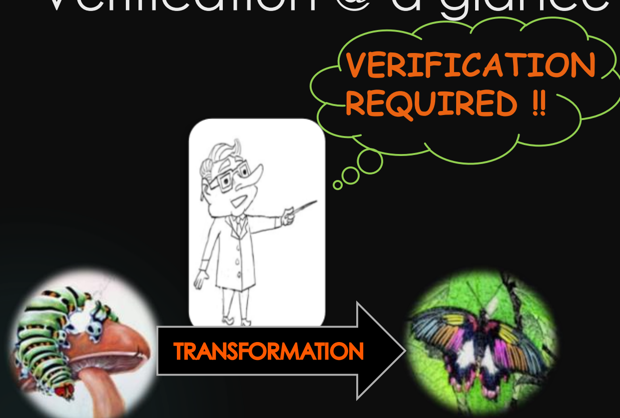
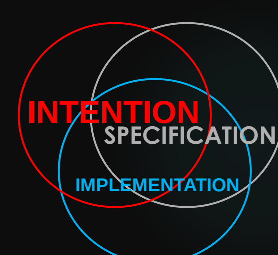
Introduction to Formal verification

Prasenjit Biswas, Nvidia

Verification @ a glance

2





- **Sources of Bugs:**
  Mismatches in between INTENTION – SPECIFICATION – IMPLEMENTATION
- **Validation:** Am I Building A Correct System?
- **Verification:** Am I Building the system Correct ?

Simulation – Two practical Examples

3

- Question :

  How long does it take to verify a 64 bit Floating Point Division Unit
- Answer :
  - There are (2^64 x 2^64) : 2^128 test cases
  - At 1test/us, it will take 10^25 years

- Question :

  How long does it take to verify a 256-bit RAM Memory Unit
- Answer :
  - There are 2^256 = 10^80 bits to test
  - At 1 test/ps and using all matters in our galaxy to build computers of the size of a single electron, it will take 10^10 years to verify 0.05%!!!

Functional Verification – Formal Methods

4

- Construct a computer based mathematical model of the system along with its random components
- Use mathematical reasoning to check functional properties of interest
  - Accurate results
    - Consideration of all cases is implicit
  - Sometimes is difficult and time consuming

Functional Verification – Formal Methods

5

- Construct a computer based mathematical model of the system along with its random components
- Use mathematical reasoning to check functional properties of interest
  - Accurate results
    - Consideration of all cases is implicit
  - Sometimes is difficult and time consuming

Formal Methods - Example

6

X → (X+1)^2 → Y

- Check if Y>X for the given system (X is a natural number)

Test vectors (X)	System output (Y)	Y>X
1	4	True
2	9	True
5	36	True
9	100	True
67	4624	True
1000	1002001	True
1000000	????	????

Formal Methods - Example

7

X ⟶ (X+1)^2 ⟶ Y

- Check if Y>X for the given system (X is a natural number)

Steps		Properties
1	Y > X	Problem statement
2	(X+1)^2 > X	Implementation
3	(X+1).(X+1) > X	Definition of Squre
4	(X+1).X+(X+1).1 > X	Distributivity
5	X.X+1.X+X.1+1.1 > X	Distributivity
6	X.X+X+X+1 > X	Multiplicative Identity
7	X.X+X+1+X > X	Additive commutivity
8	X.X+X+1 > 0	Addition cancellation
9	**True**	Natural numbers > 0

8

An Interesting bug from a real design

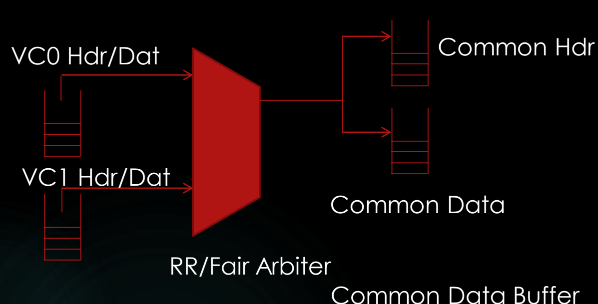
Starvation

9

- What is this ?
  - A problem encountered in multitasking
  - A process is perpetually denied necessary resources
  - Without those resources, the intended task can never be finished
  - Remedy : Alteration of scheduling policy

Starvation bug due to Shared resource

10



vc0/1_req_to_arbiter =
vc0/1_fifo_rd_req
& hdr_space_avail > 1
& vc0/1_data <= space_avail_in_dat_buffer ;

Arbiter grant Hdr for vc0/1 and then lock for corresponding data

Starvation Case:
- vc0: big data pkt [lets say 20 beats] and vc1: small data pkt [lets say 1 beat]
- Common Data Buffer has <20 slots available [due to slow drain rate]
- vc0 will always get blocked due to the optimization

Why Dynamic Verification Didn’t able to catch this?
- Very corner scenario [slow drain rate, large pkt in vc0, continues small pkt in vc1]
- Deep inside the design

Assertion Based Verification [ABV]

