COMS30026 Design Verification

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.)

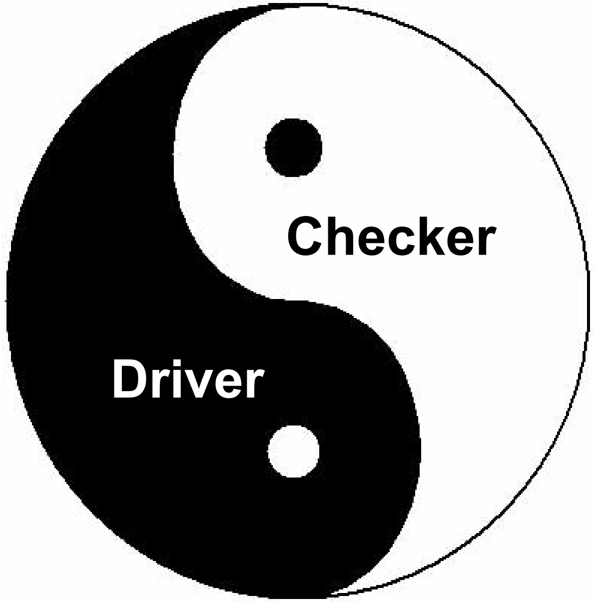
University of BRISTOL

Department of COMPUTER SCIENCE

Outline

- Fundamentals of Simulation-based Verification:
  - Strategy
    - Driving principles
    - Checking strategies
  - Working example
    - A circular buffer

    https://en.wikipedia.org/wiki/Circular_buffer



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?

Strategy of Verification

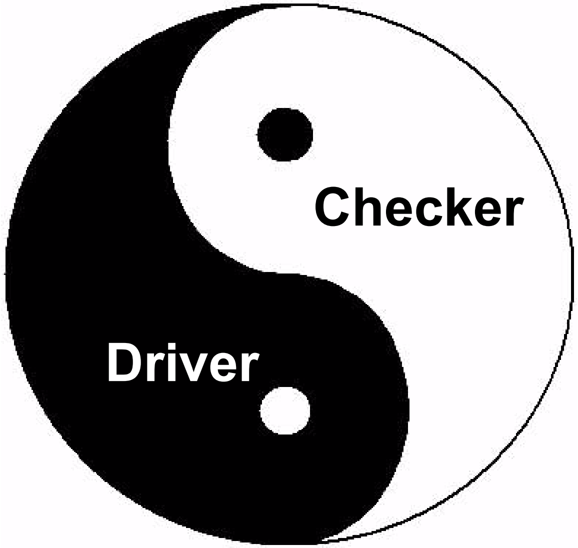
- Verification can be divided into two separate tasks
  1. Driving the design - Controllability
  2. Checking its behavior - Observability
- The basic questions a verification engineer must ask
  1. Am I driving all possible input scenarios?
  2. How will I know when a failure has occurred?

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

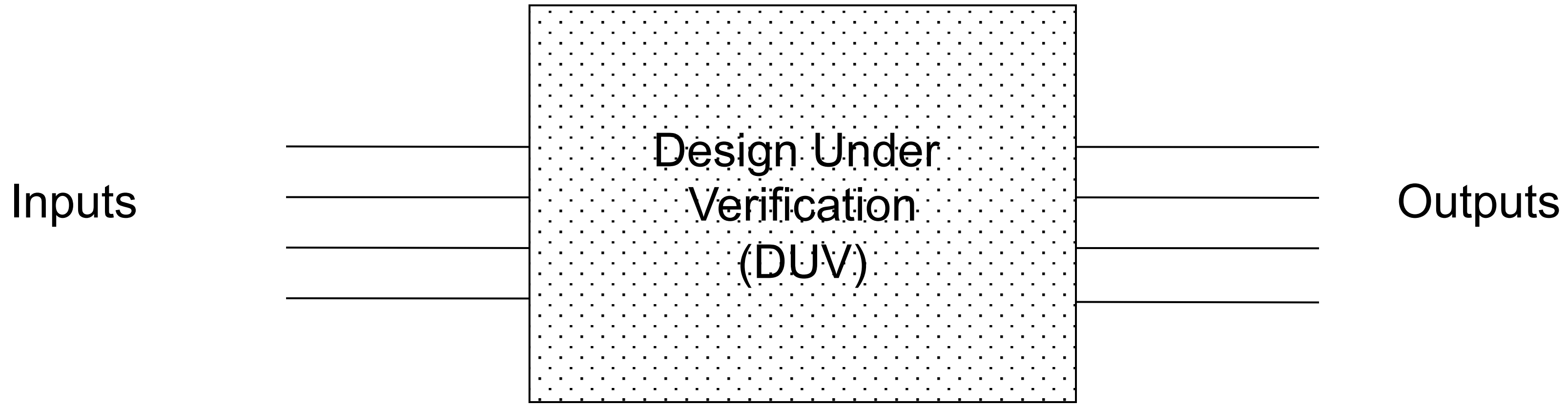
- This perfect harmony does not always exist
  - Not all failing conditions are equal
    - Same bug can lead under different failing conditions to different failures (with big difference in consequences)
  - We cannot (or don’t want to) detect all incorrect behaviors
    - Some are not important enough
    - For others we have safety nets

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 the verification objectives
  - Consider, e.g. Block vs Chip level verification
    - Both differ in the focus of verification, so the drivers and checkers will be different.

