2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Design Guidelines for Formal Verification

Anamaya Sullerey – Juniper Networks

JUNIPER NETWORKS

Impact of design style on Formal Verification

- Formal verification success and efficiency are very much dependent on the design style – not necessarily the case with simulation
- Design implementation choices can
  - Make the proofs not converge
  - Make the design unfriendly for application of formal verification techniques
  - Significantly grow the test-bench, constraints, and assertion development effort

Objectives of this paper

- Current literature is targeted towards FV engineers with emphasis on methods and techniques used in formal verification
- Focus of this paper is on
  - design styles that enable formal verification
  - processes that help adhere to these guidelines

Design Approaches

- *Functional Design* is composed of a hierarchy of modules where each module performs a well defined function with minimal side effects
- *Event driven design* is centered on performing certain actions in response to the observed events
- *Data driven design* is centered on performing certain actions based on the observed data
- A typical design uses mix of these approaches

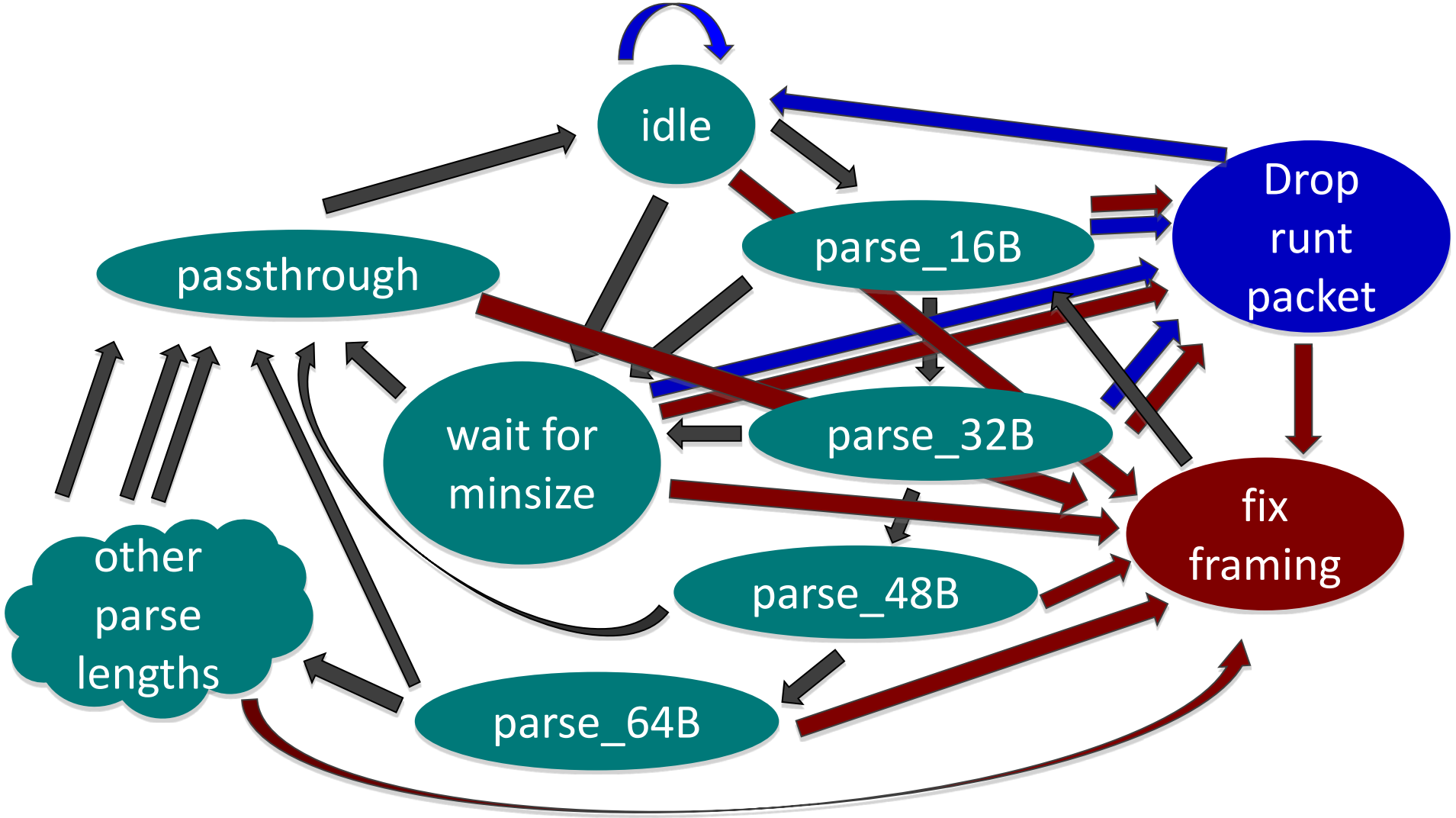
Guideline 1: Functional design paradigm

- Functional design approach is best suited for formal verification
- It facilitates application of the “divide and conquer” approach
- It reduces “proof-debug-fix” loop time resulting in speedy verification process
- “Assume-Guarantee” propagation can be applied for overall correctness

