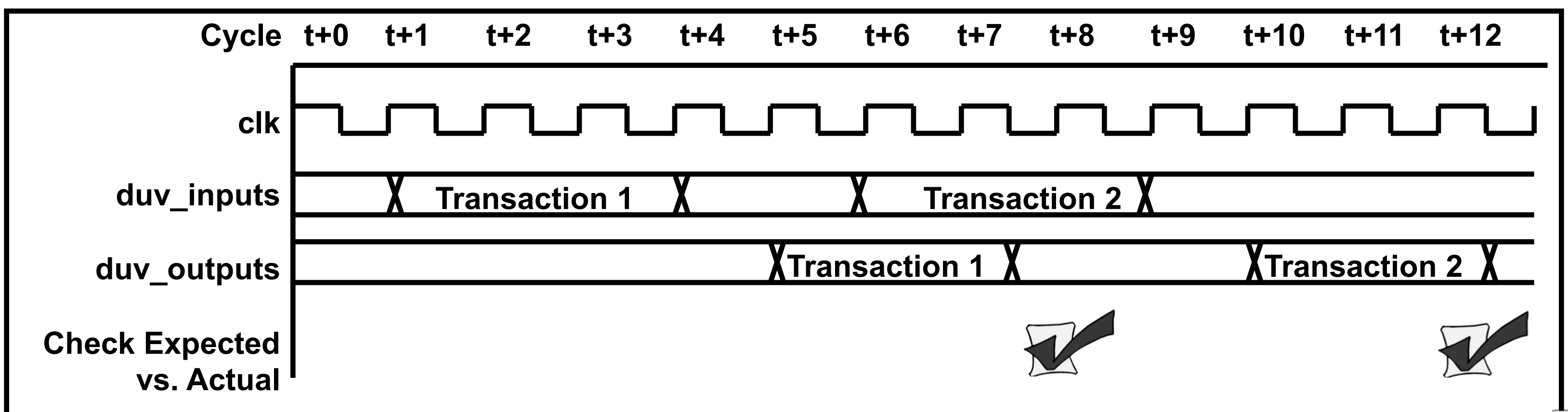
When to Check?

- 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





On-the-fly Checking

- Advantages
  - Detection can be as close as possible (in time and space) to the source of the bug
  - Can stop the test as soon as a bug occurs; no wasted simulation cycles
  - Do not require large traces and external tools to do the checking
- Disadvantages

On-the-fly Checking

Advantages

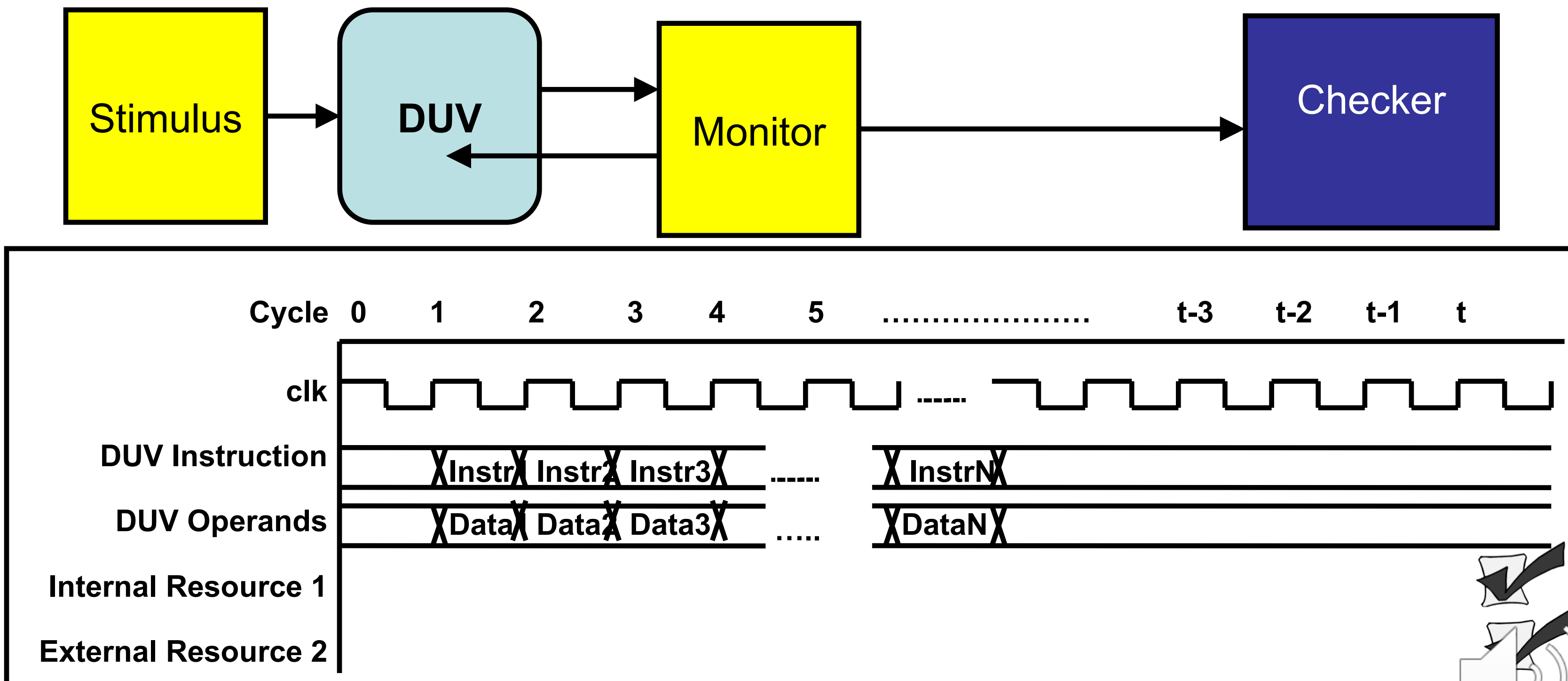
- Detection can be as close as possible (in time and space) to the source of the bug
- Can stop the test as soon as a bug occurs; no wasted simulation cycles
- Do not require large traces and external tools to do the checking

Disadvantages

- May **slow down simulation**
- Checking is limited to the allocated time and space
- Need to plan the checking in advance
  - To perform a new check, we need to add a new checker, and then rerun the simulation.

End-of-test Checking

- Checking is done **at the end of simulation**
- The checker inspects the state of internal and external resources and checks whether they are correct



End-of-test Checking

Disadvantages

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

End-of-test Checking

Disadvantages

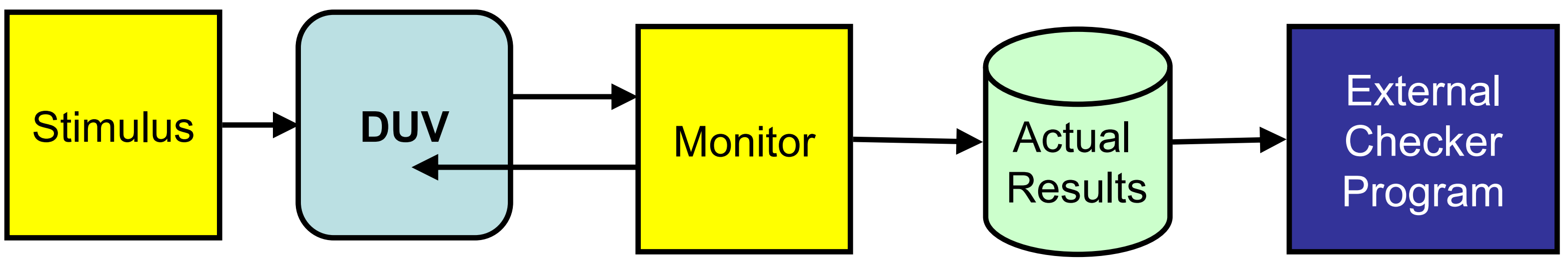
- Provides limited checking capabilities
  - Static look at the state of resources at the end of the test
- High probability of masking bugs by repeated writing to the resources during the simulation
- Hard to detect performance bugs
  - Correct things are happening, but not at the right time
- Hard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