Example
Black Box DUV

The Black Box Example



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

Verification of the Black Box

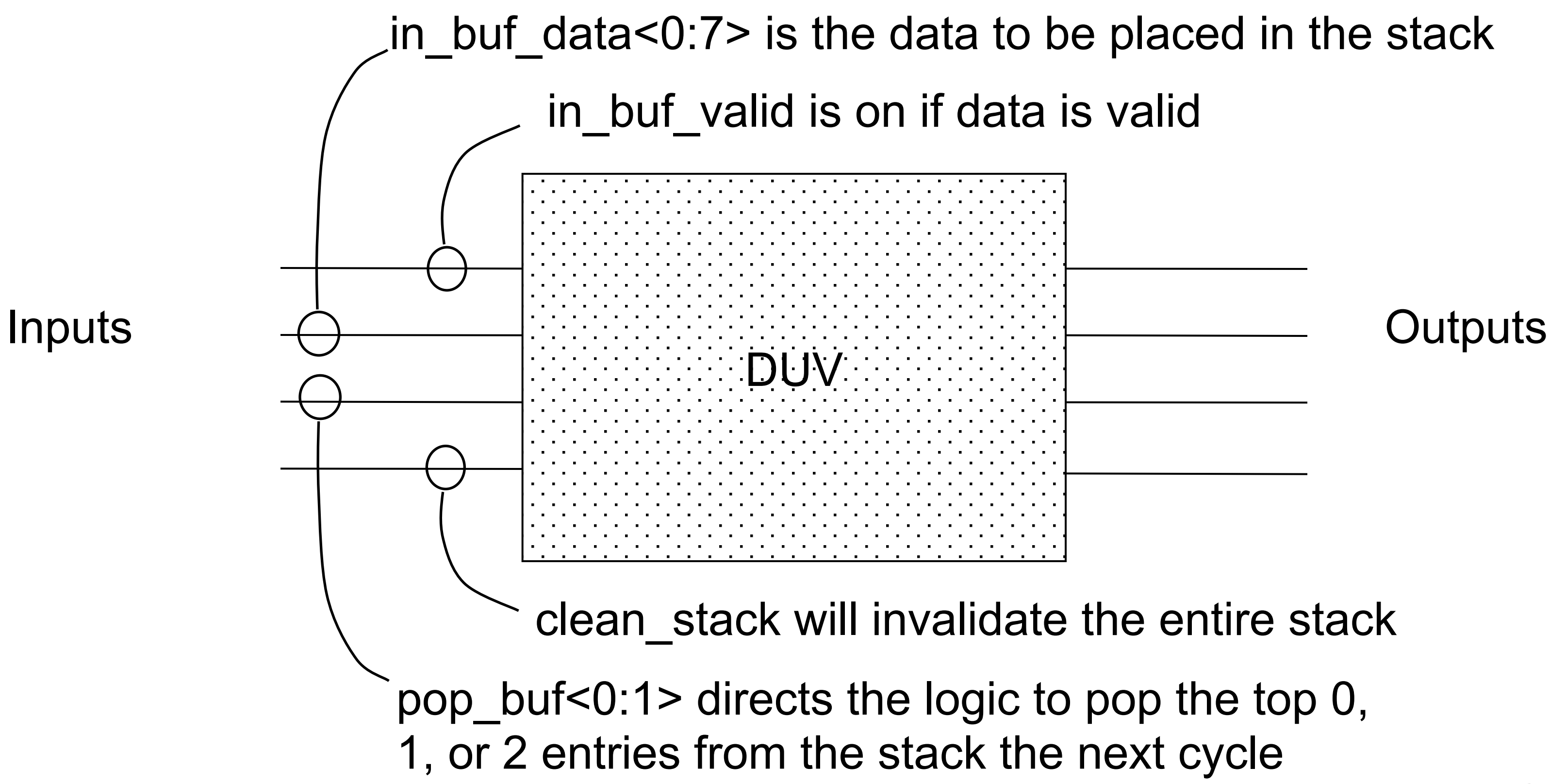
- Black box since we don't look inside it
  - What does this mean?
- The black box may have a complete documentation … or not
- To verify a black box the verification engineer must
  - understand the function and be able to
  - predict the output based on the inputs.
- It is important that the verification 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

Driving the Black Box

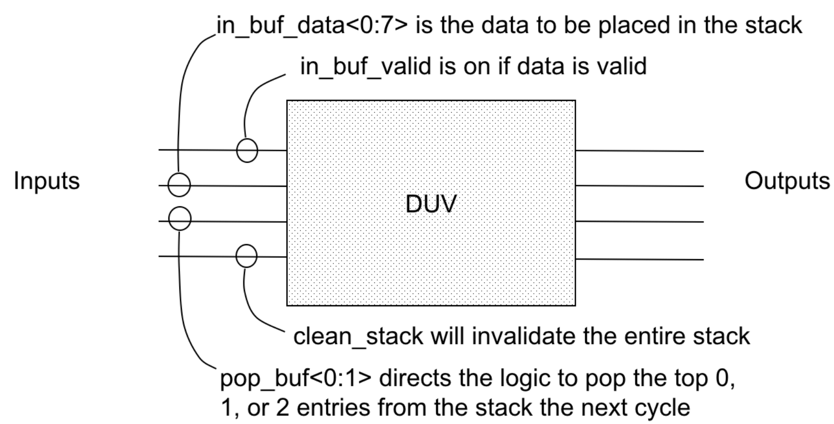
- We can start planning the stimuli even before the complete specification of the DUV is given
- The definition of the inputs can provide information and hints on
  - The interface
  - The functionality
- This information can lead to first set of stimuli
- More stimuli will be added as we learn more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

Driving the Black Box



What Can We Learn From This?

- We can build up an understanding of the design just from the input descriptions:
  - What do we know?
    - ...<fill this in, please>
    - ...
    - ...
  - What don't we know?
    - ...<fill this in, please>
    - ...
    - ...



