

Formal Technology in the Post Silicon lab

Real-Life Application Examples



Haifa Verification Conference

Jamil R. Mazzawi

Lawrence Loh

Jasper Design Automation

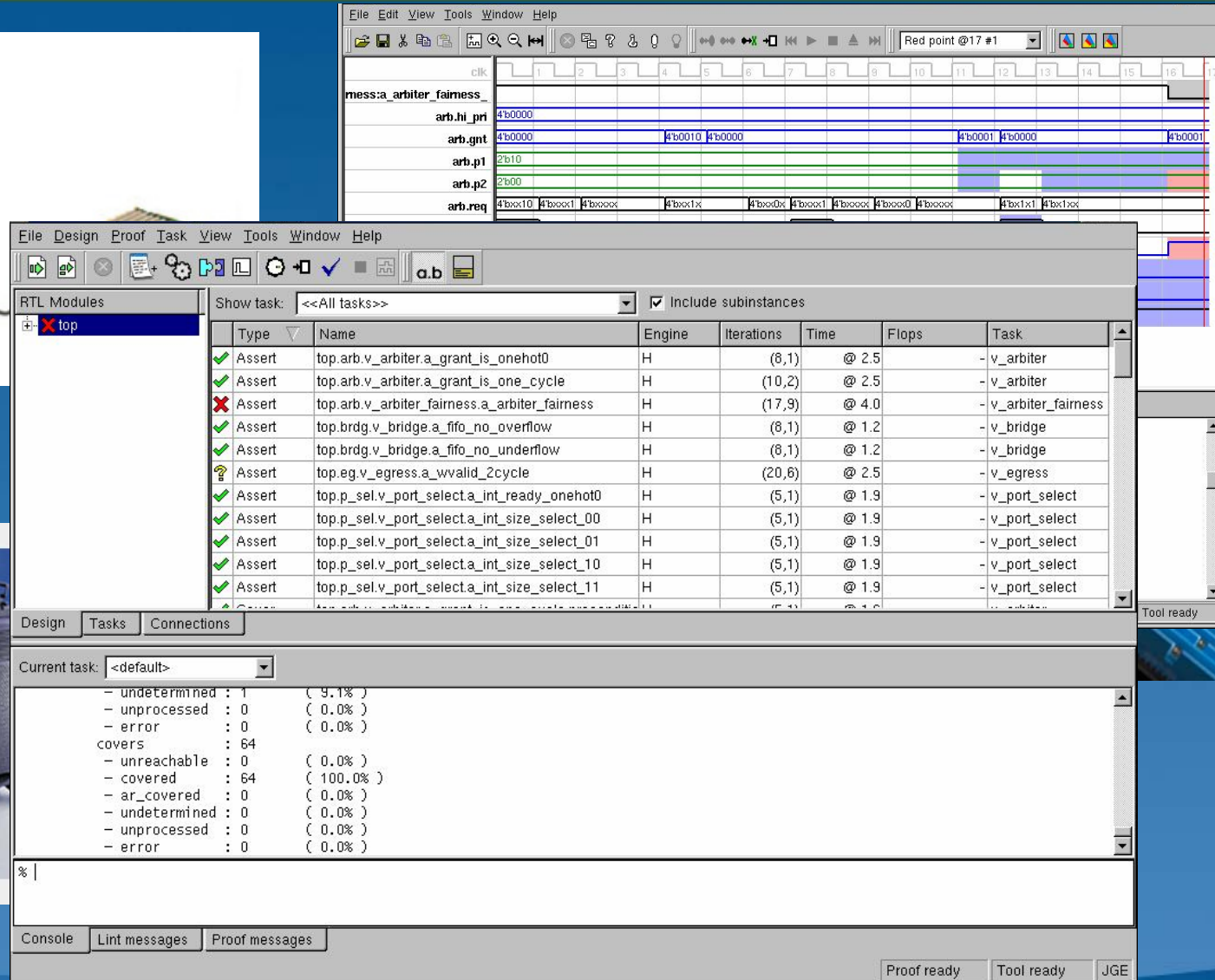
Focus of This Presentation

- Finding bugs in silicon chips
 - Post-silicon production
 - Functional bugs: Bugs originate in the RTL
 - Reproducing bugs in RTL to root-cause them after initial identification is done in the lab
- Not concerned with
 - Electrical bugs
 - Timing bugs
 - Synthesis-based bugs (?)
 - Manufacturing bugs
 - Software bugs
 - Etc...

Post-Silicon Lab, Not a Place You Would Think To Use Formal



Formal Can Play a Critical Role in the Post-Silicon Debug Lab



The screenshot displays the Jasper Design Automation software interface, which is used for formal verification and post-silicon debug. The interface is divided into several panes:

- RTL Modules:** A tree view on the left showing the hierarchy of modules, with 'top' selected.
- Task Results:** A table in the center showing the results of various tasks. The table has columns for Type, Name, Engine, Iterations, Time, Flops, and Task. The tasks listed include assertions for arbiter fairness, bridge overflow/underflow, egress validity, and port select integrity.
- Waveform Viewer:** A pane on the right showing a timing diagram for a specific signal, 'Red point @17 #1', with a red dot indicating a specific point in time.
- Design, Tasks, Connections:** Tabs at the bottom of the main pane, with 'Design' currently selected.
- Current task:** A dropdown menu showing the current task being executed, set to '<default>'. Below it, a list of task statistics is shown, including 'undetermined', 'unprocessed', 'error', 'covers', 'unreachable', 'covered', 'ar_covered', and 'error' counts and percentages.
- Console, Lint messages, Proof messages:** Tabs at the bottom of the interface for viewing logs and messages.

The interface also includes a 'Tool ready' status indicator at the bottom right.



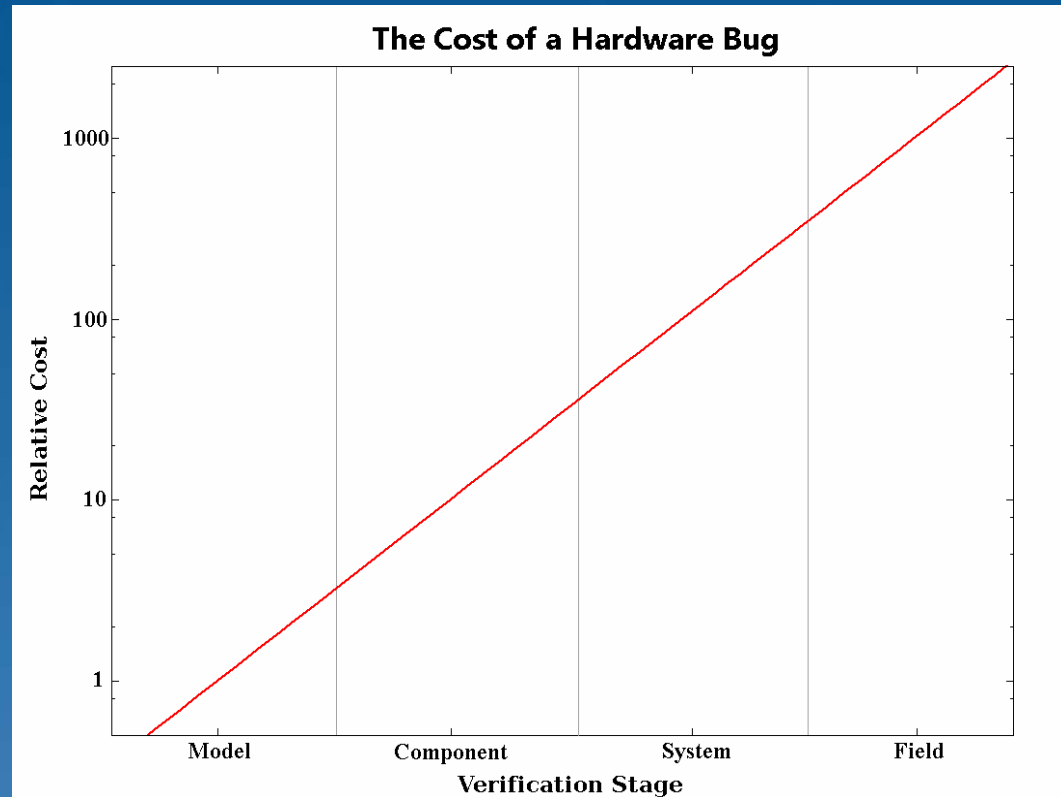
Outline

- Typical scenario from the post-silicon lab
- Simple principles of using formal in post-silicon debugging
- Two real-life case studies of using JasperGold® Formal Verification System
- Case study 1: Detecting bus protocol violation bug
 - Finding “the bug” ~6 times faster than simulation did
 - Verifying bug fix before going to silicon again
- Case study 2: Quickly isolating the block with a bug
 - Interaction between formal team and lab team
 - Two teams interacting, each using their capabilities to help the other team
 - The total power of the two teams together is much greater than either alone

Cost of Silicon Bug

Finding bugs in model testing is the least expensive and most desired approach, but the cost of a bug goes up 10× if it's detected in component test, 10× more if it's discovered in system test, and 10× more if it's discovered in the field, leading to a failure, a recall, or damage to a customer's reputation.”

