

# COMS31700 Design Verification: **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.)

# What is an assertion?

- An **assertion** is a statement that a particular property is required to be true .
  - A property is a Boolean-valued expression, e.g. in SystemVerilog.
- An assertion is a directive to a verification tool.
  - Assertions can be checked either during simulation or using a formal property checker.
- Assertions have been used in SW design for a long time.
  - `assert()` function is 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).

# HW Assertions

## HW assertions:

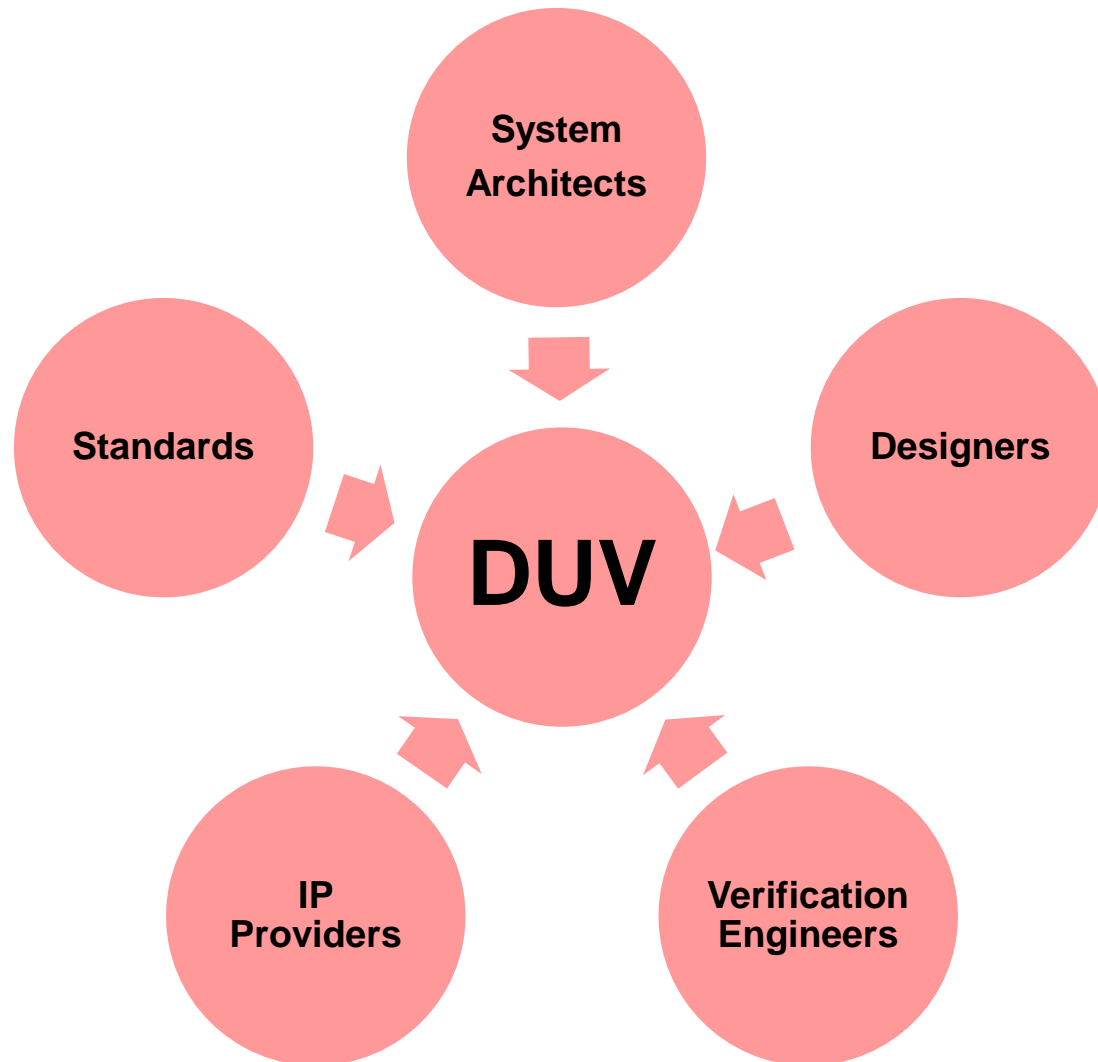
- combinatorial (i.e. “zero-time”) **conditions** that ensure functional correctness
  - must be valid at all times
    - “This buffer never overflows.”
    - “This register always holds a single-digit value.”
    - “The state machine is one hot.”
    - “There are no x’s on the bus when the data is valid.”