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

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

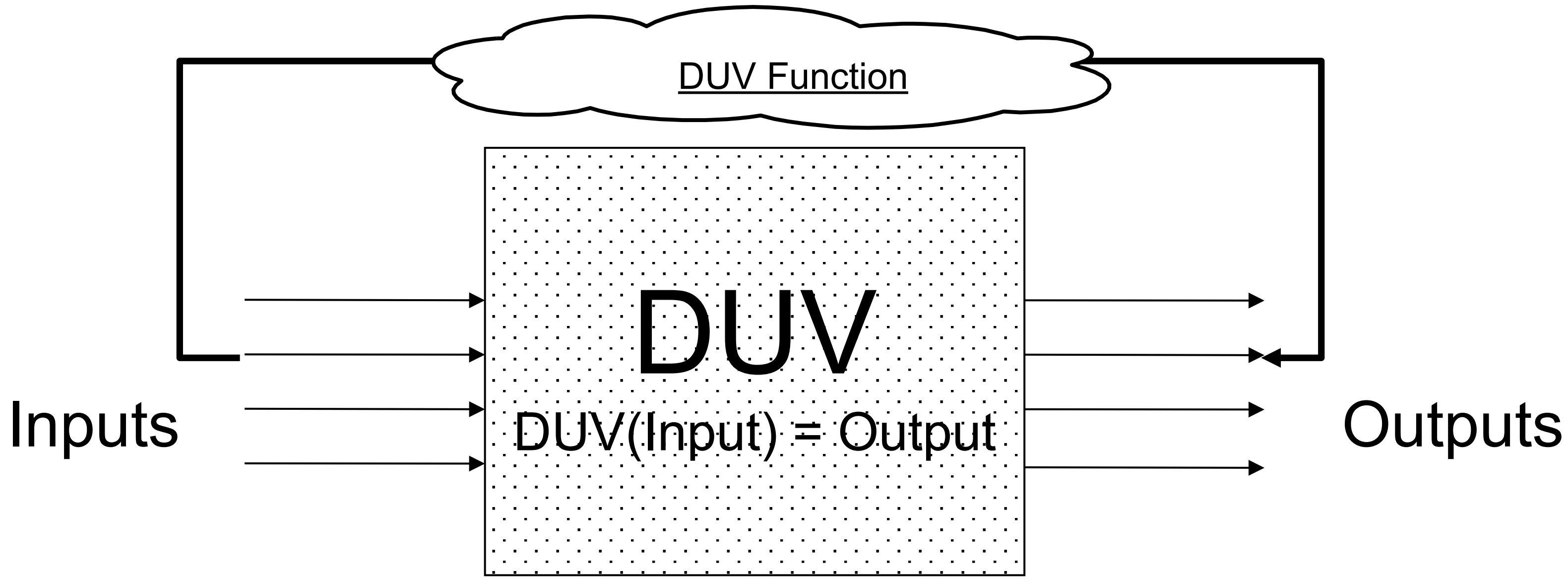
It is critically important that we record any assumptions we have made so that we can check them against the specification when it becomes available.

Checking

Where do Checkers come from?

- In microelectronic system design 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
  - The source provides us with inspiration and ideas, the implementation is the realization of these.

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
  - relevant for HW and SW alike
- **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 R1 R2
BRZ R7 L
```

Architectural (ISA-level) checking is abundant.

- The SUB and BRZ instructions are defined in the Instruction Set Architecture (ISA).
  - e.g. the (2000+ page) Arm v8-M Architecture Reference Manual is available online at https://developer.arm.com/documentation/
  - or, more locally, the XMOS xCORE-200 ISA can be downloaded from https://www.xmos.ai/file/xs2-isa-specification/
- Architecture may define that instructions must complete in order, e.g. the results of SUB must be used by BRZ.

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

Checking Based On the Architecture

Example instruction stream:

```
SUB R7 R1 R2
BRZ R7 L
```

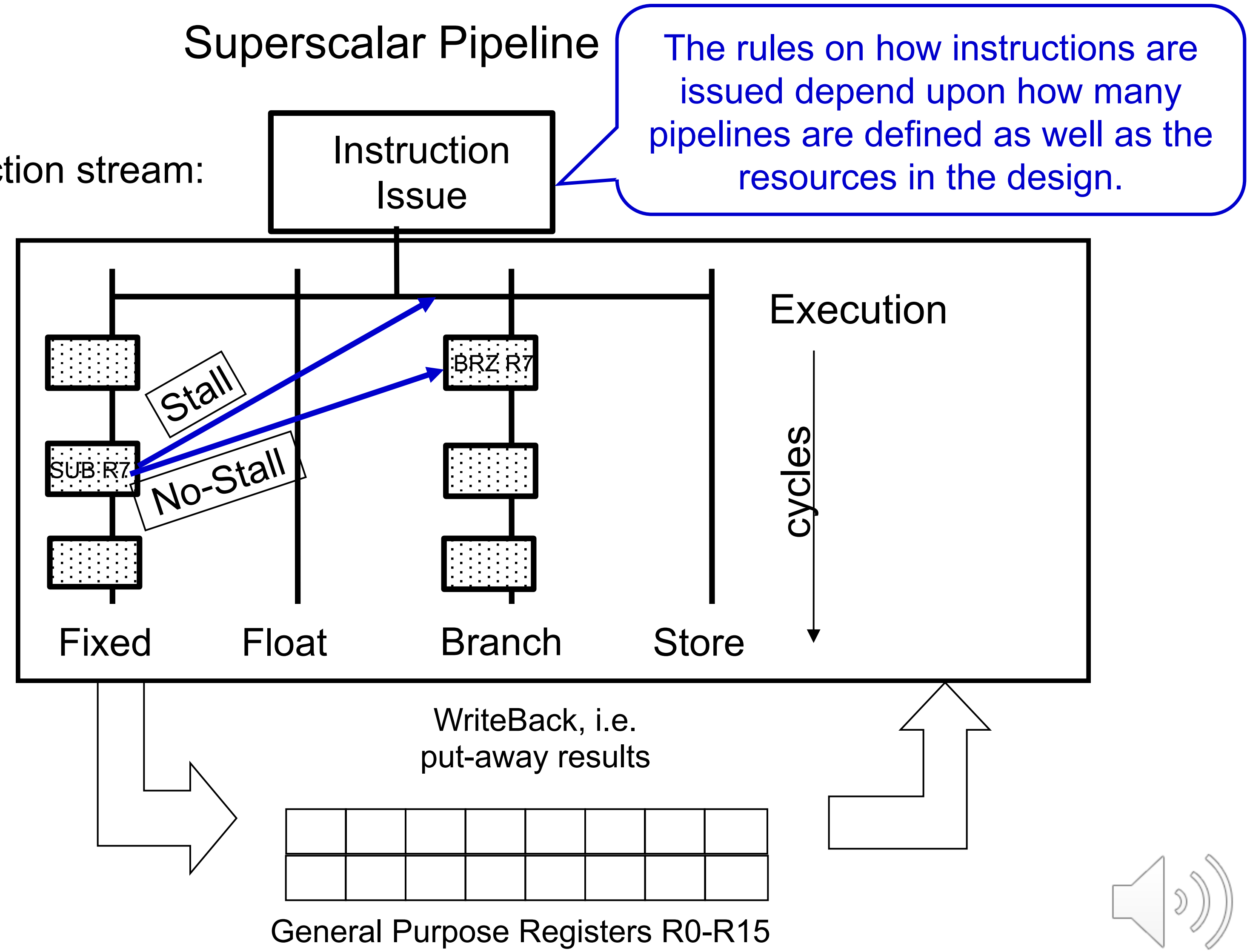
Architectural (ISA-level) checking is abundant.

- The SUB and BRZ instructions are defined in the Instruction Set Architecture (ISA).
  - e.g. the (2000+ page) Arm v8-M Architecture Reference Manual is available online at https://developer.arm.com/documentation/
  - or, more locally, the XMOS xCORE-200 ISA can be downloaded from https://www.xmos.ai/file/xs2-isa-specification/
- Architecture may define that instructions must complete in order, e.g. the results of SUB must be used by BRZ.

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

Checking Based On the Microarchitecture

Example instruction stream:
SUB R7 R1 R2
BRZ R7 L

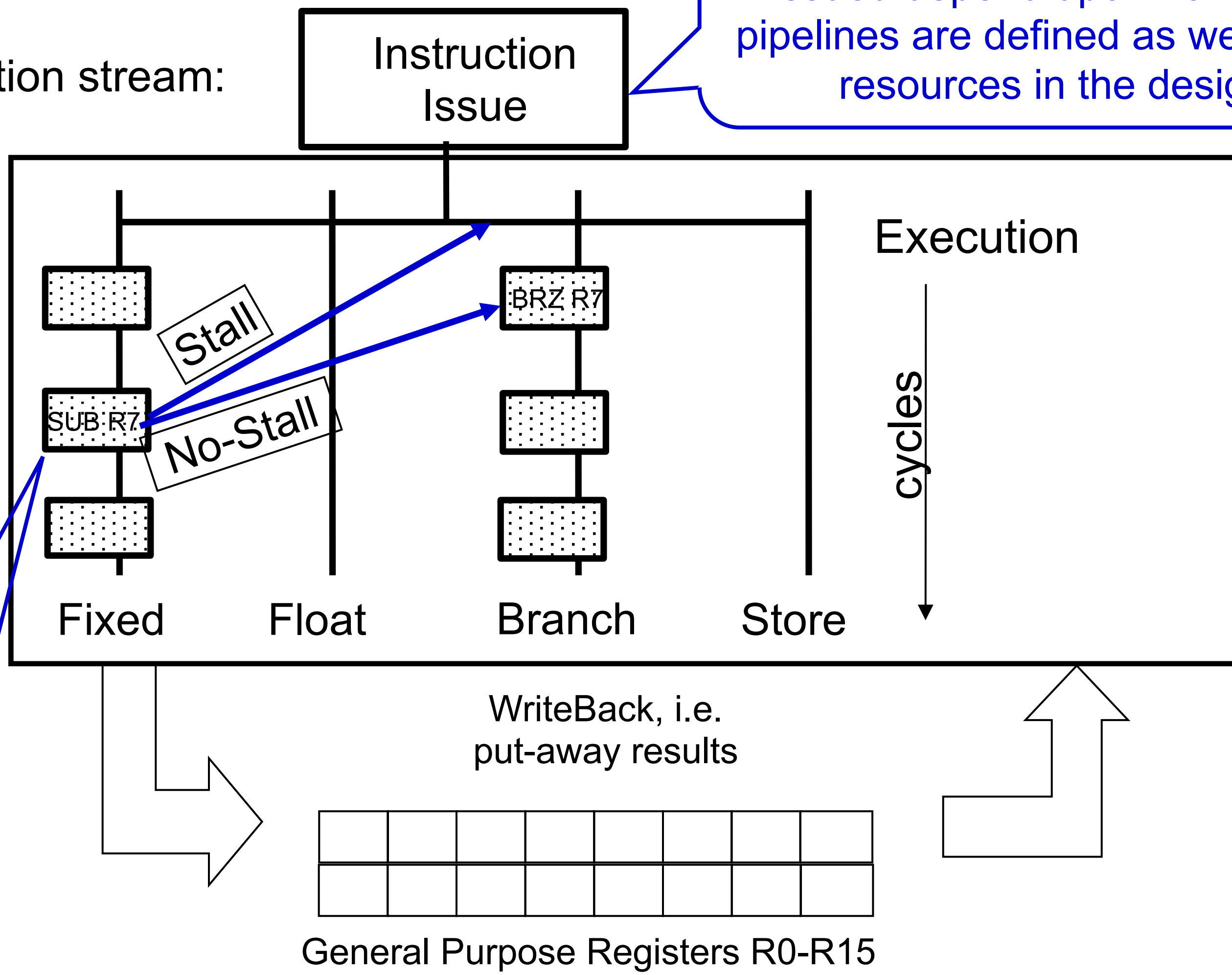


Checking Based On the Microarchitecture

Superscalar Pipeline

Example instruction stream:
SUB R7 R1 R2
BRZ R7 L

The rules on how instructions are issued depend upon how many pipelines are defined as well as the resources in the design.



The ability or inability of on-the-fly results to feed prior stages of a pipeline will affect instruction issue.

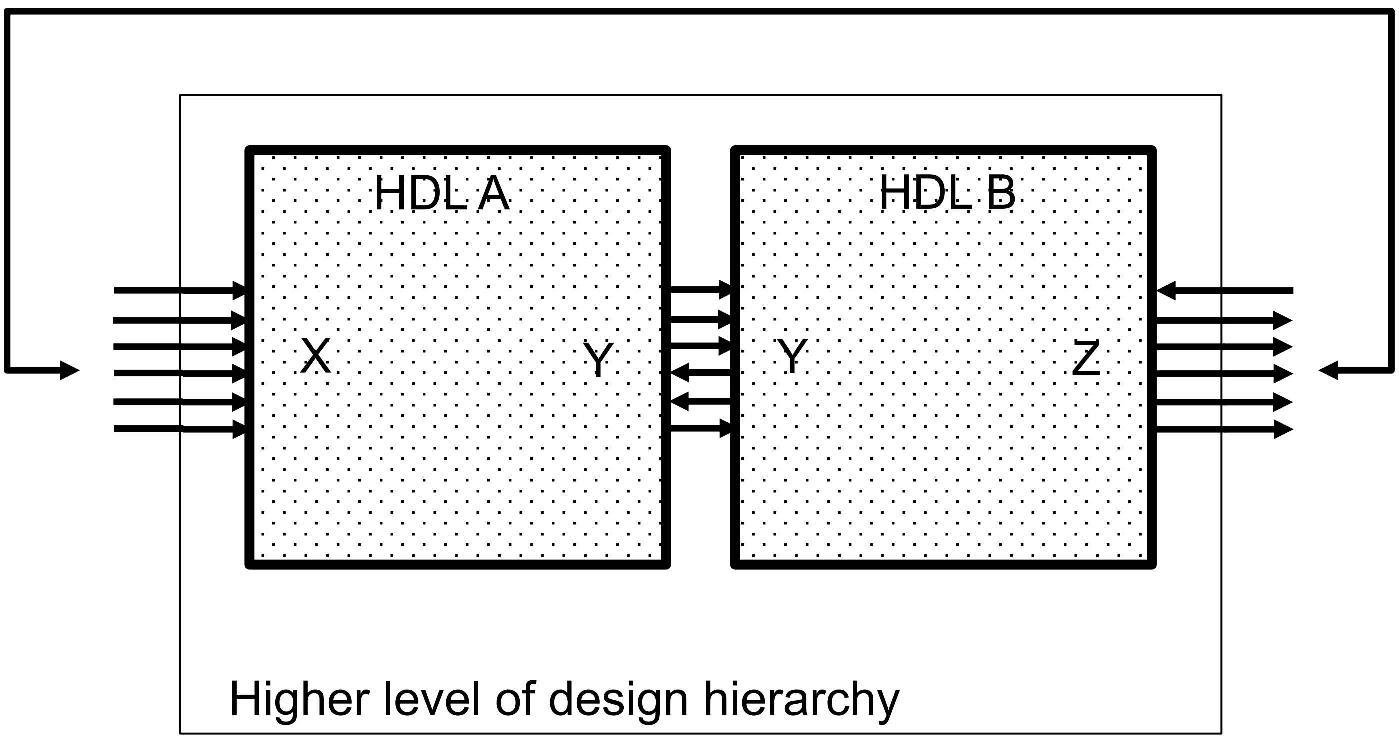
Checking Based On the Architecture and Microarchitecture

