COMS30026 Design Verification

Assertion-based Verification (Part II)

Kerstin Eder

Trustworthy Systems Laboratory

https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/

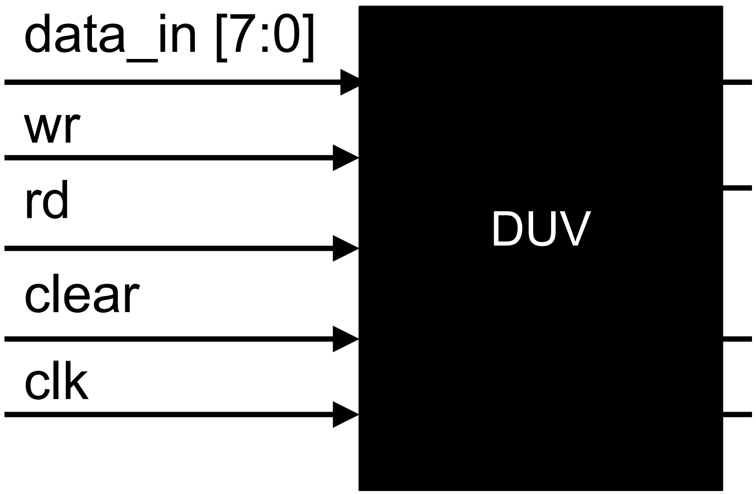
University of Bristol

Department of COMPUTER SCIENCE

CASE STUDY:
IDENTIFYING DUV PROPERTIES

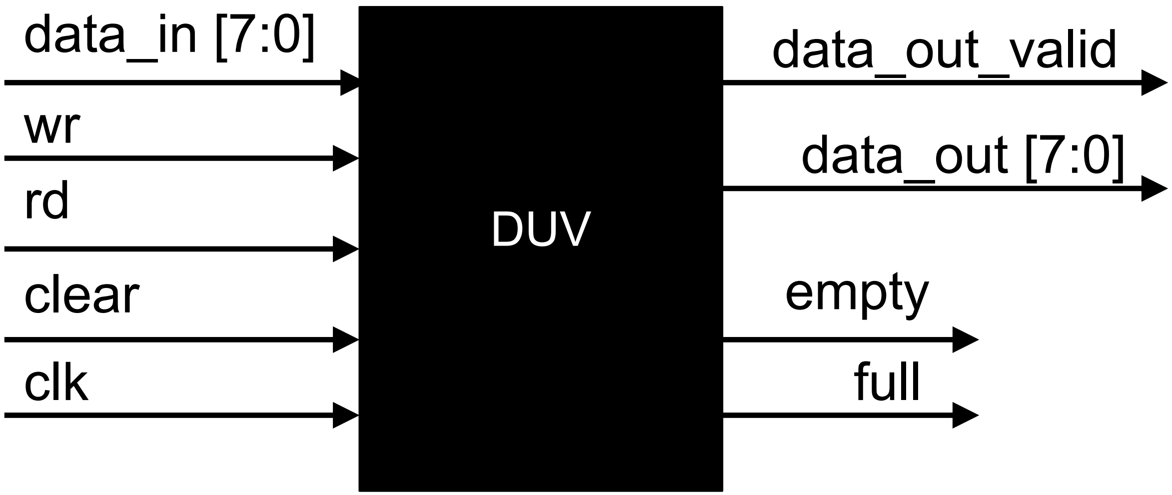
Example FIFO DUV

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

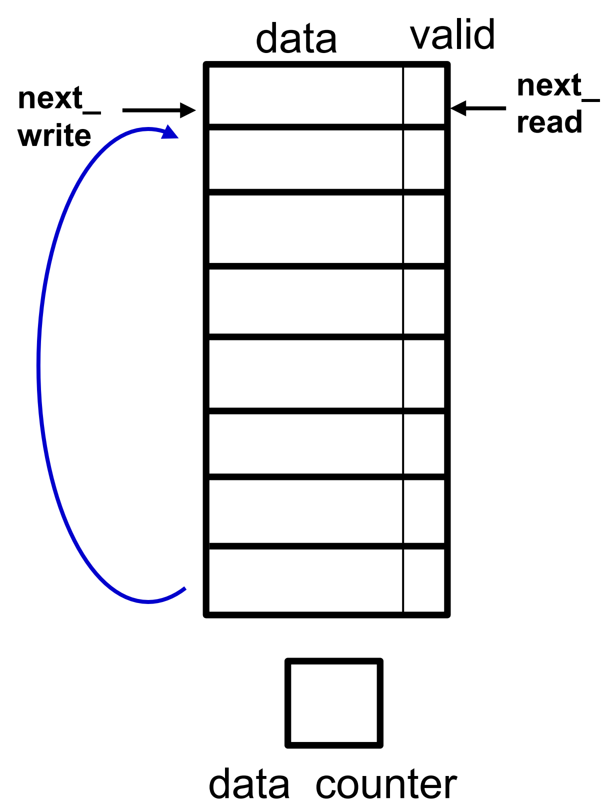
An invariant property.

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.

Implementation of the FIFO

- The actual implementation of the FIFO design is a circular buffer:
  - Logic to determine if the FIFO is full or empty: `next_read` and `next_write` as well as the `data_counter`
  - `valid` bits need to be implemented to indicate whether a data entry is valid or not
  - Wrap conditions need to be implemented to achieve a *circular buffer.*