John Bourgoïn, MIPS CEO
At a DesignCon 2006 panel



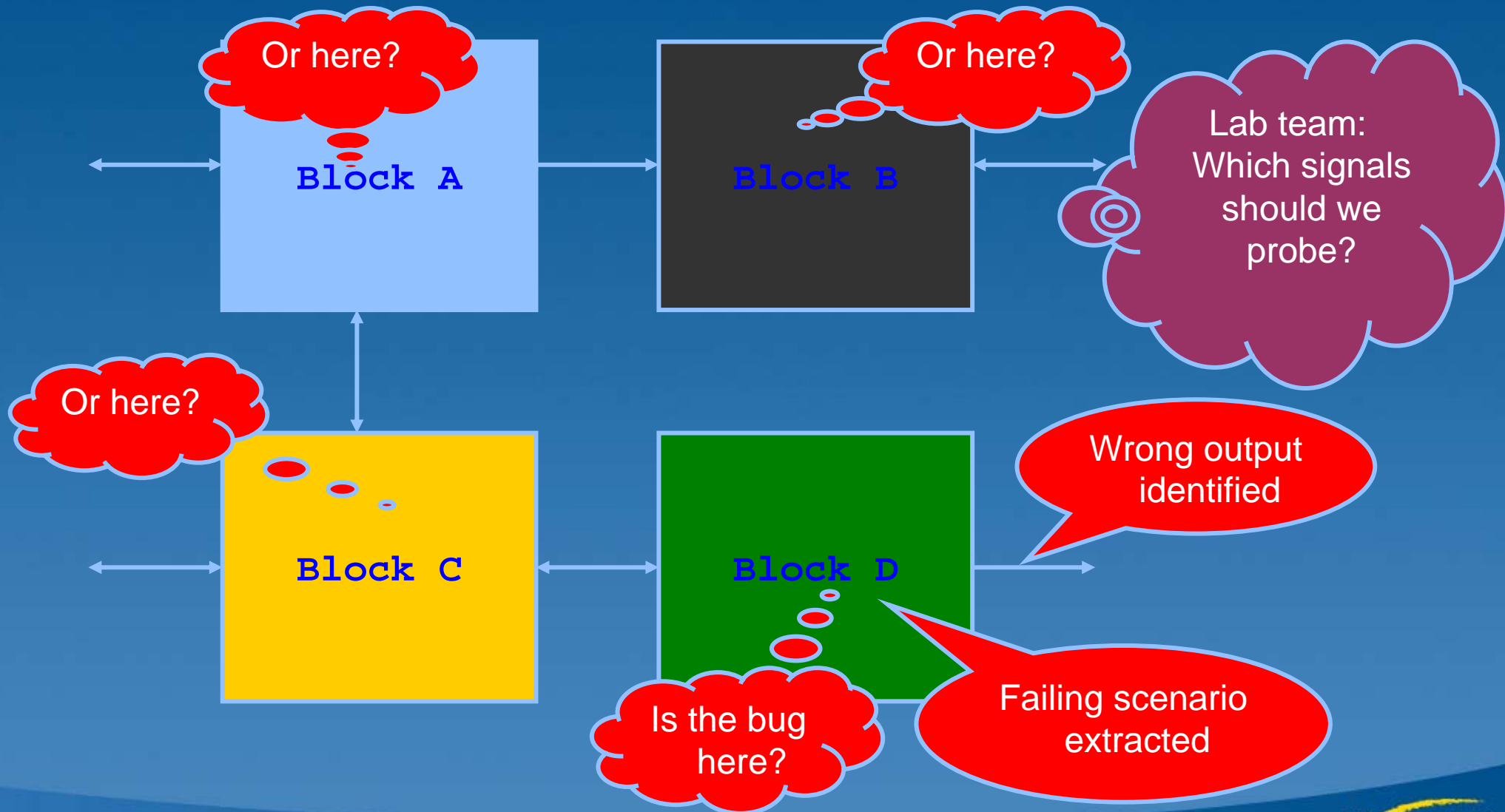
On-Chip Post-Silicon Debugging Capabilities

- Most chips now have some kind of on-chip debugging capability
 - Freeze chip, when certain event is identified
 - On-chip logic analyzer
 - Selected group of signals is mux-ed to external pins
 - Save the value of certain signals, N cycles before freeze event into some memory
 - Using scan chain to scan out all the flops
- **Common capability: Failure trace extraction**
 - Debugging team can capture a trace for number of signals a few cycles before (and maybe after) a problem is detected
 - We refer to this trace as: failure trace

Typical Scenario of a Post-Silicon Bug Isolation Process

- Silicon chip is misbehaving
 - Hanging, stopped responding
 - Dropping packets
 - Violating protocols
 - Producing wrong output
 - Etc...
- Lab team: Extract the failing trace
- Lab team now knows that
 - The chip has some illegal behavior
 - **But, how did it reach this state?**
 - The failing scenario may have taken hours (real time) to reach

Failing Scenario Identified, but Where Is the Bug?



The Dynamic-Verification Team Is Called for Help

- Here are the last few cycles of the failing scenario
- Can you please find the root-cause of the problem?
- Can you find how we reached this state, using simulations?
- We don't know where the bug is happening, but we know that it is causing block D to act incorrectly
- The bug happens after 3-4 hours run in the lab, when we inject this kind of traffic (example: only read transactions on bus X)
- Another way to say this:
 - “It took us 4 hours of real-time with random traffic of this kind to hit the bug. Let's see how you can reproduce it when your simulation time is x1000 slower.... Ah, that's only 4,000 hours of simulation.... But you can do it, we know you can.... Oh, btw, you have only 1 week to find it.”
- With simulations, the verification team is, in many cases, assigned “mission impossible”

