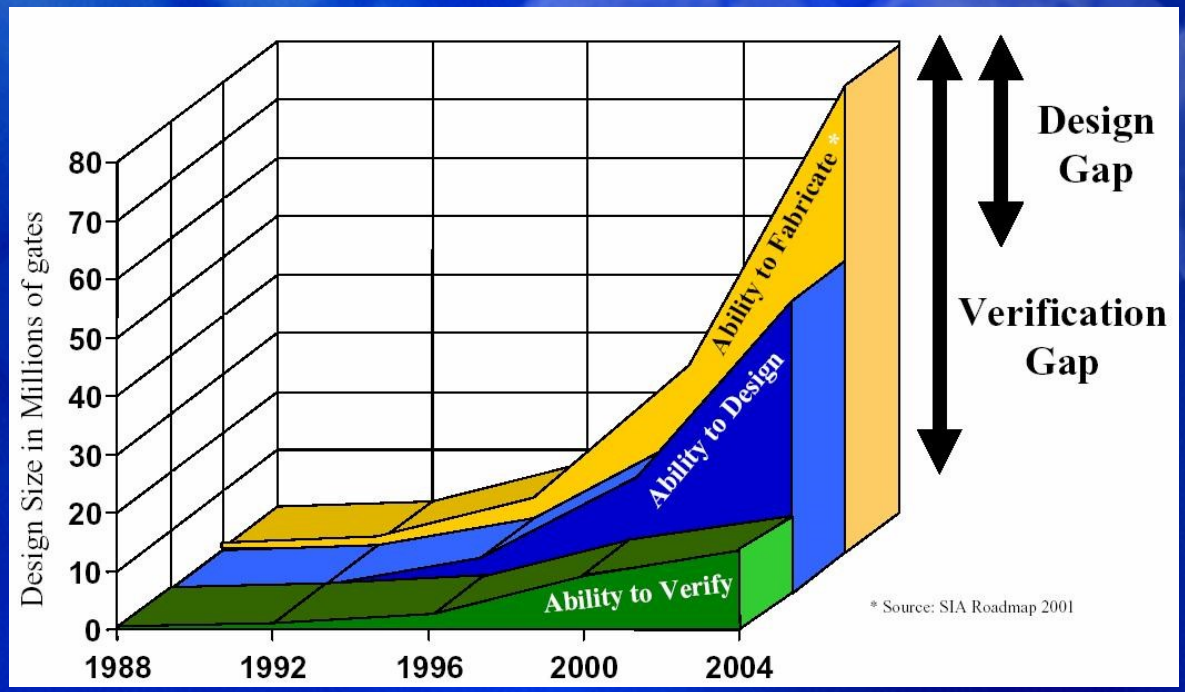
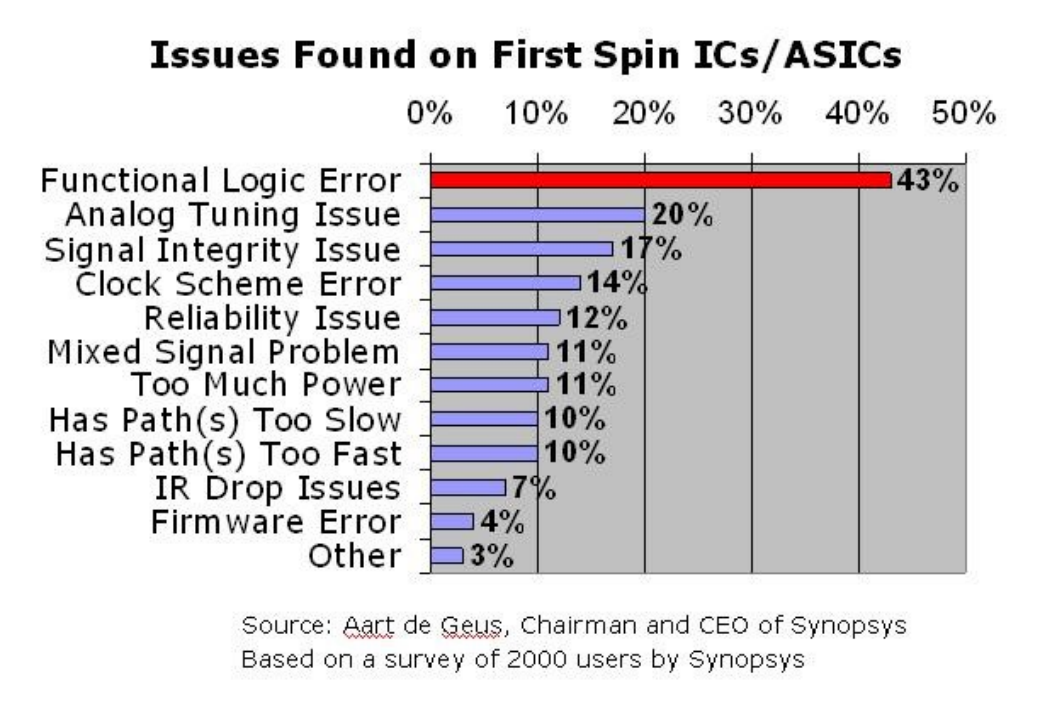
Assertion Based Verification

The Design and Verification Gap



- The number of transistors on a chip increases approximately 58% per year, according to Moore's Law.
- The design productivity, facilitated by EDA tool improvements, grows only 21% per year.
- These numbers have held constant for the last two decades.

What is the hardest challenge in design flow?



- The main problem in the design flow is to clean on the functional bugs.
- Functional Logic Error grow to be 72% at 2002, and 75% at 2004.
- Top causes are:
  - 12.7% - Goofs (for example, typos)
  - 11.4% - Miscommunication.
  - 9.3% - Micro Architecture bugs.
  - 9.3% - Logic and micro changes.

Agenda

- What is verification and coverage driven?
- ABV – Assertion Based Verification
  - What is assertion
  - Examples
- Questions
- Backup – SystemVerilog assertions
  - SystemVerilog overview
  - Advantages of SystemVerilog assertions
  - Examples

Coverage Driven Verification

How to verify that the design meets the verification goals?

- Define all the verification goals up front in terms of "functional coverage points.“
  - Each bit of functionality required to be tested in the design is described in terms of events, values and combinations.
- Functional coverage points are coded into the HVL (Hardware Verification Language) environment (e.g., Specman ‘e’).
  - The simulation runs can be measured for the coverage they accomplish.
- Focus on tests that will accomplishing the coverage ("coverage driven testing“).
  - By Fixing bugs, releasing constraints, and improving the test environment.
  - In a coverage-driven project, each verification engineer is focused on achieving the tangible results for his features.

