COMS31700 Design Verification:

Verification Hierarchy and Fundamentals of Simulation-Based Verification

Kerstin Eder

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL

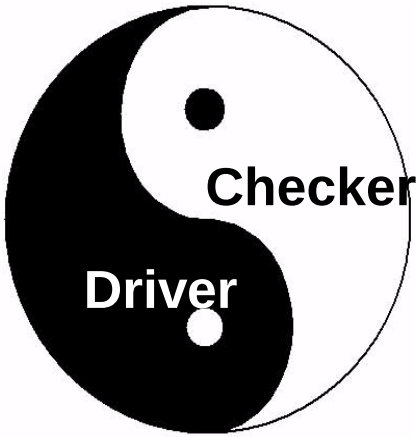
Department of COMPUTER SCIENCE

Last Time

- What is (functional) verification?
- Why verification is so important
- Verification challenges
- Verification techniques
  - Testbenches
- Reconvergence models
- Observability

Outline

- Verification hierarchy
  - Levels of verification
- Fundamentals of Simulation-based Verification:
  - Strategy
    - Driving principles
    - Checking strategies



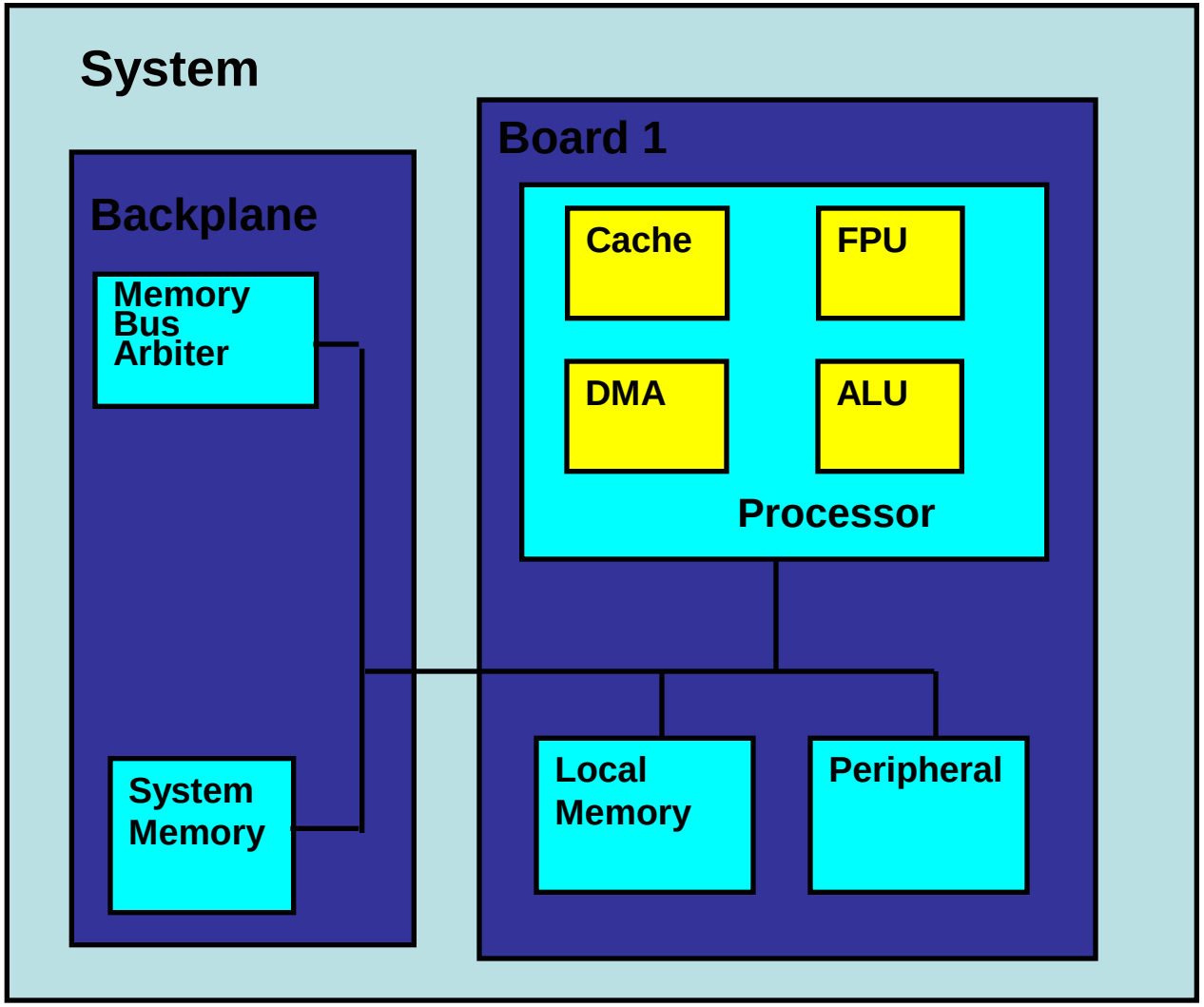
Verification Hierarchy

- Today's complex chips and systems are divided into logical units
  - Usually determined during specification / high-level design
  - Usually follow the architecture of the system
  - This practice is called hierarchical design
- Hierarchical design allows a designer to subdivide a complex problem into more manageable blocks
  - The design team combines these blocks to form bigger units, and continues to merge these blocks until the chip or system is complete