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

# Who writes the assertions?

---



# Types of Assertions

# Types of Assertions: Implementation Assertions

---

- Also called “**design**” assertions.
- Specified by the designer.
- Encode designer’s assumptions.
  - Interface assertions
    - Catch different interpretations between different designers.
- Formulate 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!)

# Types of Assertions: Specification Assertions

---

- **Also called “intent” assertions**
  - Often high-level properties.
- **Specified by architects, verification engineers, IP providers, standards.**
- 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.
  - Instead of using (implementing) a monitor and checker, in some cases writing a block-level assertion can be much simpler.

# Safety Properties

- **Safety:** Something bad does not happen
  - The FIFO **does not** overflow.
  - The system **does not** allow more than one process to use a shared device simultaneously.
  - Requests are answered within 5 cycles.
- **More formally:** *A safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property.* [Accellera PSL-1.1 2004]

Safety properties can be falsified by a finite simulation run.



# Liveness Properties

- **Liveness:** Something good eventually happens
  - The system **eventually** terminates.
  - Every request is **eventually** acknowledged.
- More formally: *A liveness property is a property for which any finite path can be extended to a path satisfying the property.* [Foster et al.: Assertion-Based Design. 2<sup>nd</sup> Edition, Kluwer, 2010.]

In theory, liveness properties can only be falsified by an infinite simulation run.

- Practically, we often 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.

# Use of Assertions

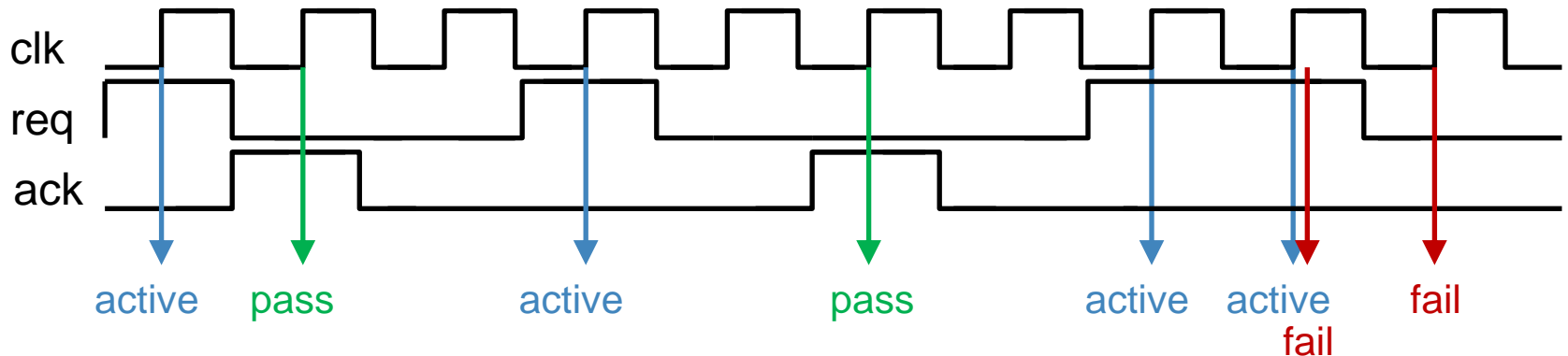
---

- Properties describe facts about a design.
- Properties can be used to write
  - Statements about the expected behaviour of the design and its interfaces
    - Combinatorial and sequential
    - (Can be used for simulation-based or for formal verification.)
  - Checkers that are active during simulation
    - e.g. protocol checkers
  - Constraints that define legal stimulus for simulation
  - Assumptions made for formal verification
  - Functional coverage points
- Remember to re-use existing assertions, property libraries or checks embedded in VIP.