Formal Technology Is Called for Help

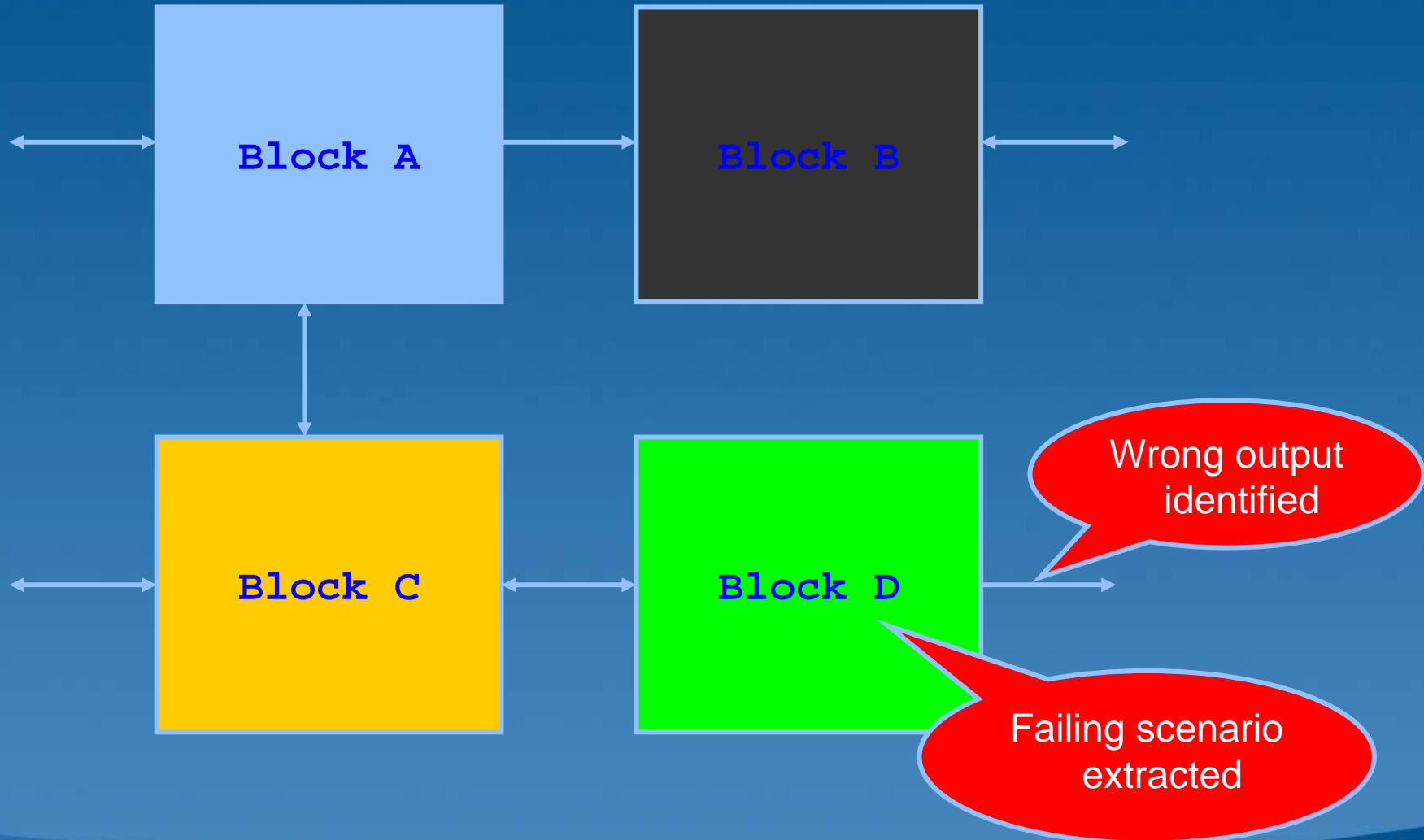
- One of the key strengths of formal is its ability to find bugs fast
 - Finding CEX (failure of a property) is usually much faster than reaching proof on the same property
 - Bug hunting
- With simulation:
 - We hope that the constrained-random generator will hit the input combination that causes the failure scenario (trigger the bug)
- With formal:
 - The formal engine can mathematically find this failure scenario starting from the extracted failure trace

Basic Flow or Process

- The following few slides outline the steps needed to find the bug
 - These are fundamentally the same steps one takes in a normal formal verification flow
- Main differences between normal and pre-silicon FV flows:
 - We are looking for one specific bug, one specific scenario
 - We are not looking for full proof or coverage completeness
 - We just need to find the scenario that leads to the illegal behavior
 - We can allow over-constraints to simplify the process
 - Example: don't allow Write transactions because the bug happens with Read transactions only
 - This allows us not to support Writes in the assertions and assumptions we write

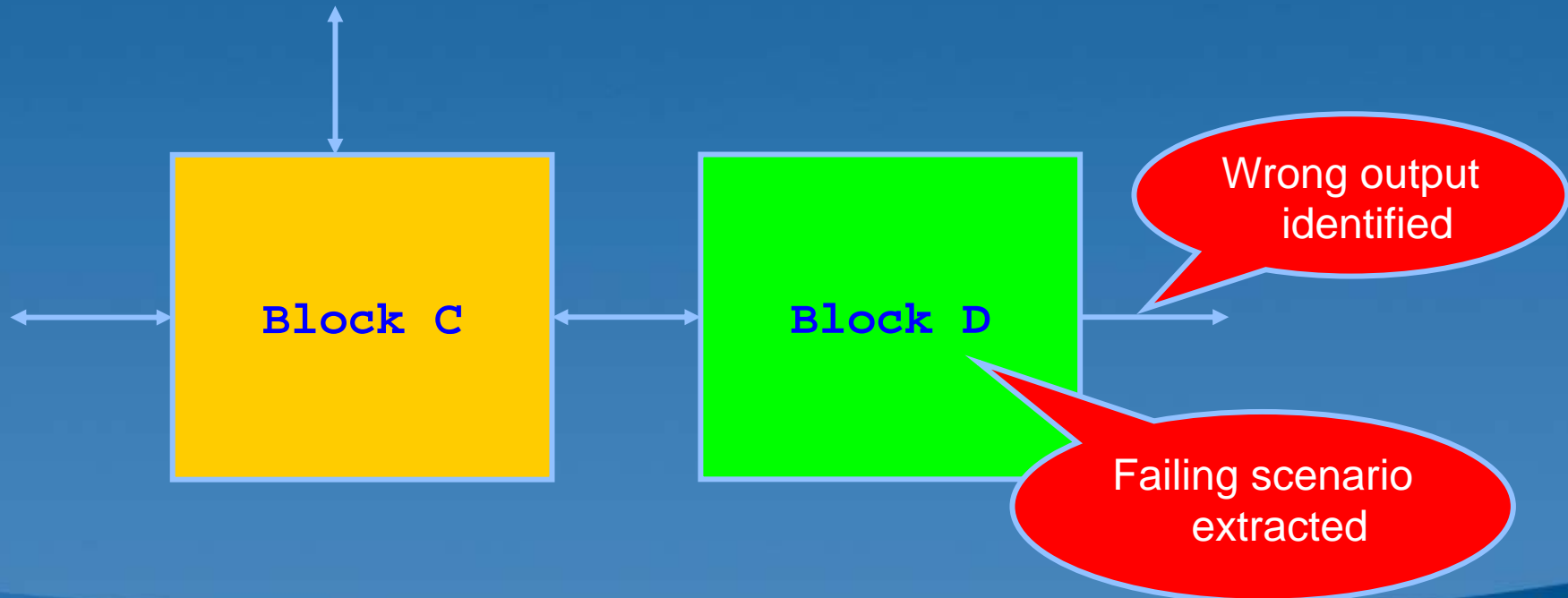
Step 1: Choose the Level or Block to Work with

- Option1: Full chip



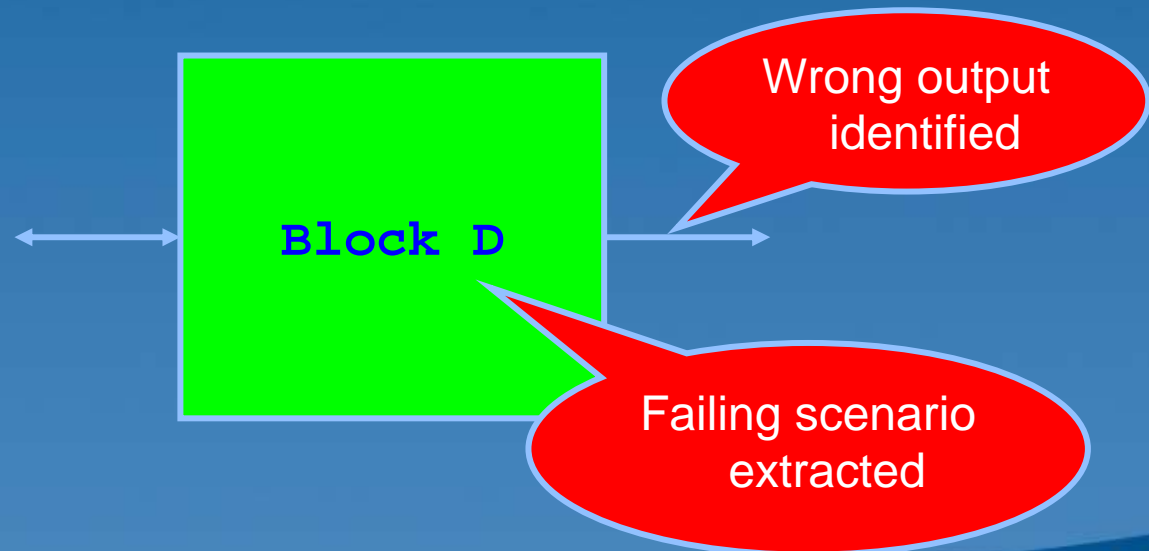
Step 1: Choose the Level or Block to Work with