11

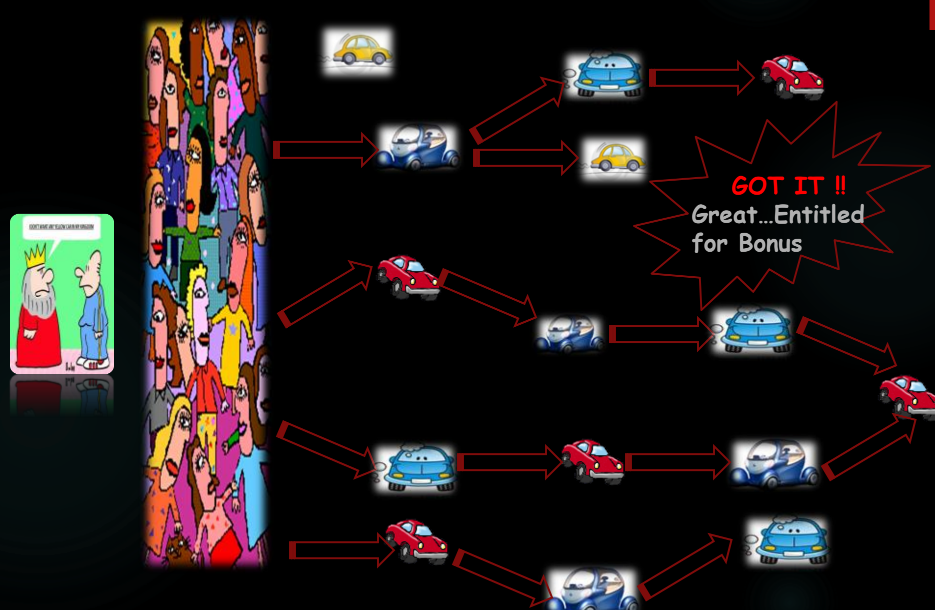
- Finite Automata [FSM] having some Inputs and Outputs
- Design Expects *Inputs obeying a set of Rules*
- Transformation Function that generates *output following a set of Rules*
- Rules are properties for Digital Design

An Introduction to ASSERTION

- Assumption
  - "An **assumption** is a proposition that is taken for granted "
  - Set of Rules all inputs should follow
- Assertion
  - A set of Rules that output should follow
  - Monitor all activities and Flag out if something un-desired happened/ a desired not happening
  - Have different severity level – Info, Warning, Error ….
- Covers
  - Some interesting [operational/Corner] scenario

"I don't want a yellow car in my Kingdom ……."

12



“But the car came on road very next day……”

13

WHAT IS THIS !!!

We have covered @#$% paths, !~*& Houses ….. Found $%#& nos. of yellow Cars …..

Moral of The Story:
“Program testing is very effective way to show the presence of **BUGs**, but it is hopelessly inadequate to show their absence ” -- Edsgar W. Dijkstra

14

Welcome to the world of Formal Verification

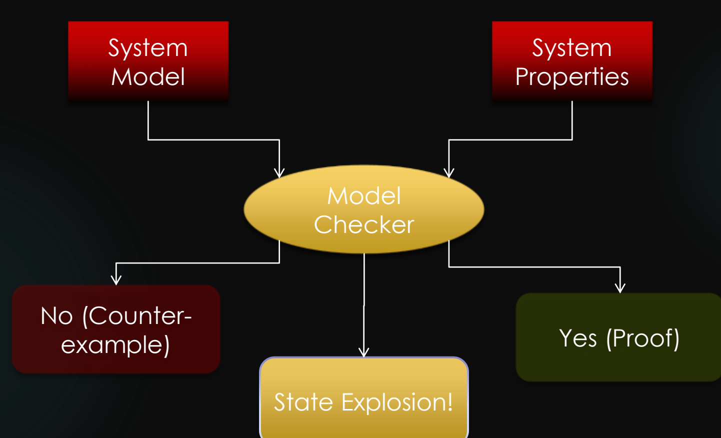
Most widely used Formal Methods

15

- Model Checking
- Equivalence checking
- Theorem proving

Simplistic Description

16



The Truth about Model Checking

17

- Can
  - prove interesting properties
    - No deadlock
    - Arbitration
    - No overflow
    - …
  - find complex bugs
  - handle unit level complexity
- Suffers from
  - state explosion
  - false alarms
- Requires
  - good knowledge about the design
  - well-defined specification
- Available tools @ industry
  - Jasper (Jasper)
  - Magellan (Synopsys)
  - IFV (Cadence)

Equivalence Checking

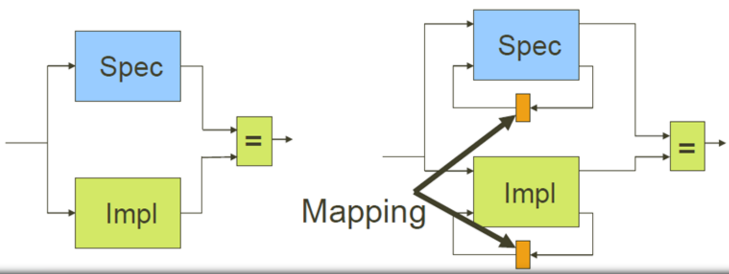
18

- Between two levels of abstraction
  - Algorithmic vs. timed model
  - Transactions vs. RTL
  - RTL vs. Gate-level
  - …
- Is not restricted to
  - Formality
  - RTL vs. gate-level
- Can handle
  - Equivalence between C and RTL
  - Clocked vs. functional models
  - Mathematical operations (add, multiply, subtract, etc.)
- Requires
  - Well-defined observation function
  - Constrained coding of the high level models
- Available tools @ industry
  - Hector (Synopsys)
  - Slec (Calypto)

Combinational Equivalence checking

19

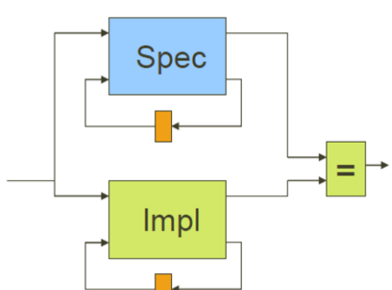
- Both designs are combinational, functions of outputs must match.
- If registers are present, must have one-to-one register mapping.