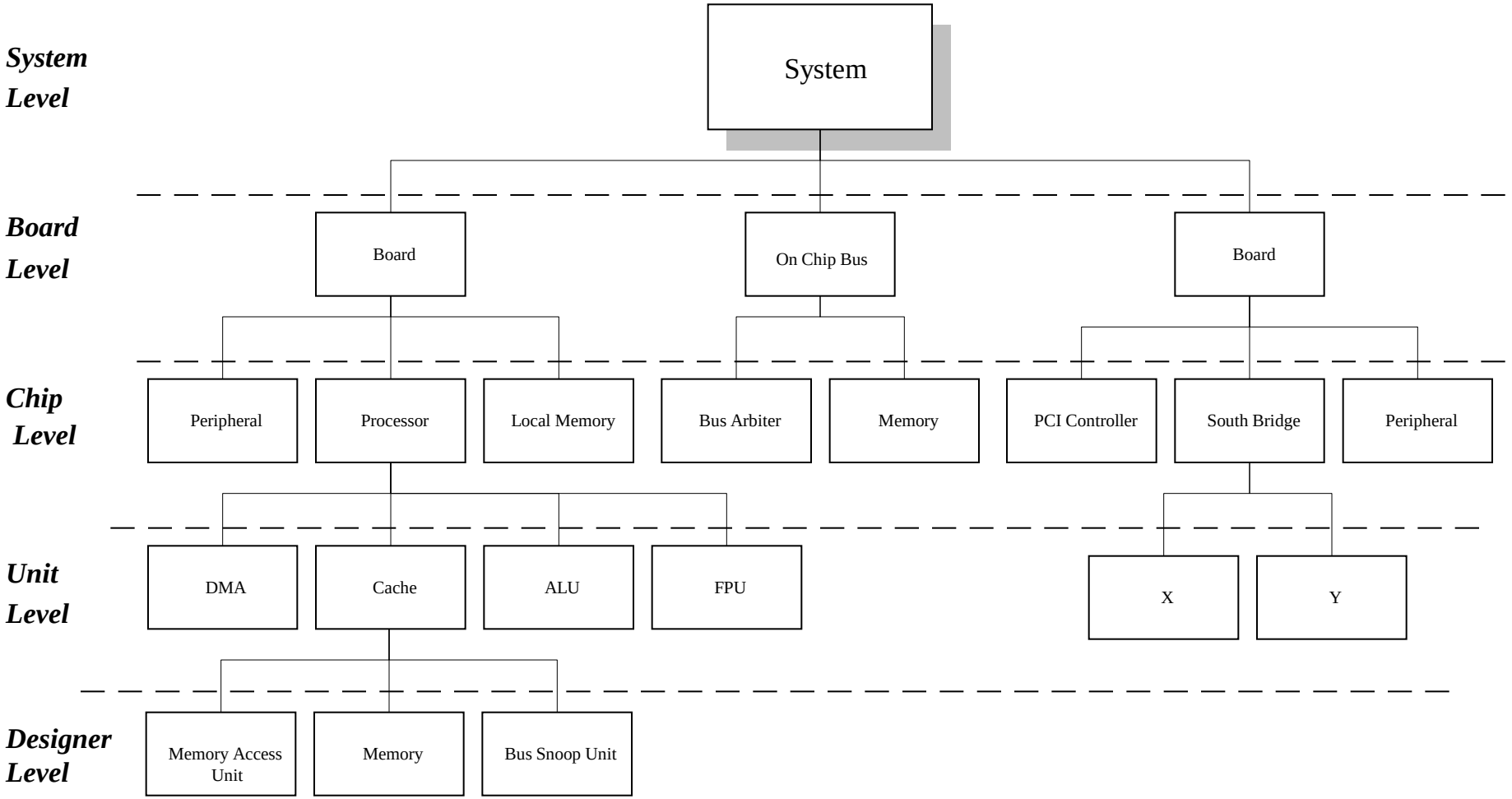
Pros and Cons of Hierarchical Design

- Pros
  - Breaks the design into manageable pieces
  - Allow designers to focus on single function / aspect of the design
- Cons
  - More interfaces to specify / design / verify
  - Integration issues

Large System Diagram



Large System Hierarchy



Levels of Verification

- Verification usually adapts to and takes advantage of the hierarchical design stages and boundaries
- Common levels of verification
  - Designer level (block level)
  - Unit level
  - (Core level)
  - Chip level
  - Board level
  - System level
  - Hardware / software co-verification

Designer (Block) Level Verification

- Used for verification of single blocks and macros
- Usually, done by the designer him/herself
- Main goal – Sanity checking and certification for a given block
- Ranges from a simple test of basic functionality to complete verification environments
- The common level for formal verification

Unit Level Verification

- A set of blocks that are designed to handle a specific function or aspect of the system
  - E.g., memory controller, floating-point unit
- Usually have formalized spec
  - More stable interface and function
- The target of first serious verification effort
- Verification is based on custom-made verification environment

Core Level Verification

- A core is a unit or set of units designed to be used across many designs
  - Well defined function
  - Standardized interfaces
- Verification need to be thorough and complete
  - Address all possible uses of the core
- The verification team can use “Verification IP” for the standardized interfaces

Chip Level Verification

- Verification of a set of units that are packaged together in a physical entity
- Main goals of verification
  - Connection and integration of the various units
  - Function that could not be validated at unit level
- Need verification closure to avoid problems in tape-out

Board/System Level Verification

- The purpose of this level of verification is to confirm
  - Interconnection
  - Integration
  - System (or board) design
- Verification focuses on the interactions between the components of the system rather then the functionality of each individual component

HW / SW Co-Verification

- Marries the system level hardware with the code that runs on it
- Combines techniques from the hardware verification and software testing domains
- This combination creates many issues
  - Different verification / testing techniques
  - Different modes of operation
  - Different speed
- Beyond the scope of this course

Which Level To Choose?

- Always choose the lowest level that completely contains the targeted function
- Each verifiable piece should have its own specification
- New or complex functions need focus
- Function may dictate verification levels
  - The appropriate level of control and observability drives decisions on which levels to verify

Which Level To Choose?

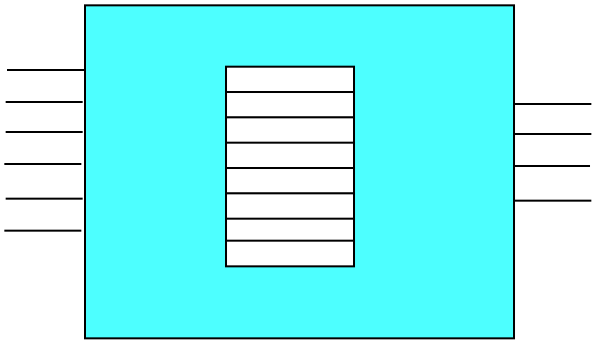
- In general, each level that is exposed to the “outside world” is mandatory
  - For example, chip level, system level
- The rest depends on many factors
  - Complexity
  - Risk
  - Schedule
  - Resources

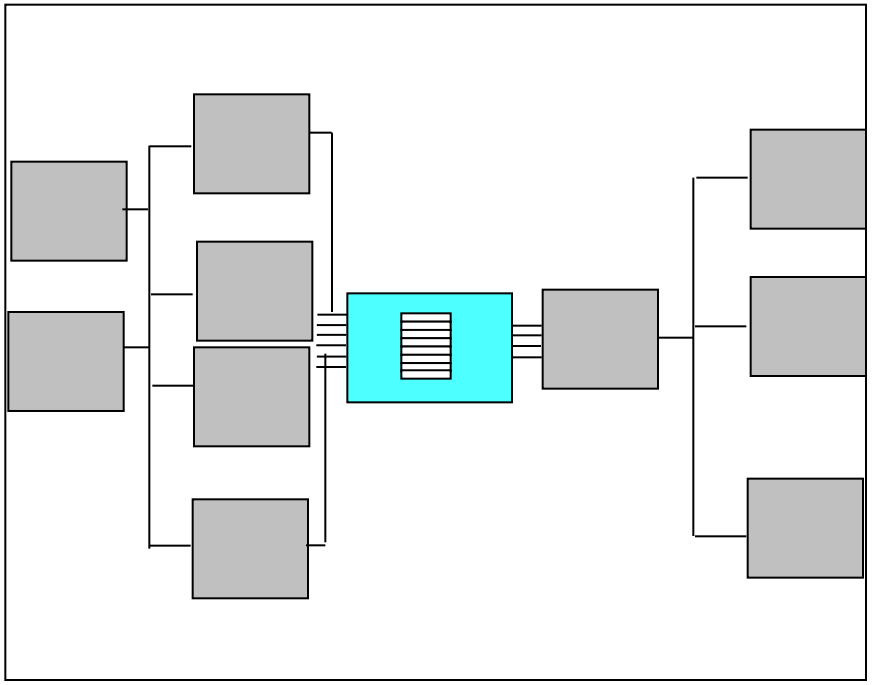
Controllability and Observability

- Controllability - Indicates the ease at which the verification engineer creates the specific scenarios that are of interest.

