

# Assertion Based Verification (ABV)

**Assertion:** An `if` statement with an error condition that indicates that the condition in the `if` statement be(come) 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.

VHDL also has `assert()` statement - but never popular.

- Only used to terminate simulation.

Revolution through Foster & Bening's OVL for Verilog.

 SystemVerilog now offers assertions to Verilog users.

---

# SW Assertion Checks

## SW 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!

Why?

# SW vs HW Assertions

## HW assertions:

- **static** (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.”

and

- **temporal conditions** - to verify 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.”



**(Some) HW assertion conditions must be evaluated over time!** → Need *temporal* assertion specification language!

# Classes of Assertions - Implementation

**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!*

Why?

# Classes of Assertions - Implementation

**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!*

Why?

REMEMBER: **Verification independence!** If the designer writes the assertion then he/she will make the same wrong assumptions made in the implementation of the design.

# Classes of Assertions - Specification

## **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 scoreboard to check whether an arbiter is fair, a block-level assertion is much simpler to specify.


# Simulation Assertions

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 way of using Verilog to specify temporal expressions.

---

# Example Assertions I

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

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



---

# Example Assertions 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.

---

# Example Assertions 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));
```

---

# Example Assertions 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.

 Controller should always maximize the green time for the major road.

---

# Example Assertions 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.



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));
```

# Example Assertions 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.

 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.