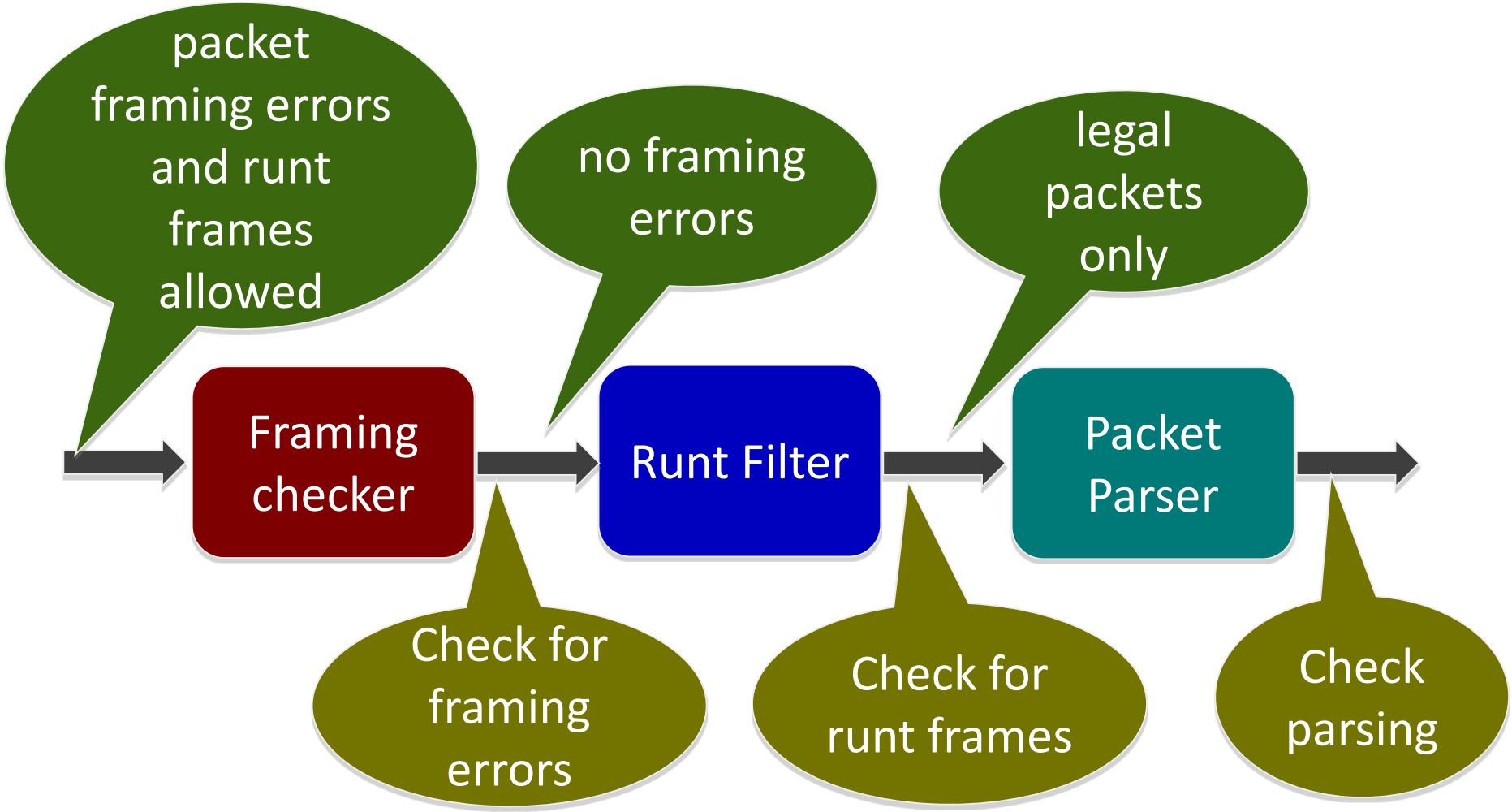
Example : Ethernet packet parser

- Interface
  - Start of packet (SOP), End of packet (EOP), Data (128 bits), Error, Valid
  - No interleaving of packets
- Functionality
  - Parses networking headers
  - Drops runt packets (<40B)
  - Fixes framing errors (SOP-EOP rules)