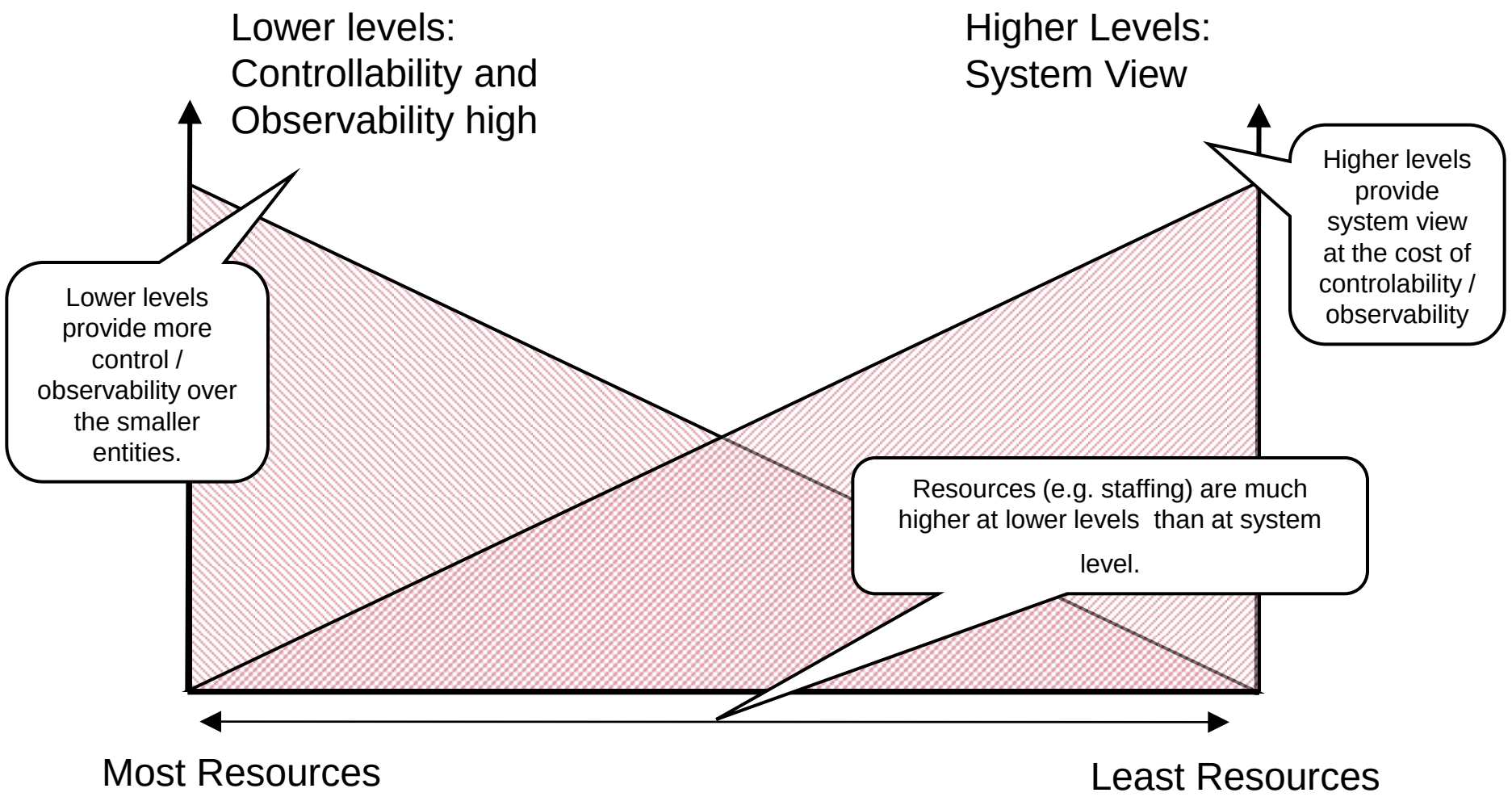
- Observability - Indicates the ease at which the verification engineer can identify when the design acts appropriately versus when it demonstrates incorrect behavior.