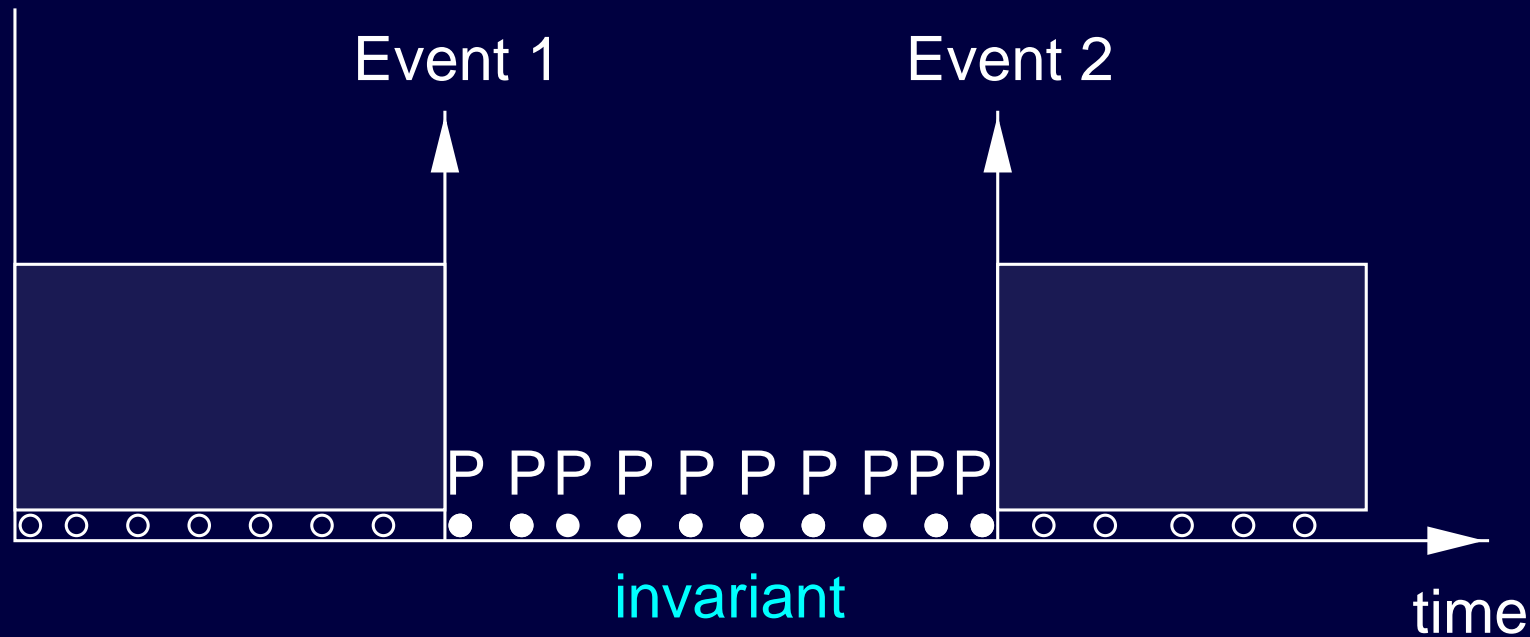
---

# Different Types of Assertions

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

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

# Invariant Assertion Window

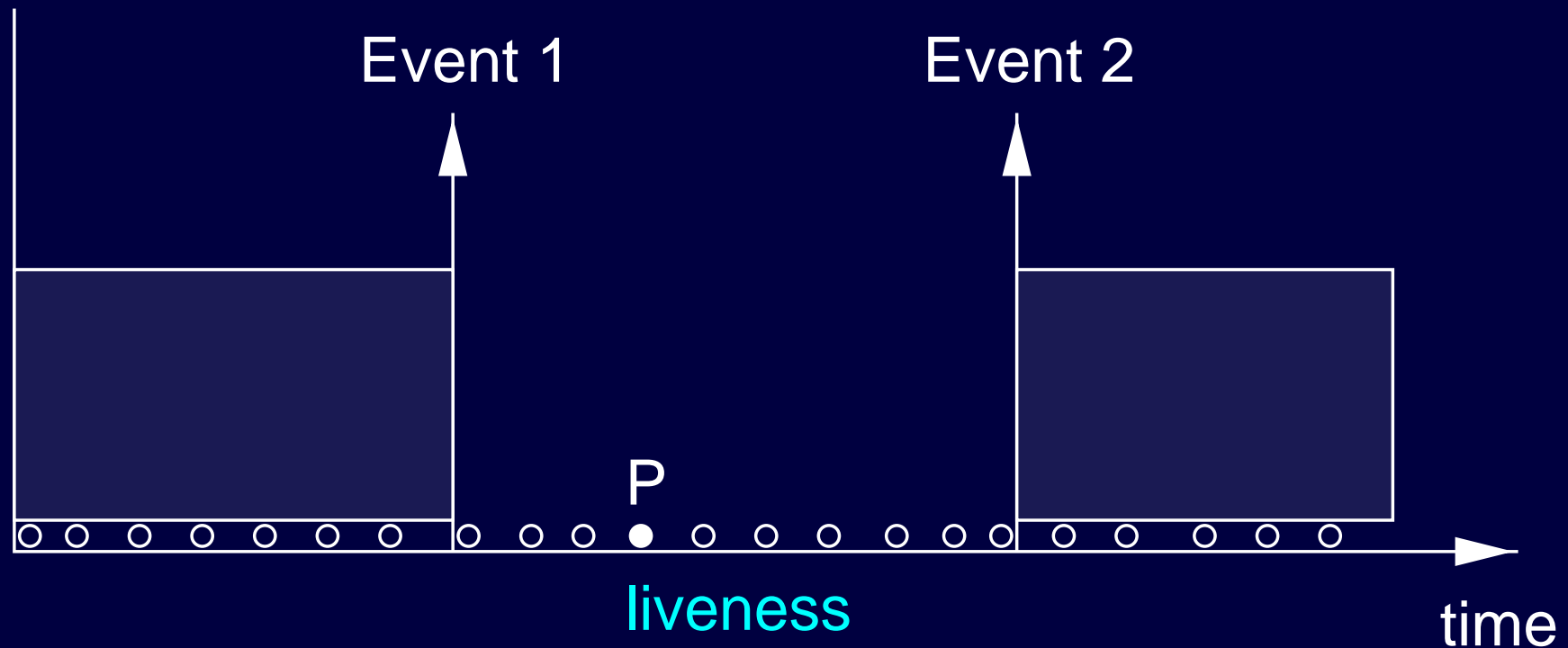


Assertion P is checked after Event 1 occurs, and continues to be checked until Event 2.