Cycle accurate equivalence

20

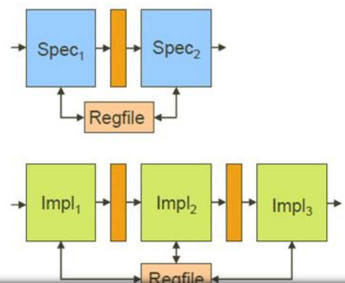
- Classical sequential equivalence
- Start execution in initial state, apply same inputs
- Outputs must match in every clock cycle
- Variants: Sequential hardware equivalence (Pixley)
  Safe replaceability (Singhal, Pixley, Aziz, Brayton)



Pipelined Equivalence

21

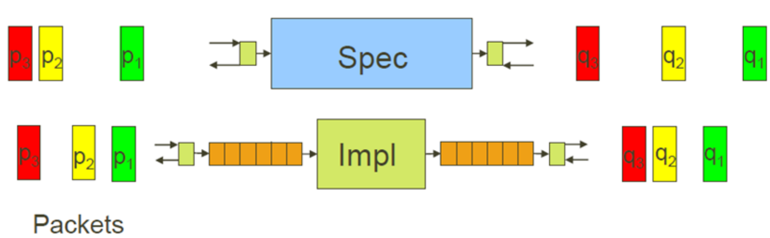
- Both designs are pipelined, but may have different number of stages
- Start in initial state
- Check results of each pipeline execution
- Pipeline can have forwarding logic, register files



Stream based Equivalence

22

- Handshaking protocol on inputs and outputs
- Input data packets may arrive temporally uncorrelated in both designs
- Output data may be generated temporally uncorrelated
- Only order and content of output packets is checked



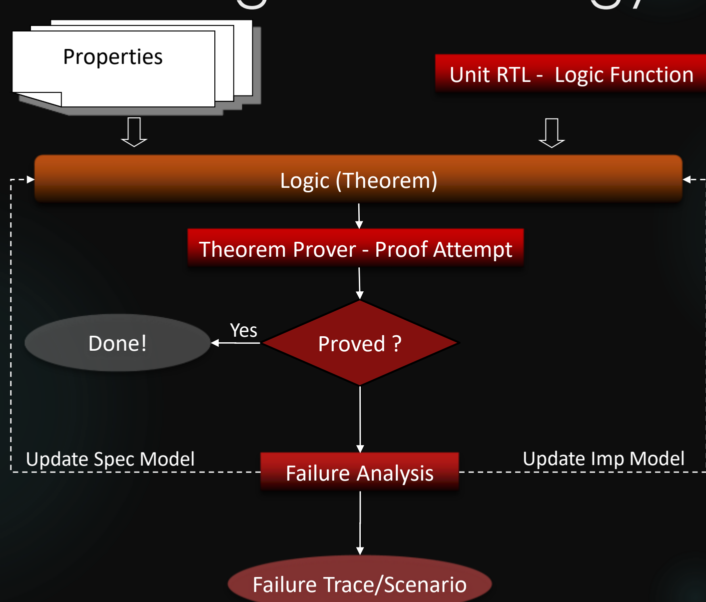
Theorem Proving

23

- Implementation and specification expressed as *formulas in a formal logic*
- Required relationship (logical equivalence/logical implication) described as *a theorem to be proven within the context of a* proof calculus
- A proof system: set of axioms and inference rules (simplification, rewriting, induction, etc.)

Theorem Proving Methodology

24



Fundamental operations

25

- For consistency, will use Verilog-like notation:

  AND: a & b

  OR: a | b

  NOT: ~a
- Sometimes AND represented as multiplication, and OR as addition
  - Like arithmetic, except 1+1 == 1
- Implication: a -> b
  - Same as: ~a | b
  - Terms: a is the *antecedent*, b is the *consequent*

Basic Boolean Identities

26

- Commutative, Associative
  - a & b == b & a
  - a & (b & c) == (a & b) & c
- Distributive both ways
  - a & (b | c) == (a & b) | (a & c)
  - a | (b & c) == (a | b) & (a | c)
- Idempotence: (a & a) == a, (a | a) == a
- DeMorgan
  - ~(a & b) = ~a | ~b
  - ~(a | b) = ~a & ~b

Implication relationships

27

a -> b
- Converse: b -> a
- Inverse: ~a -> ~b
- Contrapositive: ~b -> ~a

Which pairs are identical in truth value?
- Can be useful when restating for FV
- Use |= ("logically entails") symbol as distinct from implication when appropriate

  (a -> b) |= (~b -> ~a)

  (a -> b), (b -> c) |= (a -> c), transitive rule

  (a -> c), (b -> c) |= ((a | b) -> c), disjunctive rule

What is a Proof?

28

- Apply sequence of inference rules
- Example:
  - Known:
    S1: a, S2: (a -> b), S3: (d -> ~b))
  - Prove: ~d
    - C1: S1, S2 |= b
    - C2: S3 |= (~d | ~b)
    - C3: C1, D3 |= ~d
  - Known:
    S1: Texas A & M University is awesome,
    S2: (Awesome university -> Alumni network "is" powerful),
    S3: (Former students "failed to do" great for themselves and the community) -> Alumni network "is not" powerful))
  - Prove: Former A&M students "have done" great or themselves and the community
    - C1: S1, S2 |= Alumni network of A&M "is" powerful
    - C2: S3 |= (Former A&M students "have done" great or themselves and the community ||
      Alumni network of A&M "is not" powerful)
    - C3: C1, D3 |= Former A&M students "have done" great for themselves and the community **(Proved)**