Identifying Properties for the FIFO block

White box view:

- The value range of the `next_read` and `next_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 `next_read` pointer is incremented.
- For each valid write the `next_write` pointer is incremented.
- Data is written only to the slot indicated by `next_write`.
- Data is read only from the slot indicated by `next_read`.
- When reading and writing in the same cycle the `data_counter` remains unchanged.
  - What about a RW from an empty/full FIFO?

FORMALIZING PROPERTIES

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

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

  - and there are also temporal assertions:

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

Temporal expression in form of an implication

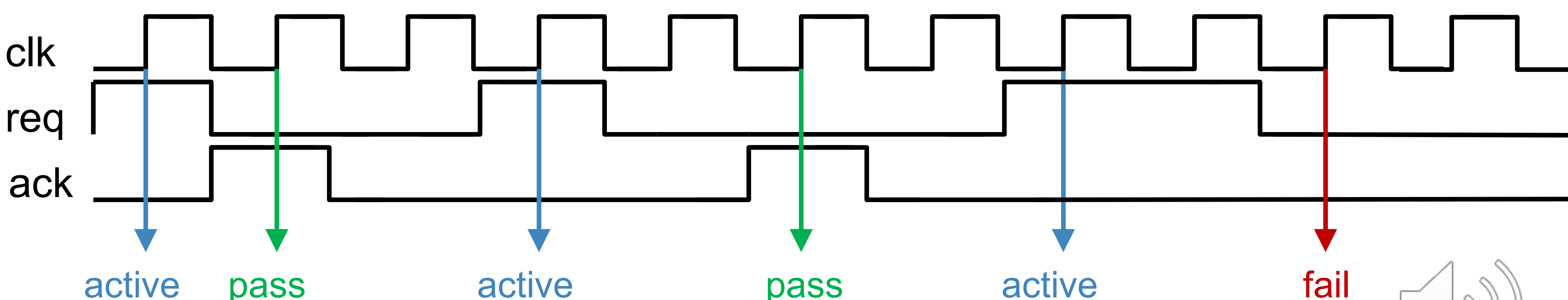
pre-condition (antecedent)

main condition (consequent)

