Driving and Checking

The Science of Verification:

- Have I thought of all possible scenarios?
- Am I driving all possible input scenarios?
- What are the corner cases?
- How will I know when it fails?

This lecture aims to encourage you to be inquisitive!

- In preparation of the 1st assignment. ;)

[Credits: This lecture is based on a set of slides sent to me by Bruce Wile from IBM.]

So, what’s the Science?

What does it mean to drive all possible input scenarios?

- Design with two bits of input and an output is trivial.

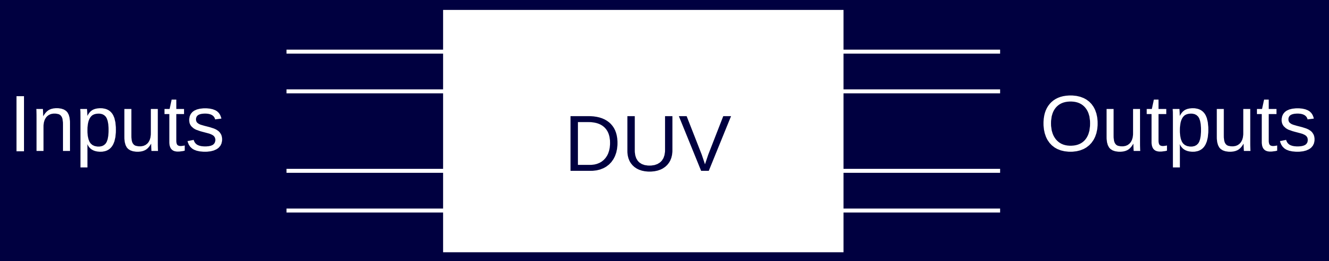
How do you do this on a complex design?

What does it mean to “know when it fails”?

- A complete set of checkers is the key!

Let’s investigate this in the context of Black Box verification.