- Event triggers and P are just Verilog expressions.

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

# Liveness Assertion Window



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.]



# How assertions work - in simulation

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


BUT:

Symptoms of low-level failures often not easy to observe/detect.

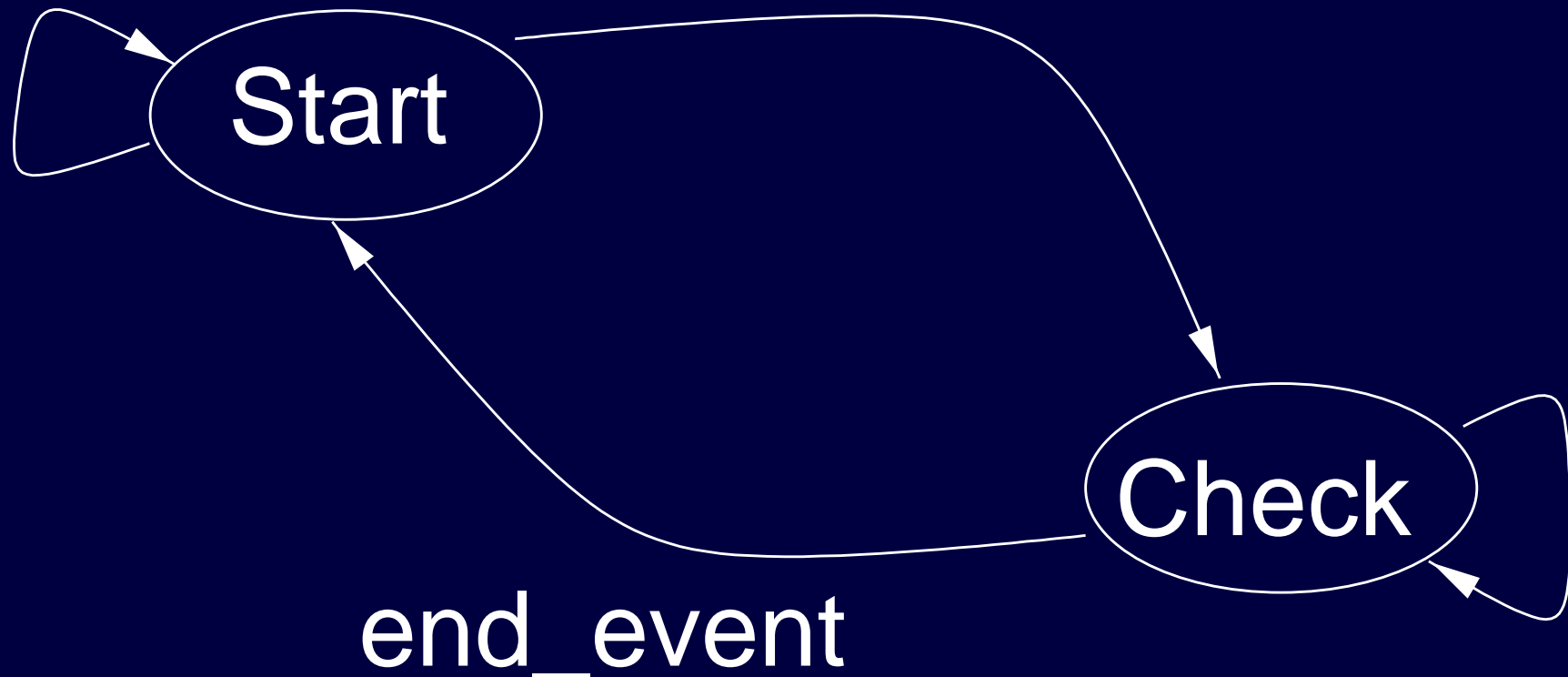
- Not visible (if at all) until affected data item reaches output and self-checking testbench flags mismatch.