How Assertions work during Simulation

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

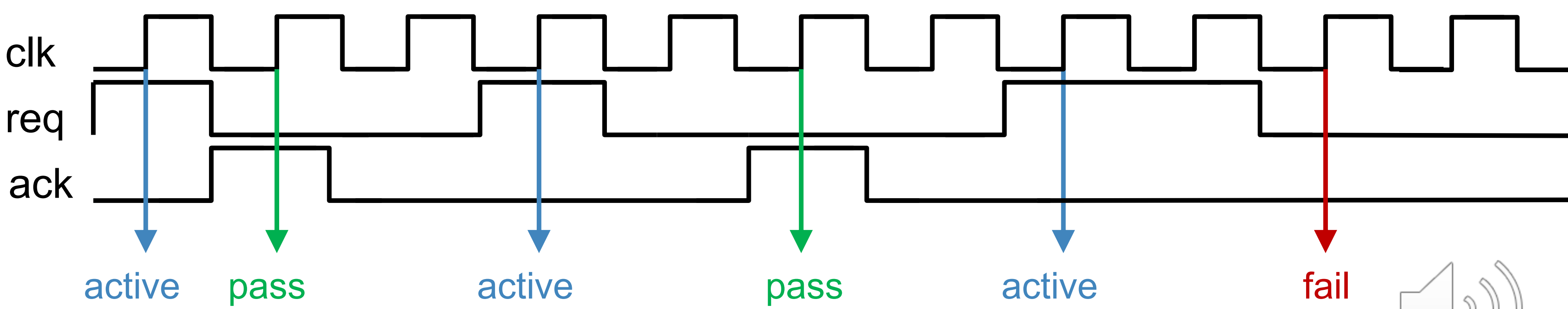
```
property req_followed_by_ack;
    @(posedge clk){ $rose (req) |=> ##[0:1] ack }
end property
p_req_ack: assert property req_followed_by_ack;
```