Black Box Verification

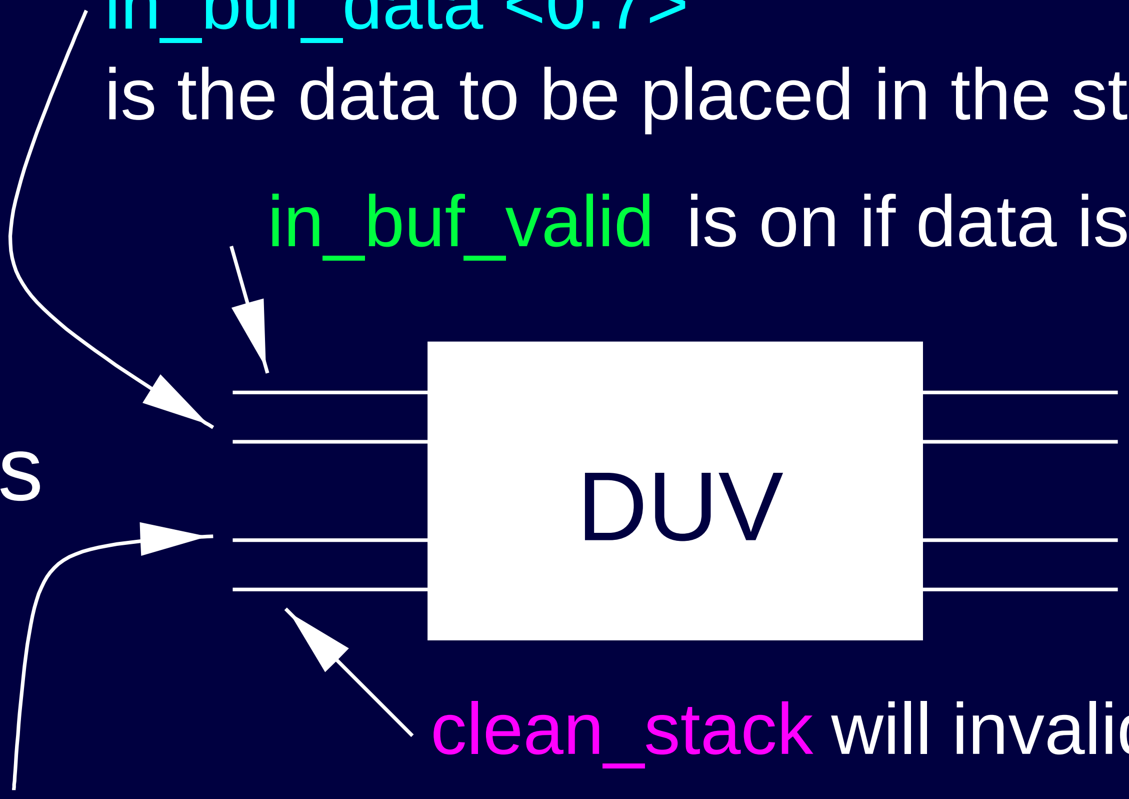


- The black box has inputs, outputs and performs some function.
- Function may be well documented...or not.
- The black box can be a full system, a chip, a unit of a chip, a module, etc.

To verify a black box you need to understand the function and be able to predict the outputs based on the inputs.

💡 This means you must **understand the specification** of the design.

Driving the Black Box



in_buf_data <0:7>
is the data to be placed in the stack

in_buf_valid is on if data is valid

Inputs

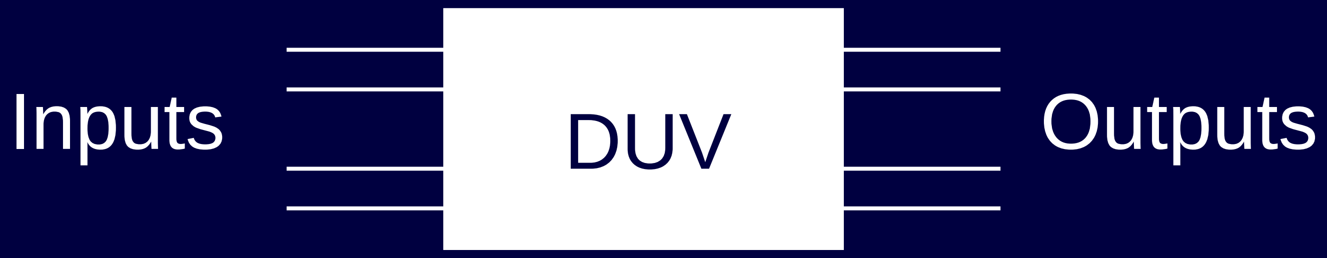
DUV

Outputs

clean_stack will invalidate the entire stack

pop_buf <0:1> directs the logic to pop the top 0, 1, or 2 entries from the stack the next cycle

Driving the Black Box

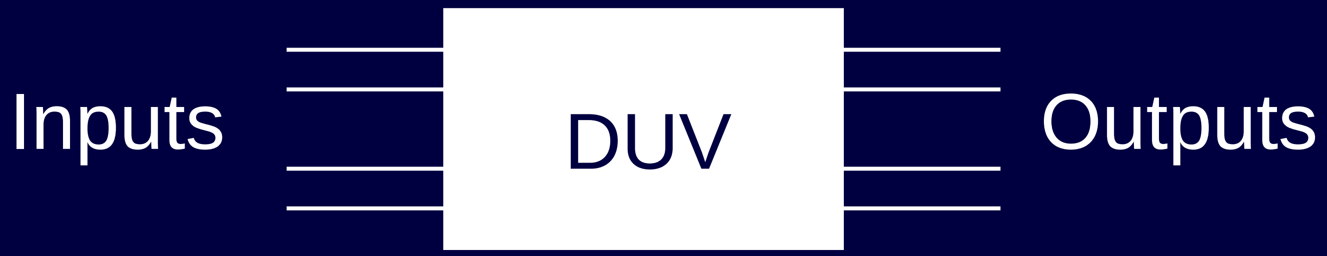


- We can start to understand the design just from the input descriptions:

What do we know?