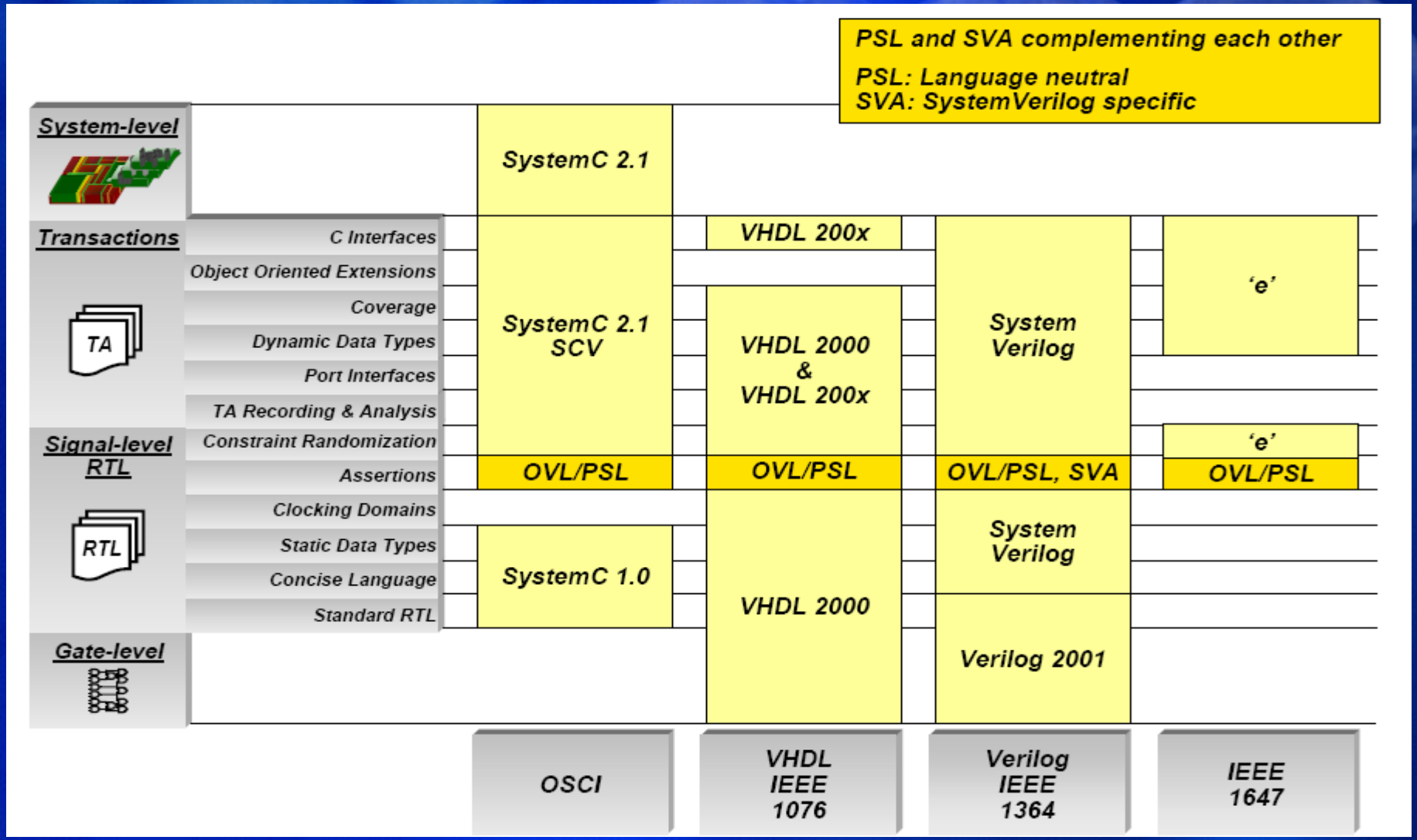
Common testing schemes

- Traditional simulation-based functional verification is good at validating baseline functionality

Disadvantages :

- The specification may not be complete
  - usually describes only the normal operating behavior
- Hard to set up all specified operations strictly from the chip inputs
- Hard to check all specified behavior strictly from the chip outputs
- Hard to locate the sources of bugs exhibited at the chip outputs
- Certain types of bug behaviors may not propagate all the way to the chip outputs
- Certain types of implementation errors are difficult to detect using functional tests

Hardware Languages for Design and Verification

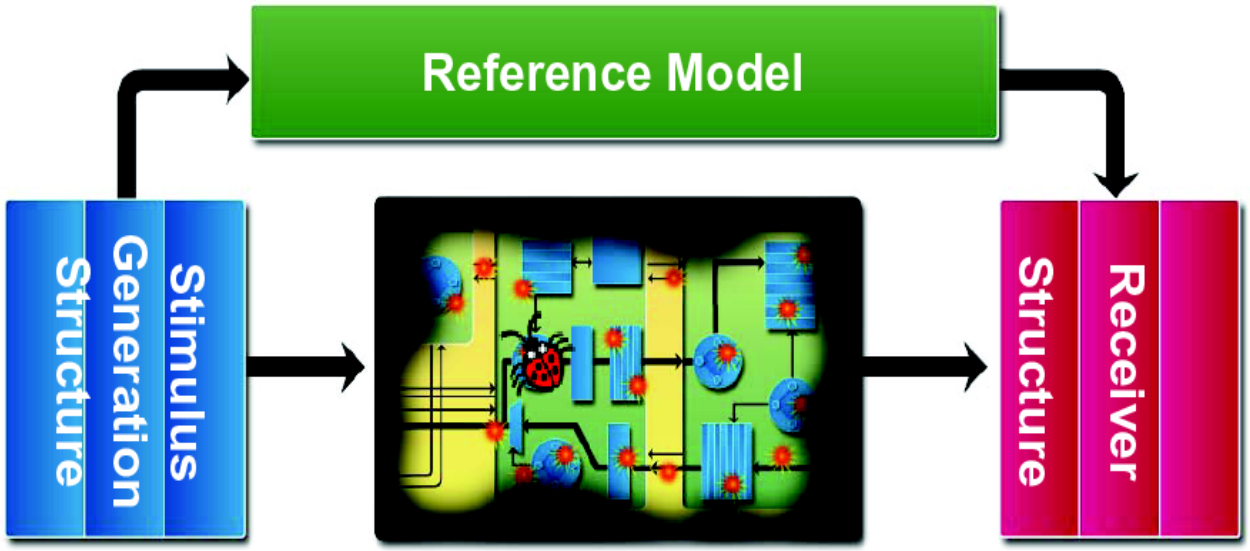


Assertions

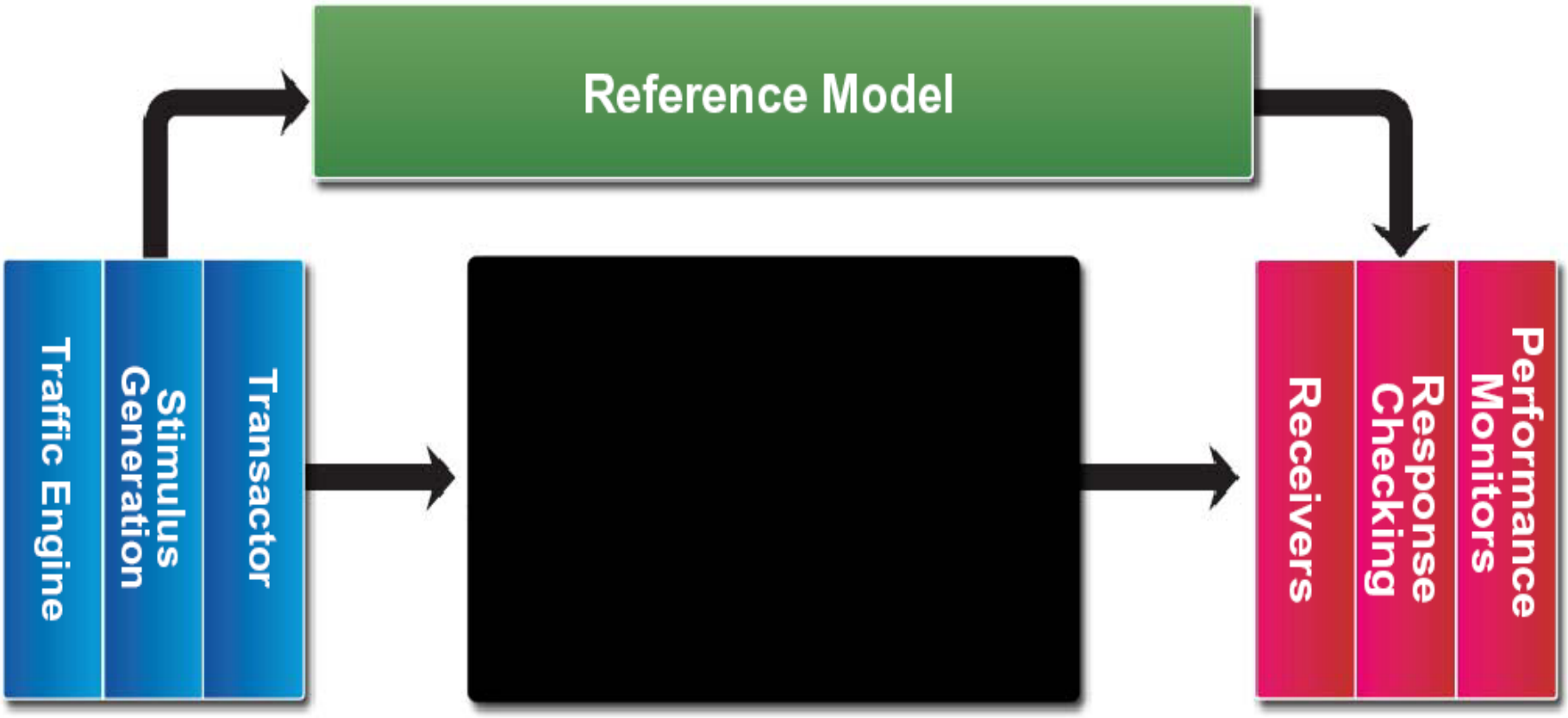
- Assertions capture knowledge about how a design should operate
- Assertions are vital to increasing both the observability and the controllability of a design
- Each assertion specifies both legal and illegal behavior of a circuit structure (which can be an RTL element, an interface, a controller, etc.) inside the design
- Assertion in English:
  - The fifo should not overflow
    (i.e., when it is full, write should not happen) or
  - TDO signal should be less then…
- Will be used in coverage-driven verification techniques.