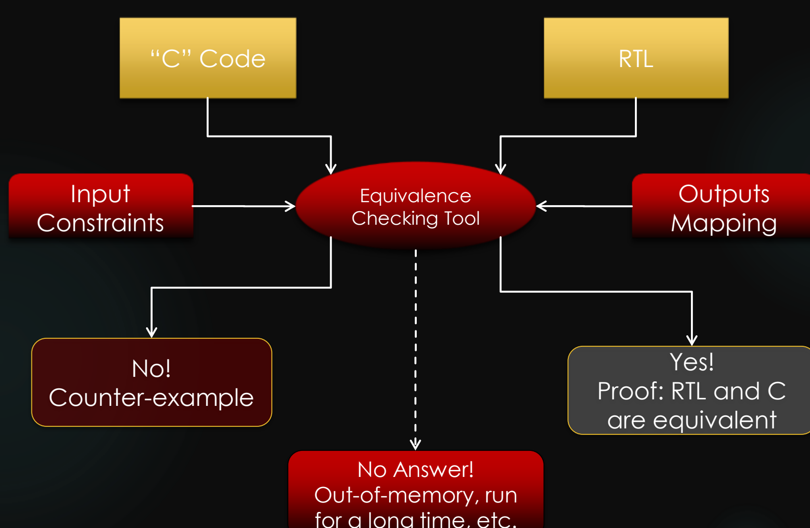
The Truth about Theorem Proving

29

- Can
  - prove complex properties
    - Computation
    - Error analysis
    - Protocols
    - …
  - handle low level complexity
- Suffers from
  - Interactions with user
  - Limited support for hardware verification
- Requires
  - good knowledge about the design
  - know the proof before you mechanize it
  - expertise-level in using the provers
- Available tools
  - HOL (U. of Cambridge)

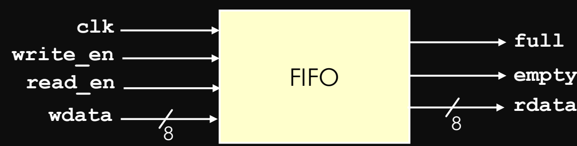
C to RTL Equivalence Checking

30



Example : The FIFO (Data transfer block)

31



Constraints

// If a FIFO is full, then there shouldn't be any further writes
```
env_no_write_when_full: assume property ((full |-> !write_en));
```

// If a FIFO empty, then there shouldn't be any further reads
```
env_no_read_when_empty: assume property ((empty |-> !read_en));
```

Assertions

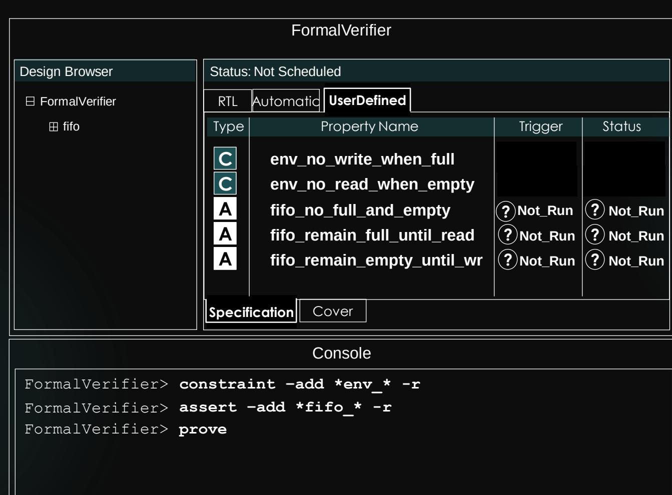
// FIFO cannot have full and empty asserted at the same time
```
fifo_no_full_and_empty: assert property (!(full && empty));
```

// FIFO must keep full asserted until a read occurs
```
fifo_remain_full_until_read:…((full & !read_en) |=> full));
```

// FIFO must keep empty asserted until a write occurs
```
fifo_remain_empty_until_write:…(empty & !write_en) |=> empty));
```

Fire Up The Model Checker

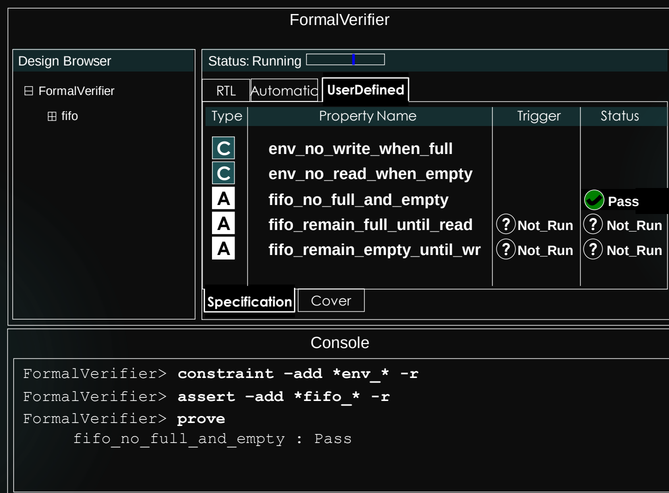
32



*Source: Cadence IFV Training Material

Analyze One At A Time

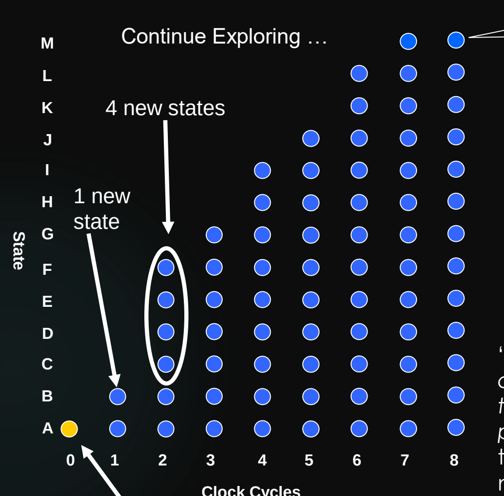
33



*Source: Cadence IFV Training Material

What Just Passed?