What don’t we know?

Driving the Black Box



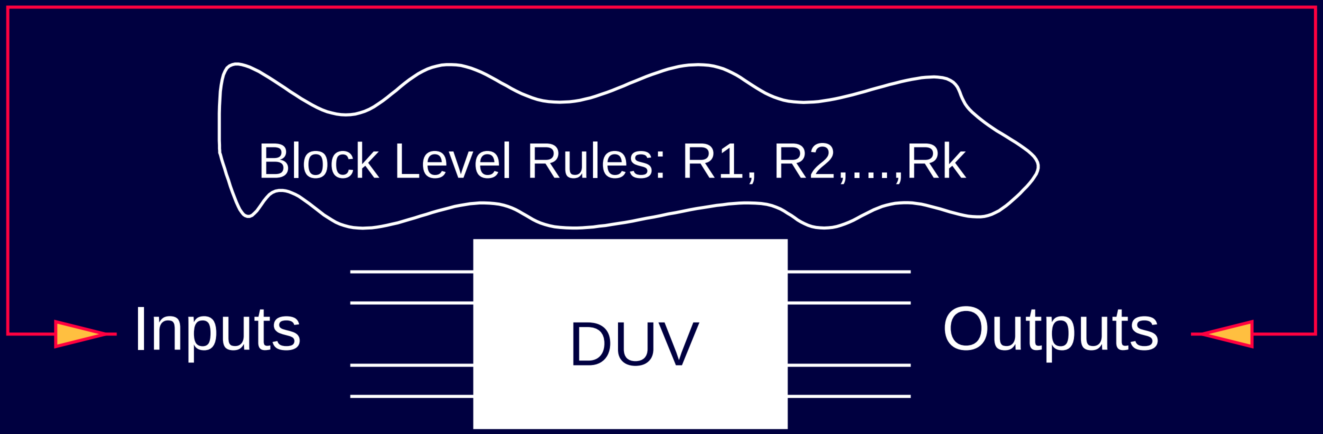
What scenarios will we need to drive?

How do we come up with the Checking?

- Understand the inputs and outputs.
- Understand the design context (up the hierarchy).
- Understand the internal structures and algorithms (the uarch).
- Understand the top-level design description (architecture).

💡 **Understand the Specification!**

Checker Source: Inputs and Outputs



The outputs can be predicted from the inputs with an understanding of the rules.

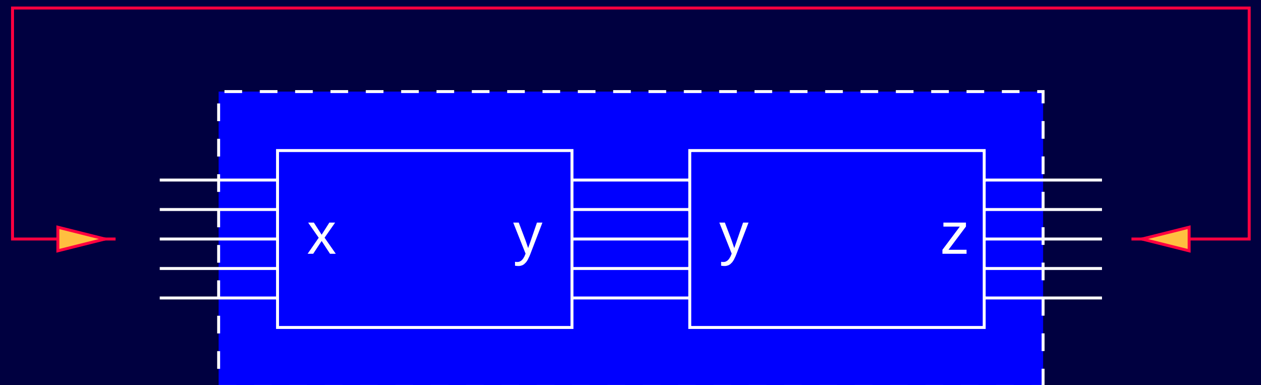
Often, all outputs of the design must be checked at every clock cycle!