Event driven approach

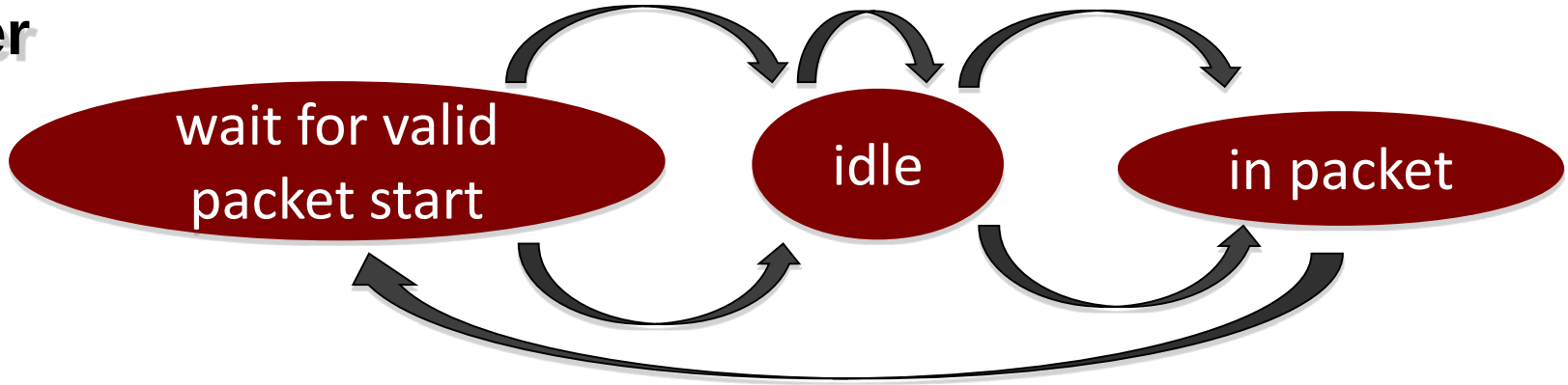


Functional design approach

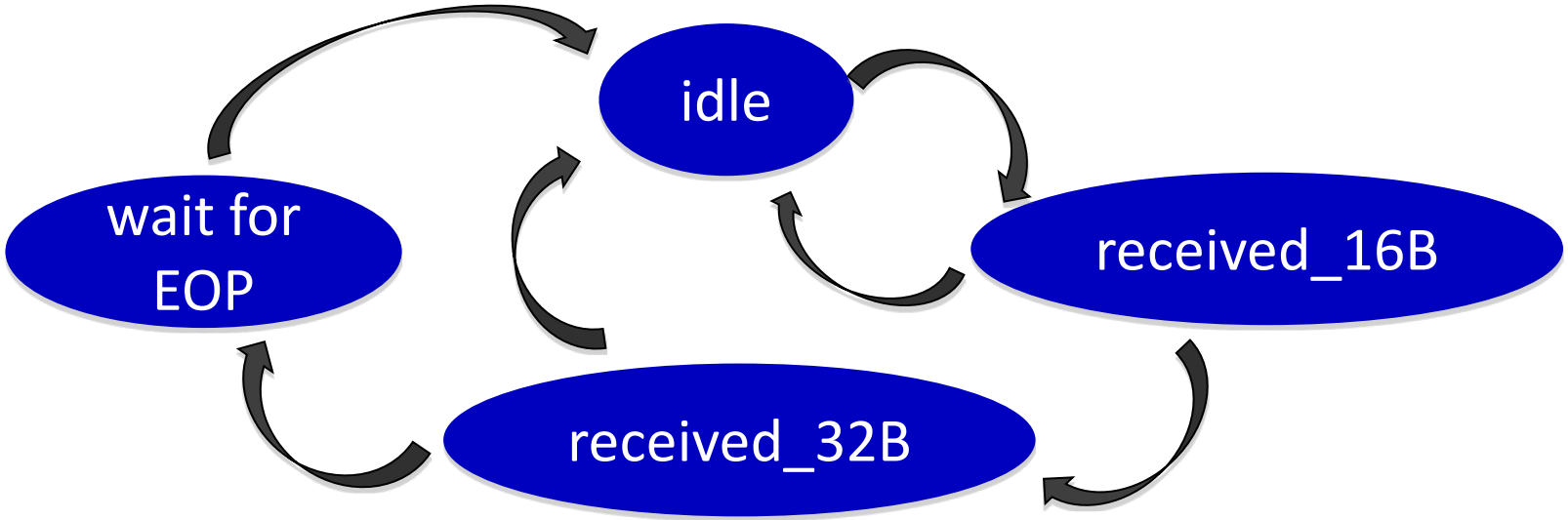


Untangled state machines - I

Framing checker

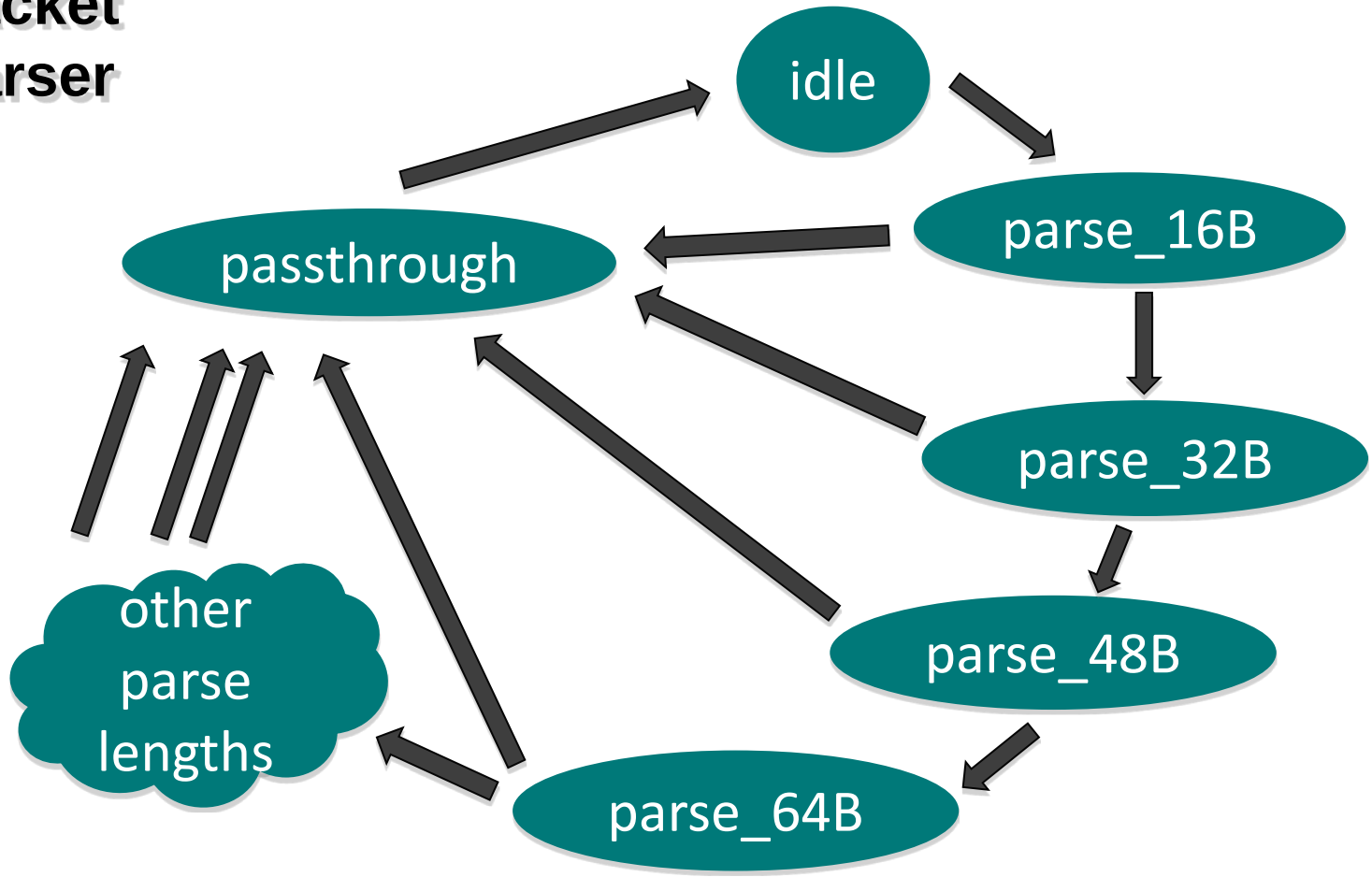


Runt filter



Untangled state machines - II

Packet parser



Guideline 2: Clear and succinct interface definitions

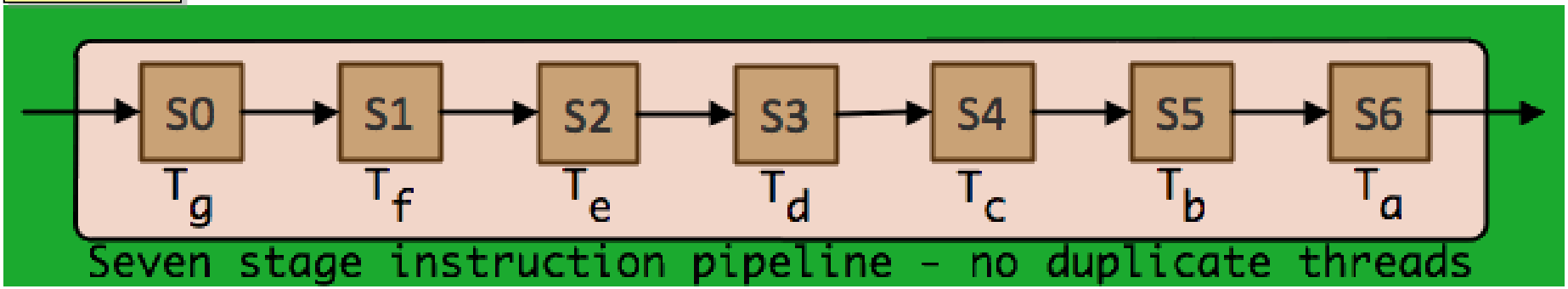
- Interface definitions affect the state space of the properties
- Interface definitions affect testbench, assertion set, and assumption set development effort
- Interface documentation is not always present for sub-blocks
- Characteristics of FV friendly interfaces
  - clean protocol definitions
  - optimal set of signals
  - explicit means of handshake and information transfer