Assertion Based Verification

- Increased Observability
- Methodology to detect bugs quickly and close to source

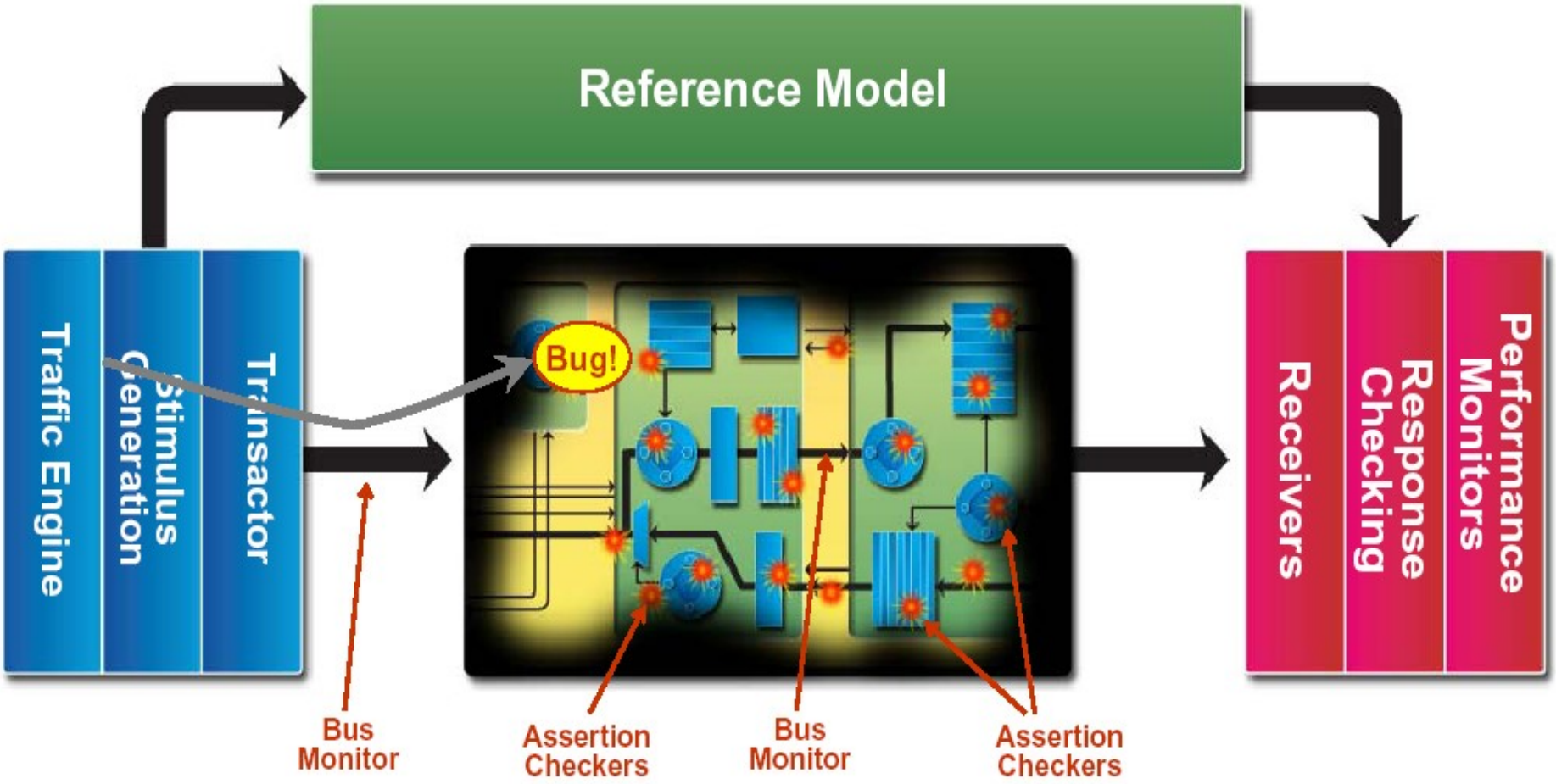


Traditional Verification



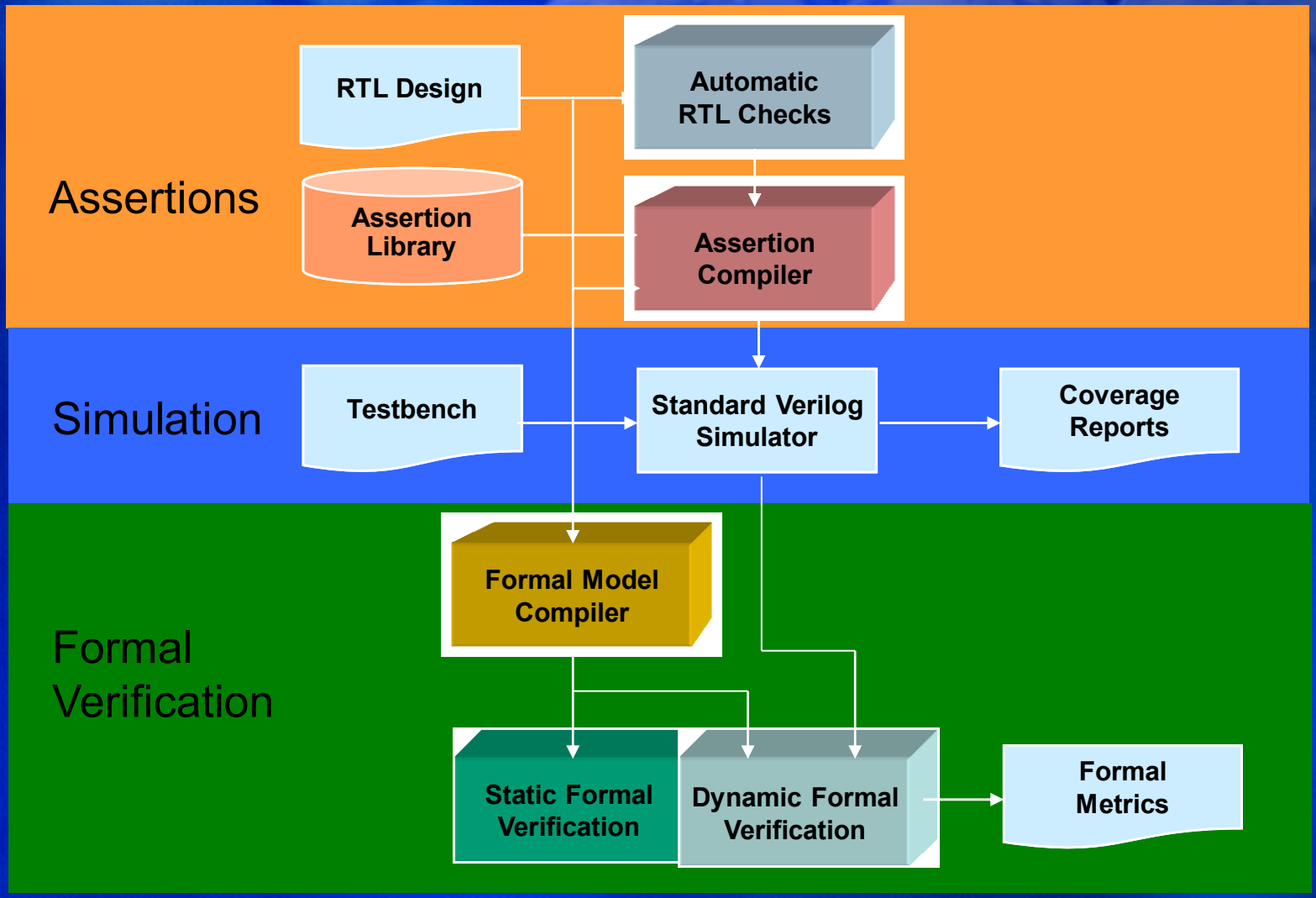
- To detect a bug using traditional methods, you must
  - Apply vectors to stimulate incorrect behavior
  - Apply vectors to propagate and detect incorrect behavior
  - Trace back through waveforms to locate the source