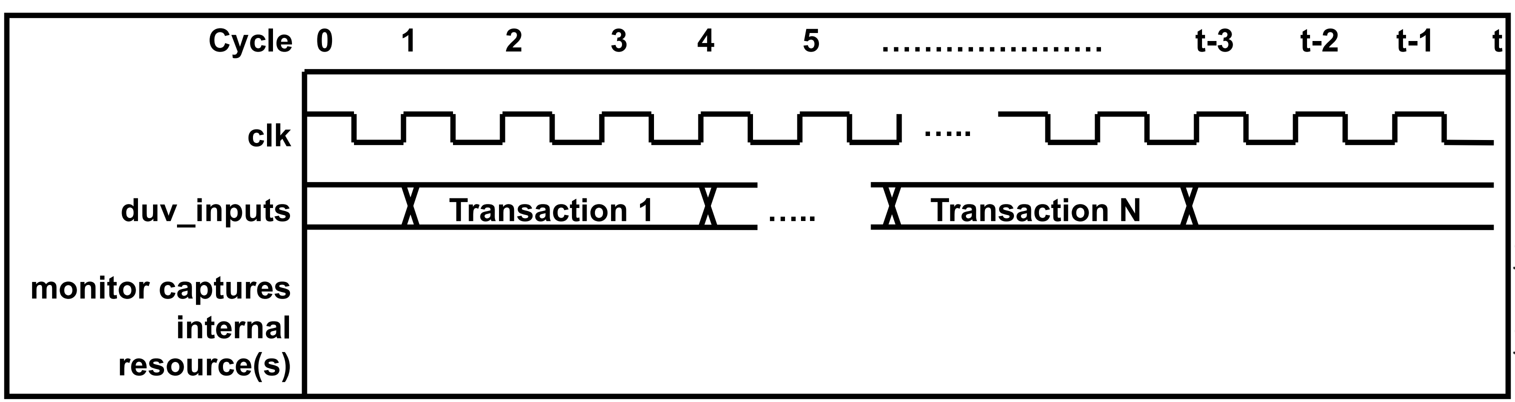
Advantages

- Simpler than other forms of checking
  - May not require a deep understanding of the DUV
- Reduces probability of false alarms
  - (because bad effects may disappear)

External Checking (Monitors)

- **Monitors** keep internal resources' values and behaviors as well as the DUV outputs 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.)

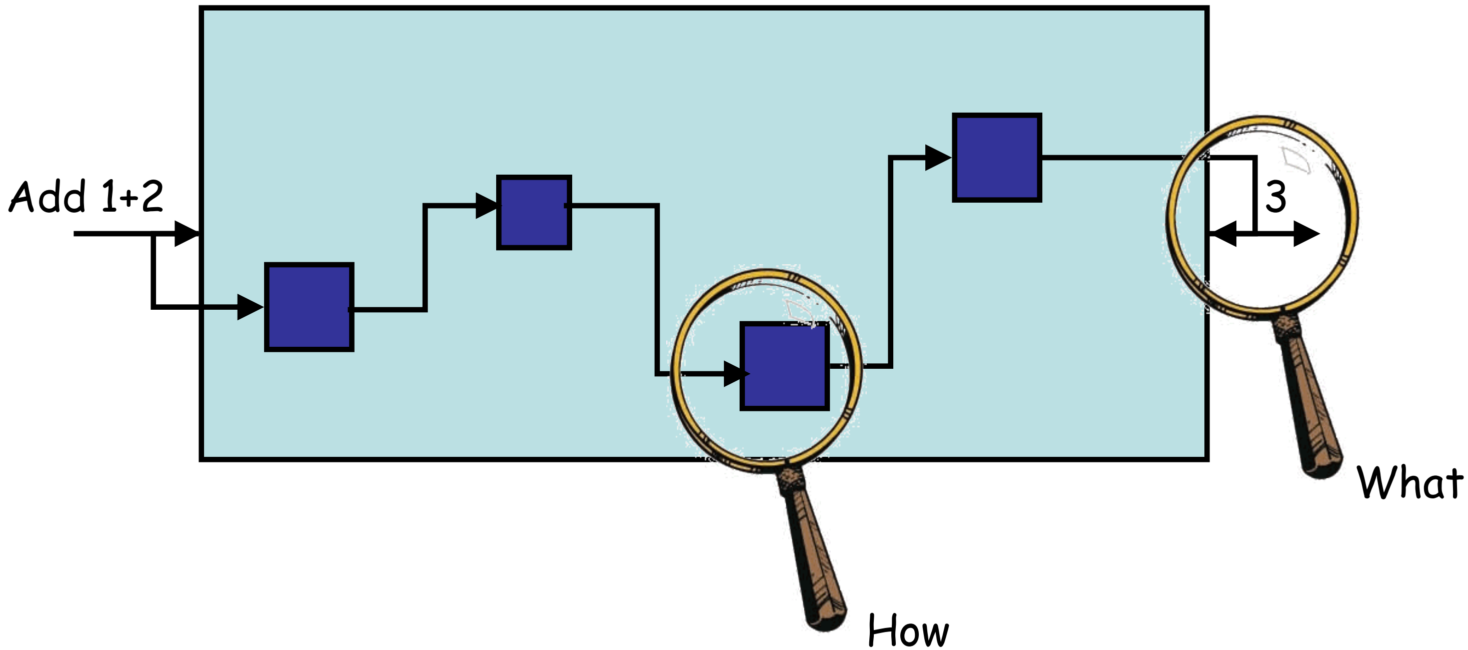
What to check

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
    - e.g. protocol compliance

(Slide from a previous lecture to remind us of where we can get inspiration for checkers from.)

Coarser Classification – The What and the How



Checking the What

- Check the final outcome of a behavior
  - Data oriented
    - But not limited to data

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 tightly focused on implementation details
    - Requires less familiarity with the DUV

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 tightly focused on implementation details
    - Requires less familiarity with the DUV
  - Low correlation between **failure**, the observed behavior that violates the spec, and **bugs/faults**, the root cause of the failure
    - Harder for debugging

Checking the How

- Check *how* things are done internally
  - Control oriented
  - Usually at lower levels of abstraction
    - Closer to implementation
  - Tighter correlation between failure and bugs/faults

the root cause of the failure

the observed behavior that violates the specification

Stimuli Generation and Checking

- In general, checking should be isolated from the stimuli generation
  - **Independence of Checking from Generation**
  - **Modularity:** ability to replace the stimuli generator
  - **Reusability:** ability to use the checkers at higher level of the design hierarchy
- Exceptions to the rule include
  - Self-checking tests
  - Golden test vectors
- *Can stimuli generation assist checking?*

Stimuli Generation and Checking

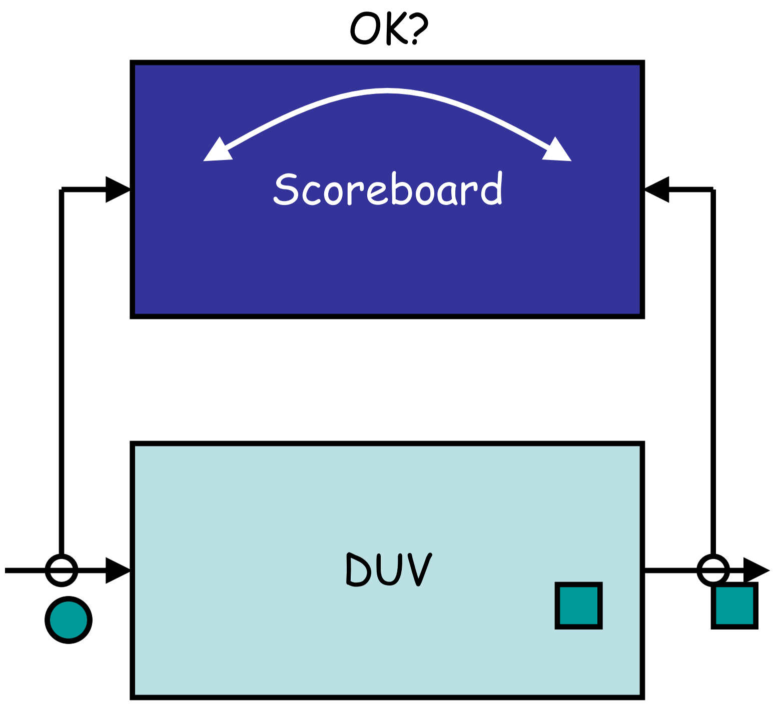
- In general, checking should be isolated from the stimuli generation
  - **Independence of Checking from Generation**
  - **Modularity:** ability to replace the stimuli generator
  - **Reusability:** ability to use the checkers at higher level of the design hierarchy
- Exceptions to the rule include
  - Self-checking tests
  - Golden test vectors
- *Can stimuli generation assist checking?*
  - The stimuli generation can assist checking by improving observability
  - Help transfer events from dark corners to the spotlight