*During simulation a design's **assertions monitor** activates.*

**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.

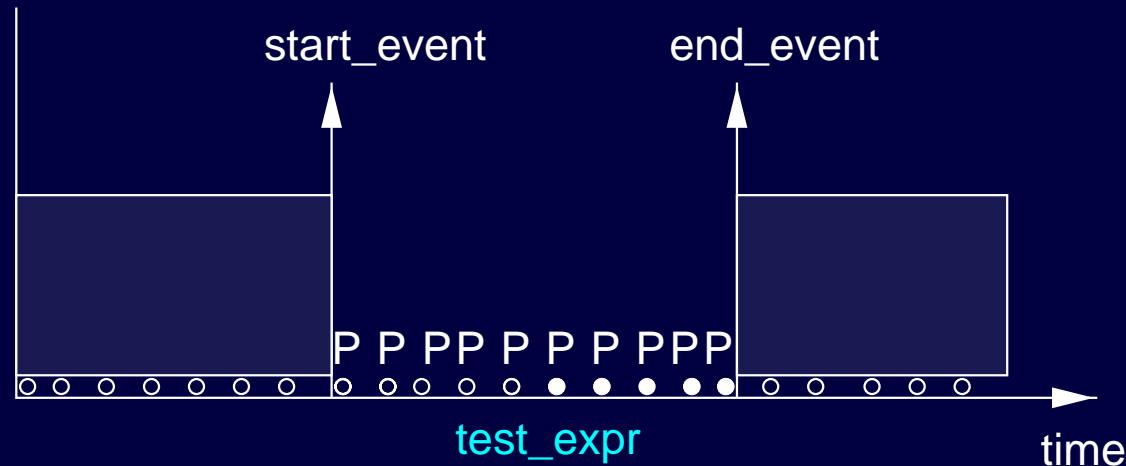
# Sequential Assertion Monitor



The **start\_event** initiates assertion validation process.  
Evaluation continues until the **end\_event** occurs.

- Distinguish between *time-bounded* and *event-bounded* monitors.

# Example: assert\_change



*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.
- FSM machine state change verification

---


# **(More) Example Assertions III**

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

# (More) Example Assertions III

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.

# (More) Example Assertions III

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.*

---

# 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?*

# 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.



# Increasing Observability

The Coverage Challenge: *Maximise the probability of stimulating and detecting bugs at minimum cost (in time, labour and computation).*

[Dill, ICCAD'99]

Goal: **Comprehensive Verification without redundant effort.**


## Shortcomings of Traditional Metrics:

- No qualitative insight into design intent.
- Limited to measuring what is controllable.
- Activating a faulty statement does not mean the bug will propagate to an observable output.

*Observability can be increased using Assertion Based Verification & Coverage.*

# Formal Assertion Checking in Practice

## Model Checkers or Property Checkers:

- Mathematical proof of property correctness for given RTL design.
  - Satisfied property *always* holds, i.e. exhaustive proof.
- Provides counter example (sequence of events) that leads to property violation.
  - Is this sequence of events possible?
- *Need to specify assumptions for proof:*
  - Assertions/constraints on inputs/state of design.
-  Correctness of proof depends on correctness of assumptions!
- Most common mistake: **Restrict input/state so much, that proof becomes trivial!**

---

# 3 Forms of Formal Assertion Checking

## 1. Static Formal Verification (SFV)

- Exhaustive formal analysis often from a reset state of the design.
  - In practice typically based on model checking, symbolic sim, SAT.


++ Proven assertions can be guaranteed to be true under any scenario.

-- Capacity limits: Fails for large designs or complex assertions.

SFV primarily provides proofs.

# 3 Forms of Formal Assertion Checking

## 2. 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.
    - Not exhaustive. Formal analysis *chooses* sim vectors.
- \_\_ Coverage metrics only loosely related to bugs.
-  *Guided sim might improve coverage but might not find bugs.*

# 3. Dynamic Formal Verification (DFV)

Simulation requires stimulus generation (test vectors).