Observability

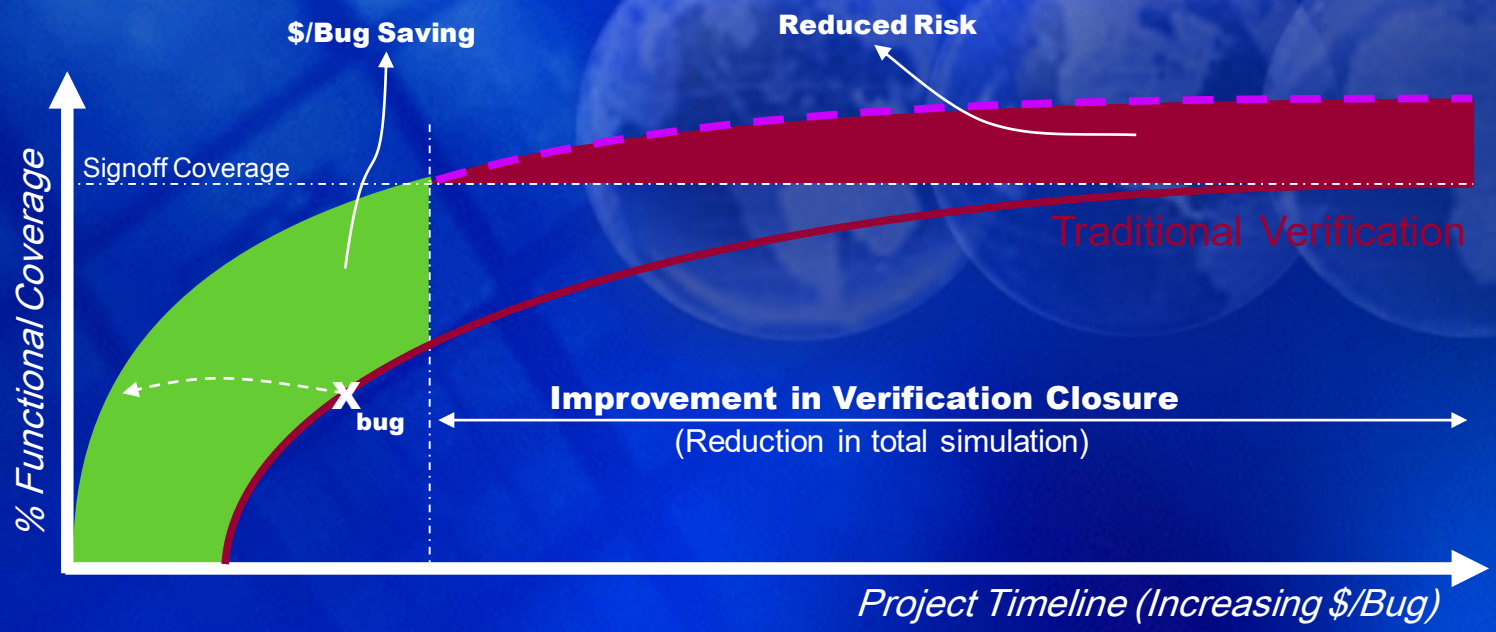


- Black Box: Observe bug when propagated to output
  - What if it never propagates to an observable point?
- ABV: Catch bugs at or near source of the problem

Complete ABV Flow



Performance of the ABV Flow



- **Assertion based verification flow provides**
  - **Find bugs faster with assertions**
  - **Find more bugs with verification hot spots**
  - **Reduce simulation cycles**
  - **Adopt the AVB/CDV methodology incrementally**

Conclusion

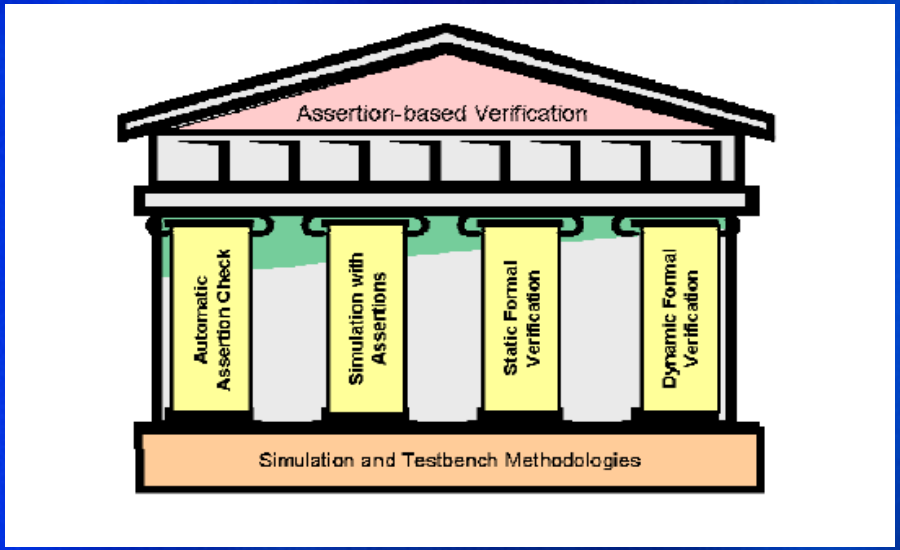
- Assertion-based verification is a shift in verification methodology
- Traditional functional verification tries to stimulate the design and observe the responses from the outside
- observability and controllability are so low that pseudo-random simulation can no longer exercise the internals of the device sufficiently

- ABV technologies and methodologies are developed to :
  - Zero in to the structure of the design
  - Responsibility of verification is also shifting from over-the-wall verification team to involve designers as well
  - Design knowledge becomes a critical criterion for successful functional verification
  - With assertions, we can detect bugs sooner
  - With formal verification, we have more direct control of the verification effort

Four Pillars of ABV

- Assertions capture the design intent by embedding them in a design and having them monitor design activities

- Pillar 1: Automatic Assertion Check
- Pillar 2: Static Formal Verification
- Pillar 3: Simulation with Assertions
- Pillar 4: Dynamic Formal Verification



SystemVerilog – assertions overview

Key benefits of SVA can be summarized as follows:

- Familiar language and syntax
- Less assertion code
- Simple hookup between assertions and the test
- No special interfaces
- Customized messaging and error severity levels
- Ability to interact with Verilog and C functions
- No mismatch results between simulation and formal tools

SystemVerilog – assertions overview

- SystemVerilog provides two types of assertions:
  - *Immediate*
  - *Concurrent*

Both intended to convey the intent of the design and identify the source of a problem as quickly and directly .

Immediate assertions

```
immediate_assert_statement ::=
      assert ( expression ) [[ pass_stmt ] else fail_stmt ][1]
```

SVA: procedural assertions

- Written to the code to be simulated – applied in simulation-based verification.
- Example:

```
assert (WR == 1'b1 && CS == 1'b0)
  $display ("INFO: memory ready"); // pass clause
  else $display ("ERROR: cannot write"); // fail clause
```

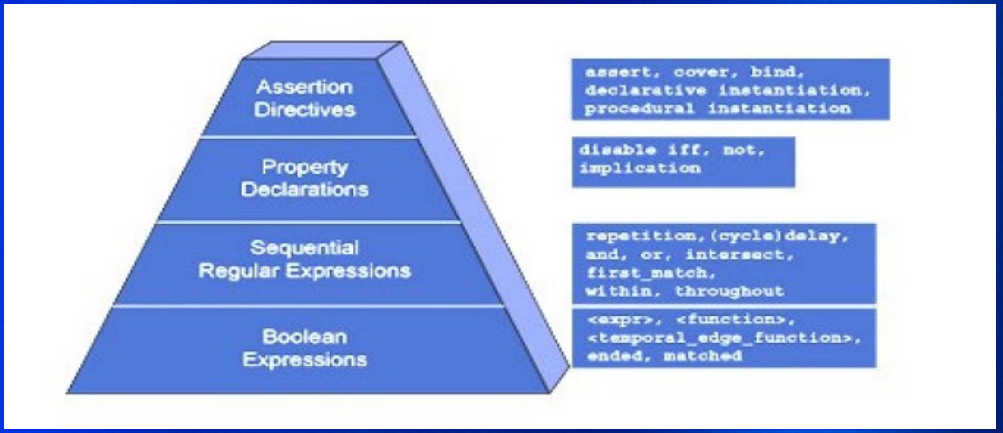
Concurrent vs. immediate

Two important differences:

- Concurrent assertions allow the specification of a temporal behavior to be checked vs. combinational condition of immediate.
- Concurrent assertions can also be instantiated *declaratively* as a module-level statement (similar to a continuous assignment).

Concurrent types:

- Boolean expressions
- Sequential expressions
- Property Declarations
- Assertion Directives



Boolean expressions

- The Boolean expressions layer is the basic layer.
- Boolean expressions specifies the values of elements at a particular point in time

Syntax :

```
concurrent_assert_statement ::=
          assert property ( sequential_expr_or_property )
             [[ pass_stmt ] else fail_stmt ]
```

SVA: declarative assertions

- Declarative assertions (similar to modules) – applied in property checking.
- Syntax:

```
assert property (property_definition_or_instance) action

action ::= [statement] | [statement] else statement
```