Checking Technologies

Scoreboards

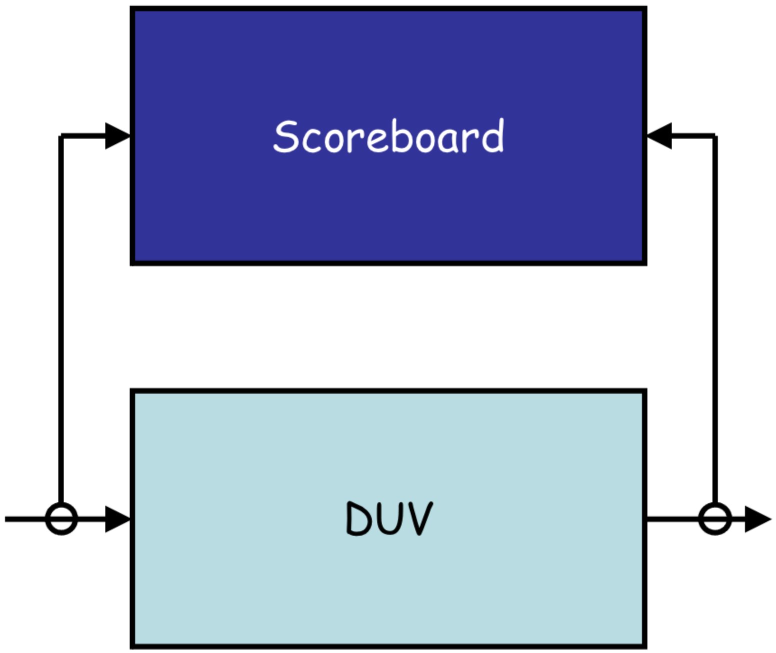
- Scoreboards are smart data structures that keep track of events in the DUV during simulation
- Usually, scoreboards are global
  - One scoreboard per verification environment
- Scoreboards are not checking mechanisms, but
  - The main purpose of using scoreboards is for checking
  - In practice, many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

Scoreboard Operation



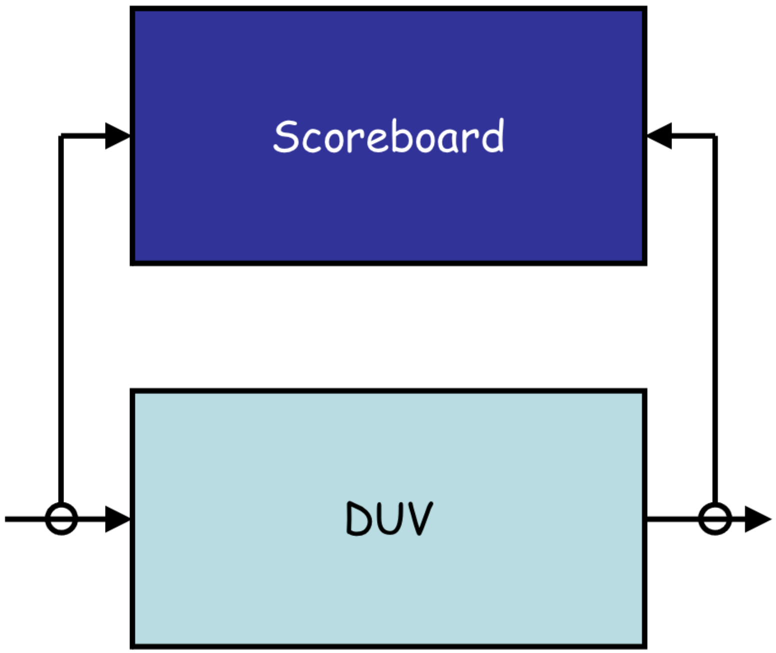
Scoreboards Overview

- Scoreboards source information from
  - the inputs and outputs of the DUV, and
  - occasionally also from internal events in the DUV.
- Scoreboards are very useful in dataflow designs
  - routers
  - cache designs
  - queues and
  - stacks



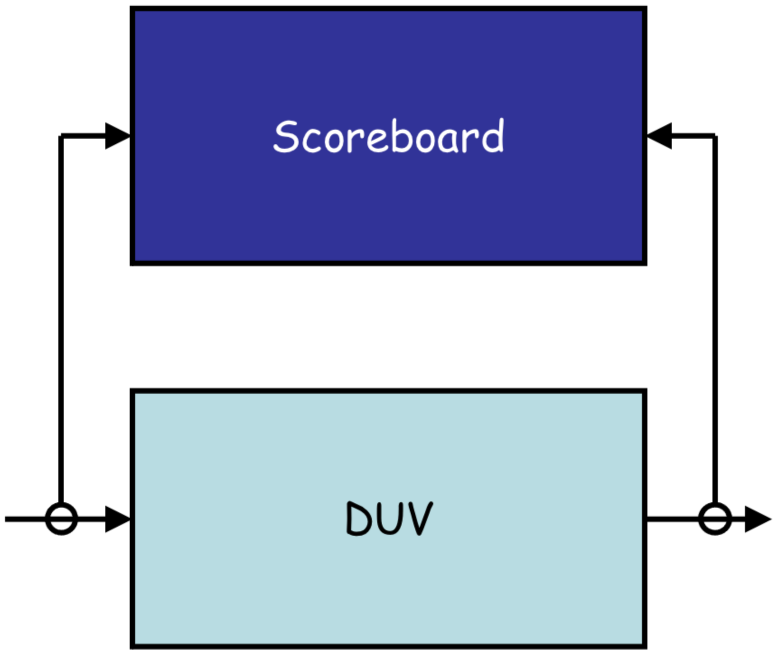
Scoreboards Overview

- Types of checks enabled using a scoreboard:
  - Matching outputs with inputs
    - No **loss** of data
      - Detect inputs with no matching output.
    - No **creation** of data
      - Detect output with no matching input.
    - No unintended **modification** of data



Scoreboards Overview

- Types of checks enabled using a scoreboard:
  - Matching outputs with inputs
    - No **loss** of data
      - Detect inputs with no matching output.
    - No **creation** of data
      - Detect output with no matching input.
    - No unintended **modification** of data
  - Timing specification
    - Delay from input to output remains within specified limits.
  - Data order, where specified
    - Are inputs processed in order of arrival?



Scoreboarding in e - 1

- Assume: DUV does not change order of packets.
  - Hence, first packet on scoreboard has to match received packet.

```
import packet_s;
unit scoreboard {
  !expected_packets : list of packet_s;
  add_packet(p_in : packet_s) is {
    expected_packets.add(p_in);
  };
  check_packet(p_out : packet_s) is {
    var diff : list of string;
     -- Compare physical fields of first packet on scb with p_out.
     -- Report up to 10 differences.
    diff = deep_compare_physical(expected_packets[0], p_out, 10);
                               check that (diff.is_empty())
           else dut_error(``Packet not found on scoreboard'',
  diff);
    -- If match was successful, continue.
    out(``Found received packet on scoreboard.'');
    expected_packets.delete(0);
  };
};
```

Scoreboarding in e - 2

Recording a packet on the scoreboard:

Extend driver such that

- When packet is driven into DUV call **`add_packet`** method of scoreboard.
  - Current packet is copied to scoreboard.
- It is useful to define an event that indicates when packet is being driven.