Guideline 3: State space as design consideration

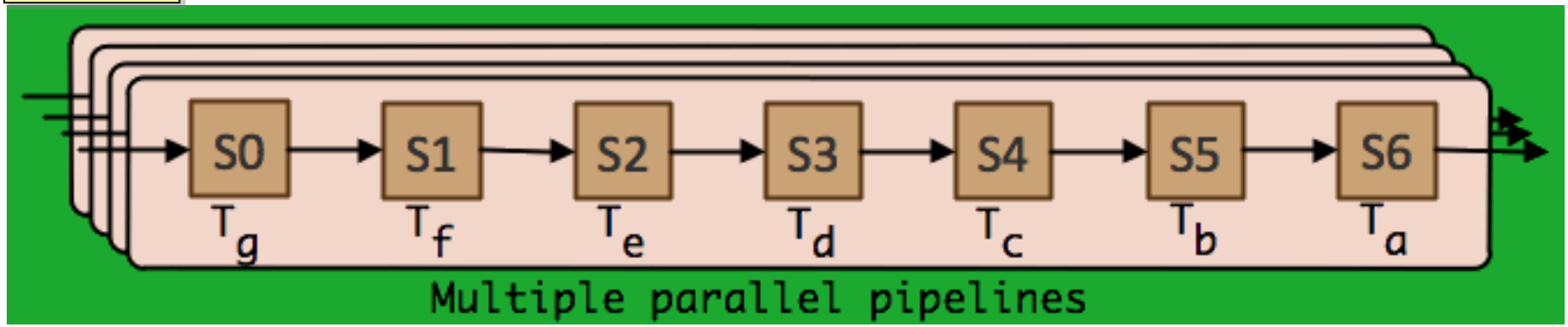
- Large blocks run into capacity limitations
  - Subdivide blocks that are too big and complex
  - Seek early feedback from the FV team for complex blocks
  - Expose Designers to formal verification to provide a feel of the tool capacity
- State space of a property is a function of
  - Cone of influence (COI) - All primary inputs and logic affecting the property
  - Connectivity within the COI

Example: Complexity and COI (I)

Case 1

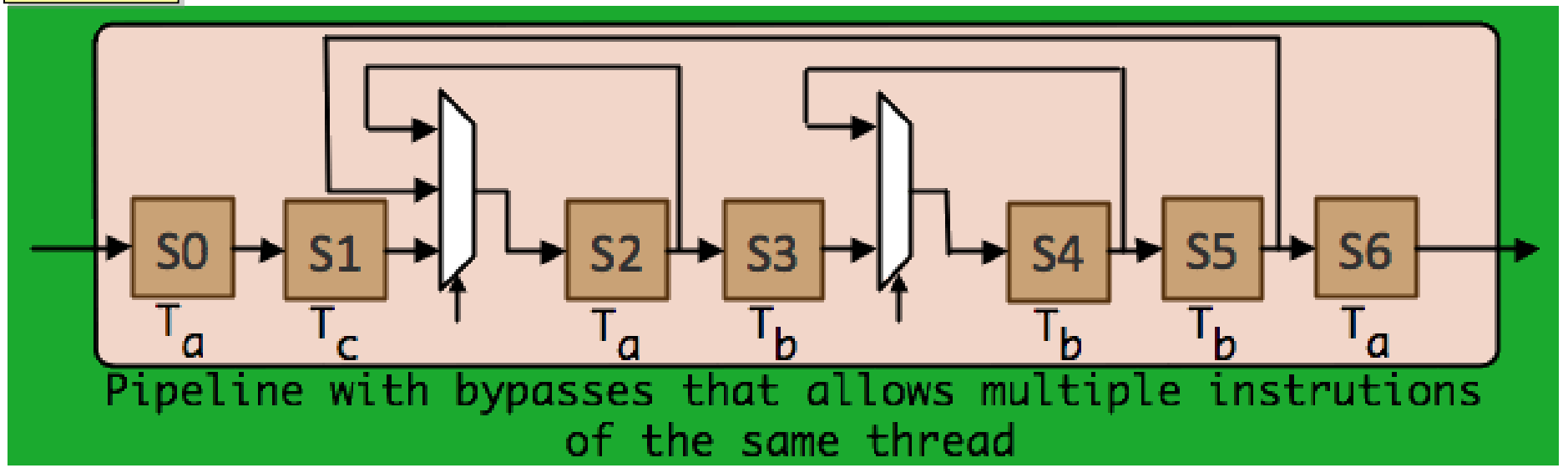


Case 2



Complexity and COI (II)