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

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

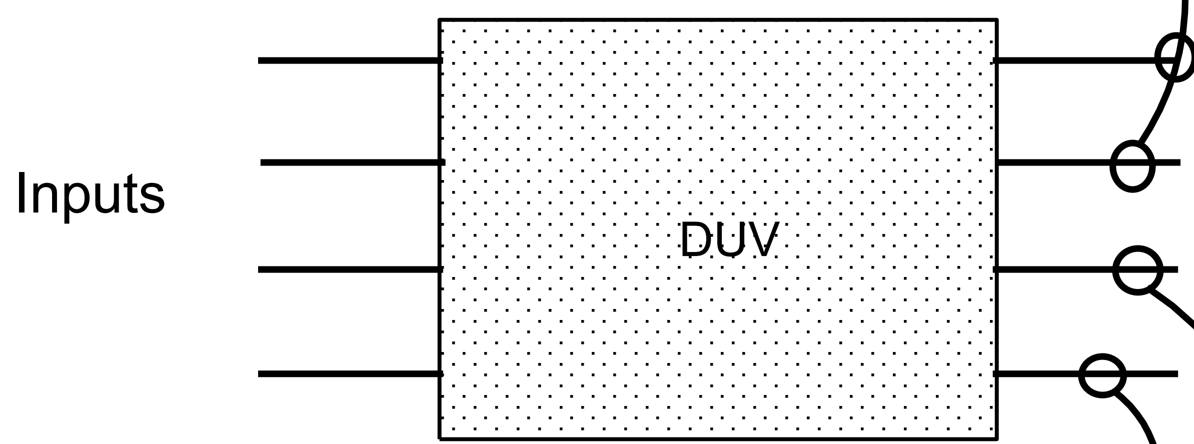


- When verifying lower levels of hierarchy such as individual blocks of HDL, the verification engineer derives checkers from an understanding of the function, properties, and context of the larger design, e.g. from how the blocks will be used in the context of the design.

Back to our example
Black Box DUV

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.

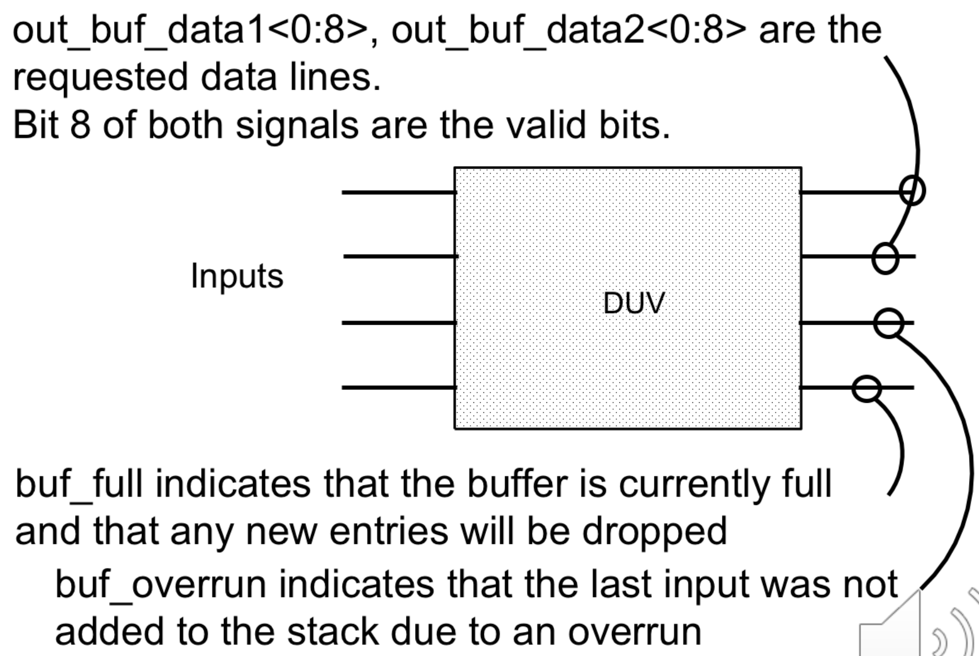


buf_full indicates that the buffer is currently full and that any new entries will be dropped

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 items become valid (for reading) one cycle after they have been 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.
- The buf_full signal 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 documentation has provided more understanding of the black box DUV.
  - What more do we know?
    - ...
  - Which information is still needed?
    - ....
- At this stage we may need some consultations with architects and potentially with designers to gain further understanding of the black box DUV.