SVA: declarative assertions, sequences

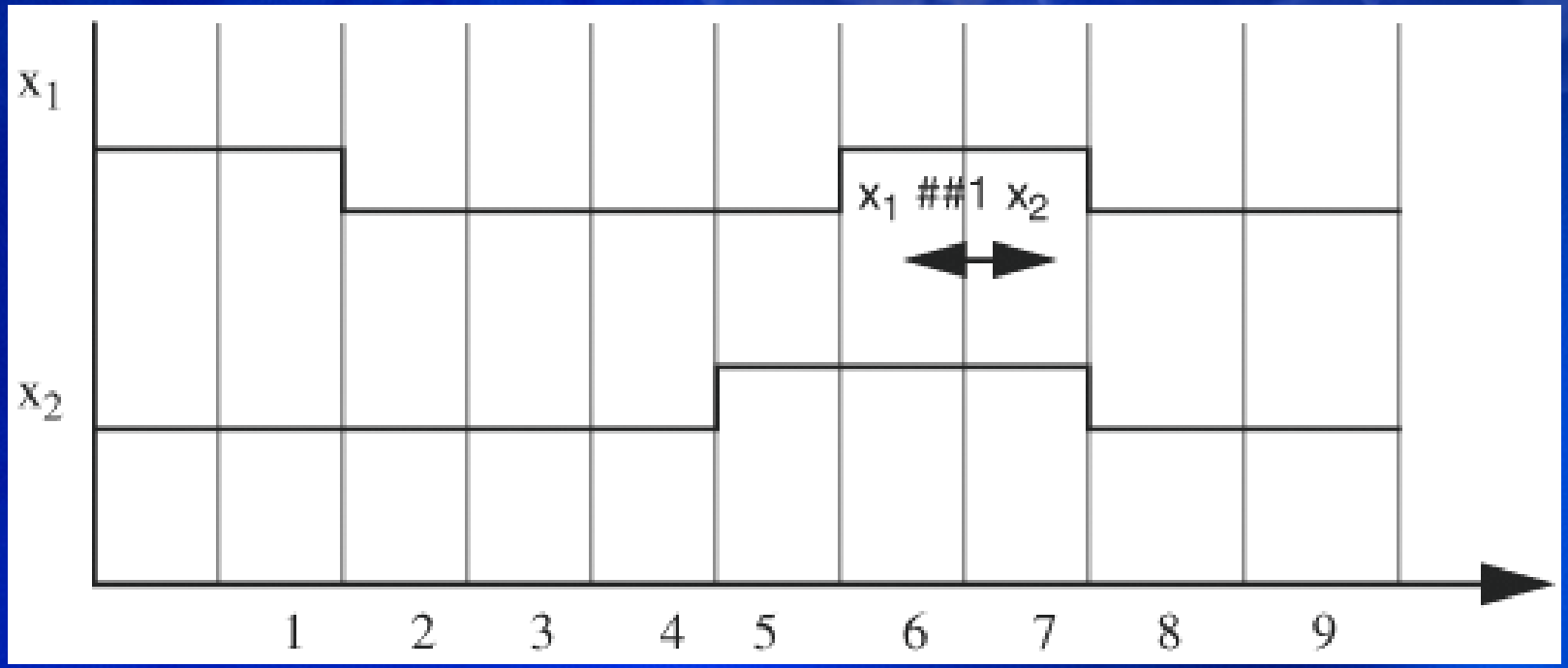
- Declarative assertions describe temporal behavior
- Main concept is a sequence:
- Sequence = $\{(B_i, T_i), i \in V\}$,
- B_i is Boolean expression at time step T_i, and V is a set of integers (e.g. clock cycle index)

SVA: sequences and waveforms

- SVA sequence constructor: ##N, where N is the delay
- x ##4 z is equivalent to (x,t),(z,t+4)
- Ranges can be described
- E.g. x ##[1,3] z stands for

  x ##1 z or x ##2 z or x ##3 z
- Many waveforms generally corresponding to a sequence!

SVA: sequences and waveforms

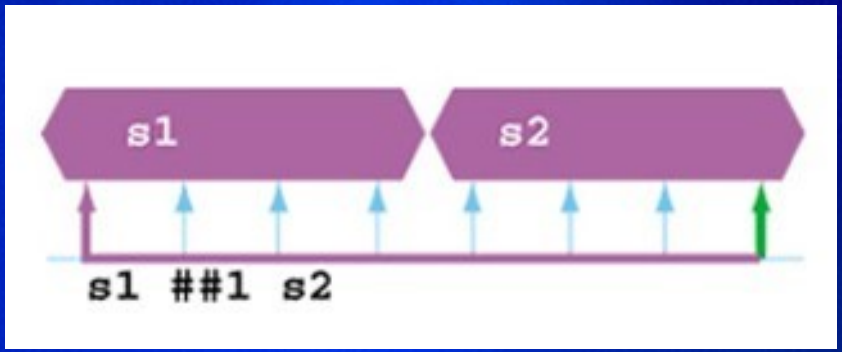


Important to understand that in SystemVerilog each element of a sequence may be either a Boolean expression or another sequence.

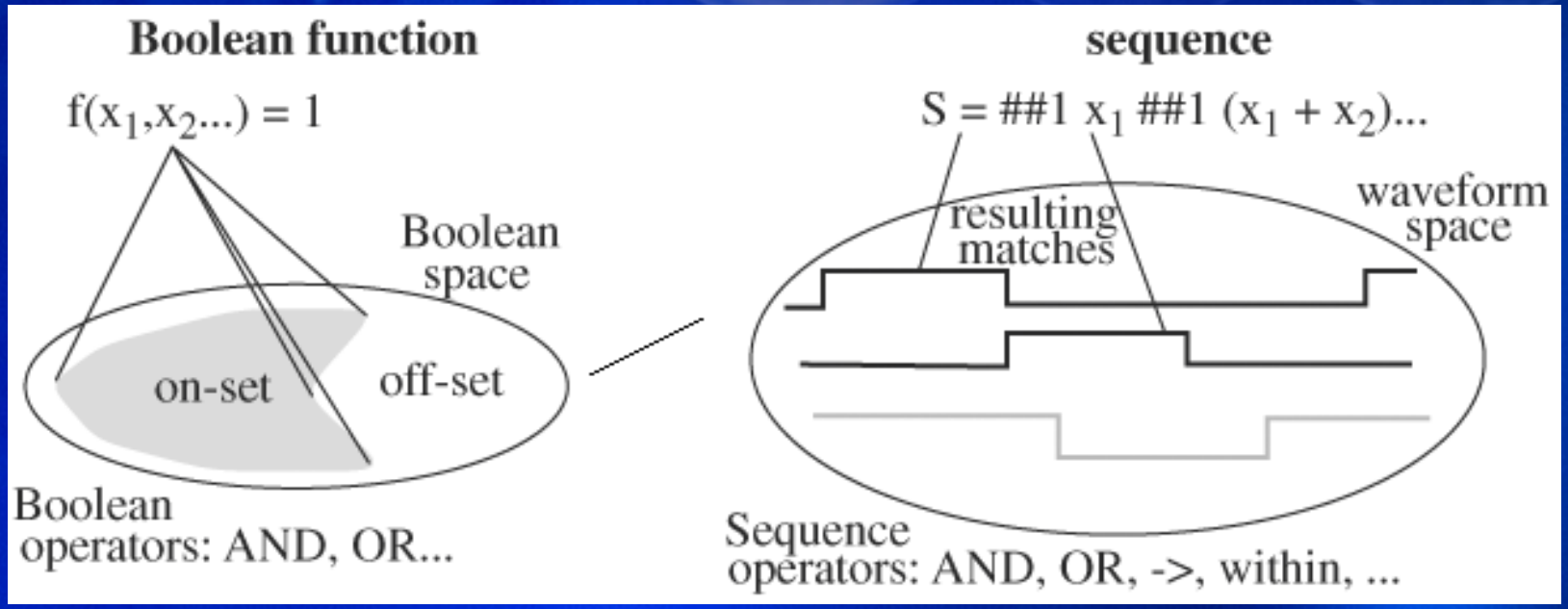
Thus, the expression

s1 ##1 s2

means that sequence s2 begins on the clock after sequence s1 ends.