Case 3



Case 2 has the largest design and possibly a larger COI for many properties, Case 3 has the highest complexity

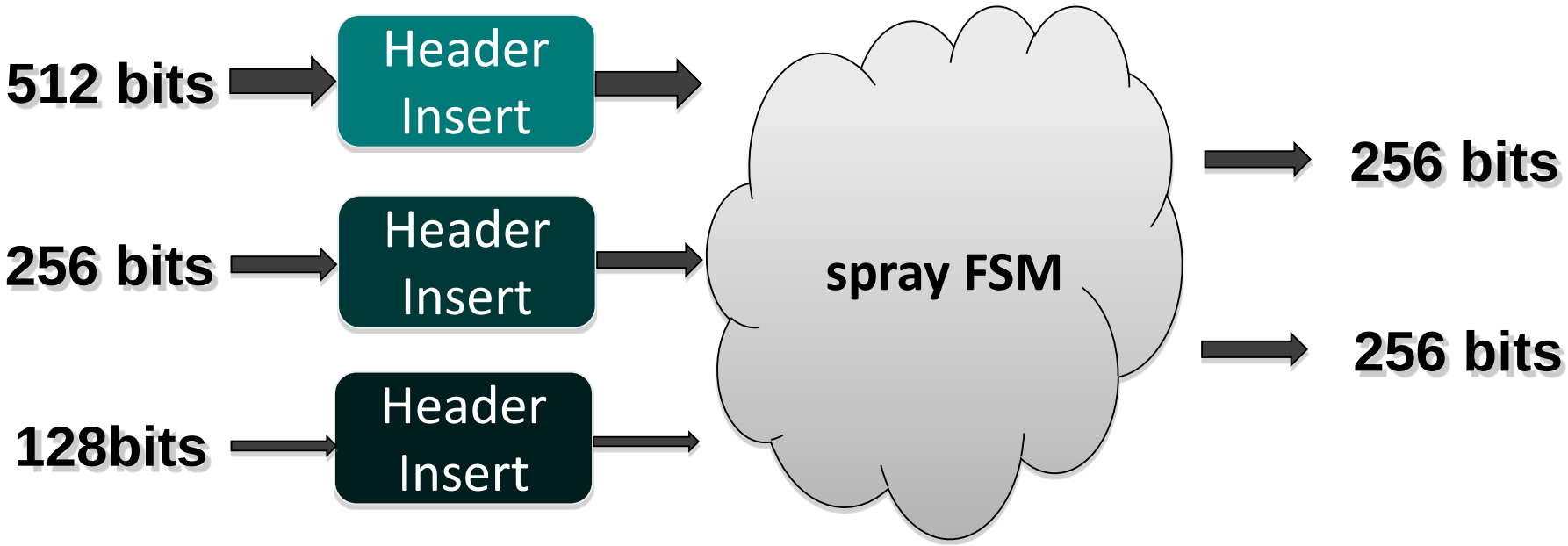
Guideline 4: Symmetry

- Symmetry is exploited by FV tools to reduce the state space
- Assertions and assumptions for symmetric designs require less effort
- Fewer unique sub-blocks in a symmetric design, less sub-blocks to verify
- Isolate asymmetry in designs that are largely symmetric