# How Assertions work during Simulation

- Temporal properties can be in one of 4 states during simulation:
  - inactive (no match), **active**, **pass** or **fail**

```
property req_followed_by_ack;  
    @(posedge clk) { $rose (req) ==> ##[0:1] ack }  
end property  
p_req_ack: assert property req_followed_by_ack;
```



# Overcoming the Observability Problem



- If a design property is violated during simulation, then the DUV fails to operate according to the original design intent.

BUT:

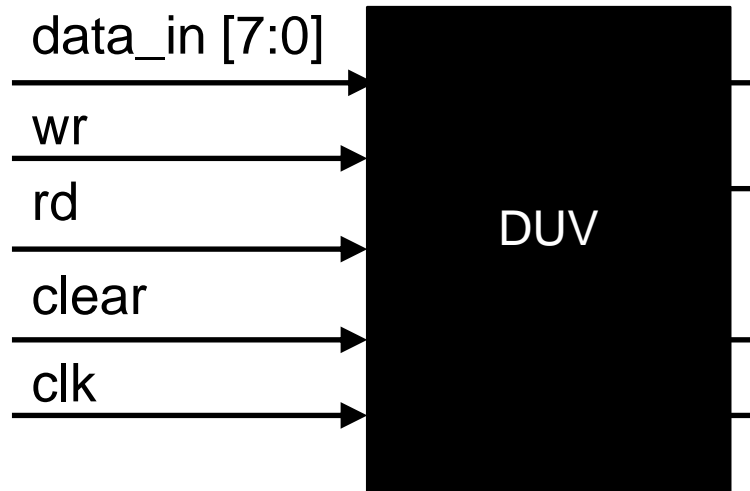
- Symptoms of low-level bugs are often not easy to observe/detect.
- Activation of a faulty statement may not be enough for the bug to propagate to an observable output.

## Assertion-Based Verification:

- During simulation assertions are continuously monitored.
- 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 bug.

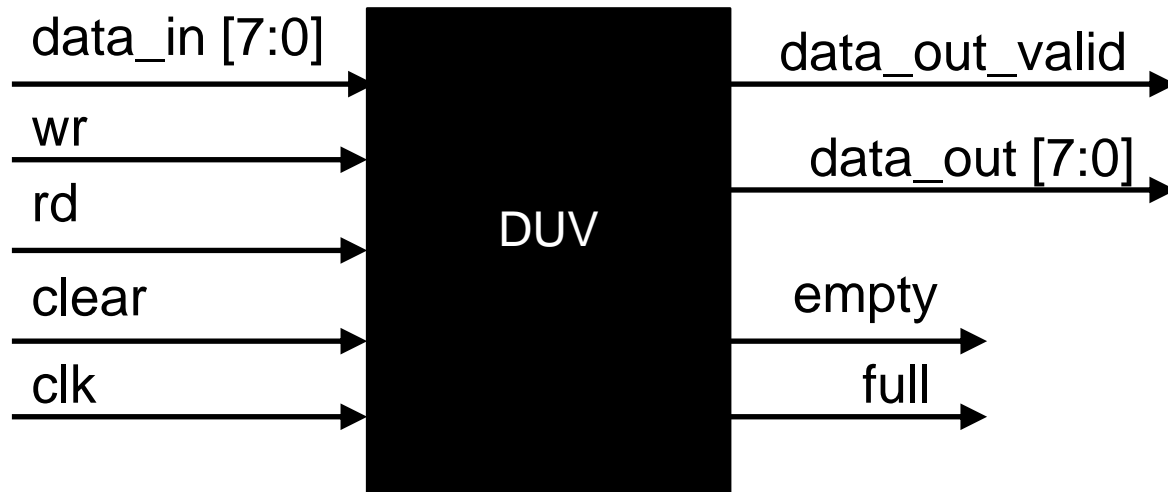
# Example FIFO DUV

# Example DUV Specification - Inputs



- Inputs:
  - wr indicates valid data is driven on the data\_in bus
  - data\_in is the data to be pushed into the DUV
  - rd pops the next data item from the DUV in the next cycle
  - clear resets the DUV