- 💡 FV can use these test vectors as a starting point.
  - Rather than starting from reset state.
- 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 focused on finding counter examples (i.e. bugs).

**DFV bridges gap between formal verification and simulation.**

- *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.

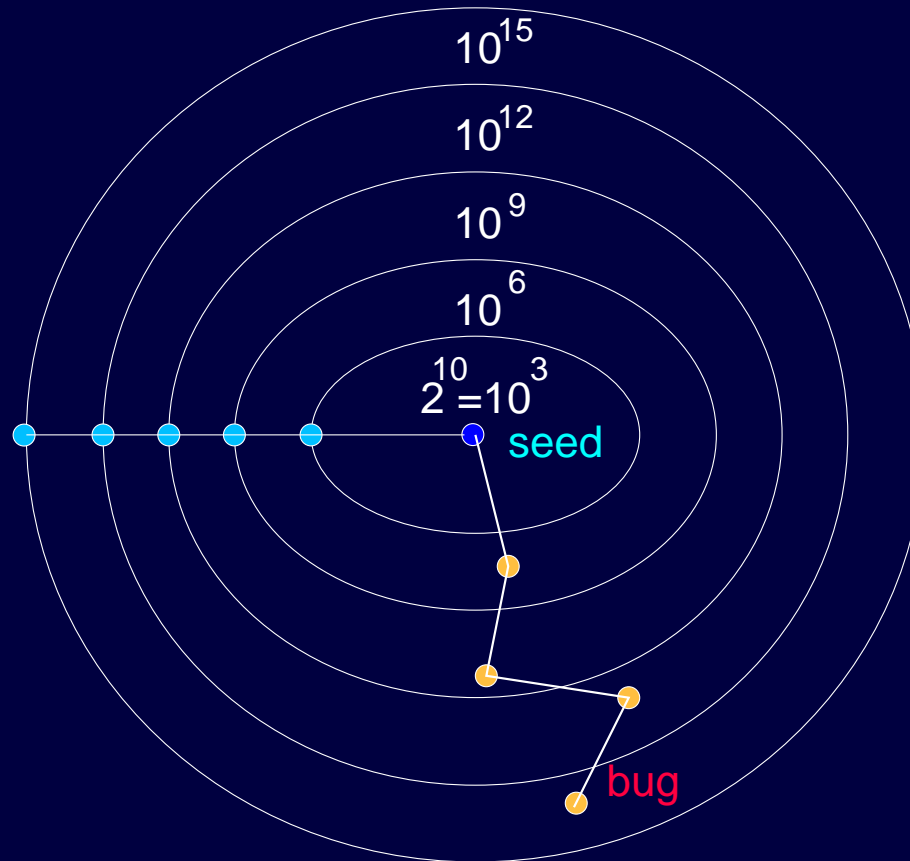
**DFV:** *Measure of depth of the exhaustive search around each seed.*

 *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).

**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

# NEW: Deep Dynamic FV (DDFV)


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.*
  - Based on new 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
- Exhaustive verification of complex assertion in 1M-gate DUV to proof radius of 50-150 cycles within 1,000 seconds.
- Assume DUV has 300 inputs that can affect the assertion at large proof radii.  $10^{90}$  possible input patterns each cycle, proof radius of 100 cycles is equivalent to about  $10^{9000}$  sim cycles.

**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 Checking in Practice

EDA support for ABV:

- *Need to provide common language to use for property specification - nobody in this industry seems to know English that well...*
- **PSL/Sugar [IBM/Accellera]**
  - temporal language that can be used both for sim and formal verification
    - \* More in lectures on Formal Functional Verification.
- OVA (Open Vera Assertions),
- OVL (Open Verification Library - Verilog Modules),
- SVA (System Verilog Assertions - SV version of OVL)