Checking for a packet on the scoreboard:

Extend receiver such that

- When a packet was received from DUV call **`check_packet`**.
  - Try to find the matching packet on scoreboard.
- It is useful to define an event that indicates when a packet is being received.

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 these two cases:
  - The input is lost or hopelessly stuck in the DUV
  - The DUV did not have enough time to handle the input

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 these 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 the delay cannot be bound?

Side Note: Graceful End-of-test

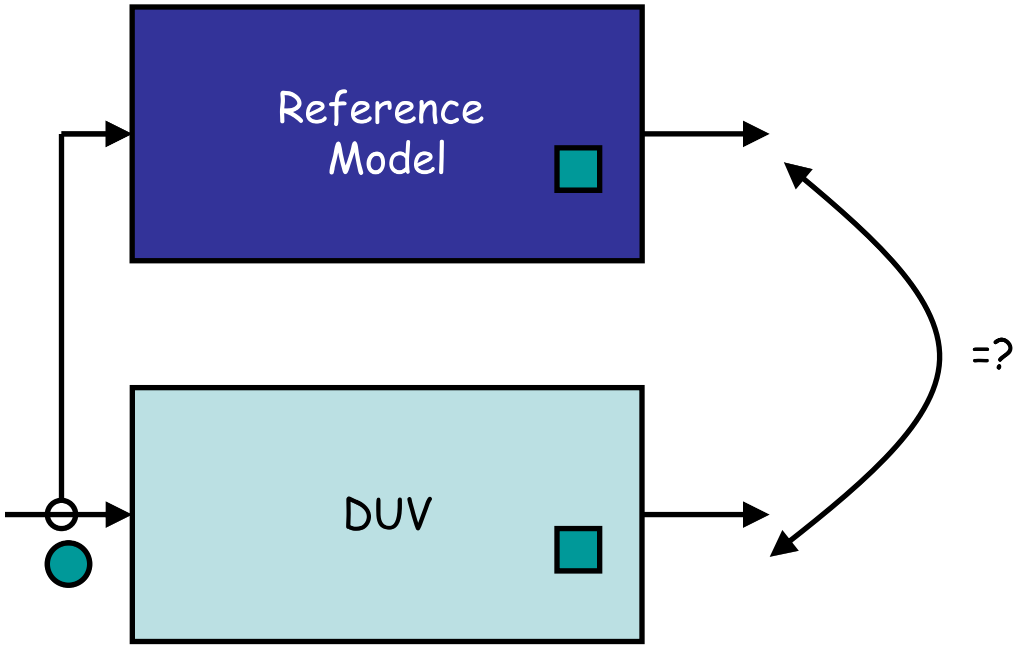
Checking that nothing is lost is very important

- If an input does not have a matching output, how can we distinguish between these 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 the delay cannot be bound?
- 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 processed

Reference Models

- A reference model is an *oracle* that tells us 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 provides 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)

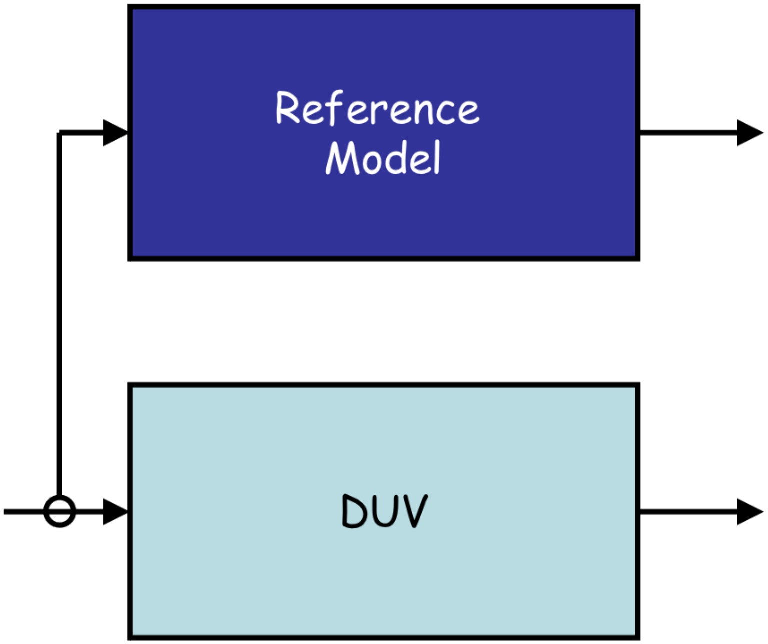
Reference Model Operation



Reference Models

Reference models have many uses

- Checking
- Aid stimuli generation
  (When?)
  - *Check the lecture on Stimuli Generation*
    - *in particular the sections on offline dynamic test generation*
- Act as “smart” protocol models
  - imitate the function of the DUV



- Vehicles for SW development

Reference Models

What can we check with a reference model?

- In principle, anything
- In practice it depends on the level of detail 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 a reference model dictates the type of information we can get out of it for checking
  - Functionally accurate models can be used only to check correctness of data, usually at the end of the test or at well defined points in time
    - timing compliance, order of execution, and similar properties need other means of checking

Levels of Abstraction

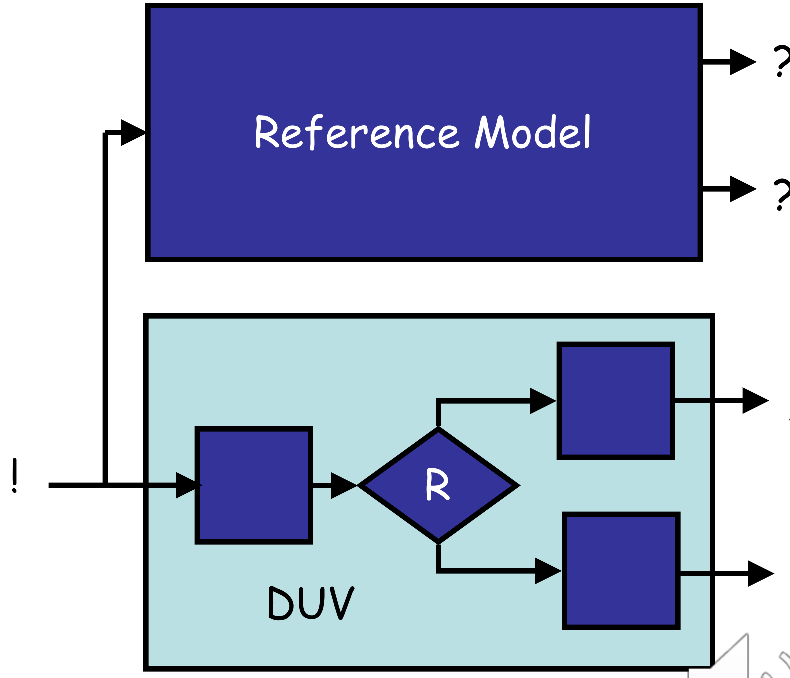
- The level of abstraction in a reference model dictates the type of information we can get out of it for checking
  - Functionally accurate models can be used only to check correctness of data, usually at the end of the test or at well defined points in time
    - timing compliance, order of execution, and similar properties need other means of checking
  - Cycle accurate models can be used for checking all aspects of I/O behavior

Levels of Abstraction

- The level of abstraction in a reference model dictates the type of information we can get out of it for checking
  - Functionally accurate models can be used only to check correctness of data, usually at the end of the test or at well defined points in time
    - timing compliance, order of execution, and similar properties need other means of checking
  - Cycle accurate models can be used for checking all aspects of I/O behavior
  - Cycle accurate and latch accurate models can be used also for checking the internal state of the DUV
    - This type of model is sometimes called deep functional reference model