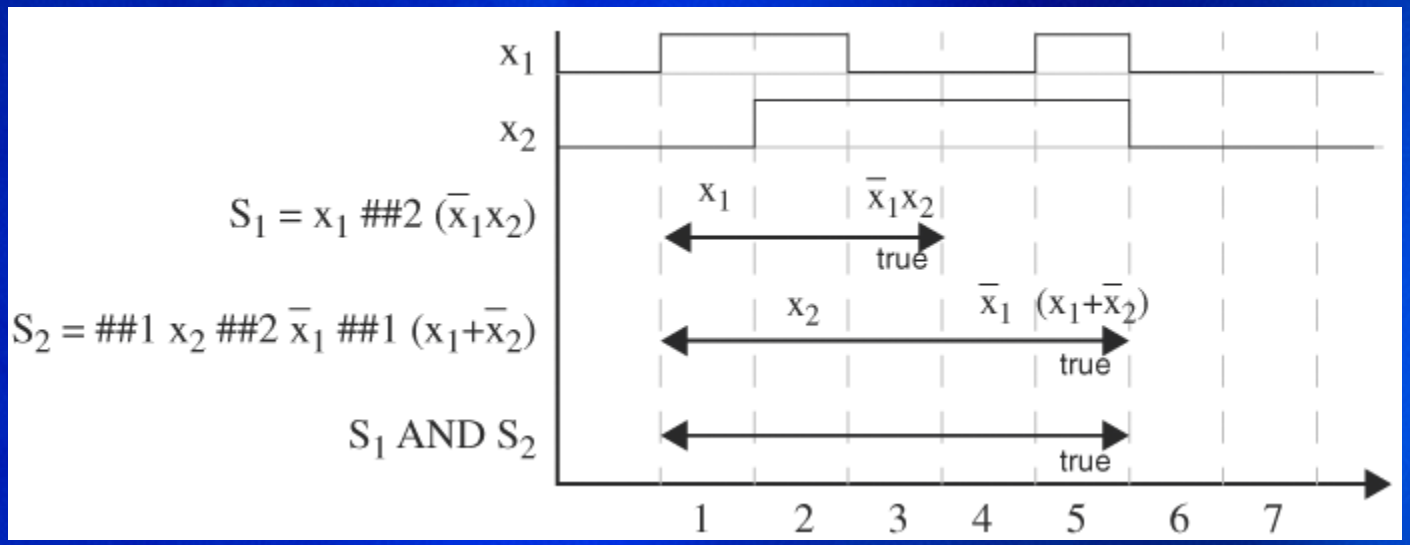


Analogy with Boolean functions



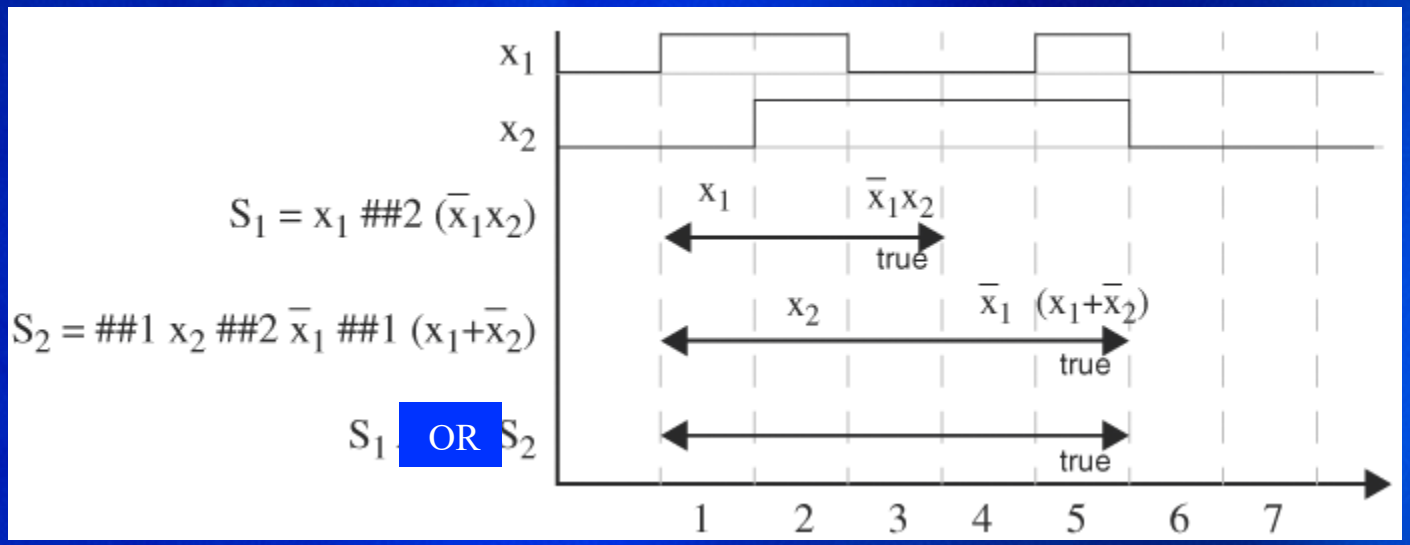
SVA. Sequence Operators: AND

- s1 AND s2 is true if s1 and s2 have both become true:



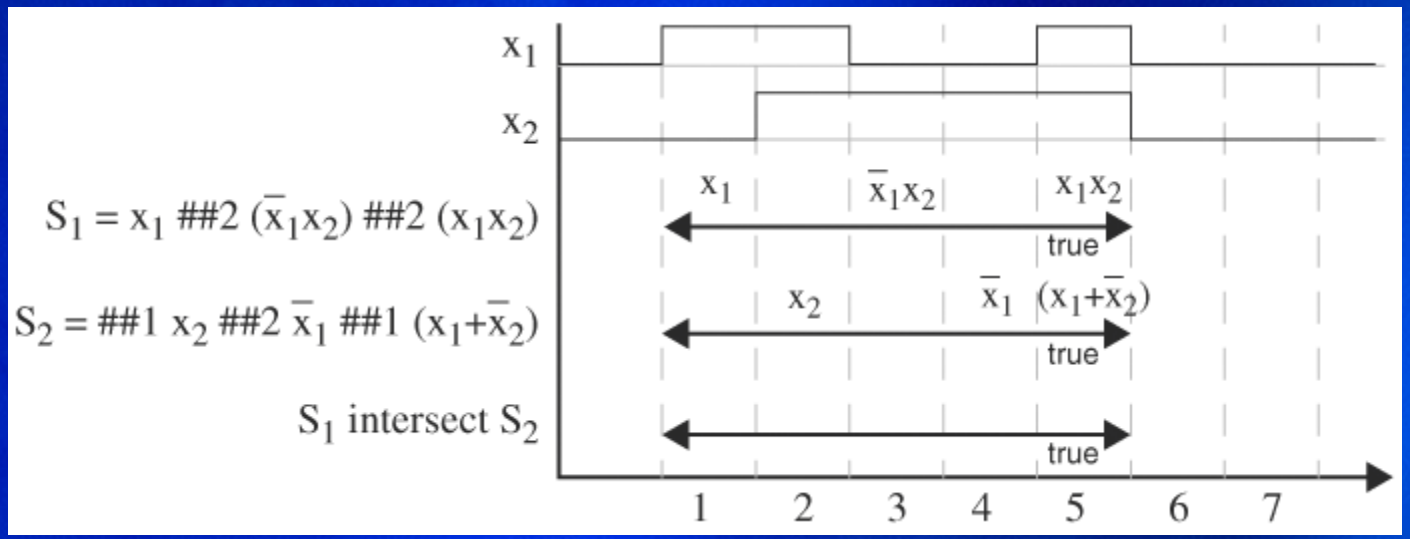
SVA. Sequence Operators : OR

- s1 OR s2 is true if s1 or s2 has become true:



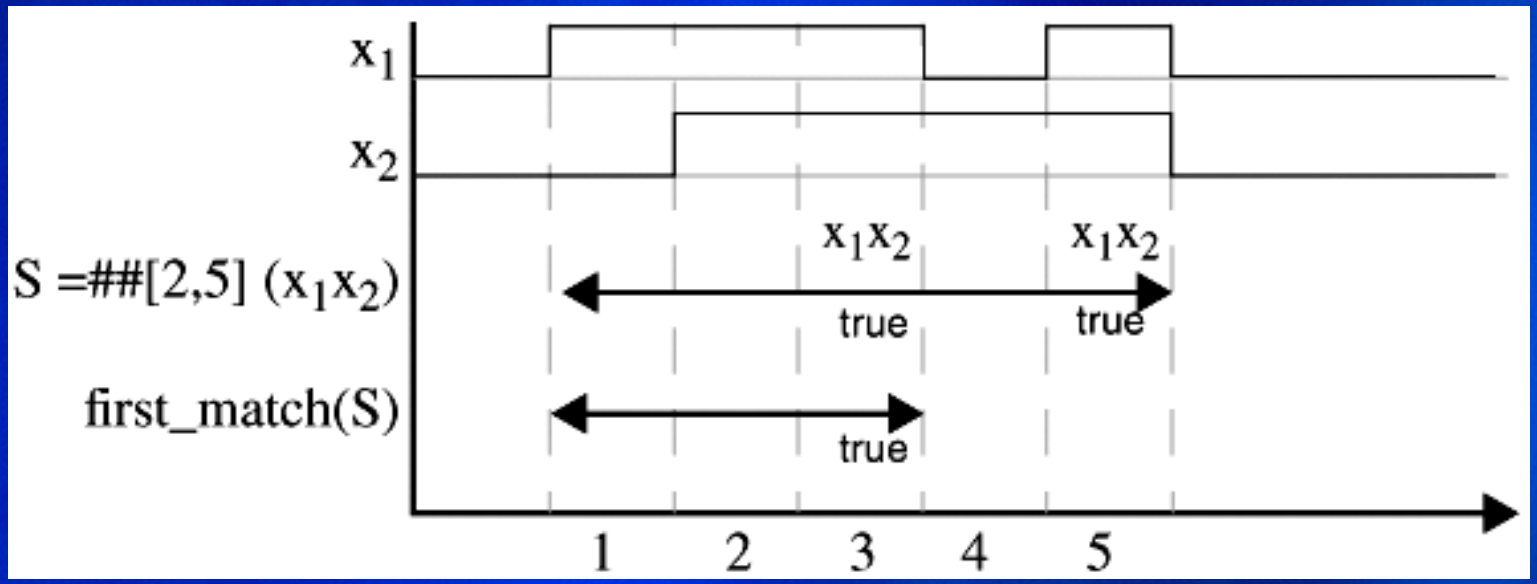
SVA. Sequence Operators: intersect

- Similar to AND, but requires that the sequences begin and end at the same time:



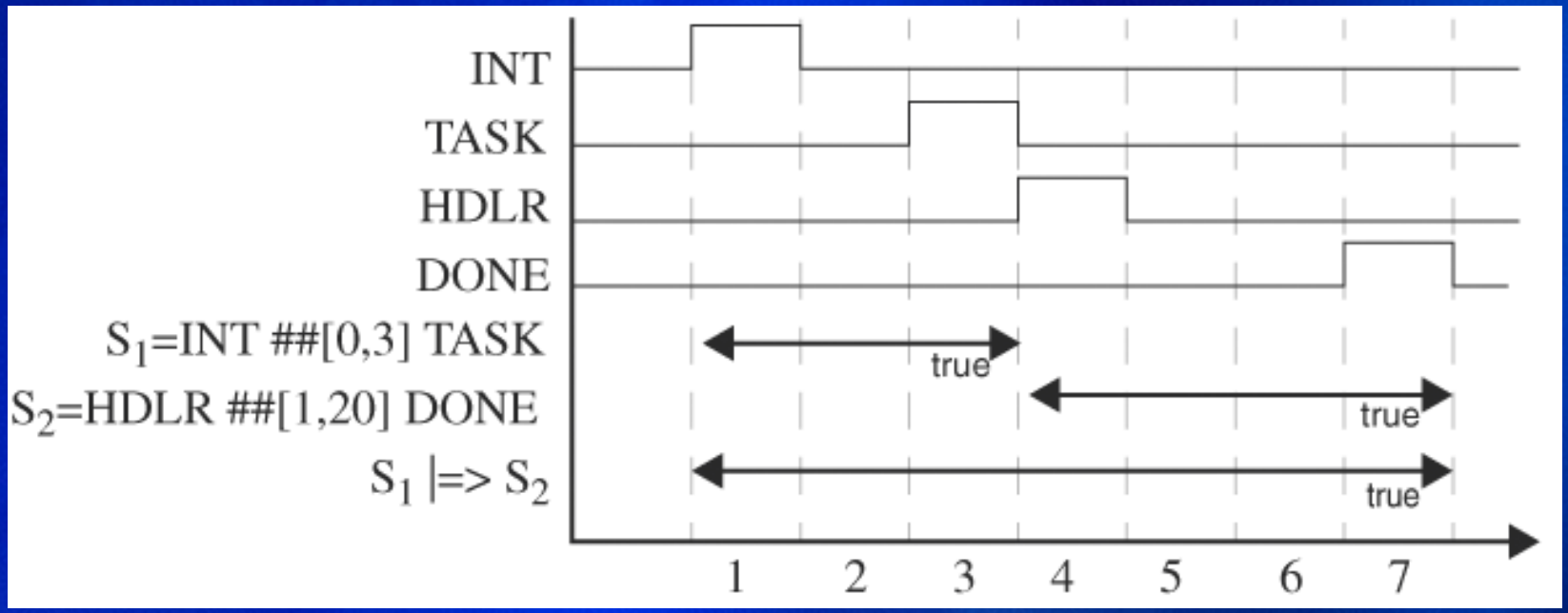
SVA. Sequence Operators: first_match

- Is true when a sequence becomes true the first time:



SVA. Operations: implication

- S1 → S2, i.e. ¬S1 + S1·S2
- If S1 becomes true at time t then start evaluating S2, whose result determines the result of the implication.
- If S1 is false, then the implication is true.



SVA. Sequence Operators: throughout

- S = {(B_i,T_i),i ∈V}
- E throughout S is{(E · B_i,T_i),i ∈V}
- E.g. (x_1 + x_2) throughout (##1 x_3 ##2 x_4) is equivalent to:

  (##1 (x_1 + x_2)x_3 ##2 (x_1 + x_2)x_4)

- **The optional list of arguments allows for specification of sequences as a generic temporal relationship .**

```
sequence seq1 (a,b);
        a ##2 b;
endsequence
```

- Some operations on sequences :

Operation	Syntax	Explanation
Concatenation	seq1 ##1 seq2	seq2 begins on the clock after seq1 completes
Overlap	seq1 ##0 seq2	seq2 begins on the same clock on which seq1 completes
Ended Detection	seq1 ##1 seq2.ended	seq2 completes on the clock after seq1 completes, regardless of when seq2 started.
Repetition	seq1[*n:m]	Repeat seq1 a minimum of n and maximum of m times. May result in multiple matching sequences
First Match Detection	first_match(seq1)	If seq1 has multiple matches, use the first and ignore the rest
Or	seq1 or seq2	Compound sequence that matches when either seq1 or seq2 matches
And	seq1 and seq2	Matches when one sequence matches on or after the other sequence also matches
Length-matching And	seq1 intersect seq2	Matches on cycles at which both seq1 and seq2 match
Condition qualification	cond throughout seq	cond is true for every cycle of seq
Within	seq1 within seq2	seq1 starts on or after seq2 and ends on or before the end of seq2

Property declarations

- More general behaviors to be specified.
- Properties allow you to invert the sense of a sequence
  - As when the sequence should *not* happen.
- Disable the sequence evaluation.
- Specify that a sequence be implied by some other occurrence.

Syntax :

```
property p1;
   @(posedge clk) disable iff (test) not abort_seq;
endproperty
```

"as long as the test signal is low,
check that the abort_seq sequence does not occur."

Assertions directives (Procedural)

- System implication - "when this happens, then that will happen“ thus require the writer to specify the trigger assertion.
- Declarative assertions, require additional work by the designer to use them effectively.

Consider a finite state machine design.

Assertion

"when in state ACK, if foo is high, then req should be held low for 5 clocks."

```
always @(posedge clk)
      case (st)
      ACK:
            if (foo == 1)
              begin // when in this state, req should be low for 5 clocks
              ...
              end
      ...
      endcase
```

State machine cont'

This assertion could be coded declaratively as:

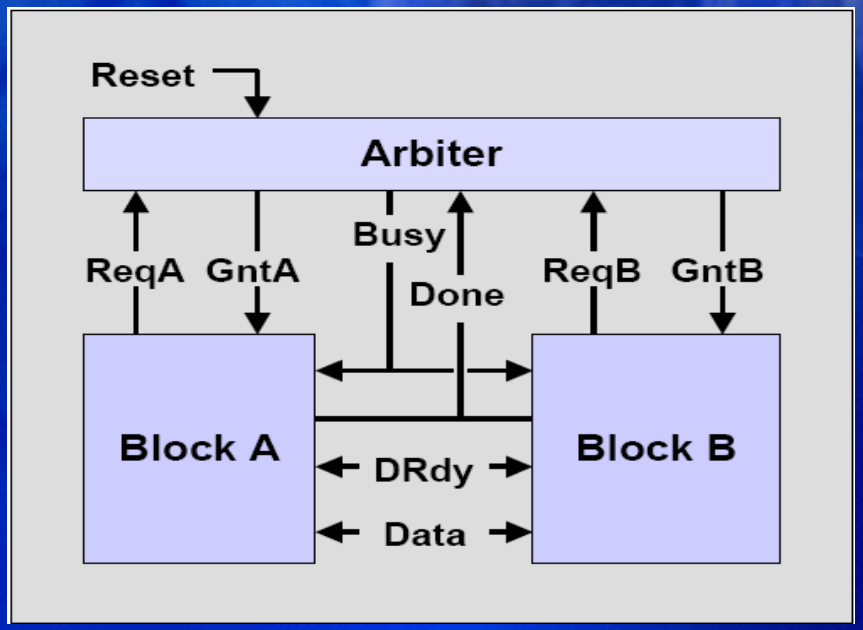
```
P4: assert property(
    @(posedge clk) (st == ACK) && (foo == 1) |-> !req[*5]);
```

The fact that the assertions are syntactically part of the language allows them to be embedded *procedurally* in the RTL code and automatically infer this information, as in:

```
always @(posedge clk)
        case (st)
        ACK:
              if (foo == 1)
                begin
                   P5: assert property (!req[*5]);
                   ...
                end
```

Assertion example

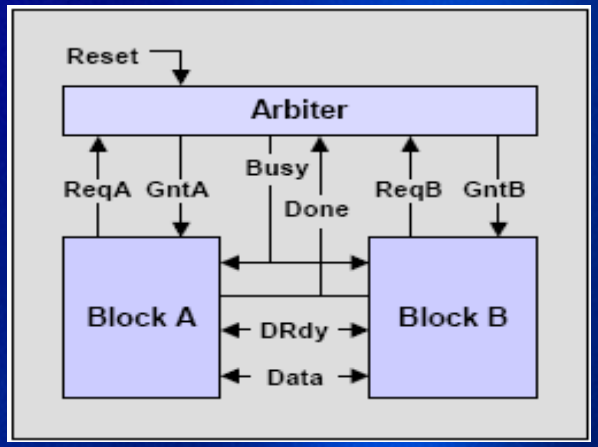
Two blocks A,B exchange data via a common bus :