- Option 2: Last two blocks



Step 1: Choose the Level or Block to Work with

- Option 3: Single block



Step 2: Define Your Property: `not(illegal_scenario)`

- Start from the description of the problem
 - We have a trace that shows the illegal scenario
 - Or we know that the problem happens when a write trans is followed by another write trans
- All we need to do is define a property that states that:
 - **This scenario cannot happen**
- Examples:
 - If we know the problem happens when FSM_X goes from state_A to state_B, and this is not allowed:

```
assert (not ( (fsm_x==state_A) ##1 (fsm_x==state_B) ))
```
 - If the problem happens when some FIFO overflows, and it is not supposed to:

```
assert (not (fifo_x.overflow))
```
 - If the problem happen when slave_x is responding to a read transaction:
 - Define properties that ensure this slave is adhering to all the protocol rules for read transactions

Step 3: Optional: Write Input Constraint, as Needed

- **Input Assumptions**

- Optional step: may not need
- Based on the interface spec between Block C and Block D
- Only legal inputs can happen

- If needed: add constraints to prevent scenarios you don't want to support

Block A

Assertions: Not (illegal scenario)

Case Study 1:

Memory Controller Violating Bus Protocol

- SoC Chip, with a CPU and multiple peripherals
- Chip had problems in the market and was re-called
 - It hangs in certain conditions, in the field
- Bug was identified in the post-silicon lab as...
 - DDR2 memory controller is hanging and causing the bus to hang
 - Bug happens with Read transactions to the DDR2 memory controller (no problem in Write)
 - Suspect that the memory controller (bus slave) is violating the bus protocol
- The DDR2 memory controller with the bug is IP from a well-known IP vendor
- Simulation team worked for 3-4 months (with random simulation) until they were able to root-cause the bug
- Imagine the cost of this bug
- Imagine the relationship between simulation team, Chip-Company, IP-Vendor, and Chip-Company's customer during this time

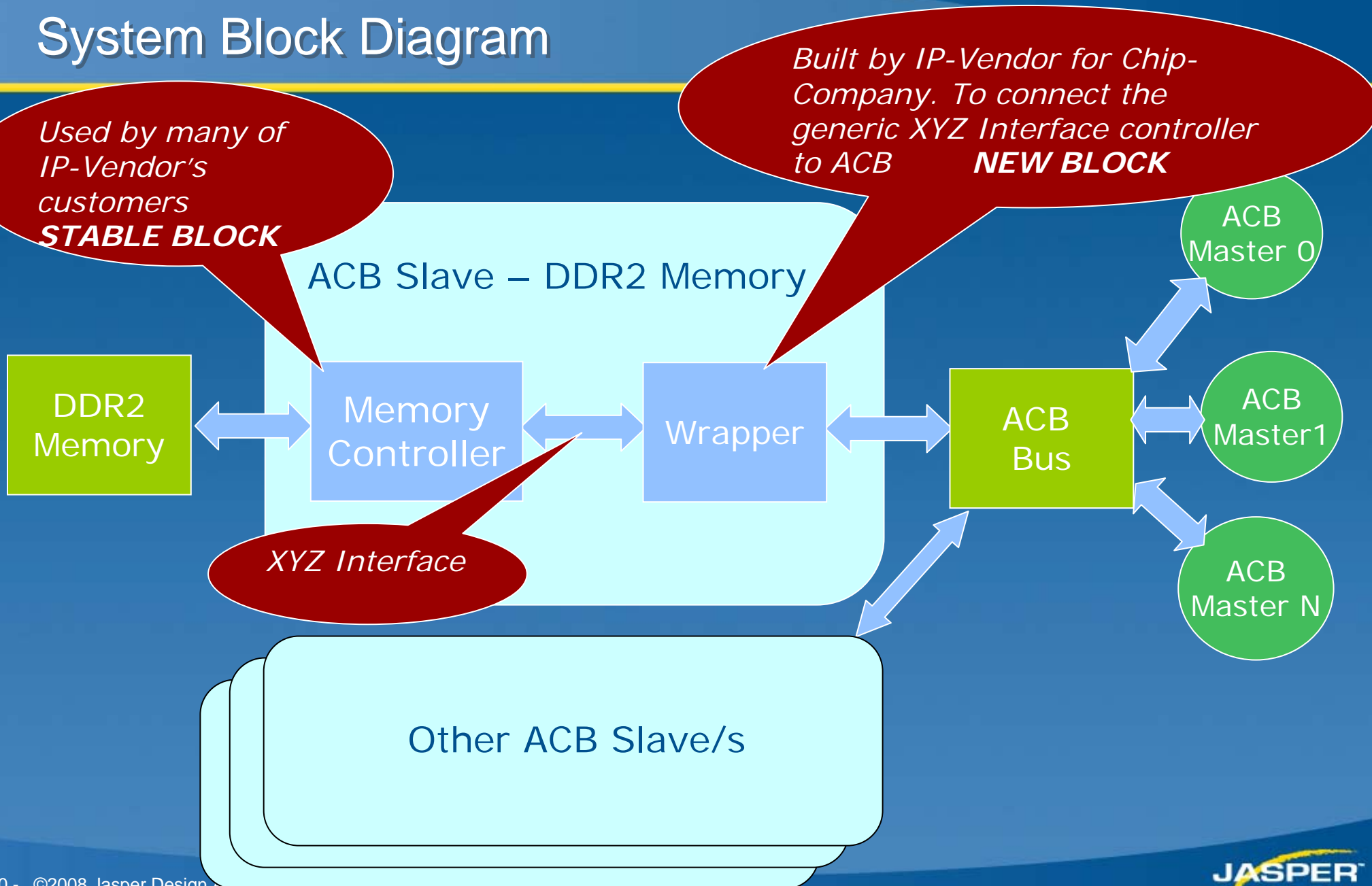
The following names used in this presentation are aliases to protect identity etc...