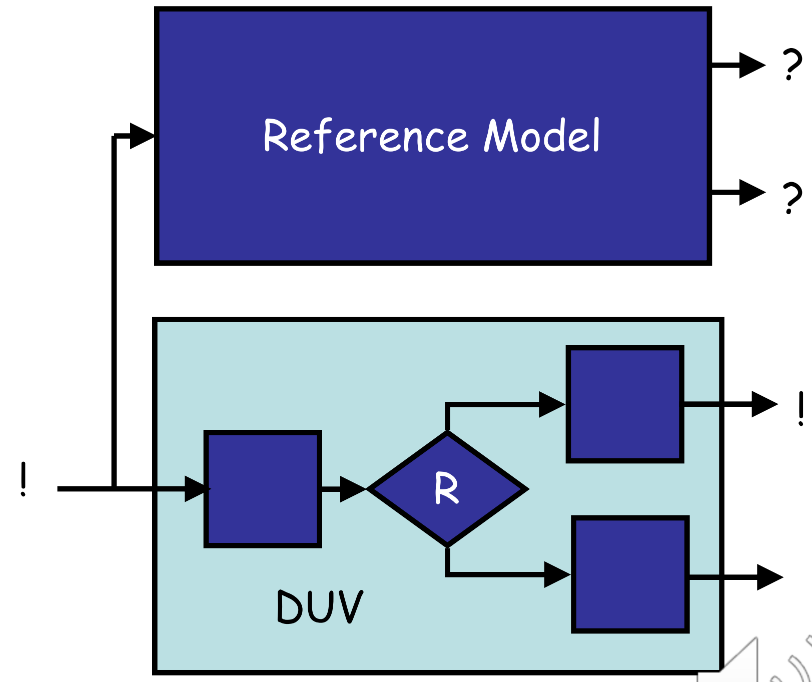
Impure Reference Model

- Sometimes it is impossible (or very hard) for the reference model to duplicate significant decisions made by the DUV



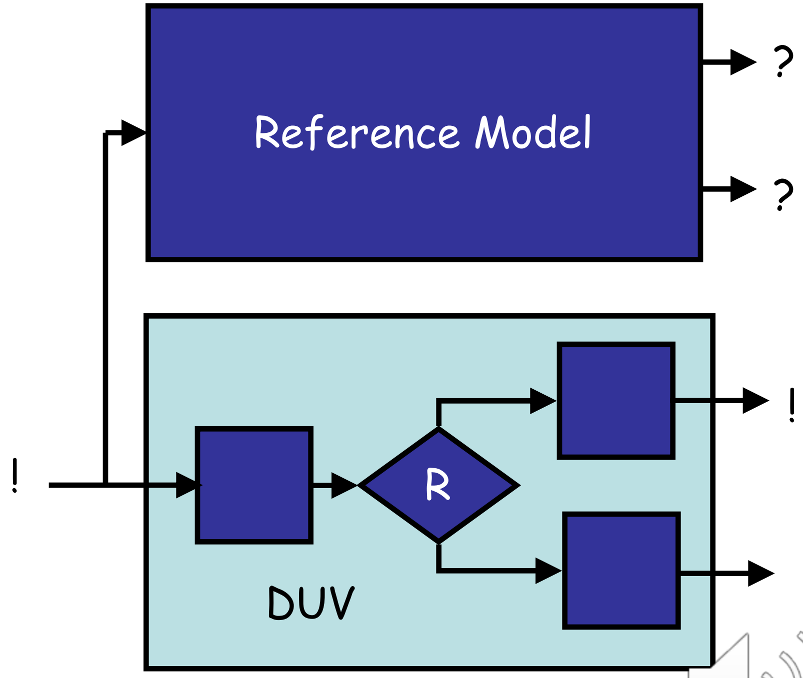
Impure Reference Model

- Sometimes it is impossible (or very hard) for the reference model to duplicate significant decisions made by the DUV



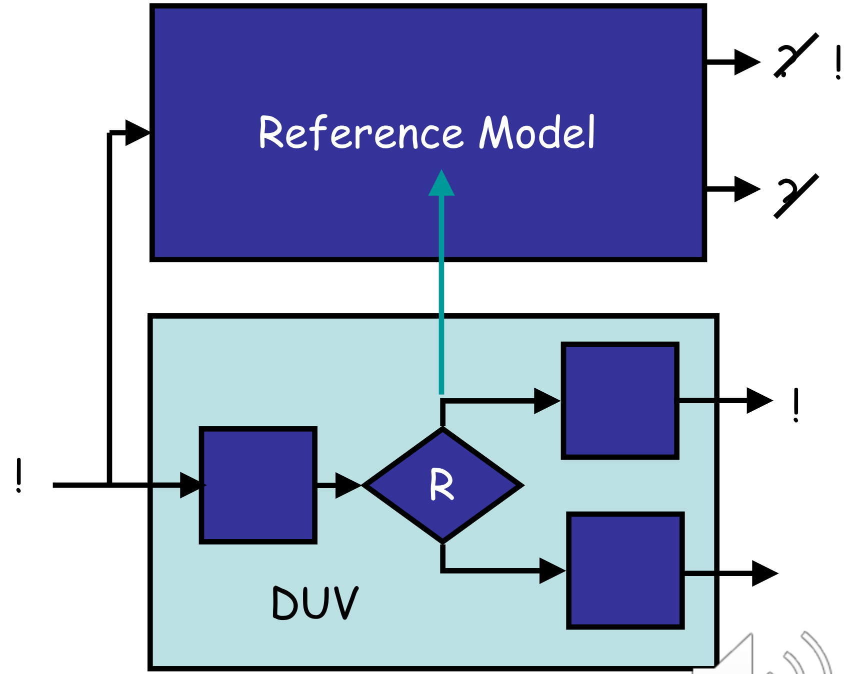
Impure Reference Model

- 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!



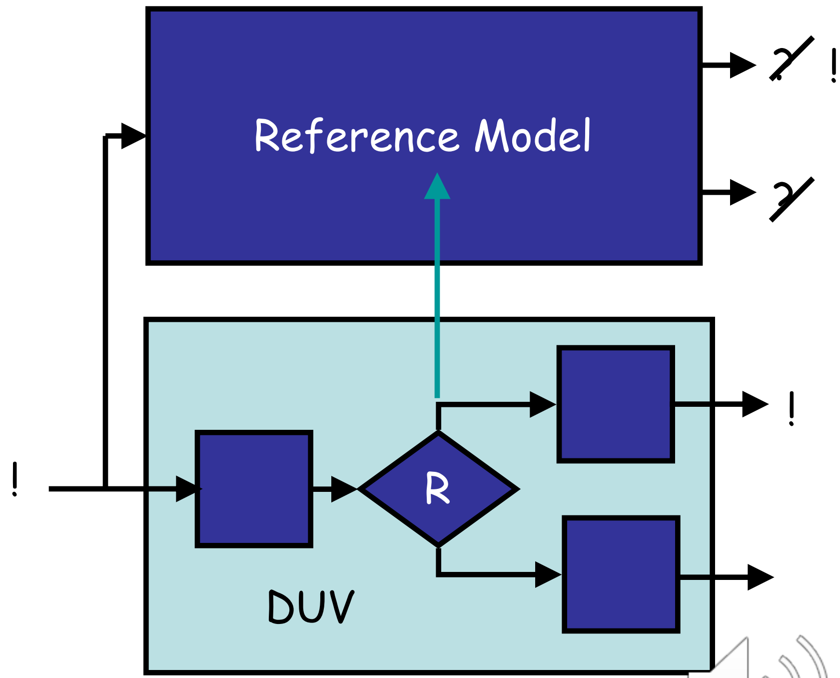
Impure Reference Model

- 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!