34



Fix point = all reachable states explored

No additional reachable states →
all reachable states explored

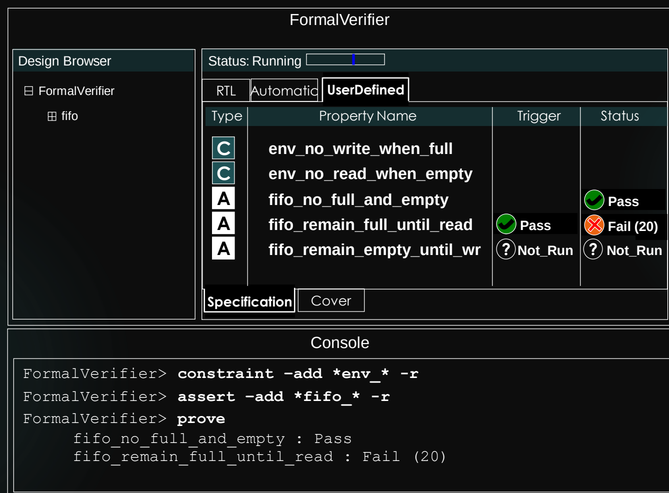
Assertion Passes →
no reachable state in the design can *violate* the assertion

"…*any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern*. The duration of this repeating pattern cannot exceed the number of internal states of the machine…"

*Source: Cadence IFV Training Material

Point Of Failure

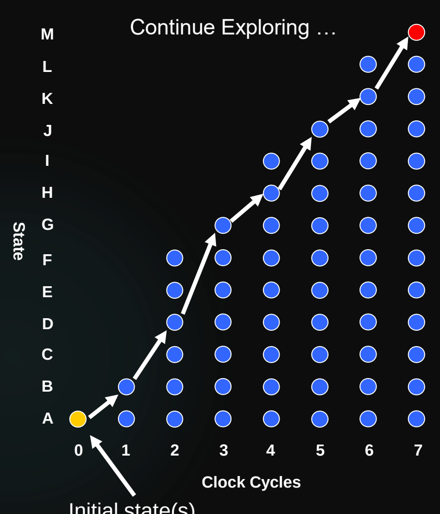
35



*Source: Cadence IFV Training Material

What Just Failed?

36



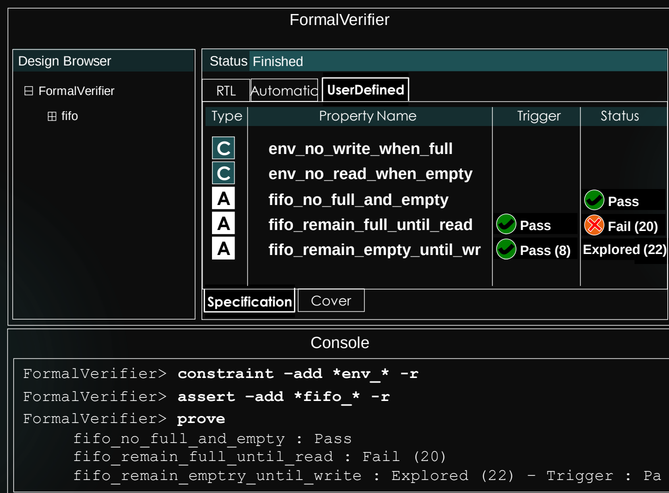
Assertion Fail →
at least one state found to violate the assertion

Trace the path back to the initial state
- The trace is called a Counter-Example
- Reported as waveforms to the user

*Source: Cadence IFV Training Material

Analyzing Further…

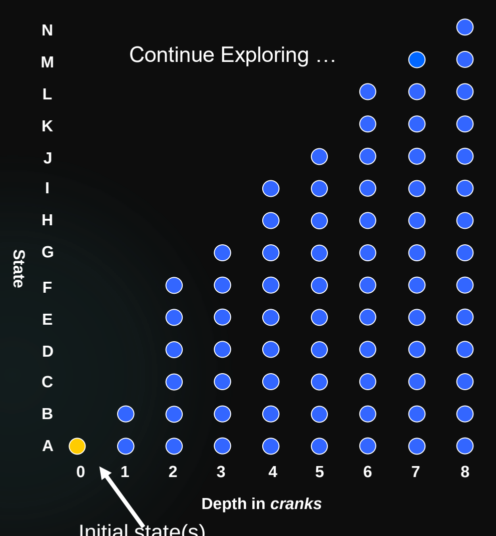
37



*Source: Cadence IFV Training Material

Until We Cannot Anymore

38

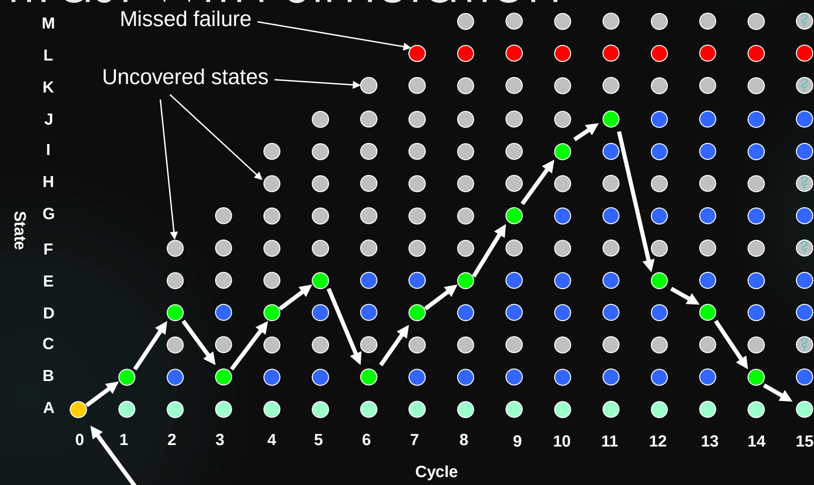


Additional state space still being reached implies that the proof is not complete

Effort (time-out) is defined by user
Result is explored with reported depth. Up to that depth, no state is found to *violate* the assertion

Contrast With Simulation

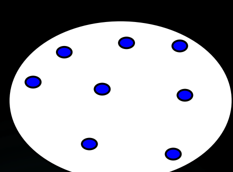
39



- **Misses many reachable states (C, F, H, K, and M)**
- **Covers many states multiple times (A, B, D, and E)**

Motivation for Formal Verification

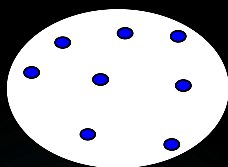
40



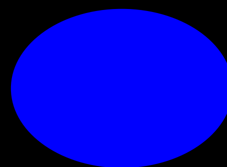
Simulation: spot coverage of design space