Ideas for Checkers of 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 / Internals	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 / Internals	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.

Ideas for Checkers of the Black Box

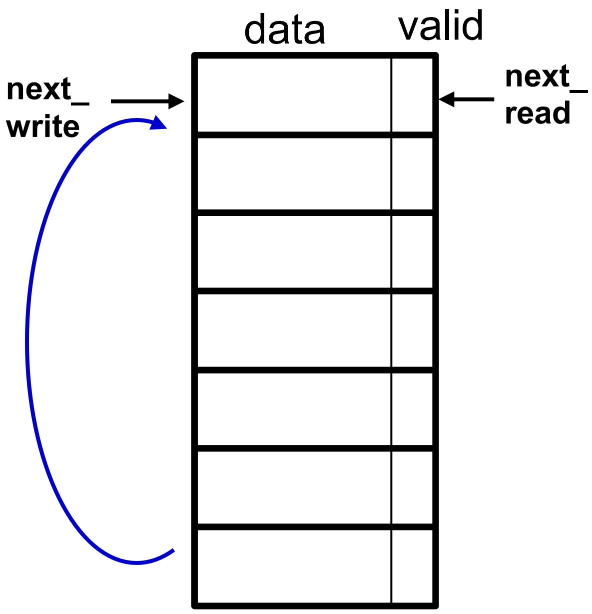
Checker	Checker Source	Checker implementation
The design returns the correct data	Inputs and Outputs, Architecture	A fundamental check on the black box is that the returned data matches the sent data. The verification code must keep an independent copy of all DUV data in order to check the data outputs coming from the design.
Buffer overflow	Microarchitecture / Internals	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 / Internals	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.

Ideas for Checkers of the Black Box

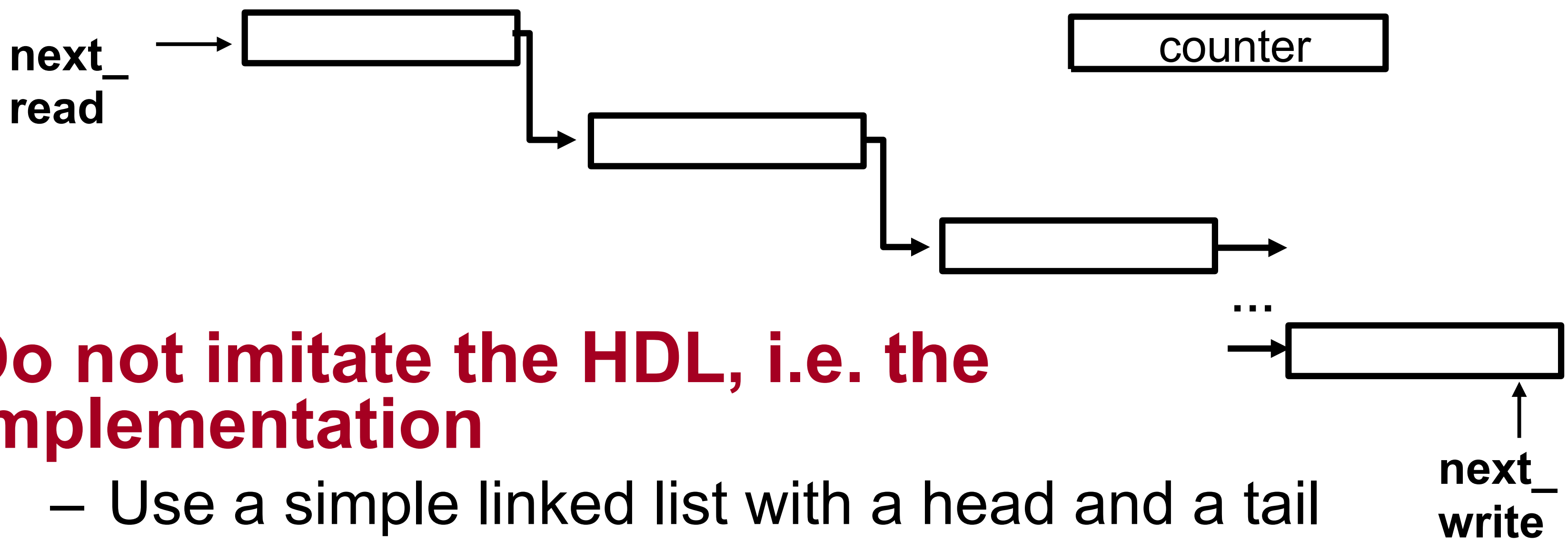
Checker	Checker Source	Checker implementation
The design returns the correct data	Inputs and Outputs, Architecture	A fundamental check on the black box is that the returned data matches the sent data. The verification code must keep an independent copy of all DUV data in order to check the data outputs coming from the design.