Impure Reference Model

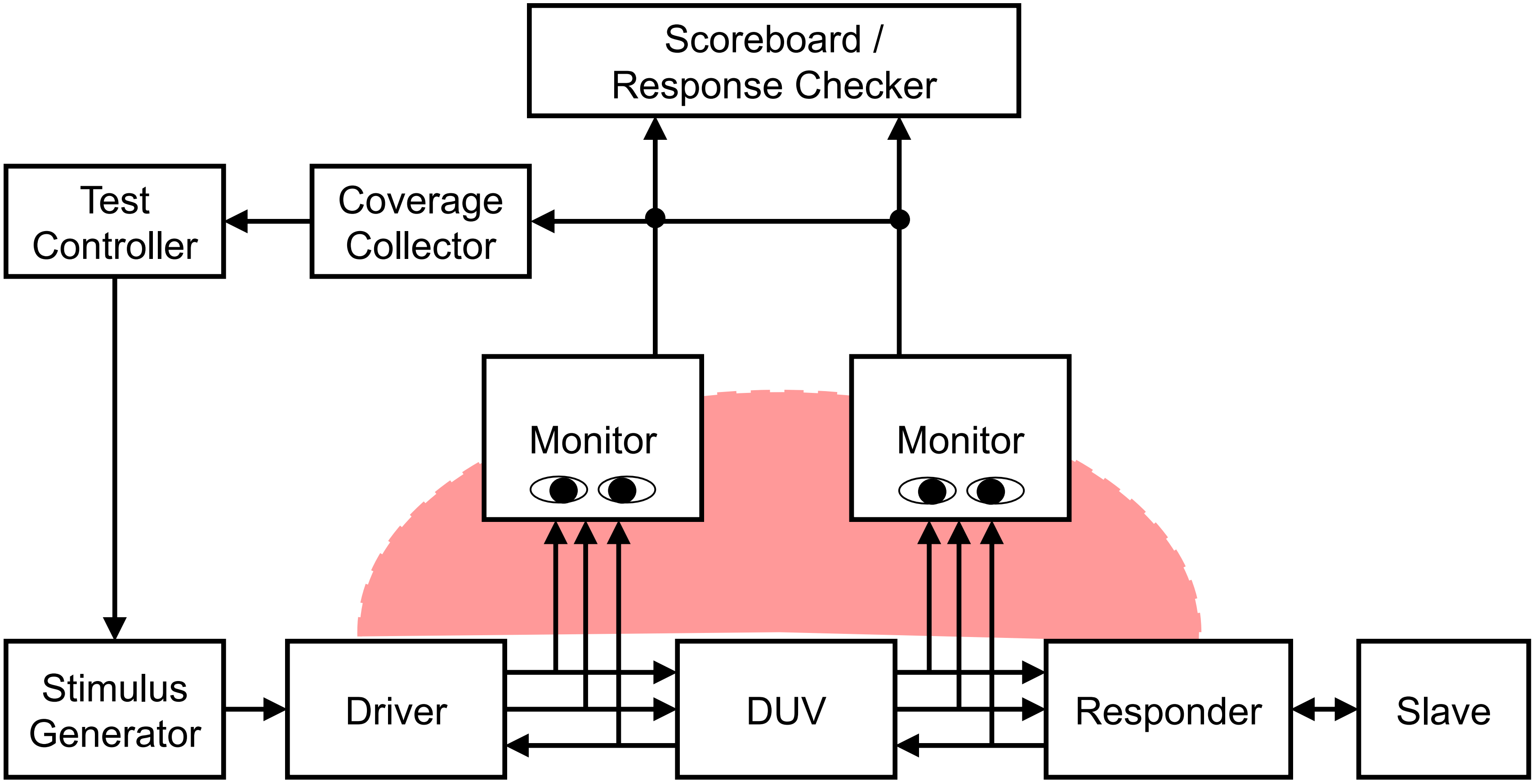
- Sometimes it is impossible (or very hard) for the reference model to duplicate significant decisions made by the DUV
- Possible solution:
  Use information from the DUV to assist the reference model!



What are the shortfalls of impure reference models?