Controllability and Verification Levels

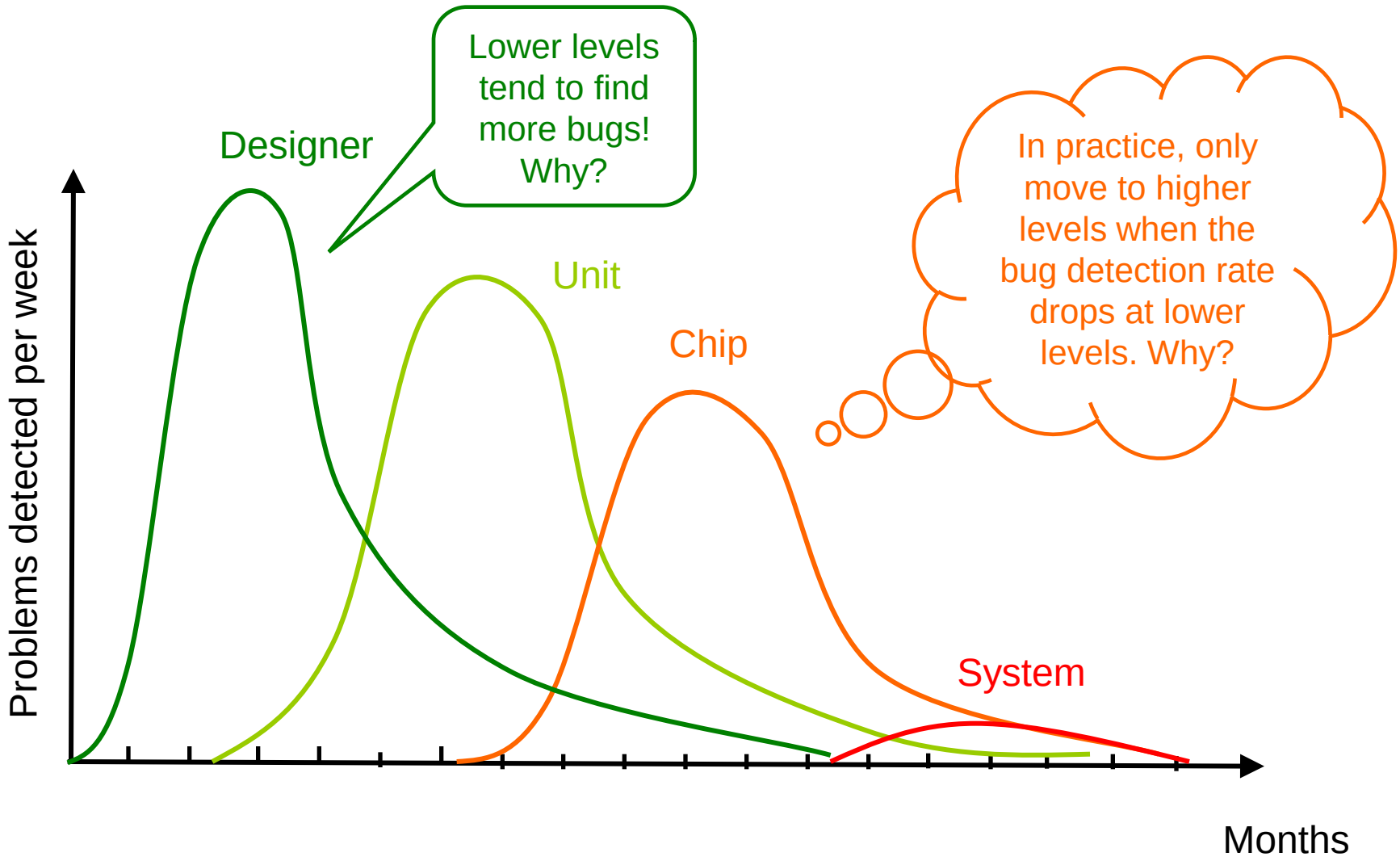




Trade-offs in Verification Hierarchy



Bug Detection



Fundamentals of Simulation-based Verification

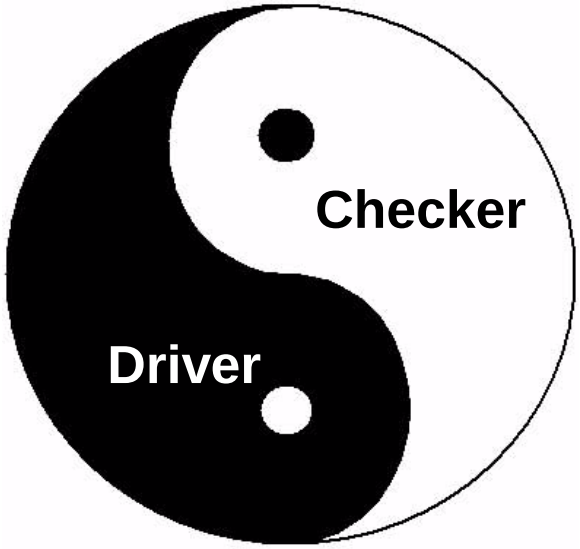
The Strategy of
Driving & Checking

Strategy of Verification

- Verification can be divided into two separate tasks
  1. Driving the design - Controllability
  2. Checking its behavior - Observability
- The basic questions a verification engineer must ask
  1. Am I driving all possible input scenarios?
  2. How will I know when a failure has occurred?

The Yin-Yang of Verification

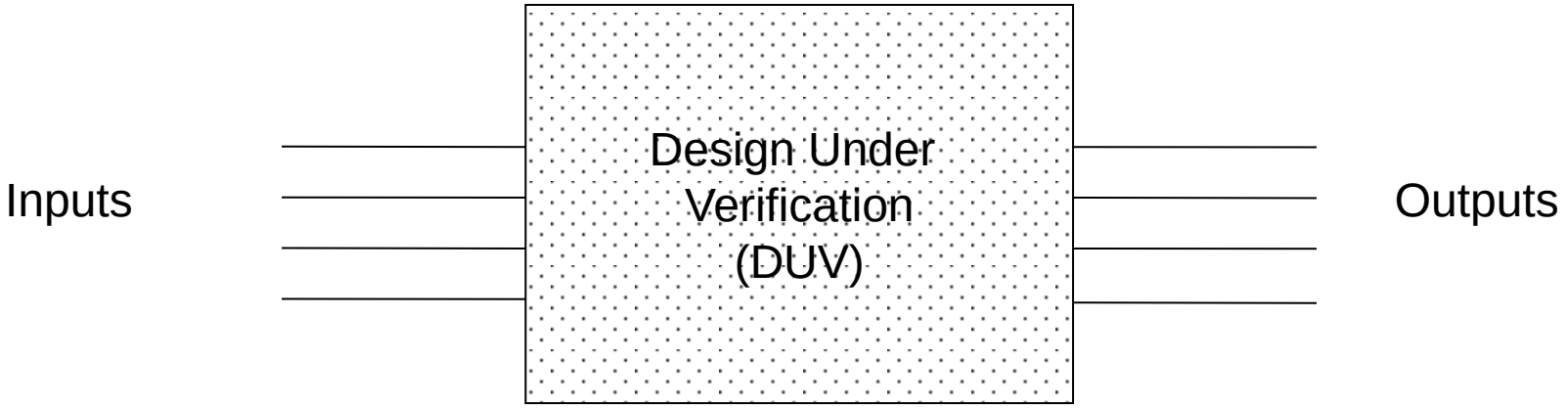
- Driving and checking are the yin and yang of verification
  - We cannot find bugs without creating the failing conditions
    - Drivers
  - We cannot find bugs without detecting the incorrect behavior
    - Checkers



Comments on Yin and Yang

- This perfect harmony does not always exist
  - Not all failing conditions are equal
    - Same bug can lead under different failing conditions to different failures (with big difference in consequences)
  - We cannot (or don't want to) detect all incorrect behaviors
    - Some are not important enough
    - For others we have safety nets
- The right balance is a function of the level of verification and specific needs
  - Example: Block vs Chip level verification – difference in drivers and checkers and in focus of verification.

The Black Box Example



- What does it mean to
  - Drive all input scenarios
  - Know when the design fails