# Example DUV Specification - Outputs



## ■ Outputs:

- `data_out_valid` indicates that valid data is driven on the `data_out` bus
- `data_out` is the data item requested from the DUV
- `empty` indicates that the DUV is empty
- `full` indicates that the DUV is full

# DUV Specification

---

- High-Level functional specification of DUV
  - The design is a FIFO.
  - Reading and writing can be done in the same cycle.
  - Data becomes valid for reading one cycle after it is written.
  - No data is returned for a read when the DUV is empty.
  - Clearing takes one cycle.
  - During clearing read and write are disabled.
  - Inputs arriving during a clear are ignored.
  - The FIFO is 8 entries deep.



# Identifying Properties for the FIFO block

## Black box view:

- Empty and full are never asserted together.
- After clear the FIFO is empty.
- After writing 8 data items the FIFO is full.
- Data items are moving through the FIFO unchanged in terms of data content and in terms of data order.
- No data is duplicated.
- No data is lost.
- data\_out\_valid only for valid data, i.e. no x's in data.

An invariant property.

# Identifying Properties for the FIFO block

---

## White box view:

- The value range of the read and write pointers is between 0 and 7.
- The data\_counter ranges from 0 to 8.
- The data in the FIFO is not changed during a clear.
- For each valid read the read pointer is incremented.
- For each valid write the write pointer is incremented.
- Data is written only to the slot indicated by `nxt_wr`.
- Data is read only from the slot indicated by `nxt_rd`.
- When reading and writing the data\_counter remains unchanged.
  - What about a RW from an empty/full FIFO?

# Property Formalization

- Property Formalization Languages

- Most commonly used languages:

- **SVA** and
    - PSL [IEEE – 1850]

- Assertions can be combinatorial

```
property mutex;  
  { !(empty & full) }  
end property
```

Boolean  
expression

Temporal  
expression  
in form of an  
implication

- or temporal

```
property req_followed_by_ack;  
  @(posedge clk) { $rose (req) | => ##[0:1] ack }  
end property
```

pre-condition  
(antecedent)

main condition  
(consequent)

# Introduction to Writing Properties using SVA

---

To formalize basic properties using SVA we need to learn about:

- Sequences
  - Cycle delay and repetition
- Implications
- `$rose`, `$fell`, `$past`, `$stable`

# Sequences

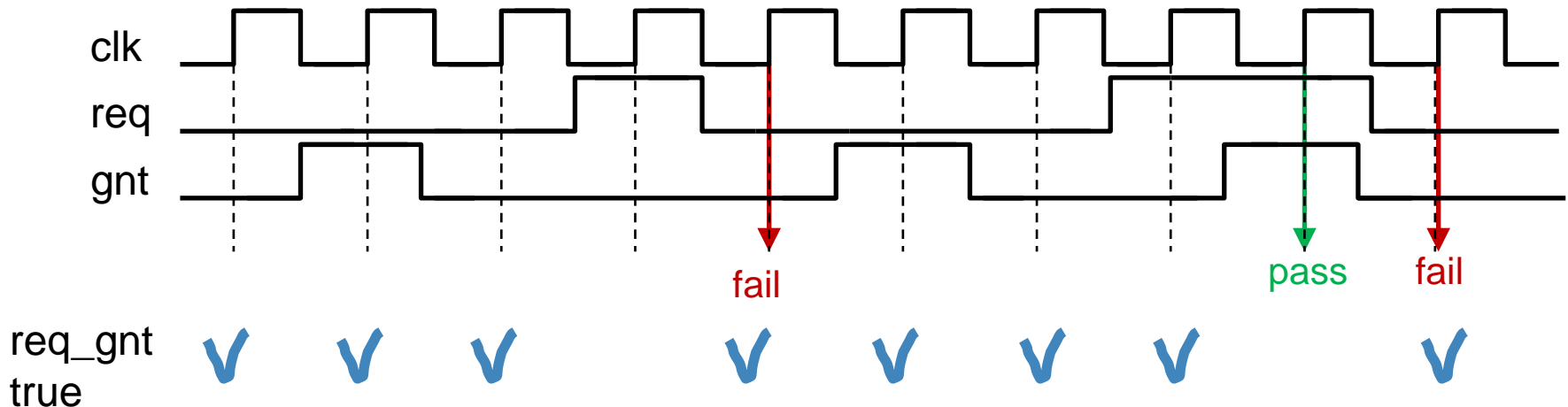
- Useful to specify complex temporal relationships.
- Constructing sequences:
  - A Boolean expression is the simplest sequence.
  - `##` concatenates two sequences.
  - `##N` cycle delay operator - advances time by `N` clock cycles.
    - `a ##3 b` `b` is true 3 clock cycles after `a`
  - `##[N:M]` specifies a range.
    - `a ##[0:3] b` `b` is true 0,1,2 or 3 clock cycles after `a`
  - `[*N]` consecutive repetition operator
    - A sequence or expression that is consecutively repeated with one cycle delay between each repetition.
      - `a [*2]` exactly two repetitions of `a` in consecutive clock cycles
  - `[*N:M]` consecutive repetition with a specified range
    - `a[*1:3]` covers `a`, `a ##1 a` or `a ##1 a ##1 a`