However, if the outputs are not specified clock-cycle for clock-cycle, then verification should not be done clock-cycle for clock cycle!

NOTE: **Response verification should not enforce, expect, nor rely on an output being produced at a specific clock cycle.**

[Credits: Source from Yaron Wolfsthal, IBM Haifa, “Specification-based Verification”]

Checker Source: Design Context



Block Level Rules: R1 <> R(x,y) + R(y,z)

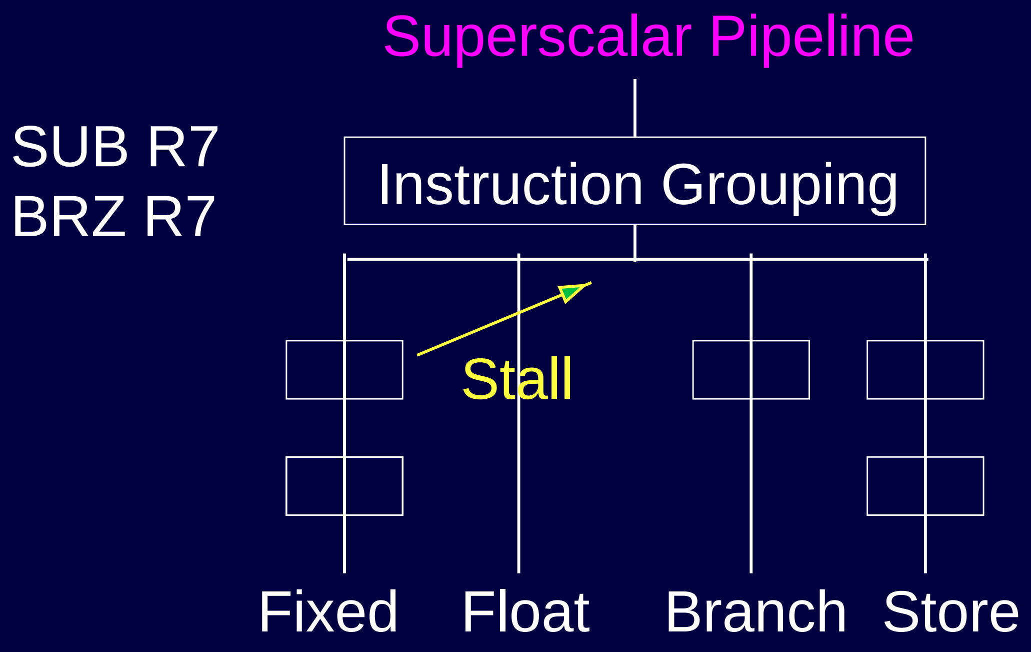
R1 implies certain behaviour on lower-level modules.

Some checkers will come from understanding the **context of the higher level(s) of design,** i.e. the environment in which the design will be used.

- **What are interesting questions regarding our black box buffer/stack?**

[Credits: Source from Yaron Wolfsthal, IBM Haifa, “Specification-based Verification”]