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;
```



Writing Properties using SVA

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

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

Implications

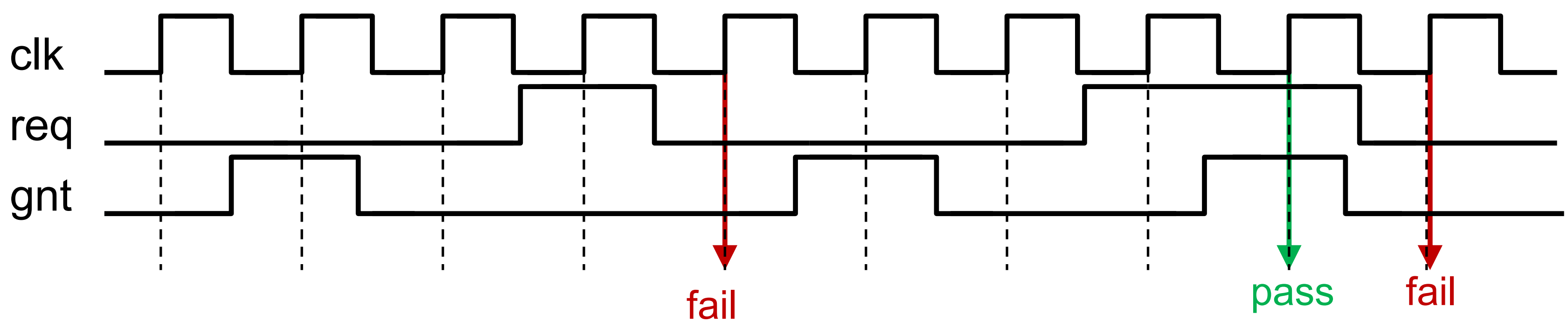
- Properties typically take the form of an implication.
- SVA has two implication operators:
- `|=>` represents logical implication

non-overlapping implication

  - `A|=>B` is equivalent to `(not A) or B`,

    where `B` is sampled one cycle after `A`.

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

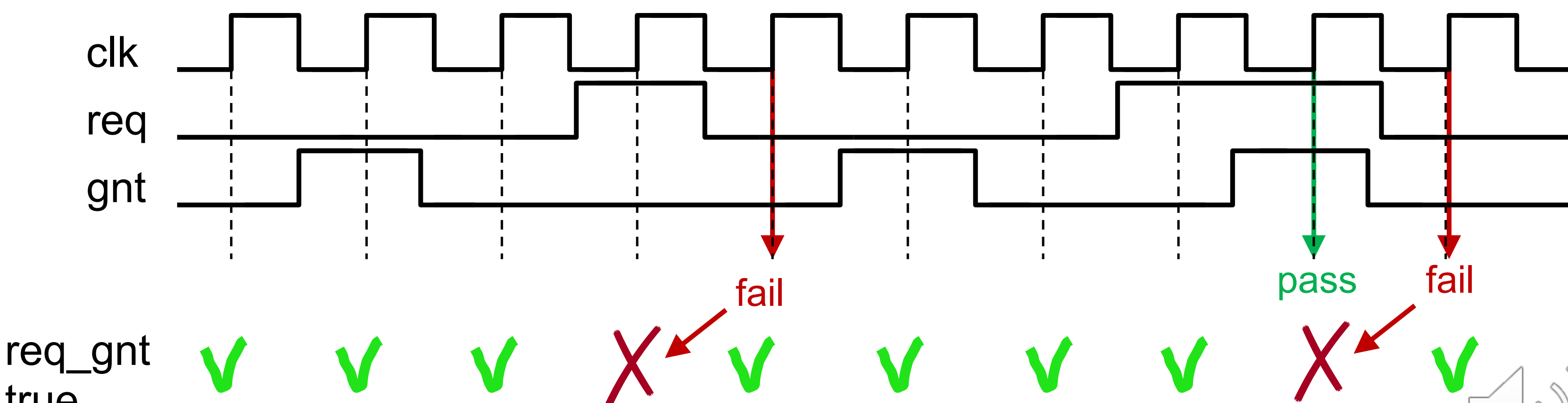


Implications

- Properties typically take the form of an implication.
- SVA has two implication operators:
- `|=>` represents logical implication
  - `A|=>B` is equivalent to `(not A) or B`,
    where `B` is sampled one cycle after `A`.

non-overlapping implication

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



Implications

- SVA has another implication operator:
- `|->` represents logical implication
  - `A|->B` is equivalent to `(not A) 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 |-> 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 behavior.

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 timing delay range.
    - `a ##[0:3] b` b is true 0,1,2 or 3 clock cycles after a
  - `[*N]` consecutive repetition operator
    - Allows to specify 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 within a specified range
    - `a[*1:3]` covers `a`, `a ##1 a` or `a ##1 a ##1 a`

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`

CASE STUDY:
FORMALIZING PROPERTIES

Example FIFO DUV

Formalization of key DUV Assertions

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

This is a safety property!

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.

```
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

- 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);
```

- 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);
```

USING ASSERTIONS

All my assertions pass – now what?

- 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 pre-condition is false!
    - These are **vacuous** passes.
    - **We need to know how often the property passes non-vacuously!**

Assertion Coverage

- Measures how often an assertion condition has been evaluated.
  - Many simulators count only **non-vacuous** passes.

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

  - Add assertion coverage points using:

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

  - Coverage can also be collected on sub-expressions:

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

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 failure.

Costs and benefits of ABV

- Costs include:

- Benefits include:

Costs and benefits of ABV

- Costs include:
  - Simulation speed
  - Writing the assertions
  - Maintaining the assertions
- Benefits include:

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
  - ABV enables finding more bugs faster
  - Improved localisation of bugs for debug using assertion labels
  - ABV promotes the measurement of functional coverage
  - Improved qualification of test suite based on assertion coverage
  - ABV facilitates the uptake of formal verification
  - Re-use the 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 etal.: Assertion-Based Design. 2nd 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 design 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 capture intent and to assert correctness of the new features
- 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 of assertions
- Safety and liveness properties
- Implementation vs specification assertions
- Introduction to basics of SVA as a property formalization language
- Importance of Assertion Coverage
- Costs vs benefits of using ABV