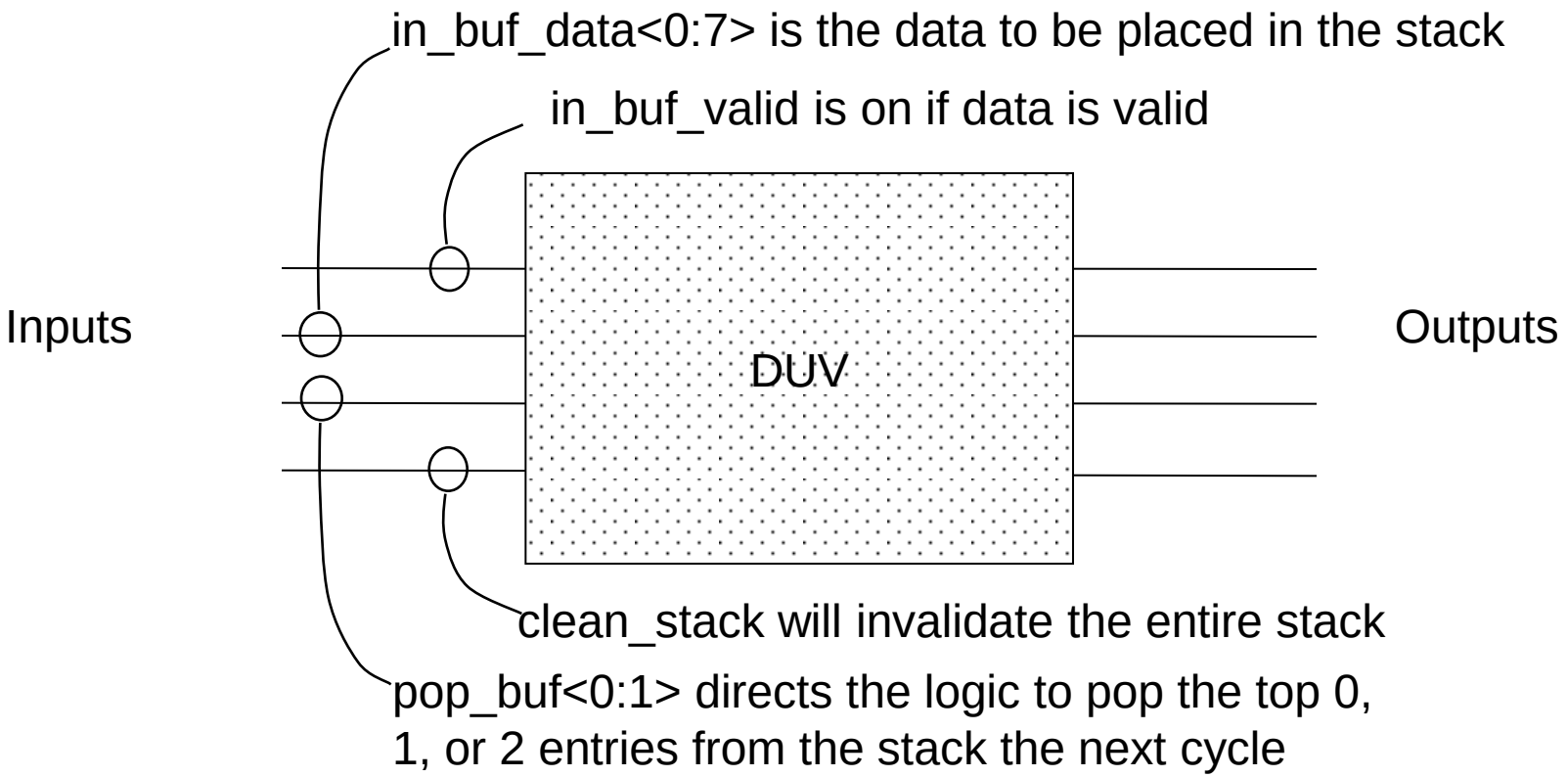
Verification of the Black Box

- Black box since we don't look inside it
  - What does this mean?
- The black box may have a complete documentation … or not
- To verify a black box the verification engineer must understand the function and be able to predict the output based on the inputs.
- It is important that the verification team obtain the input, output and functional description of the black box from a source other than the HDL designer
  - Standard specification
  - High-level design
  - Other designer that interfaces with the black box
  - …

Driving the Black Box

- We can start planning the stimuli even before the complete specification of the DUV is given
- The definition of the inputs can provide information and hints on
  - The interface
  - The functionality
- This information can lead to first set of stimuli
- More stimuli will be added as we learn more details on the DUV

Driving the Black Box



What Can We Learn From This?

- We can start understanding the design just from the input descriptions:
  - What do we know?
  - What don't we know?

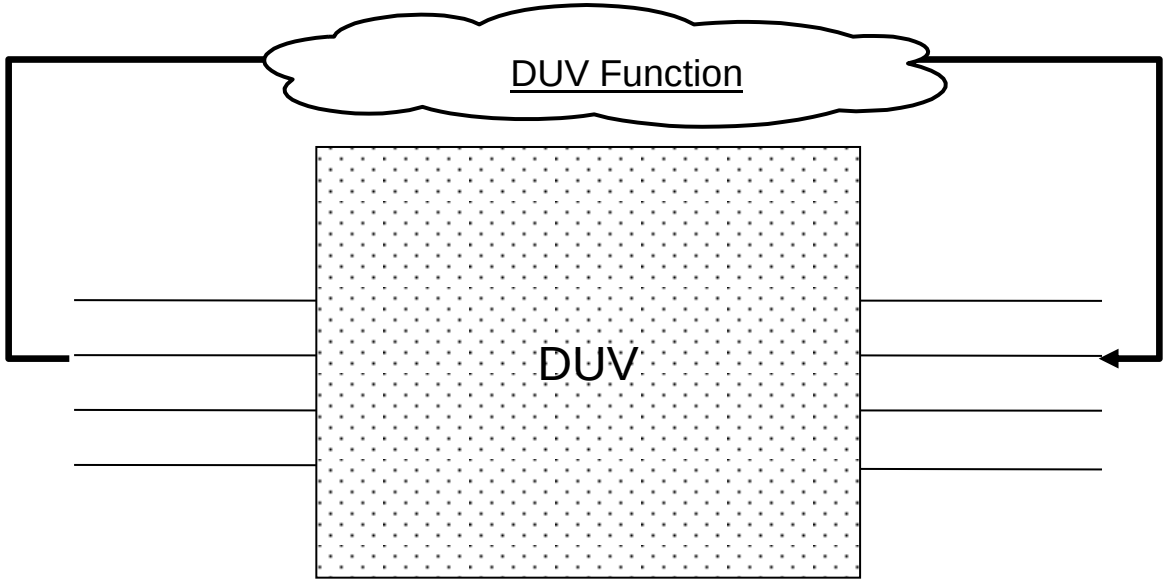
What can we set up?

- Writing to the stack
  - Back-to-back writes
  - Long sequences of writes
- Reading from the stack
  - All three possible reads (0, 1, 2 reads)
  - Back-to-back and long sequences
- Corner cases
  - Reading from an empty stack (and almost empty)
  - (Writing to a full stack (and almost full))
- Combinations and scenarios
  - Two or three of read, write, clean

Checking Strategies

- There are five main sources of checkers
  - The inputs and outputs of the design (specification)
  - The architecture of the design
  - The microarchitecture of the design
  - The implementation of the design
  - The context of the design (up the hierarchy)
- Note that the *source of checkers* and their implementation are two different issues

Checking Based On the DUV I/O



- Check the output signals of the DUV based on
  - The input signals
  - Understanding of the specification of the DUV

Checking Based On the DUV I/O

- The most basic type of checking
- Must be present unless we are certain that this type of checking is covered by other types of checking
- The checker need not (and should not) imitate the design
- Checking is easier than implementing the DUV
  - Can use higher level of abstraction
  - Need to verify the outputs instead of generating them
- Verification should not enforce, expect nor rely on an output being produced at a specific clock cycle (Why not?)