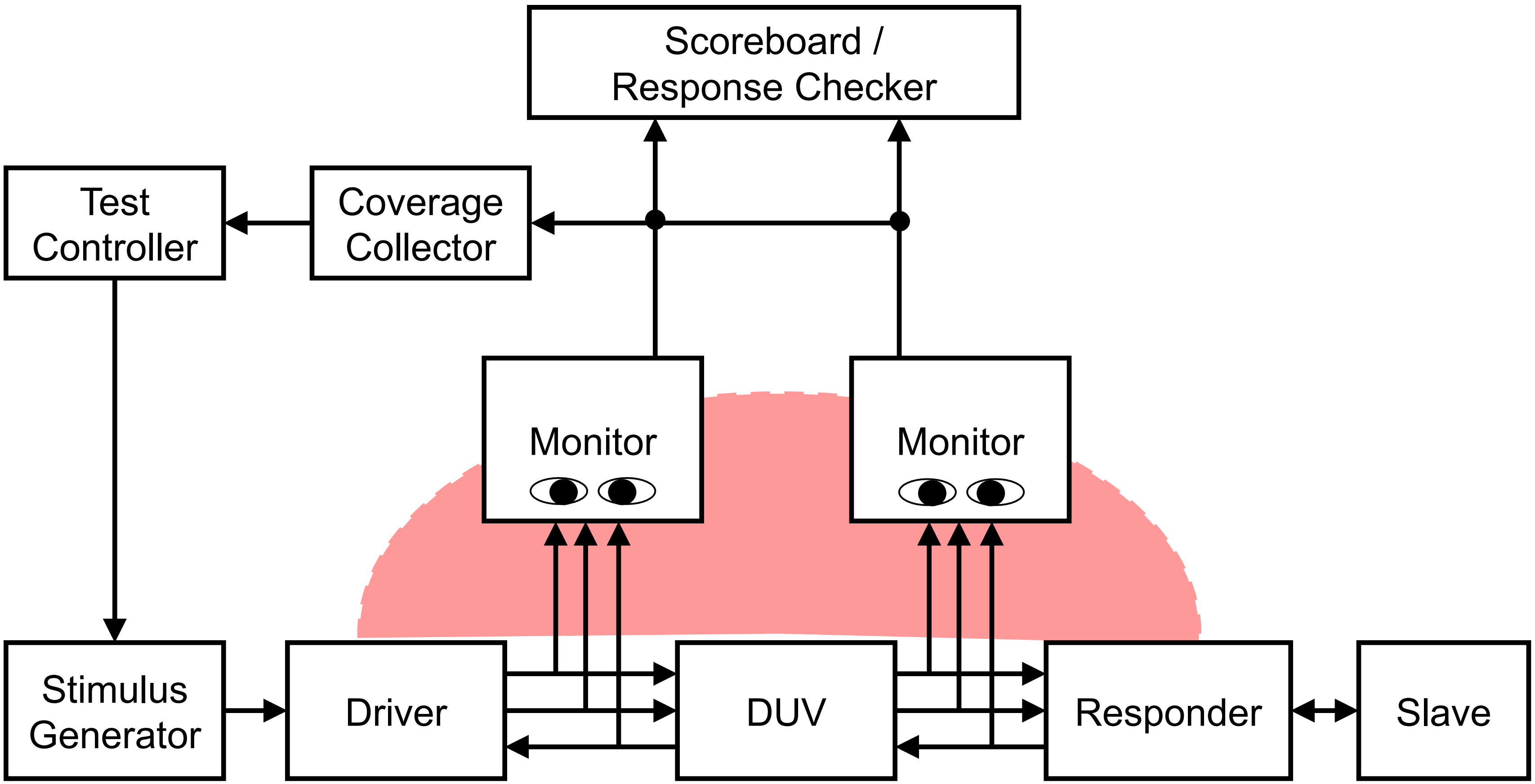
Contemporary TB Architecture

Contemporary Testbench Architecture



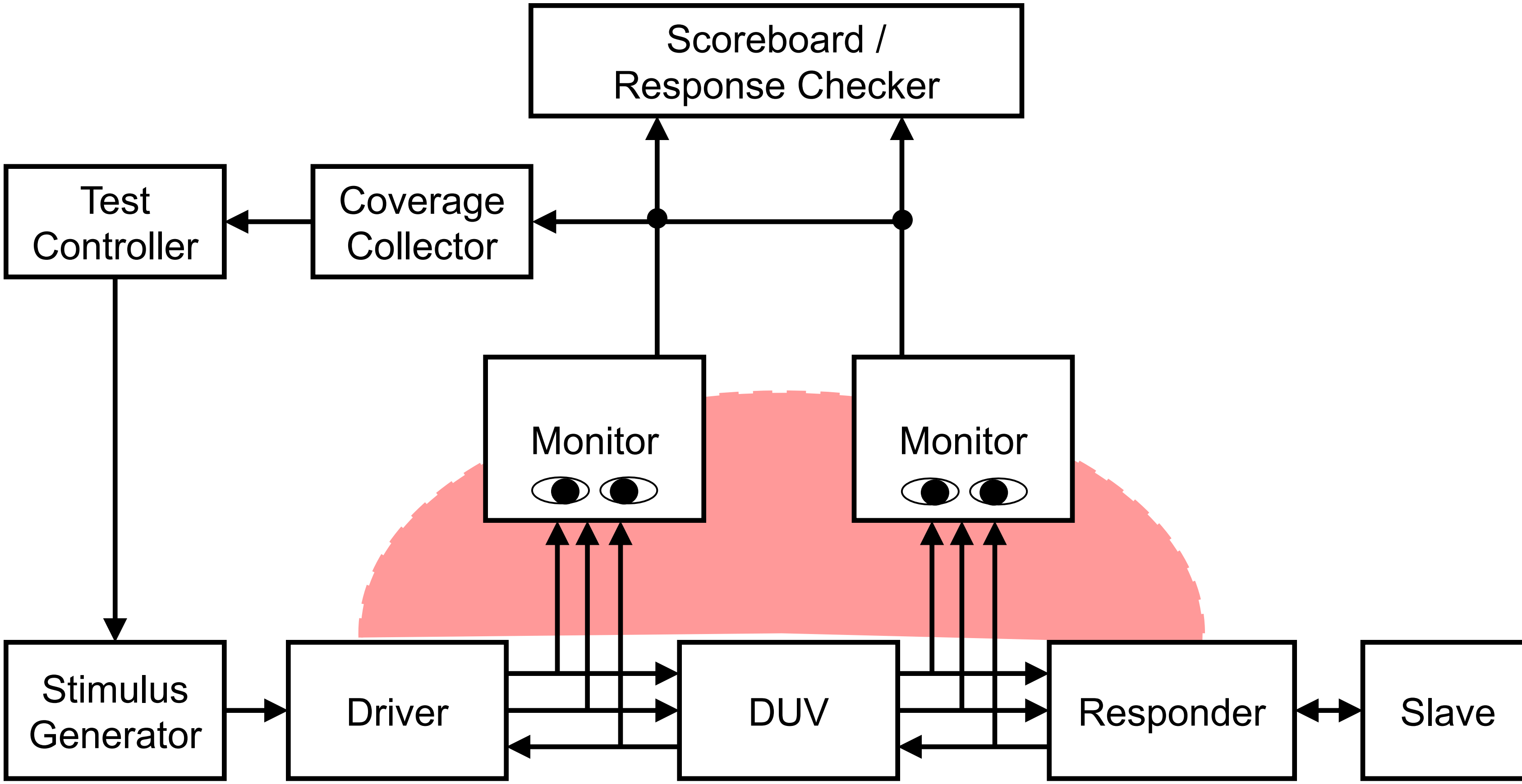
H. Foster: "Response checkers, monitors and asserstions". In Practical Design Verification by Pradhan and Harris (editors). Cambridge, 2009.

Contemporary Testbench Architecture



H. Foster: "Response checkers, monitors and asserstions". In Practical Design Verification by Pradhan and Harris (editors). Cambridge, 2009.

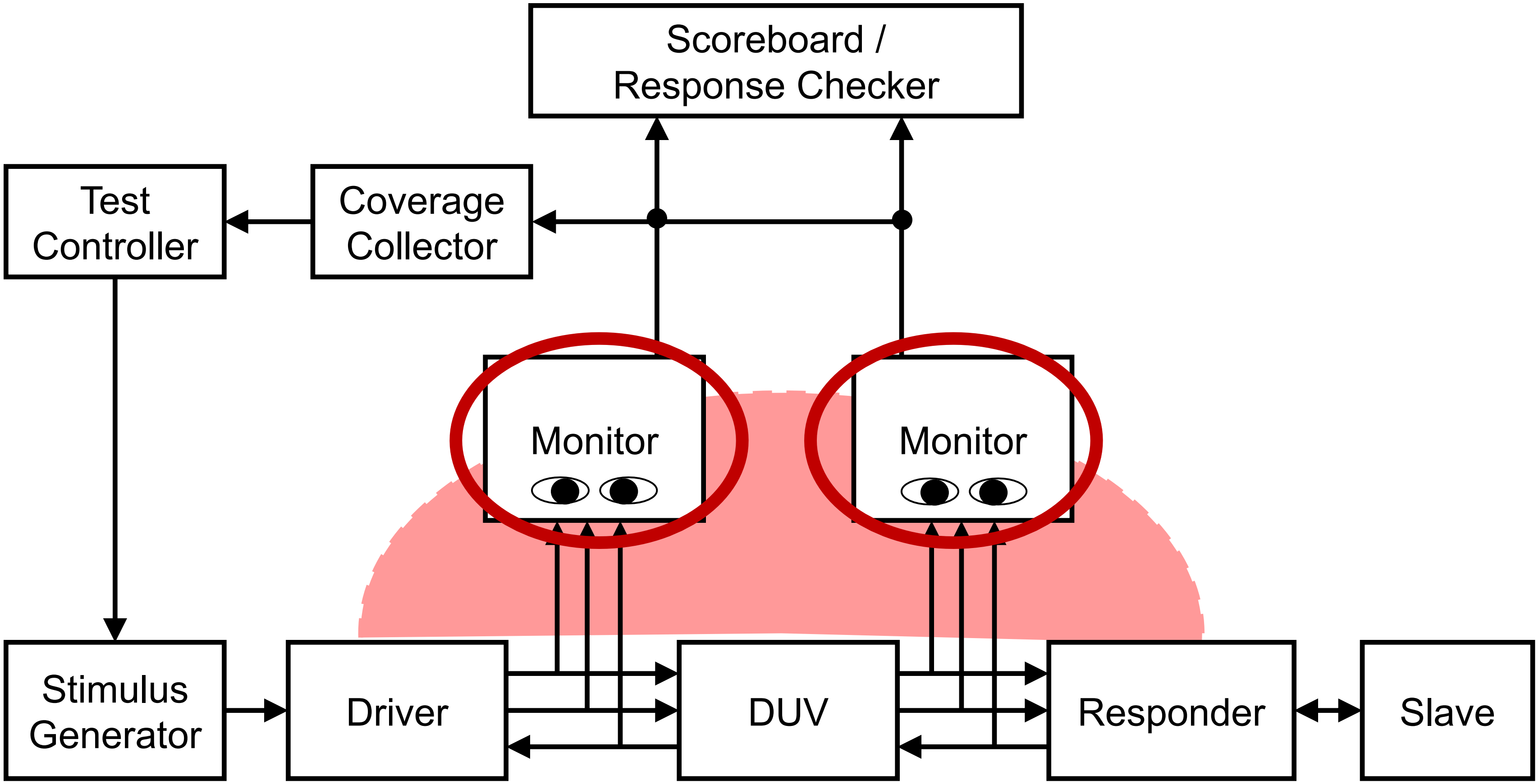
Contemporary Testbench Architecture



H. Foster: "Response checkers, monitors and asserstions". In Practical Design Verification by Pradhan and Harris (editors). Cambridge, 2009.