# Implications

- Properties typically take the form of an implication.
- SVA has two implication operators:
- $\mid\Rightarrow$  represents logical implication
  - $A \mid\Rightarrow B$  is equivalent to  $(\text{not } A) \text{ or } B$ ,  
where  $B$  is sampled one cycle after  $A$ .

non-overlapping  
implication

```
req_gnt: assert property ( req  $\mid\Rightarrow$  gnt );
```



# Implications

- SVA has another implication operator:
- $| \rightarrow$  represents logical implication
  - $A | \rightarrow B$  is equivalent to  $(\text{not } A) \text{ or } B$ ,  
where B is sampled **in the same cycle as** A.

```
req_gnt_v1: assert property ( req | => gnt );
```

```
req_gnt_v2: assert property ( req | -> ##1 gnt );
```

The overlapping implication operator  $| \rightarrow$  specifies behaviour in the same clock cycle as the one in which the LHS is evaluated.

Delay operator  $##N$  delays by N cycles, where N is a positive integer including 0.

**Both properties above are specifying the same functional behaviour.**

# Useful SystemVerilog Functions for Property Specification

- `$rose` and `$fell`
  - Compares value of its operand in the current cycle with the value this operand had in the previous cycle.
- `$rose`
  - Detects a transition to 1 (`true`)
- `$fell`
  - Detects a transition to 0 (`false`)
- Example:

```
assert property ( $rose(req) ==> $rose(gnt) );
```



# Useful SystemVerilog Functions for Property Specification

- `$past (expr)`
  - Returns the value of `expr` in the previous cycle.
  - Example:

```
assert property ( gnt |-> $past(req) );
```

- `$past (expr, N)`
  - Returns the value of `expr` `N` cycles ago.
- `$stable (expr)`
  - Returns true when the previous value of `expr` is the same as the current value of `expr`.
  - Represents: `$past (expr) == expr`

# Property Formalization

# Formalization of key DUV Assertions

- System Verilog Assertion for:
  - Empty and full are never asserted together.

Is this a safety or a liveness property? Why?

```
property not_empty_and_full;  
@(posedge clk) !(empty && full);  
endproperty  
mutex : assert property (not_empty_and_full);
```

This label is useful for debug.

# Formalization of key DUV Assertions

- System Verilog Assertion for:
  - Empty and full are never asserted together.

This is a safety property!

```
property not_empty_and_full;  
@(posedge clk) $onehot0({empty, full});  
endproperty  
mutex : assert property (not_empty_and_full);
```

Alternative encoding: **\$onehot0** returns true when zero or one bit of a multi-bit expression is high.