Checking Based On the Architecture

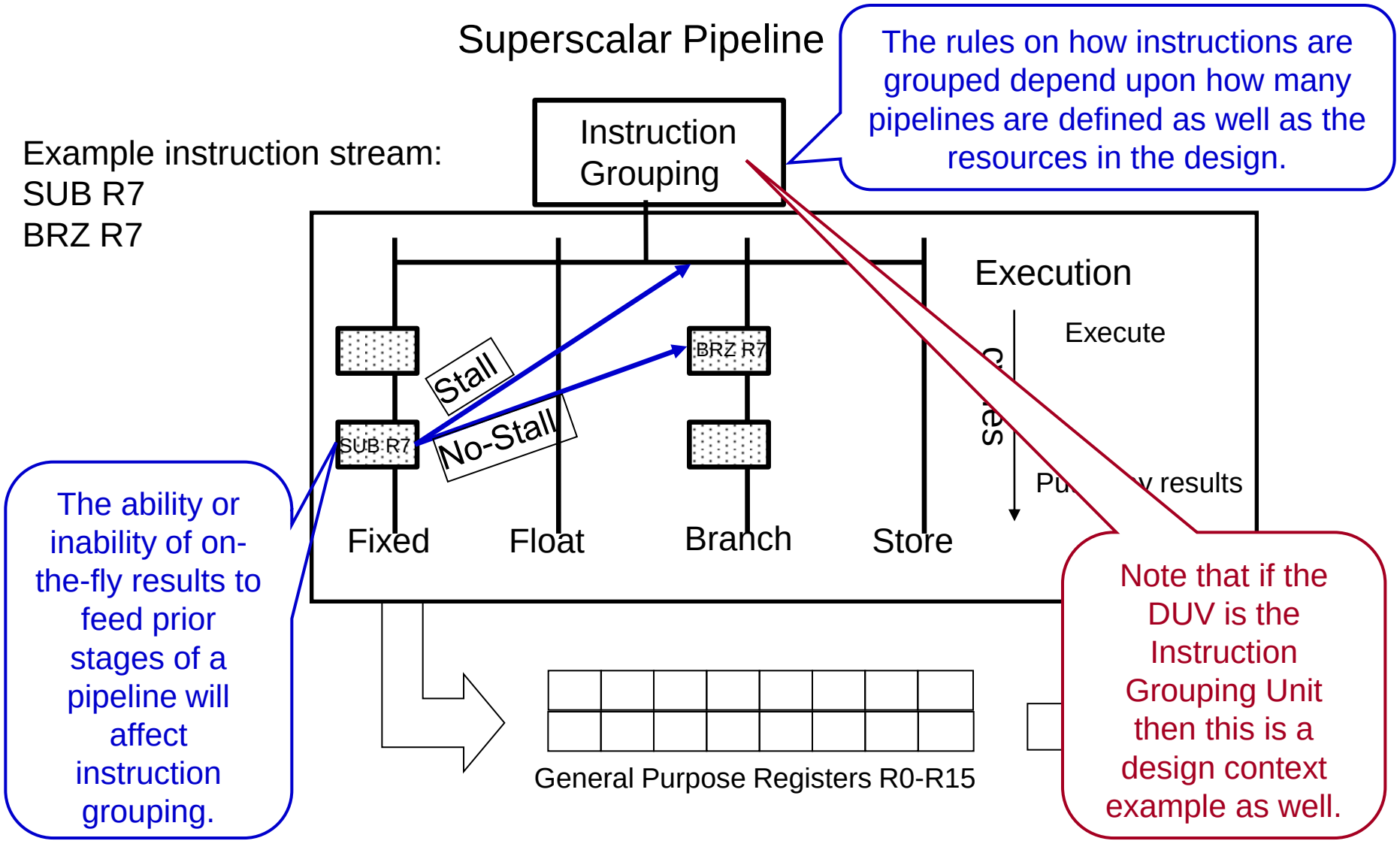
Example instruction stream:
SUB R7
BRZ R7

Architectural checking is abundant.

- The SUB and BRZ instructions are defined in the ISA.
- Architecture defines that instructions must complete in order.
- Architecture defines that results of SUB must be used by BRZ.

Many checkers have their roots in the Architecture of the design!

Checking Based On the Microarchitecture



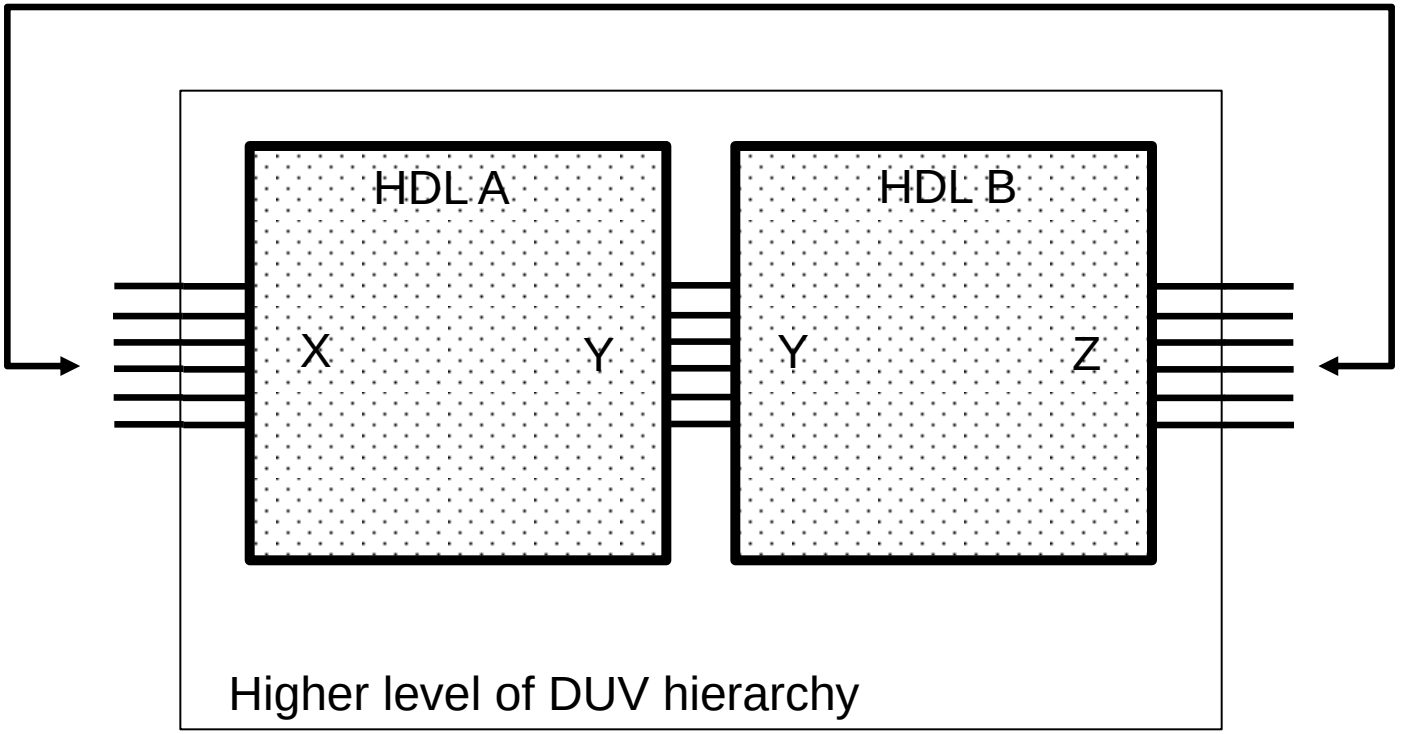
Checking Based On the Architecture and Microarchitecture