Contemporary Testbench Architecture

H. Foster: "Response checkers, monitors and asserstions". In Practical Design Verification by Pradhan and Harris (editors). Cambridge, 2009.



Monitors

- Monitors are TB components that observe the inputs, outputs, or internals of the DUV.
  - Monitors watch activity of the DUV.
    - Black box: DUV inputs and outputs
    - Grey box: potentially selected internals
  - Monitors can convert low-level signals to transactions.
  - Monitors can flag simple timing and protocol errors.
  - Monitors collect functional coverage.
  - Monitors update the scoreboard.
  - Monitors don’t drive DUV pins; they are “passive”.
    - Monitors are self-contained and don’t cause “side effects”.
    - Monitors are re-usable at different levels of abstraction.

Types of Monitors

- Input monitors:
  - Collect inputs to the DUV and pass them to scoreboard.
  - Can have checker components.
- Output monitors:
  - Observe the outputs from the DUV and pass them to the scoreboard.
  - Can have checker components.
- Coverage monitors:
  - Collect inputs, outputs and selected internal signals.
  - Permit analysis of stimulus and functionality coverage.

Rule-based Checking

- Checks that a set of rules hold in the DUV
- Essentially, all checking is rule-based, e.g.

```
if (not "something") then error
```

- where the "something" can be
  - Value of a register matches value in reference model

Rule-based Checking

- Checks that a set of rules hold in the DUV
- Essentially, all checking is rule-based, e.g.

```
if (not "something") then error
```

- where the "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

Rule-based Checking

- Checks that a set of rules hold in the DUV
- Essentially, all checking is rule-based, e.g.

> if (not "something") then error

- where the "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

- Rules can come from many sources
  - All levels of the design process
    - Spec, high-level design, lower levels of design, implementation
  - Behavior of neighboring units

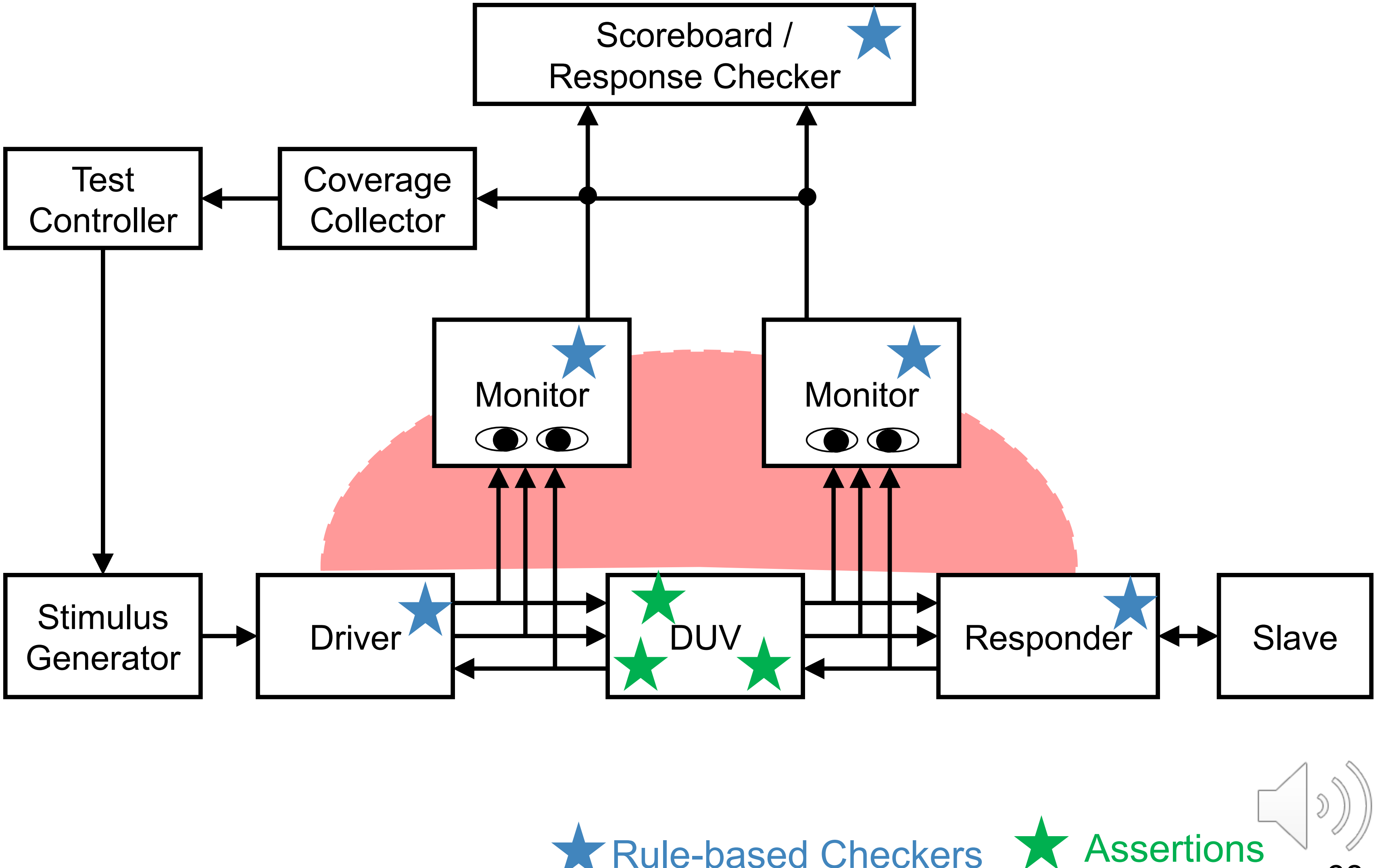
Rule-based Checking

- Rules can come from many sources
  - All levels of the design process
    - Spec, high-level design, lower levels of 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

- Rules can come from many sources
  - All levels of the design process
    - Spec, high-level design, lower levels of 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 checkers embedded in the DUV code are called assertions
  - Lecture on Assertion-Based Verification

Contemporary Testbench Architecture



H. Foster: "Response checkers, monitors and assertions". In Practical Design Verification by Pradhan and Harris (editors). Cambridge, 2009.

Self-Checking Testbenches

- Knowledge of the DUV's functionality can be built into the TB.
  - This *automates* the checking process.
  - Verification engineers encode their knowledge of correct DUV functionality into the checkers, monitors and scoreboard using:
    - Golden Vectors,
    - Reference Models,
    - Protocols or Transactions,
    - Assertions.
- This results in a **"self-checking" TB**.
  - Checkers are "always" active.

Checking: Practical Aspects

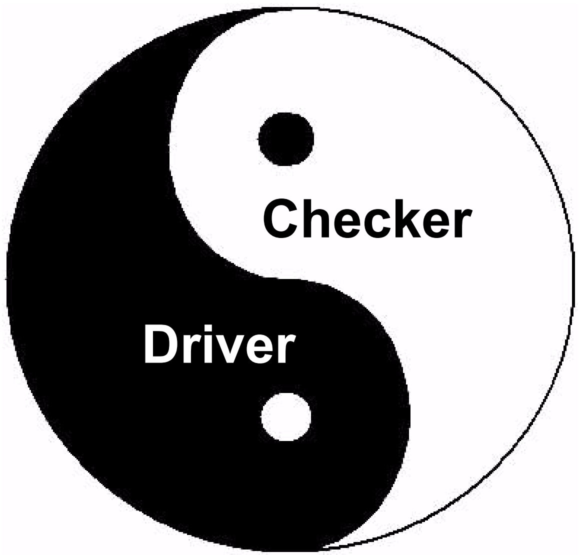
Consider the following:

1. The cost of *implementation and maintenance* of checkers vs the cost of *debugging* (without checkers).

2. The cost of mistakes
   – Missed detection
      ▪ 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?

Summary



- ✓ (Stimuli Generation)
- ✓ Checking
- ABV
- Coverage

Coverage Driven Verification Methodology

➔ Coverage Directed Test Generation