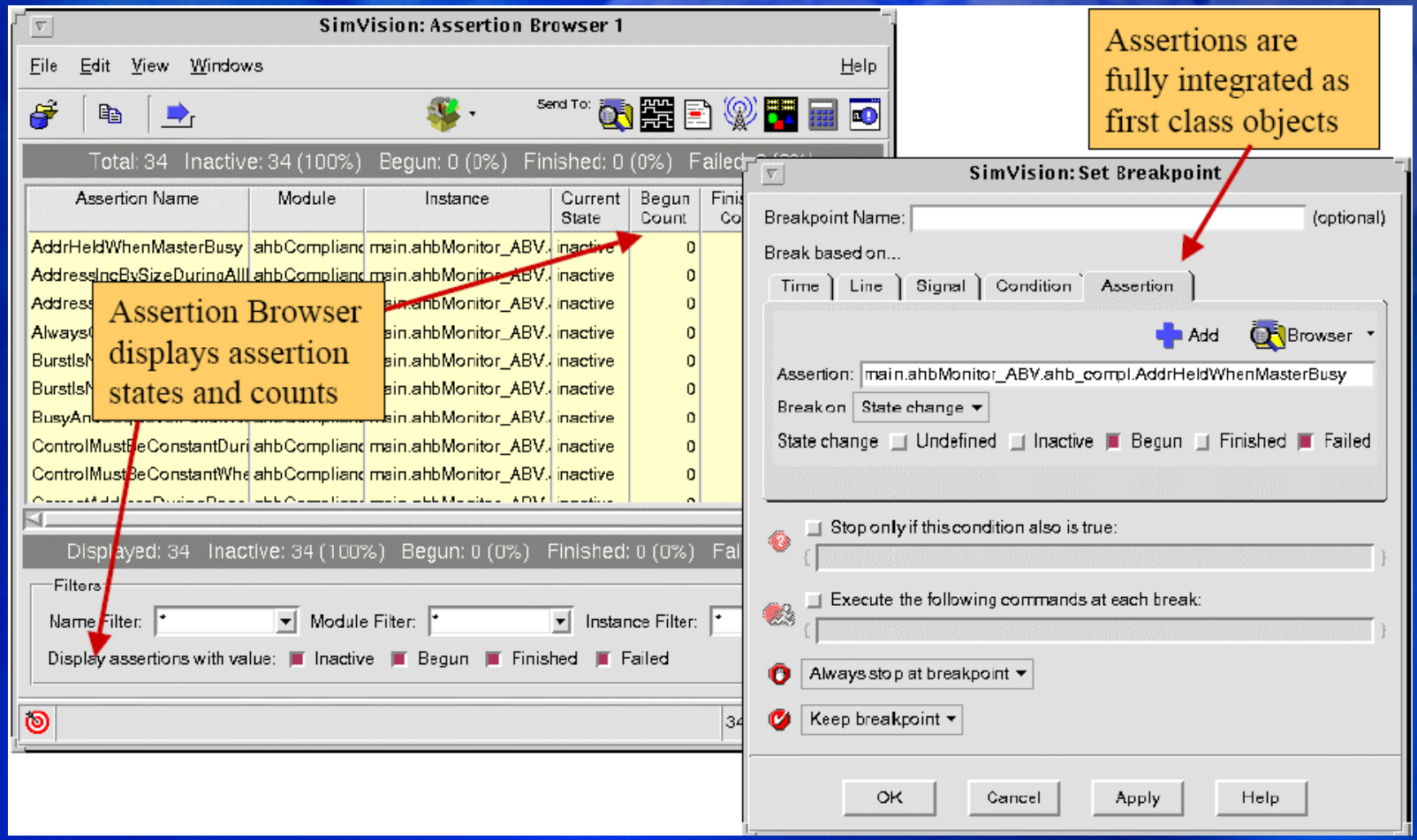
- A and/or B sends ‘Req’ to the Arbiter.
- Arbiter sends ‘Gnt’ back to A or B, making it Master.
- Arbiter sets ‘Busy’ while A or B is Master.
- Master sets ‘DRdy’ when Data is on the bus.
- Master sets ‘Done’ in the last cycle of a grant.

Some Assertions to Check :

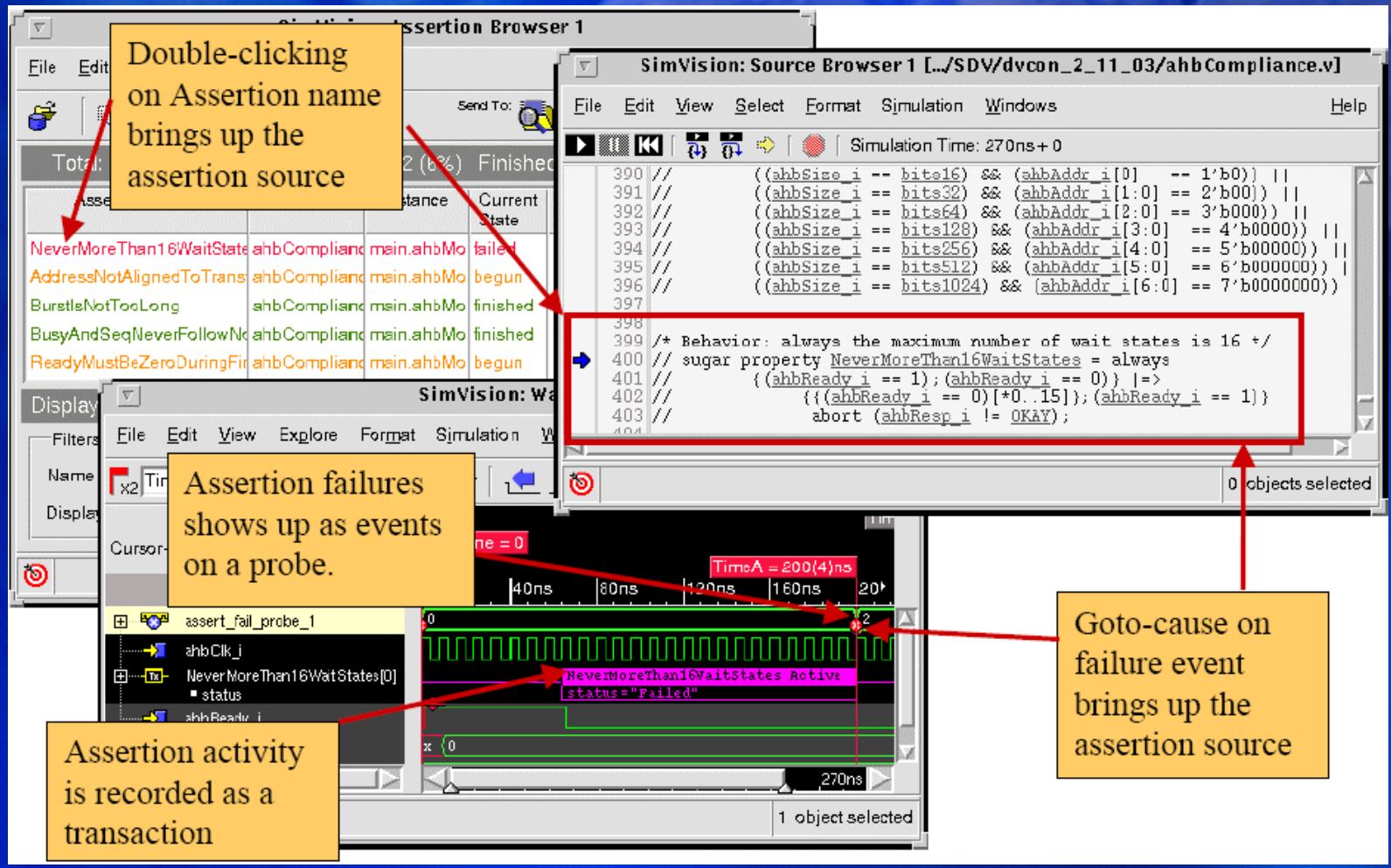
- A Grant never occurs without a Request.
  - Assert never GntA && !ReqA
- If A (B) receives a Grant, then B (A) does not.
  - Assert always GntA -> !GntB
- A (B) never receives a Grant in two successive cycles.
  - Assert never GntA && nextGntA
- A Request is eventually followed by a Grant
  - Assert always ReqA -> eventually GntA



Tool Support



Assertion Debug Environment



Conclusion

- SystemVerilog is a unified language that contains design and verification constructs
- Assertions communicate to the test bench and vice-versa.
- Customize messaging resulting from the assertions
- Create calls to Verilog and C functions dependent on the success or failure of the assertions
- Minimize assertion code by inferring design control structures

Furthermore, SystemVerilog assertions will eliminate the debugging of mismatches between simulation and formal tools

Had Fun ?!..... Come any time

Barak & Daniel

VLSI Seminar