Checker Source: uArchitecture

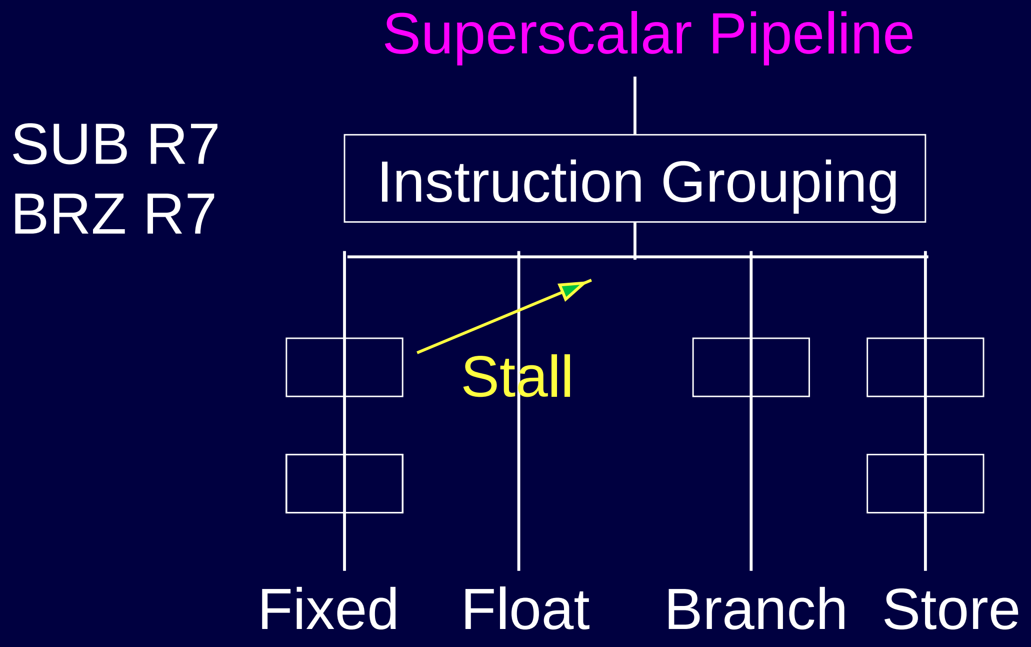


The rules on how instructions are grouped depend upon how many pipelines are defined as well as the resources in the design.

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

Many checkers are derived from the uarch design.

Checker Source: Architecture



Architectural checking is abundant.

- The SUB and BRZ commands are defined by the architecture.
- Architecture defines that commands must complete in order.

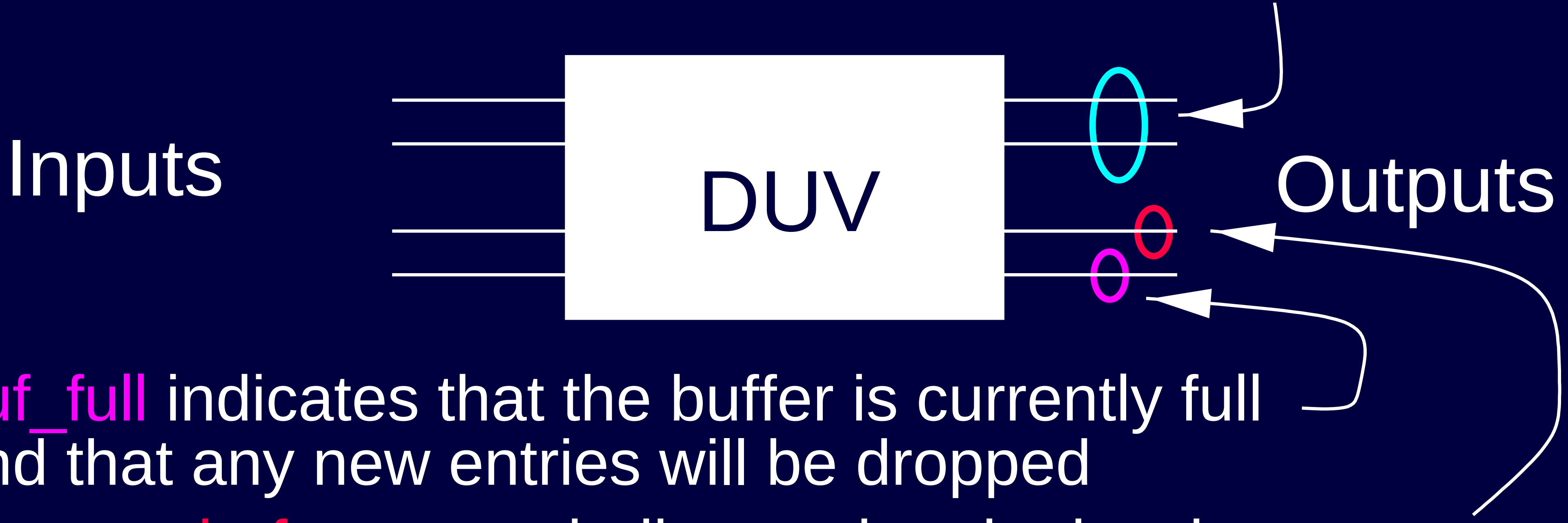
■ Architecture defines that results of SUB must be used by BRZ.