Buffer overflow	Microarchitecture / Internals	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 / Internals	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 a circular buffer:
  - Logic required to determine if design is full or empty: `next_read` and `next_write` and potentially a `counter`
  - `valid` bits need to be implemented
  - Wrap conditions need to be implemented to achieve a *circular buffer.*



...and the Checking Counterpart



Do not imitate the HDL, i.e. the implementation

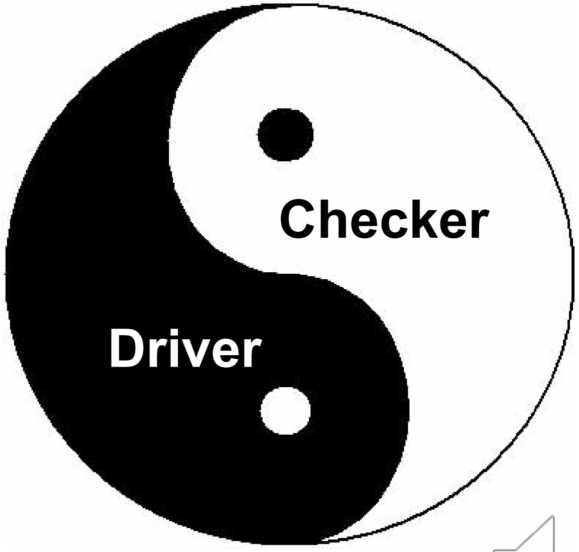
- Use a simple linked list with a head and a tail
- Counter is inc/dec as the driver sends/requests data
  - Much simpler
  - Can predict behavior exactly

- We need **high-level verification languages** to specify the design intent:
  - Expressive, flexible and declarative
  - Allow abstraction from implementation detail

Bug Hunting

Remember, to find a bug you need both, **driving & checking**:

- 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:

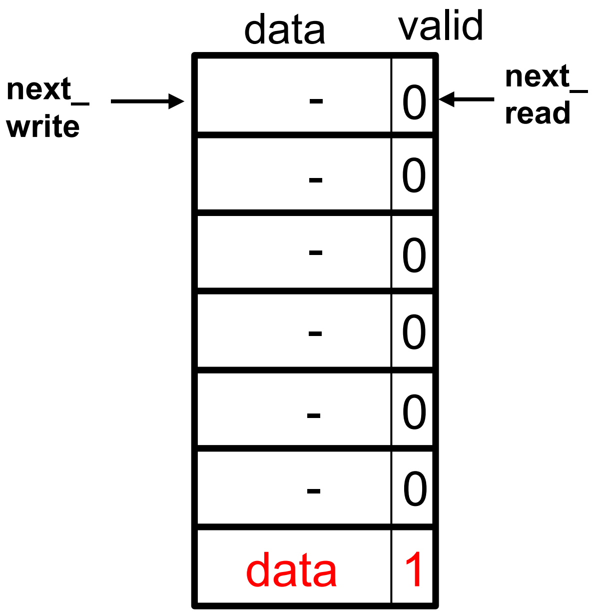
- If the in_buf_valid = 1 (with data) is on in the same cycle as the clean_stack, the logic puts the data in the stack but resets the pointers as intended.