- Chip-Company
- IP-Vendor
- ACB Bus
- XYZ Interface

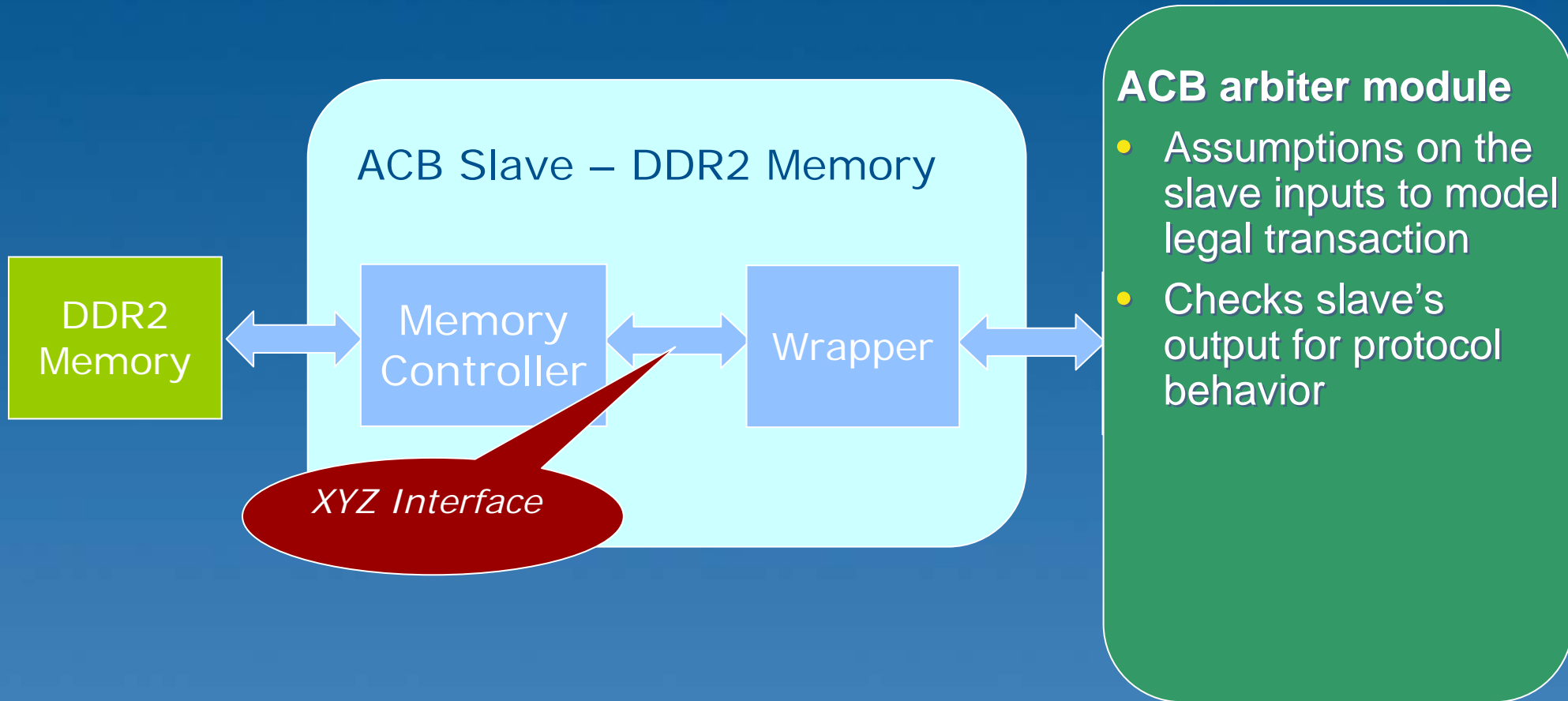
Formal Is Called for Help

- Formal was called to help **after the fact**, to see how fast it can be done with formal
- Formal engineer was given the same information the simulation team got (no cheating)
- The bug was found after 2.5 weeks
 - A good part of this time was spent ramping up on the design and protocols involved
 - Once setup is complete and properties are written, actual run time to find the CEX was under 1 minute
 - Compared to weeks of simulations!
- Later, formal was re-run **on the fixed RTL code**
 - Two other bugs were found

System Block Diagram



Verification Strategy: Step 1: Model the ACB Arbiter



Verification Strategy: Step 2: Option A

Model the DDR2 Interface, Include Memory Controller

DDR2 model

- Assumptions on inputs
- Checks outputs

ACB Slave – DDR2 Memory

Memory
Controller

Wrapper

ACB arbiter model

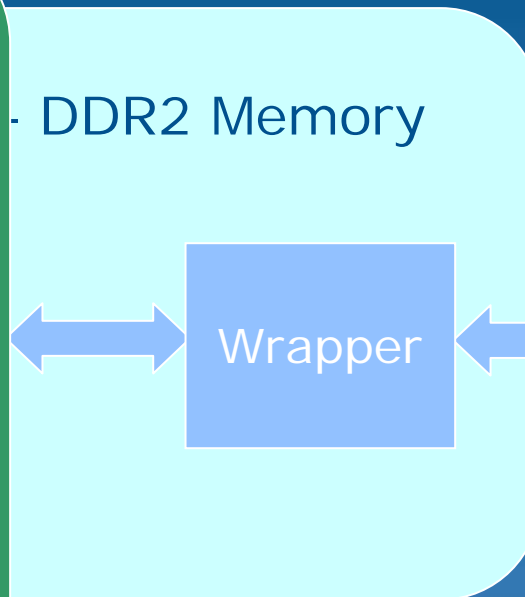
- Assumptions on the slave inputs to model legal transaction
- Checks slave's output for protocol behavior

Verification Strategy: Step 2: Option B

Remove the Memory Controller and Model the XYZ Interface

XYZ model

- Assumptions on wrapper inputs
- Sample wrapper output to model controller state



ACB Arbiter model

- Assumptions on the Slave inputs to model legal transaction
- Checks Slave's output for protocol behavior

Verification Strategy: Final Decisions

- We ended up using Option B
- The memory controller is considered stable; the wrapper is new code
 - The bug is probably in the wrapper code
 - Avoid the complexity of the DDR2 protocol
- We focused on writing and proving properties to check compliance of the wrapper (as a slave) with the ACB bus protocol
- Important: This is post-silicon verification, not pre-silicon verification
 - Shortcuts are allowed, anything to make us find the bug faster
 - Write properties only where the bug is suspected to be
 - Use assumes to prevent certain scenarios from happening (like Write trans)
 - Put assumes on internal signals:
`assume (top.addr_decoder.legal_address == 1)`

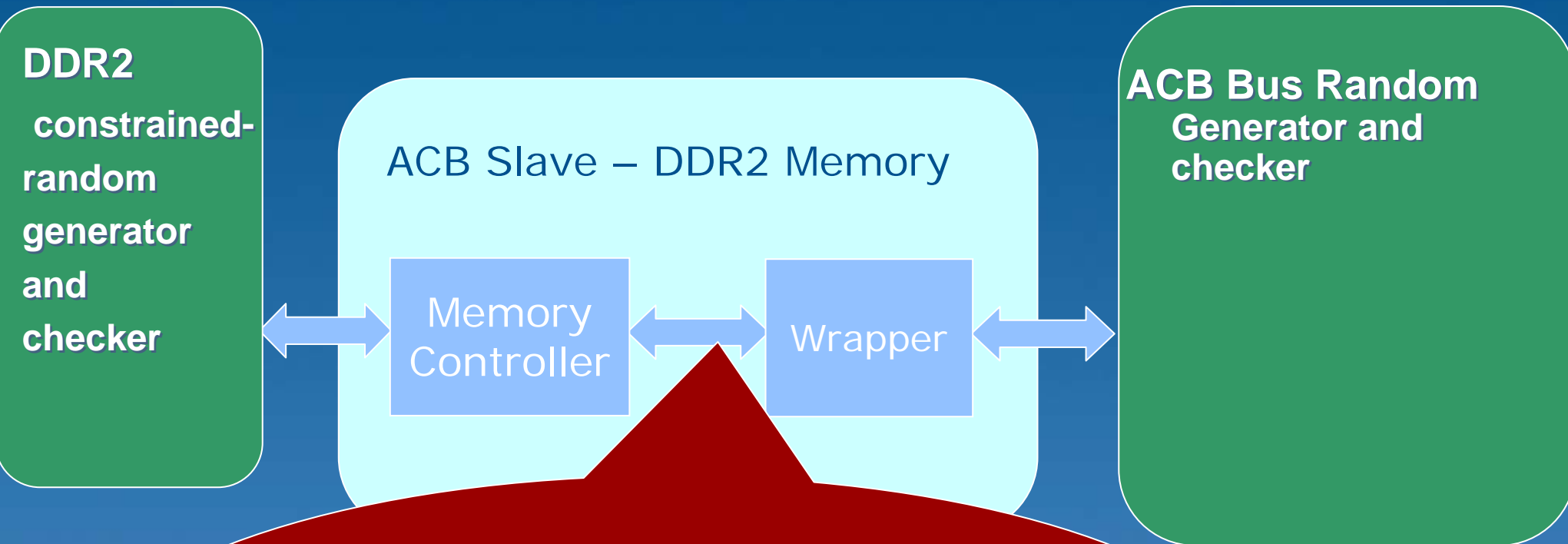
Specification: In Plain English

- For a transaction of size M beats, the slave needs to return M rd_ack
- If the last rd_ack comes at Cycle N, the rd_complete needs to be asserted at either Cycle N-1 or N
- If rd_complete is given at cycle N-1, cycle N must have a valid beat
- Design decision: Always give rd_complete at cycle N-1 (never at N)

Main Assertion – Code Example: Single-Beat Transaction

```
property P_rdComp_is_one_before_last_rdAck_single_beat;  
  @( posedge clk ) disable iff (!rst_n)  
    (m0_active_rd & ACB0_SI_rdComp &  
     m0_trans_is_single_one_beat)  
  | ->  
  (  
    ( ACB0_SI_rdDAck & (m0_trans_length == 1))  
    or ( !ACB0_SI_rdDAck ##1 ( ACB0_SI_rdDAck &  
                                (m0_trans_length == 1)) )  
    ) ;  
endproperty
```

Why the Bug Was Hard to Find with Coverage-Driven Random Simulations



Bug started here, very specific timing relationship that the memory controller produced

1- Limited controllability:

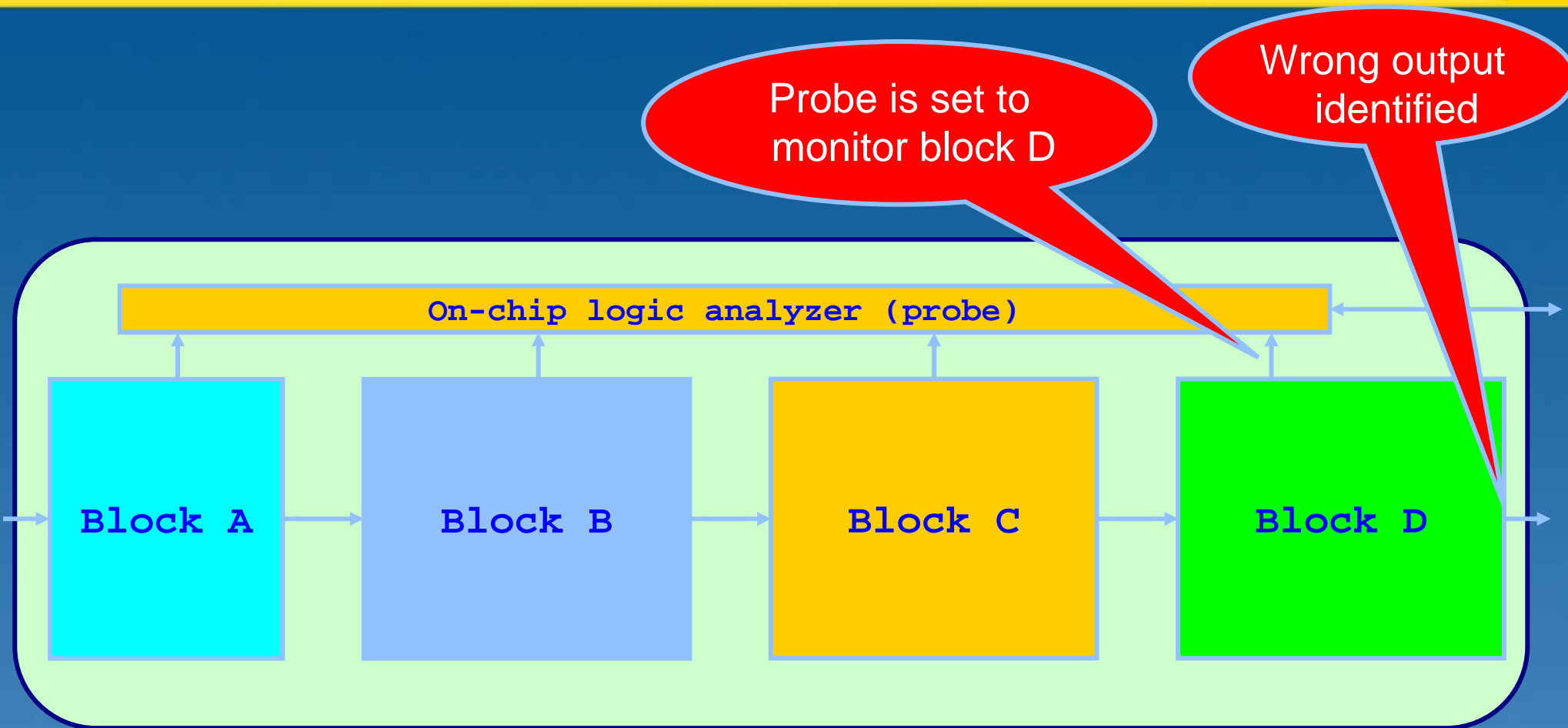
It is hard to hit this combination randomly when you are driving random traffic from the DDR2 memory side

2- No functional coverage was defined for all timing relationships

Testcase 2: Formal Team Hand-in-Hand with Lab Team

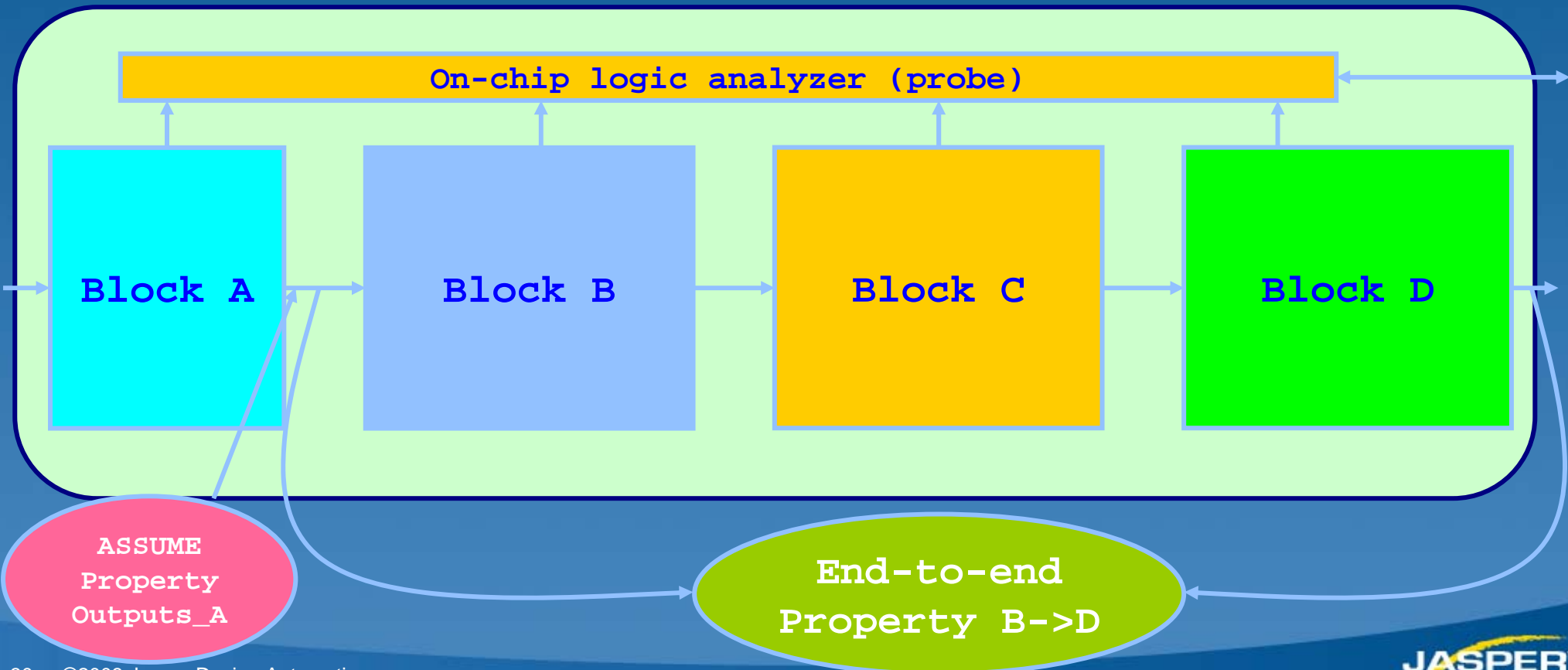
- Existing customers
- Existing experience with formal
- Never used formal for post-silicon before
- Formal is called for help once the bug is identified in the lab
- Formal team worked with the lab team hand-in-hand

High-Level Block Diagram



With Formal: Write End-to-End Property

- An End-to-end property was written, from Inputs of B to Outputs of D
 - Block A was not relevant
- The property was written based on the illegal trace found by the lab team
- Only inputs that cause the bug are allowed (others are constrained out)



With Formal: Write End-to-End Property

- An End-to-end property was written, from Inputs of B to Outputs of D
 - Block A was not relevant
- The property was written by the lab-team
- Only inputs that could be constrained out)