Most checkers have their roots in the Architecture of the design.

Checking the Black Box

out_buf_data_1 <0:8> and out_buf_data_2 <0:8> are the requested data lines.

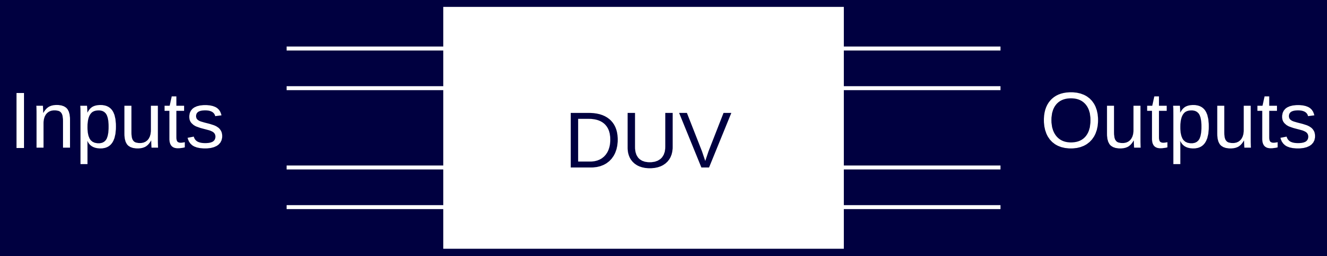
Bit 0 of both signals are the data valid bits.



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

Checking the Black Box


Inputs

DUV

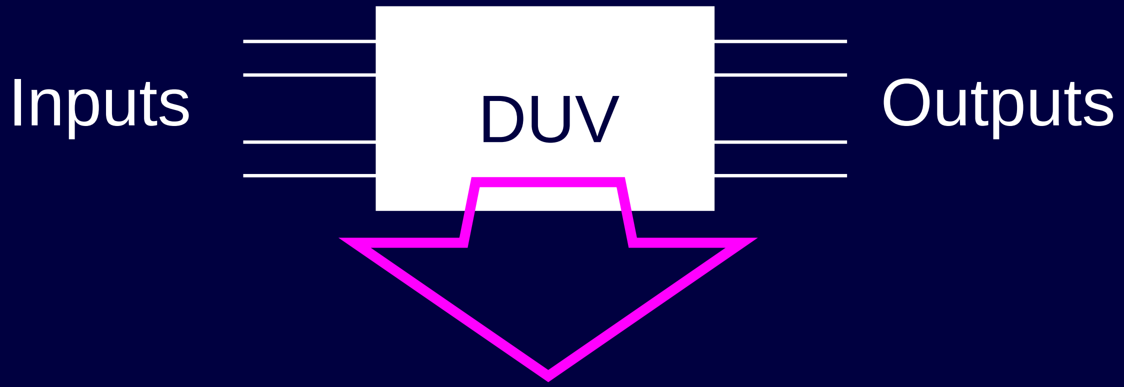
Outputs

- The outputs give us an insight into the scenarios we need to create.