Motivation for Formal Verification

41



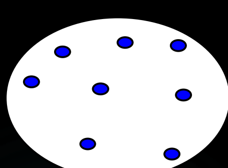
Simulation: spot coverage of design space



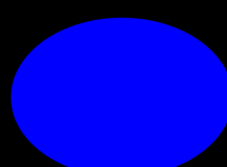
Formal Verification (ideal case): full coverage of design space

Motivation for Formal Verification

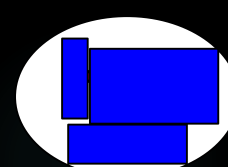
42



Simulation: spot coverage of design space



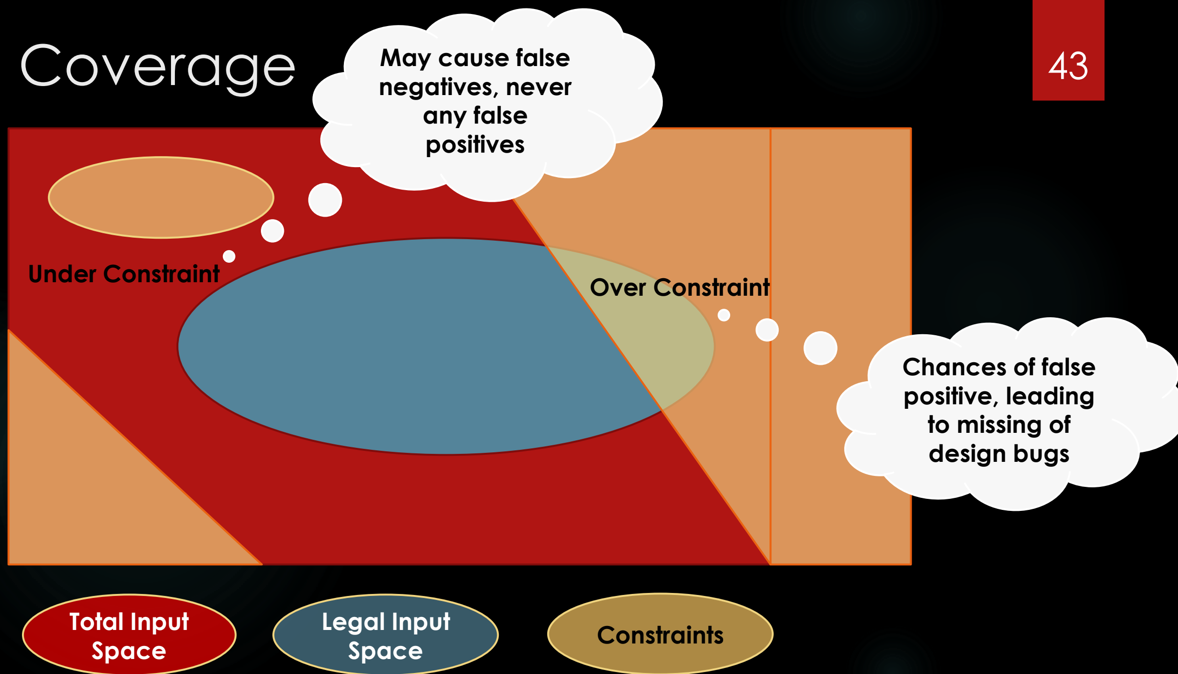
Formal Verification (ideal case): full coverage of design space



Formal Verification (real life): full coverage in some areas

Coverage

43



Mechanics Informs Thought

44

Problem: How do I verify the fairness of a round-robin arbiter?

Simulation:

- I think I will write a behavioral predictor for the arbitration
- Need to think of test scenarios to exercise the DUT
  - Client A requests twice, while B requests once
  - All clients request simultaneously and keep requesting …
- How many clients can hang off the arbiter again?
- Aaargh! I think I will just randomize the inputs

Formal :

- How do I come up with invariants for fairness?
- Example: If there are N agents, and an agent is requesting, it must get a grant within N grants from the arbiter. Must prove this for all the agents in the system.

Introducing the ARBITER

45

- mem-arbiter (input *r1, r2, clk,* output *g1,g2*)
  - Specification/requirements :
    1. Request line *r1* has higher priority than request line *r2*. Whenever *r1* goes high, the grant line *g1* must be asserted for the next two cycles.
    2. None of the request lines are high, the arbiter parks the grant on *g2* in the next cycle.
    3. The grant lines, *g1* and *g2*, are mutually exclusive.

Properties in temporal logic

46

- Request line *r1* has higher priority than request line *r2*. Whenever *r1* goes high, the grant line *g1* must be asserted for the next two cycles.
  - $G\,[r1 \rightarrow Xg1 \wedge XXg1]$
  - r1 |=> g1 ##1 g1
- When none of the request lines are high, the arbiter parks the grant on *g2* in the next cycle.
  - $G\,[\neg r1 \wedge \neg r2 \rightarrow Xg2]$
  - !r1 && !r2 |=> g2
- The grant lines, *g1* and *g2*, are mutually exclusive.
  - $G\,[\neg g1 \vee \neg g2]$
  - (!g1) | | (!g2)

Is it consistent ?