**Information from lab-team
helping formal-team
narrow their scope of work**

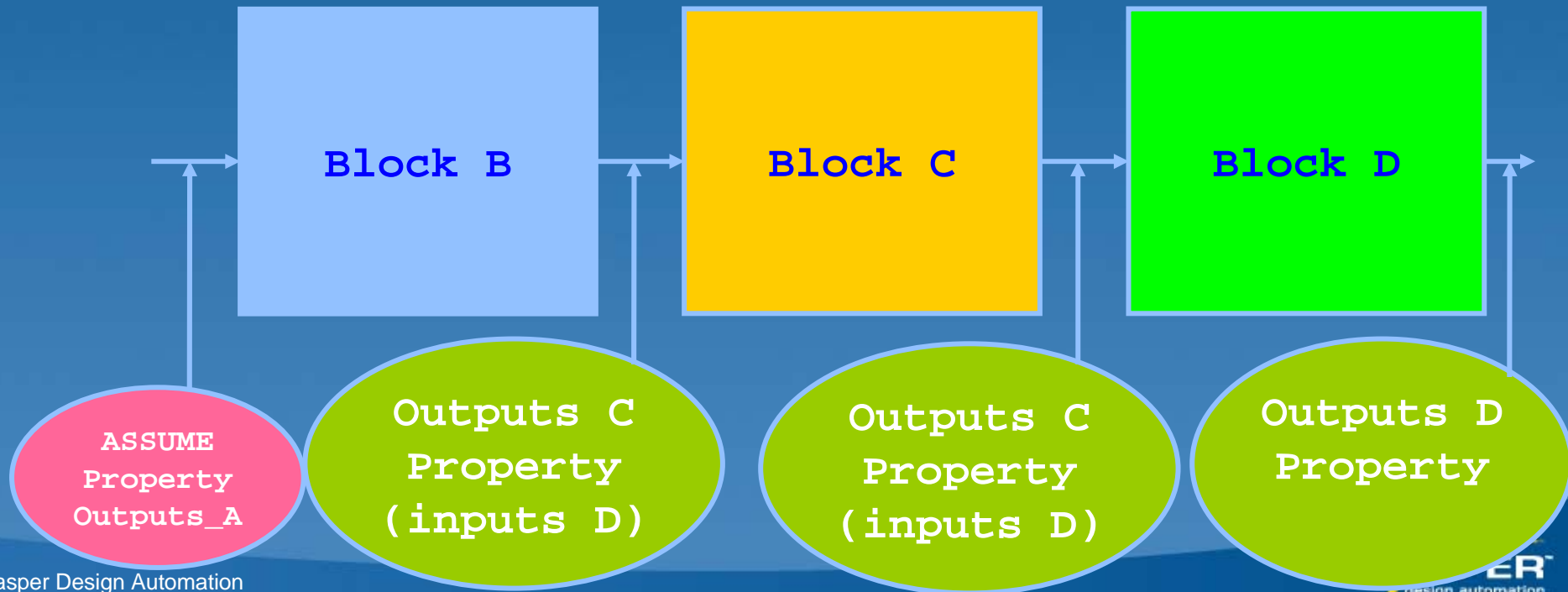
Block D

**ASSUME
Property
Outputs_A**

**End-to-end
Property B->D**

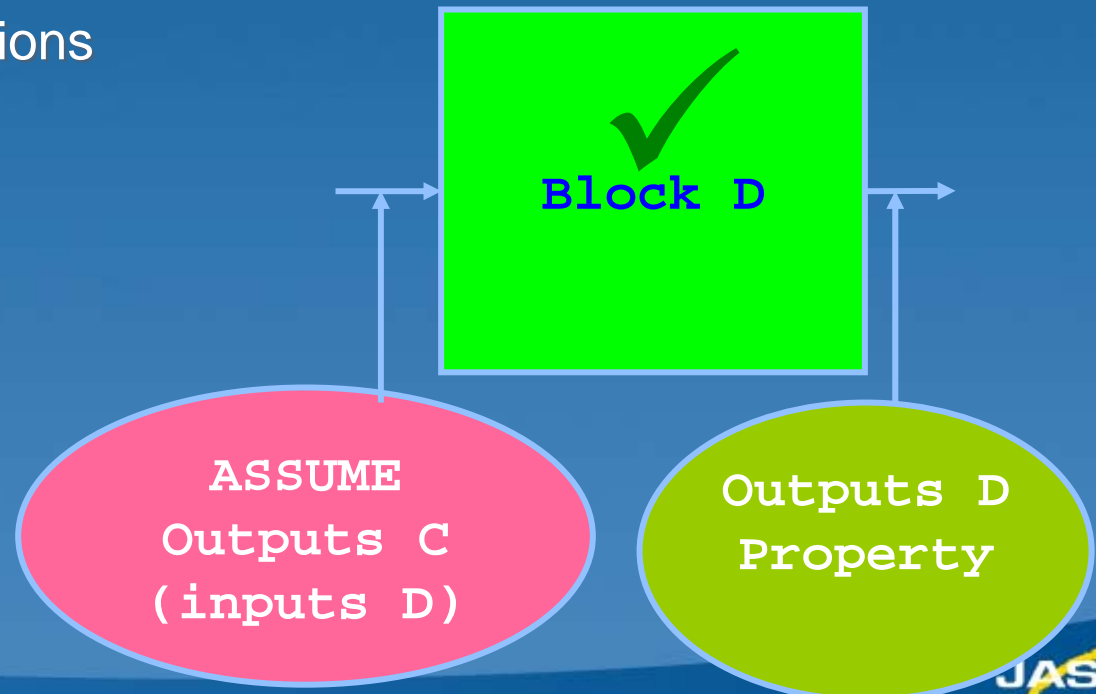
Break the End-to-End Property into 3 Properties

- The end-to-end property was broken into 3 properties



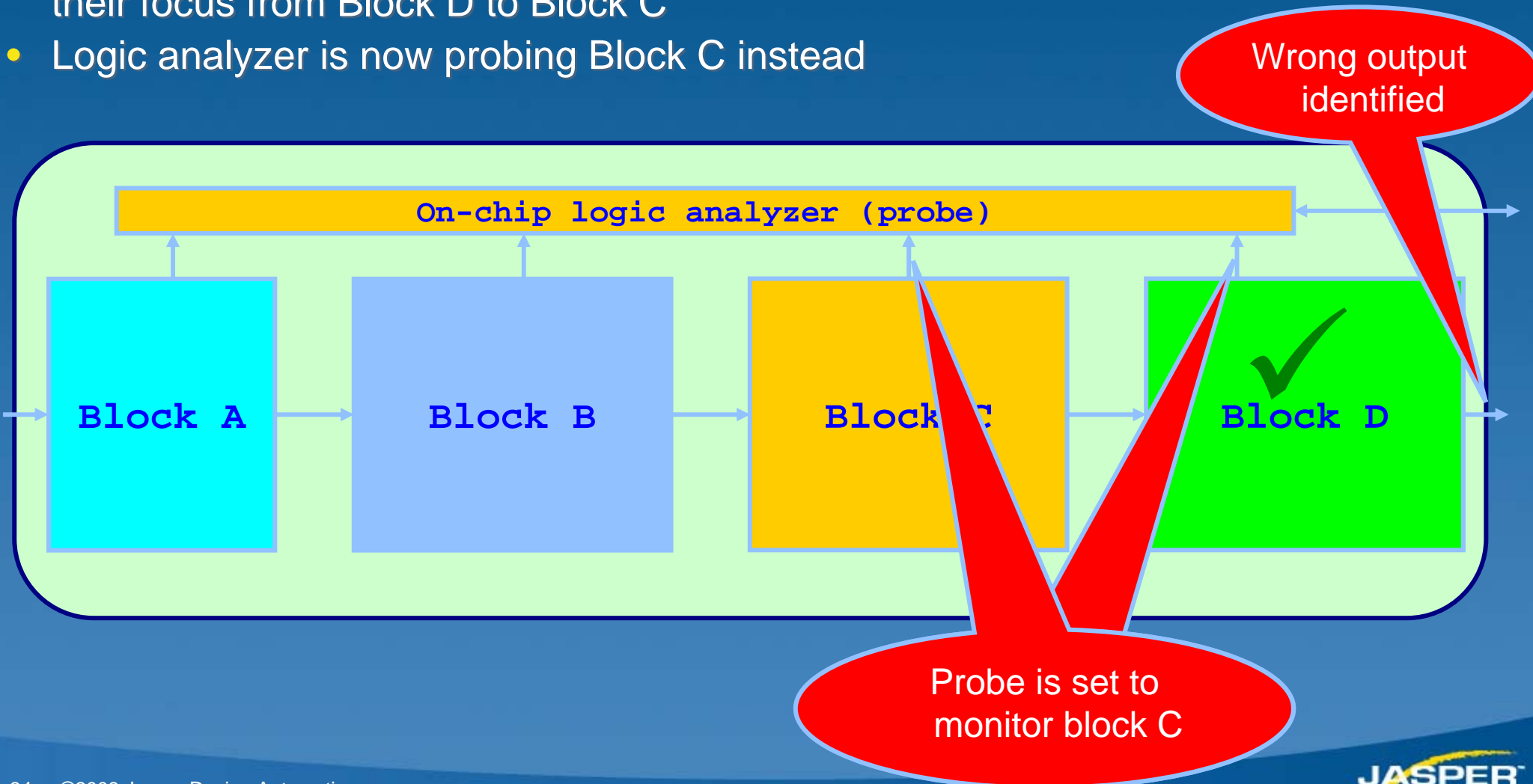
Since Block D Is the “Suspect,” Start Proving Its Properties