- Check that architectural and microarchitectural mechanisms in the DUV are operating as expected
  - Buffers: overflow and underflow
  - Invalid states and transitions
  - Pipelines
  - Writeback and forwarding logic
  - Reorder buffers
  - …

Checking Based On the Implementation

- Check items that are related to specific implementation details
  - Cyclic buffers for queues
  - Pipeline buffer stages
  - …

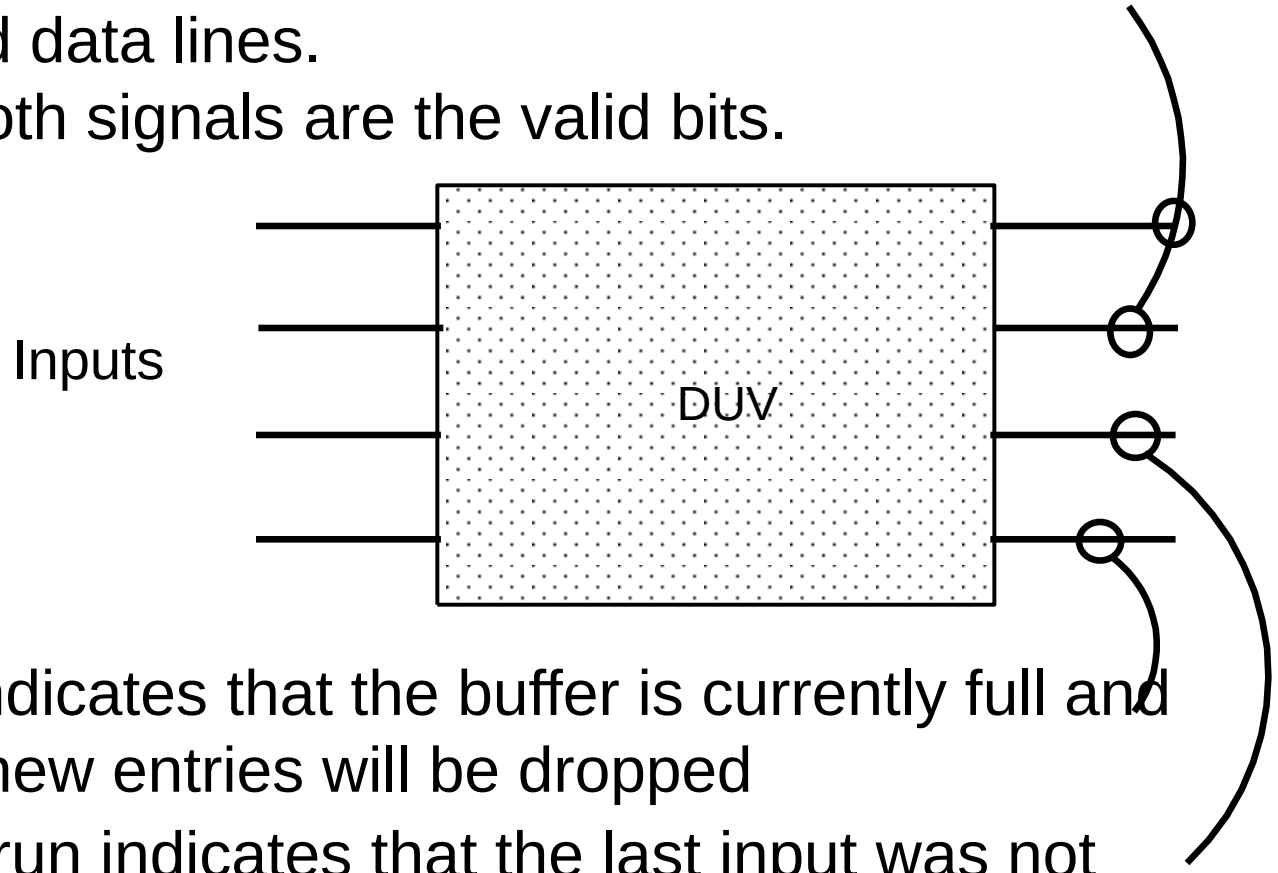
Checking Based On the Design Context



- When verifying lower levels of hierarchy such as individual blocks of HDL, the verification engineer derives checkers from an understanding of the function, properties, and context of the larger design.

Output Definition of the Black Box

out_buf_data1<0:8>, out_buf_data2<0:8> are the requested data lines.
Bit 8 of both signals are the valid bits.



buf_full indicates that the buffer is currently full and that any new entries will be dropped

buf_overrun indicates that the last input was not added to the stack due to an overrun

What Can We Learn From This?

- The outputs give an insight into the scenarios we need to create.
  - What more do we know?
  - Which information is still needed?

Documentation Reveals

- The stack is 7 entries deep.
- The data become valid (for reading) one cycle after it is written.
- We can read and write at the same time.
- No data is returned for a read if the stack is empty.
- Cleaning takes one cycle.
  - During that time we cannot read or write.
  - Inputs arriving with a clean command are ignored.
- The clean command turns the valid bit off on all 7 entries.
- Buffer full is valid one cycle after the buffer is filled.
  - This is why we need the buf_overrun signal.
- The “stack” is a FIFO.

What Can We Learn From This?

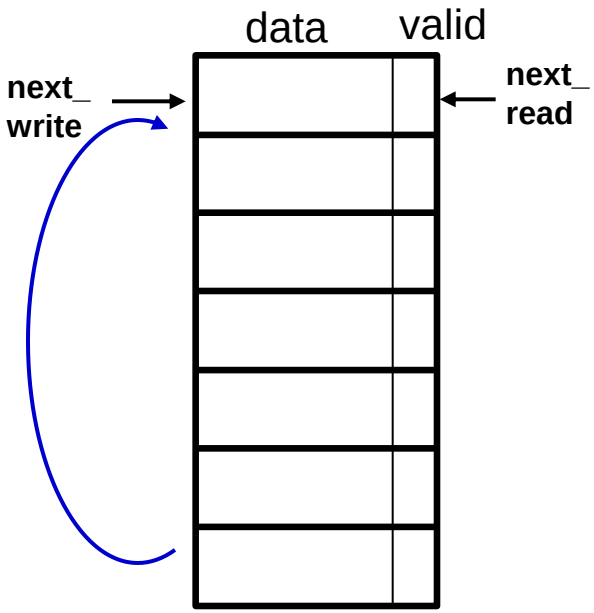
- The design insight and consultations with architects/designers have provided more understanding of the black box DUV.
  - What more do we know?
  - Which information is still needed?