Remove it …

47

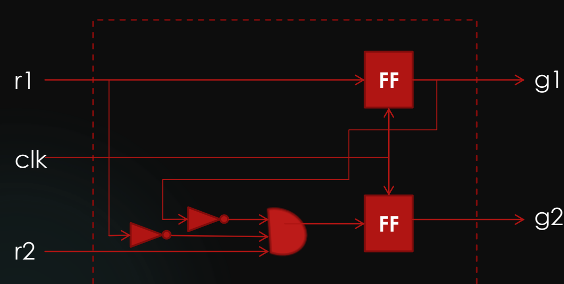
- G [¬g1 -> g2]
- !g1 |-> g2

- Have I written enough properties?
  1. G [r1 -> Xg1 ∧ XXg1] i.e r1 |=> g1 ##1 g1
  2. G [¬g1 -> g2] i.e !g1 |-> g2
  3. G [¬g1 v ¬g2] i.e (!g1) | | (!g2)

- Does the spec cover any behavior where *g1* is required to be high?
  - YES
- Does it enforce *g2* to be high?
  - NO
- Should any extra property be added?
  - G [¬r1 ∧ X¬r1 -> XX ¬g1]
  - !r1 ##1 !r1 |=> !g1

Property Verification

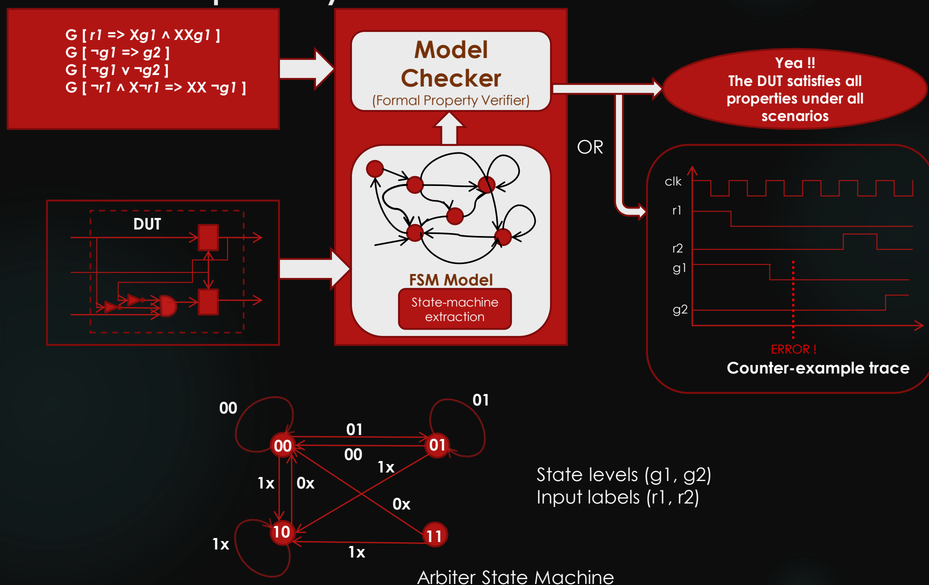
48




```
module arbiter( r1, r2, g1, g2, clk )
input clk, r1, r2;
output g1, g2;
reg g1, g2;
always @( posedge clk )
begin
g2 <= r2 & ~r1 & ~g1;
g1 <= r1;
end
endmodule
```

Formal Property Verification Platform

49



Types of Properties

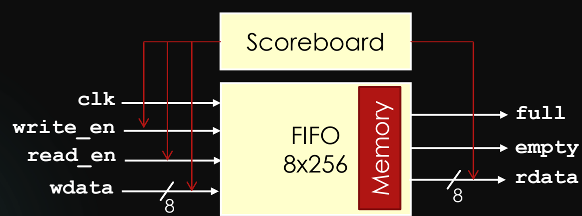
50

- **Safety :** "Something bad won't happen."
  G(~ (grant & busy))
- **Liveness :** "Something good will happen."
  F(grant)
- **Fairness :** "Something happens infinitely often."
  G(F(!busy))
  - Usually considered subset of liveness

Down The Data Path – FIFO revisited

51

Problem: How do I verify the data integrity of my FIFO DUT?



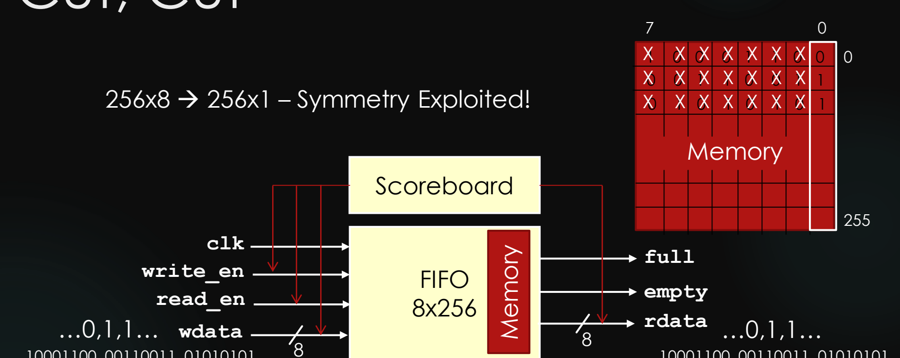
Problem Characterization: What is data integrity for the FIFO?

- The DUT should not corrupt any data
- The DUT should not drop any data
- The DUT should not duplicate any data
- The DUT should not re-order the data

Cut, Cut, Cut

52

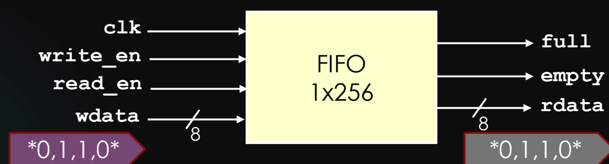
256x8 → 256x1 – Symmetry Exploited!



The basic Abstractions

53

Key Insight: A *suitably aligned* input sequence of 0s, followed by two consecutive 1s, followed by 0s can detect *all* of the integrity errors



Problem Characterization: What is data integrity for the FIFO?

- The DUT should not corrupt the data *0,1,0*
- The DUT should not drop any data *0,1,0*
- The DUT should not duplicate any data *0,1,1,1,0*
- The DUT should not re-order the data *0,1,0,1,0*

Enumerating The Alignment

54

Applying the idea in simulation would involve enumerating the alignment

Test-1 : 1,1
Test-2 : 0,1,1
Test-3 : 0,1,1,0
Test-4 : 0,0,1,1,0 …

Applying the idea in formal would involve…

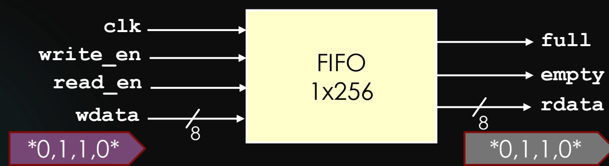
Two invariants specify the pattern …….