- Run the proof engines on the properties on Block D's outputs
- Property was proven
 - Block D does not have the bug
 - The failure trace cannot happen
- So why do we see this trace in the lab???!
 - Maybe the input D assumptions do not hold on outputs of C



Block D Is Cleared, Block C Is the Suspect

- Using the exhaustive answer from formal team, the lab team moved their focus from Block D to Block C
- Logic analyzer is now probing Block C instead



Block D Is Cleared, Block C Is the Suspect

- Using the exhaustive
- their focus from Blo
- Logic analyzer is n

removed

**Information from formal
team helping lab team
focus their scope of work**

Wrong output
identified

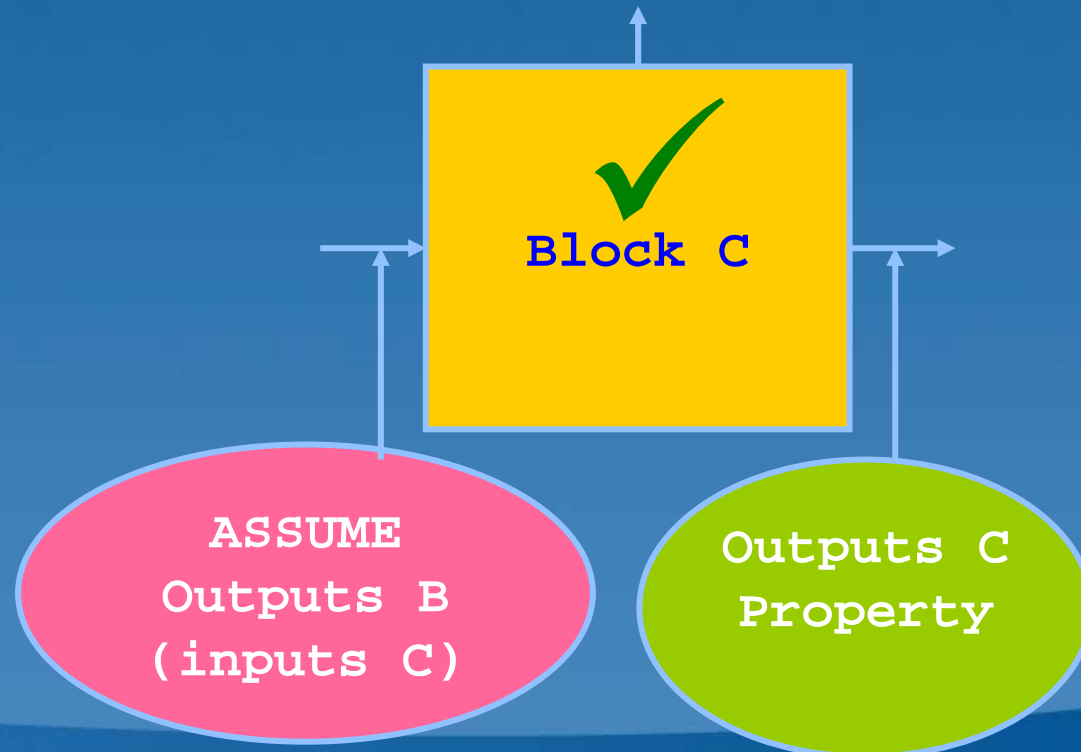
Block A

Block D

Probe is set to
monitor block C

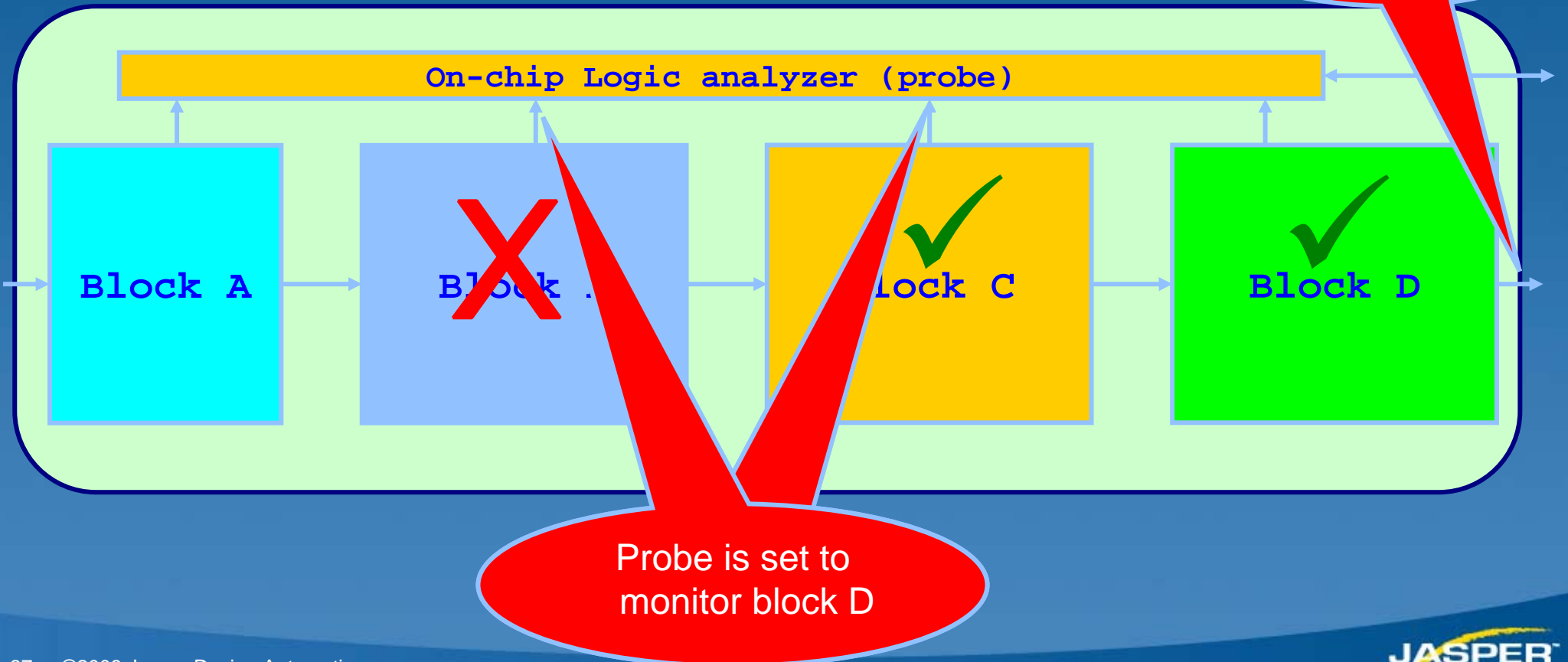
Formal Team Proves the Properties on C's Outputs

- Using the information from the lab team...



Block C Is Cleared, Block D Is the “Suspect”

- Lab team moves the focus to Block B, with full confidence the bug is there
- Probe is moved to Block B
- Bug is found by the lab team when they



Summary

- Formal can play key role in the post-silicon lab
- Saves time, \$\$\$, and reputation
- Use the power of formal for bug-hunting
- Case Study 1:
 - Formal totally wins over simulations: seconds vs. weeks of run time
 - Found another bug in the fixed RTL!
- Case Study 2:
 - Better approach: Use formal in the lab from day 1 (once a bug is found)
 - Formal team and lab team work hand-in-hand, feeding information to each other
 - Use exhaustiveness of formal to rule out the existence of the bug in a given block
 - Information from each team helps the other team focus their efforts
- You need Formal tool with capacity
- You need experience in formal, ahead of time
- Maybe, if formal was used in pre-silicon verification, we wouldn't be doing post-silicon verification 😊