Checking the Black Box

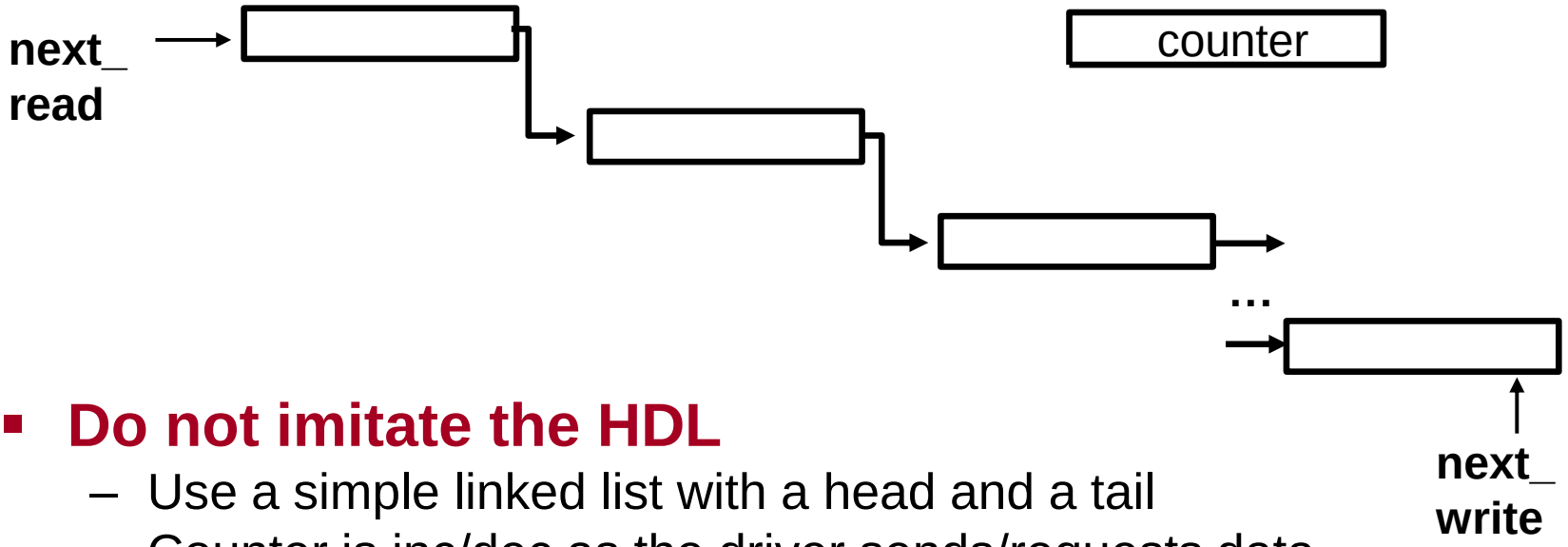
Checker	Checker Source	Checker implementation
The design returns the correct data	Inputs and Outputs, Architecture	A fundamental check on the black box is that the returned data matches the sent data. The verification code must keep an independent copy of all DUV data in order to check the data outputs coming from the design.
Buffer overflow	Microarchitecture	The verification code must keep a count of how much data is in the design. This allows prediction and checking of the buf_full and buf_overrun outputs.
Data becomes valid at the right time	Microarchitecture	The design description stipulates that the driver may read data from the design the cycle after it sends it. Therefore, the verification team should write a checker to verify that the data is not valid too early/late and that it can be read the following cycle.
Check all outputs all of the time	Design context	The out_buf_data wires should never contain valid data unless the driver performed a read and there was data in the design. Similarly, the buf_full and buf_overrun wires should only be active during a full or overrun condition.

HDL Implementation of the Black Box

- The actual implementation of the design in the black box example might be:
  - Logic required to determine if design is full or empty: `next_read =/= next_write`?
  - Valid bits need to be implemented
  - Wrap conditions need to be implemented



And the Checking Counterpart

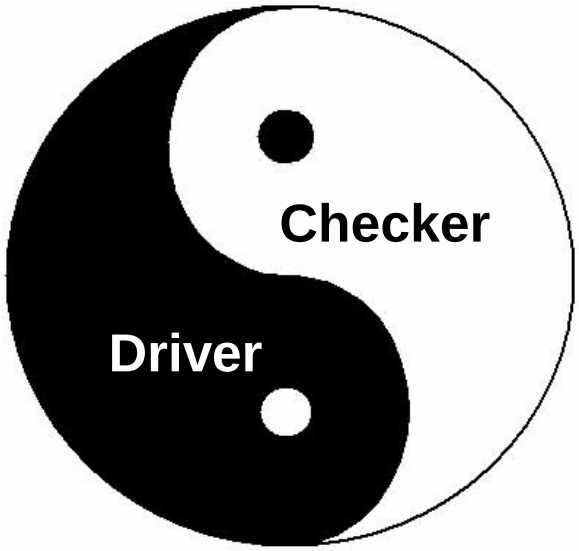


- **Do not imitate the HDL**
  - Use a simple linked list with a head and a tail
  - Counter is inc/dec as the driver sends/requests data
- Much simpler
- Can predict behavior exactly
- Need **high-level verification languages** to specify design intent:
  - Expressive, flexible and declarative
  - Allow abstraction from implementation detail

Driving and Checking

- You need both or you get nothing!

- To find a bug:
  - Your driver must create the failing scenario, and
  - Your checker must flag the behaviour mismatch.



Bug hunting…(I)

Given this bug in our simple stack:

(Which of course is never “given”... ;)

- When clean_stack => 1, the data valid bits should all be cleared.
- The next_write pointer and next_read pointer are supposed to be set to the top of the stack.

BUT:

- If the in_buf_valid => 1 (with data) is on in the same cycle as the clean_stack, the logic puts the data in the stack but resets the pointers as intended.
- This only occurs when the stack has 6 valid entries, because the bug is in the logic that is trying to set the buf_full output.

So, somewhere in the stack, there is a valid bit == 1 that should not be on.

Bug hunting… (II)

What will it take to create a scenario that uncovers this bug?

- 1. There must be 6 valid entries.
- 2. Send a clean and a data entry on the same cycle.
- 3. Start sending new entries.
- Need to send at least 6 new entries in order to move the pointers to the valid entry that shouldn't be valid.

Driving designs into corner cases can be quite difficult!

Bug hunting… (III)

What do you have to check to find this bug?

- This bug could manifest itself in a few ways:
  - The buf_full comes on because the next write points to a valid entry.
  - Read returns data when no data should be returned.
  - buf_overrun comes on too soon, as the write pointer detects that it is pointing to a valid entry when another write comes on.

More on Observability

- The chances that the verification engineer would think of such a scenario (without knowing about the bug) are slim.
- Part of the problem is the need to flush the erroneous state to the observed output.
- The probability of detecting the bug should increase if we could **detect it earlier**:
  - Reduce the probability of erasing the erroneous state
  - Reduce the probability of keeping it hidden
- For this we need better **observability**!
  - Levels of observability: black box, grey box, white box

Summary

Verification Engineers need to be inquisitive.

- Identify interesting driving scenarios.
- Find sources for checkers:
  - I/O, design context, uarch, architecture and implementation.
- Familiarize yourself with the specification of the design.
- Don’t take understanding for granted. If in doubt - ask!
- Work in close collaboration with architects/designers.
- Don’t re-implement the design - abstract, cheat, ...
  - Behavioural models are allowed to “cheat”.
    - Return random data (e.g. memory modelling)
    - Look ahead in time
    - Predetermine answers
- Select the right level for verification.

Driving and Checking: You need both SKILLS to uncover bugs!