- The *first time* we see 1, it must be followed by another 1
- We must not see more than two 1s

Assume-Assert Duality

Assume:
- The first time we write 1 we must write 1 the next time
- We must not write more than two 1s

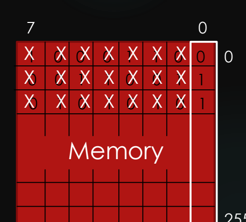
Assert:
- The first time we read 1 we must read 1 the next time
- We must not read more than two 1s

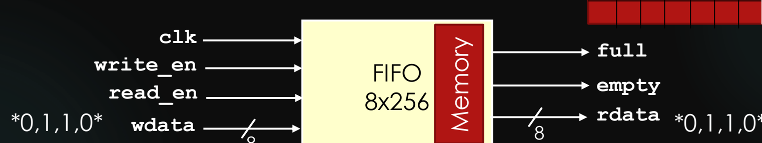


Application :
- Asynchronous FIFOs can (and should) be verified this way
- Generalized to almost every data-transport design you can think of including
  - Memory controllers
  - DMA controllers
  - Bridges and cross-bars etc.

Anything Else Cut-worthy?

256x8 → 256x1 – Symmetry Exploited!

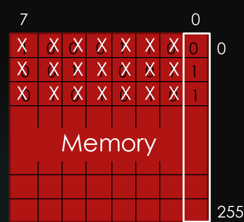




Key Insights
- The memory array is not very interesting i.e. can be assumed correct
- We really, only need to track the 2 consecutive 1s

And… Cut

57



Each time we write a 1 into the RAM we record the address

Write 0 to Address 0x?? — Don't Care

Write 1 to Address 0xD — 0 0 0 0 1 1 0 1 A

Write 1 to Address 0xC — 0 0 0 0 1 1 0 0 B

Each time we read we compare with the recorded addresses. Send 1 on match, else 0.

Read from Address X

```
rdata[0] <= (X == A) ? 1 :
            (X == B) ? 1 :
            0;
```

256x8 → 2x8!

Compositional Reasoning

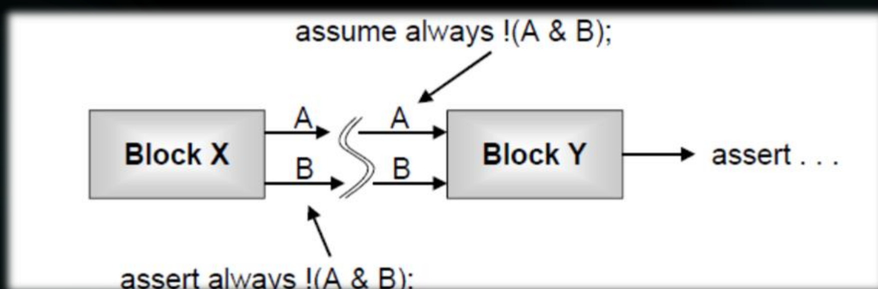
58

- Process of reducing an analysis of a larger concurrent system to reasoning about its individual functional pieces
- Effective for managing proof complexity and state explosion during a formal proof
- Transfers the burden of proof from the global component to the local functional component level
- Global properties can be inferred from independently verified functional component properties
- Example : Assume-Guarantee, Formal Abstraction (next slides)

Assume-Guarantee

59

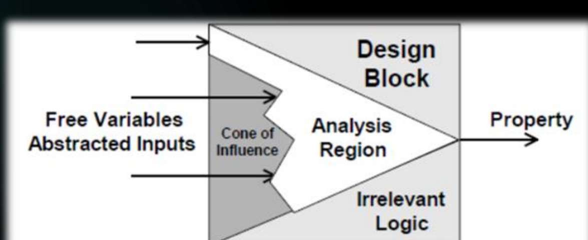
- Prove properties on a decomposed block using a set of assumptions about another neighboring block
- Prove these assumptions separately on the neighboring block



Formal Abstraction

60

- First prove properties on a subsection of the formal analysis block
- Then the driving logic for this subsection is abstracted
- The design logic is thus ignored in favor of the proved properties
- Key proponent :
  - If a property holds on the formal abstractions (the generalization), then it holds on the entire cone of influence (the actual design logic)



Remark : if a property fails on the formal abstraction, then it might be necessary to include additional logic into a larger analysis region, forming a new abstraction that eliminates the false negative.

Some Formal Methods Myths

61

- Formal methods can only be used by mathematicians
  - They are primarily based on mathematical concepts that is usually transparent to the user
- The reasoning process is itself prone to errors, so why bother?
  - We opt to reduce design bugs not eliminate them
- Using formal methods tends to slow the design process
  - The early detection of design bugs are allows us to speed up the overall design process

Welcome Questions from Green minds !!

Few References and acknowledgements :

- Formal HW Verification course by Prof. Sofiene Tahar
- Book : A Roadmap for Formal Property Verification by Prof. Pallab Dasgupta, Springer Publication
- DVCON paper "Guidelines for creating a FV testplan", authored by Harry Foster, Lawrence Loh, Bahman Rabii and Vigyan Singhal
- Introduction to FV talk @Nvidia-B'lore by Abhishek Datta
- Book : William K. Lam (2005) Hardware Design Verification: Simulation and Formal Method-Based Approaches
- Cadence Training Material
- Yogesh Mahajan, Deepanjan Roy, Mark Bezdany, Kalyan K, Abhishek Datta, Prosenjit Chatterjee (nVIDIA)