# Formalization of key DUV Assertions

- System Verilog Assertion for:
  - After clear the FIFO is empty.

```
property empty_after_clear;  
@(posedge clk) (clear |-> empty);  
endproperty  
a_empty_after_clear : assert property (empty_after_clear);
```

**Beware of property bugs!** Know your operators:

- seq1 |-> seq2, seq2 starts in last cycle of seq1 (overlap)
- seq1 |==> seq2, seq2 starts in first cycle after seq1

We need: `@(posedge clk) (clear |==> empty);`

# Formalization of key DUV Assertions

- System Verilog Assertion for:
  - On empty after one write the FIFO is no longer empty.

```
property not_empty_after_write_on_empty;  
@ (posedge clk) (empty && wr ==> !empty);  
endproperty  
a_not_empty_after_write_on_empty : assert property  
    (not_empty_after_write_on_empty);
```

Assertions can be  
monitored during  
simulation.

Assertions can also  
be used for formal  
property checking.

**Challenge:**  
**There are many more interesting assertions.**

# Corner Case Properties

- **FIFO empty:** When the FIFO is empty and there is a write at the same time as a read (from empty), then the read should be ignored.

```
property empty_write_ignore_read;  
  @(posedge clk) (empty && wr && rd | =>  
                  data_counter == $past(data_counter)+1);  
endproperty  
a_cc1 : assert property (empty_write_ignore_read);
```

- **FIFO full:** When the FIFO is full and there is a read at the same time as a write, then the write (to full) should be ignored.

```
property full_read_ignore_write  
  @ (posedge clk) {full && rd && wr | =>  
                  data_counter == $past(data_counter)-1};  
endproperty  
a_cc2: assert property (full_read_ignore_write);
```

# All my assertions pass – what does this mean?

- Remember, simulation can only show the presence of bugs, but never prove their absence!
- An assertion has never “fired” - what does this mean?
  - Does not necessarily mean that it can’t be violated!
    - **Unless simulation is exhaustive..., which in practice it never will be.**
  - It might not have fired **because it was never active.**
  - Most assertions have the form of **implications.**
  - Implications are satisfied when the antecedent is false!
    - These are **vacuous** passes.
    - **We need to know how often the property passes non-vacuously!**
- How do you know your assertions are correctly expressing what you intended?



# Assertion Coverage

- Measures how often an assertion condition has been evaluated.
  - Many simulators count only **non-vacuous** passes.
  - Option to add assertion coverage points using:

```
assert property ( (sel1 || sel2) ==> ack );  
cover property ( sel1 || sel2 );
```

- Coverage can also be collected on sub-expressions:

```
cover property ( sel1 );  
cover property ( sel2 );
```

# Costs and benefits of ABV

- Costs include:
  - Simulation speed
  - Writing the assertions
  - Maintaining the assertions
- Benefits include:
  - Explicit expression of designer intent and specification requirements
    - Specification errors can be identified earlier
    - Design intent is captured more formally
  - Enables finding more bugs faster
  - Improved localisation of errors for debug
  - Promote measurement of functional coverage
  - Improved qualification of test suite based on assertion coverage
  - Facilitate uptake of formal verification tools
  - Re-use of formal properties throughout design life cycle

Intellectual step of  
property capture **forces** you  
to think earlier!

# Do assertions really work?

- **Assertions are able to detect a significant percentage of design failures:**

[Foster et al.: Assertion-Based Design. 2<sup>nd</sup> Edition, Kluwer, 2010.]

- **34%** of all bugs were found by assertions on DEC Alpha 21164 project [Kantrowitz and Noack 1996]
- **17%** of all bugs were found by assertions on Cyrix M3(p1) project [Krolnik 1998]
- **25%** of all bugs were found by assertions on DEC Alpha 21264 project - The DEC 21264 Microprocessor [Taylor et al. 1998]
- **25%** of all bugs were found by assertions on Cyrix M3(p2) project [Krolnik 1999]
- **85%** of all bugs were found using OVL assertions on HP [Foster and Coelho 2001]

- **Assertions should be an integral part of a verification methodology.**

# 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

# Summary

---

In ABV we have covered:

- What is an assertion?
- Use and types of assertions
- Safety and Liveness properties
- Introduction to basics of SVA as a property formalization language
- Importance of Assertion Coverage
- Costs vs benefits of using assertions