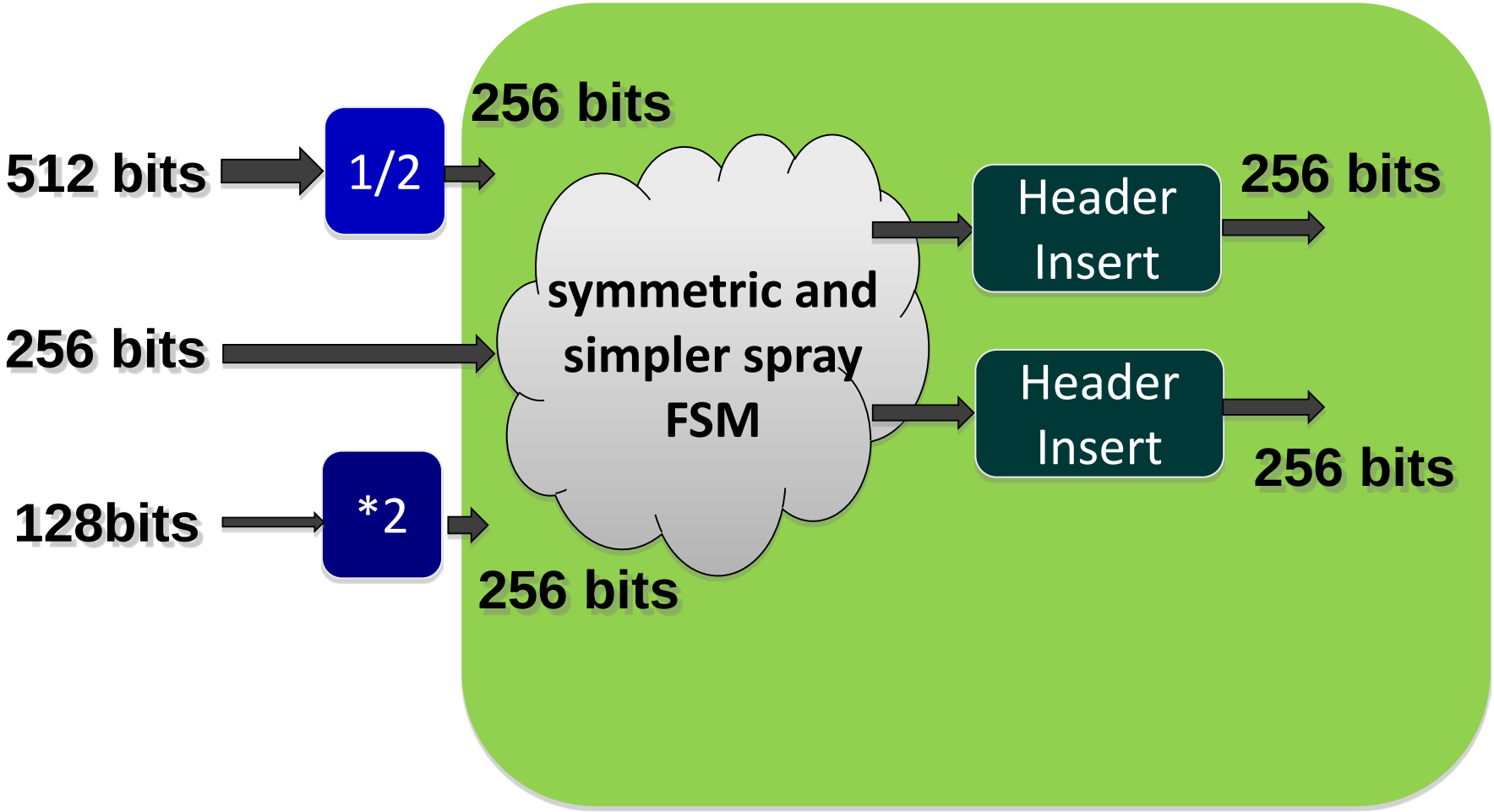
Example: Isolating symmetry

- Consider a logic working on packets that has
  - three input source interfaces (128bits, 256bits, and 512bits)
  - two output destination interfaces (256bits, 256bits)
  - A packet spray engine
- Function of this design is to
  - add a 512 bit header to all incoming packets
  - arbitrate among the source interfaces and spray packets uniformly to the destination interfaces