What more do we know?

What info do we still need from the architect/designer?

Design Insight into the Black Box



💡 **Use consultations with system architects and designers.**

- Stack is 7 deep.
  A new stack entry is valid for reading the next cycle.
  The stack is reset the cycle after the read command.
  Inputs arriving with a clear command are ignored.
  The clean command turns the valid bit off on all 7 entries.
  No data is returned for a read if the stack is empty.
  The “stack” is a FIFO. (A queue in fact!)

Checking the Black Box

- What checkers do we need to implement?
- What is the checker specification source?

  (– I/O, design context, uarch or architecture?)
- What kind of bugs would each checker uncover?

■

Checking the Black Box

- What checkers do we need to implement?
- What is the checker specification source?

  (– I/O, design context, uarch or architecture?)
- What kind of bugs would each checker uncover?

■ Is correct data always returned?

Buffer overflow/full

💡 Check overflow even when no overflow is expected.

Does stack data become valid at the right time?

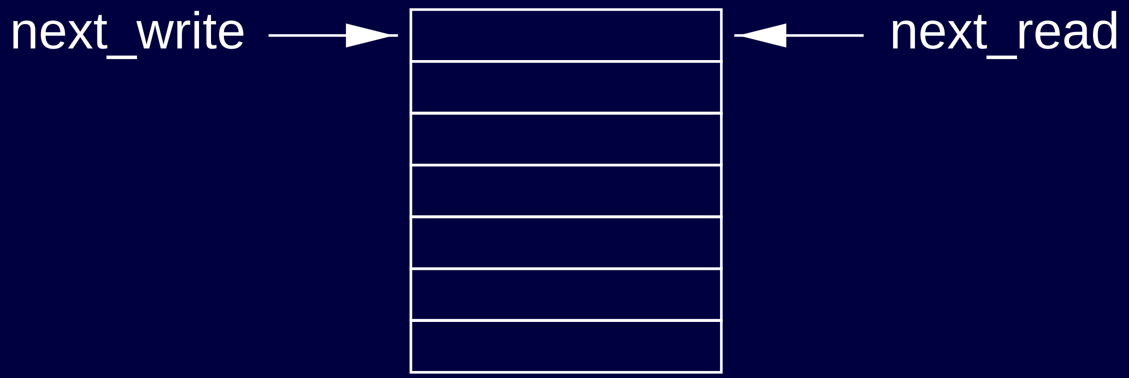
Check all outputs all the time. WHY?

Check data outputs when no request was made...

... (lots more)

Checking: Don't re-implement the design!

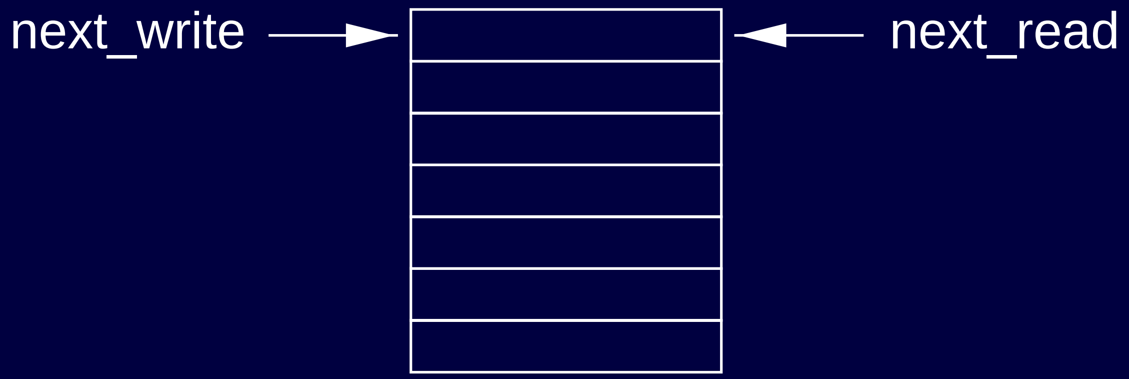
The actual implementation of the stack in our black box example might be:



- Logic required to determine if stack is full or empty.
  - next_read == next_write
- Valid bits need to be implemented.
- Wrap conditions must be implemented.