- This only occurs when the stack has 6 valid entries, because the bug is in the logic that is trying to set the buf_full output.

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

But,where?

... resulting in this situation ☹

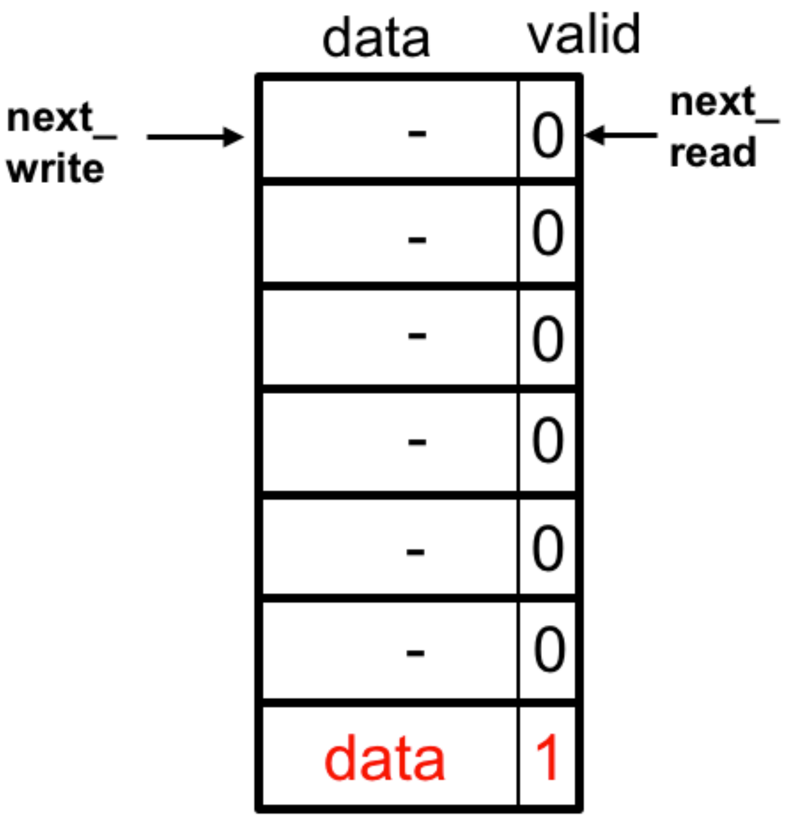
- This only occurs when the stack has 6 valid entries, because the bug is in the logic that is trying to set the buf_full output.
- The new data item is therefore put into the 7th data slot with the valid bit set to 1.



Bug hunting… (II)

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

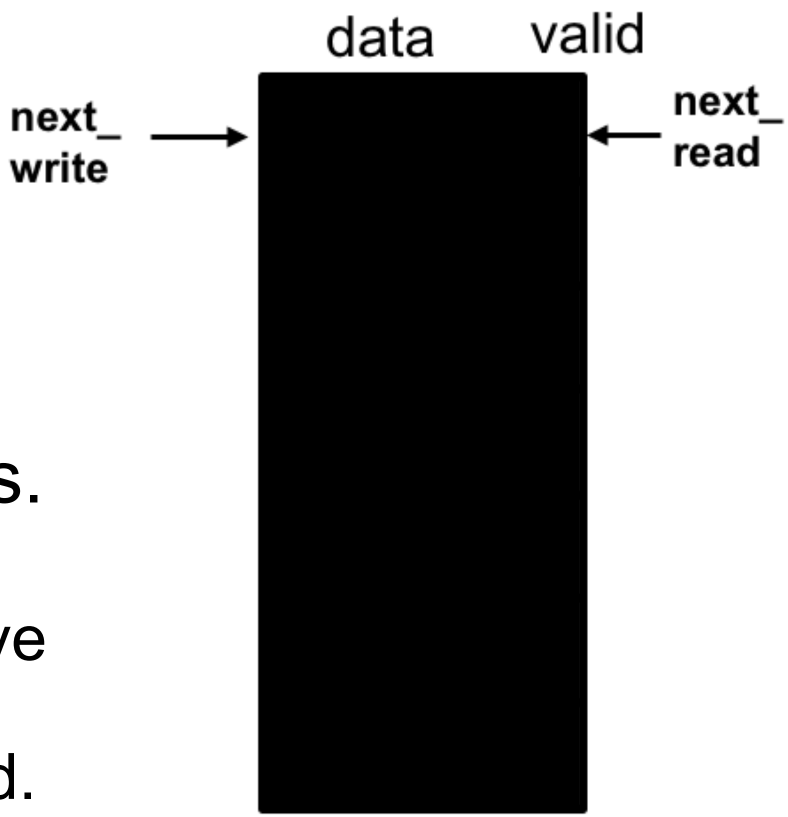
1. There must be 6 valid entries.
2. Send a clean and a data entry on the same cycle.



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.
   – We 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.

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.
- Which of the above may occur depends on the actual implementation, e.g. the control logic that sets the full and overrun signals.

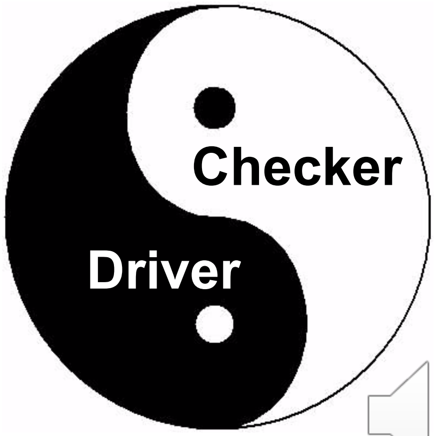
Reflections on our bug hunting

- 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 & Checking: You need both (SKILLS) to uncover bugs!