Asymmetric implementation



Symmetric implementation

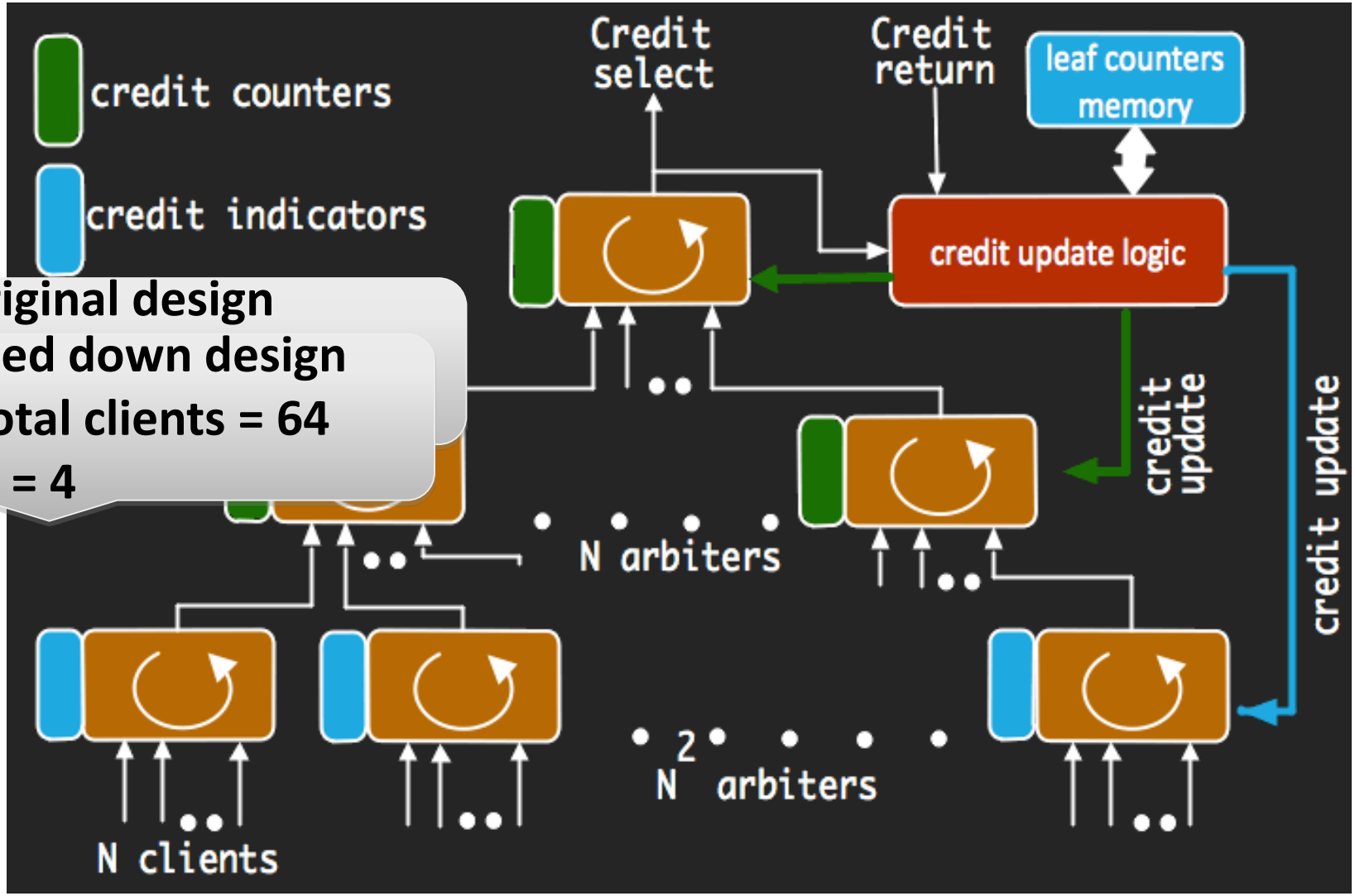


Symmetric design

Guideline 5: Parameterized designs

- Parameterization is utilized to scale down large designs without affecting their key aspects
- Designs can be parameterized using System Verilog parameters or Verilog pre-processors
- Formal verification of scaled down design allows proofs to converge
- Formal verification of scaled down design provides quick turnaround time

Example: Parameterized scheduler



Original design

Scaled down design

- **Total clients = 64**
- **N = 4**

Guideline 6: Assertions for the design invariants

- Design blocks have invariants (rules about state, events etc) around which the code is structured
  - One hot bit vector
  - Guarantee of a grant for any request in N cycles
  - Certain timeout never happens in low power state
- Most violations of design invariants lead to a design bug
- Assertions based on the design invariants
  - allow quick debug
  - guides the tool with other proofs

Guideline 7: Code structure

- Poor code layout increases the effort required for FV and makes the process error prone
- Isolate independent, complex, deep state logic (like LFSRs, Crypto functions) into separate modules for easy abstraction
- Instantiate memories outside of logic
- Create expressions composed of meaningful intermediate terms - helps in cut-point insertion and partial proofs

Guideline 8: Error isolation

- Many designs process a large set of independent symmetric contexts (network flows, cache lines)
- A common technique applied for such designs models a single context
- Proof of a property for the modeled context proves the correctness for all contexts
- Illegal inputs are part of the input space of the unconstrained contexts
- This technique works only if an error in one context does not affect the state of any other context

Example: MESI protocol verification

- Valid states for a cache line: modified(M), exclusive(E), shared(S), and invalid(I)
- Cache-controller design keeps state for all cache-lines and operates on few of those at a given time
- Read operation is allowed in M, E, and S states, similar rules for write and other operations
- Single cache line modeled for formally verifying the design
- Illegal state or operation on other cache lines should not affect modeled cache line

Adherence to the guidelines

- Provide good literature on Formal Verification to the designers – wiki-pages, papers
- Provide good examples of formal friendly designs
- Encourage designers to do formal verification
- Have formal test plans
- Involve formal verification team early in the design process
- Make formal verification requirements a part of various reviews

Reviews

Guideline	Review
Functional design paradigm	Micro-architecture
Clear and succinct interfaces	Interface
Keeping state space as design consideration	Micro-architecture, Formal test plan
Symmetry	Micro-architecture, Formal test plan
Parameterization	Formal test plan
Capturing design invariants	Assertion and coverage
Code structure	Code
Error isolation	Functional spec, Micro-architecture

Acknowledgements

- Grateful for the feedback of the reviewers
  - Sanjeev Singh, Jonathan Sadowsky, and David Talaski @ Juniper
  - Erik Seligman @ Intel