Checking: Don’t re-implement the design!

The actual implementation of the stack in our black box example might be:

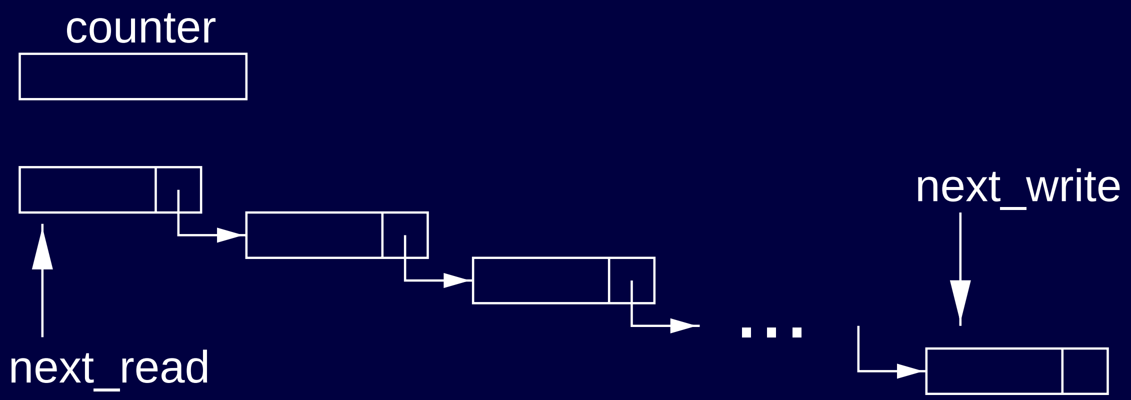


- Logic required to determine if stack is full or empty.
  - ■ next_read == next_write
- Valid bits need to be implemented.
- Wrap conditions must be implemented.

💡 Array implementation of circular fixed length queue.

Checking: Don’t re-implement the design!

Our verification checker implementation is simpler.



- Simple linked list with a head (next_read) and a tail (next_write).
- Counter is inc/dec as the driver sends/requests data.

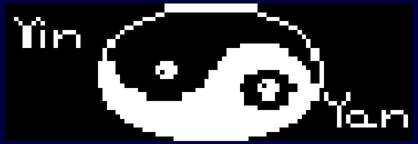
💡 **Need high-level verification languages!**

– expressive, flexible and declarative: Specify design intent.

– Allow abstraction from implementation details.

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 mismatch.**

Driving and Checking the Black Box

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 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.

Driving and Checking the Black Box

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

Driving and Checking the Black Box

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.

Driving and Checking the Black Box

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.

Driving and Checking the Black Box

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.

Behavioural Design Environment

One of the most difficult concepts that new verification engineers hit is that **your behavioural model of the design's environment can "cheat".**

- The behavioural model only needs to make the DUV think the real logic is connected to its interface.
- **The behavioural model can:**
  - predetermine answers ■ cache modelling
  - return random data
    - ■ Memory modelling: Memory controller - memory (behavioural model).
  - look ahead in time
    - ■ Branch prediction: Look ahead in instruction stream!

(• Often 1/2 work load - speed up.)

Summary

Verification Engineers need to be inquisitive.

- Identify interesting driving scenarios.
- Find sources for checkers: I/O, design context, uarch or architecture.
  - Familiarize yourself with the specification of the design.
  - Don't take understanding for granted. If in doubt - ask!
  - Work in close collaboration with designers.

  - Don't re-implement the design - abstract, cheat, ...

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