6.S965
Digital Systems Laboratory II

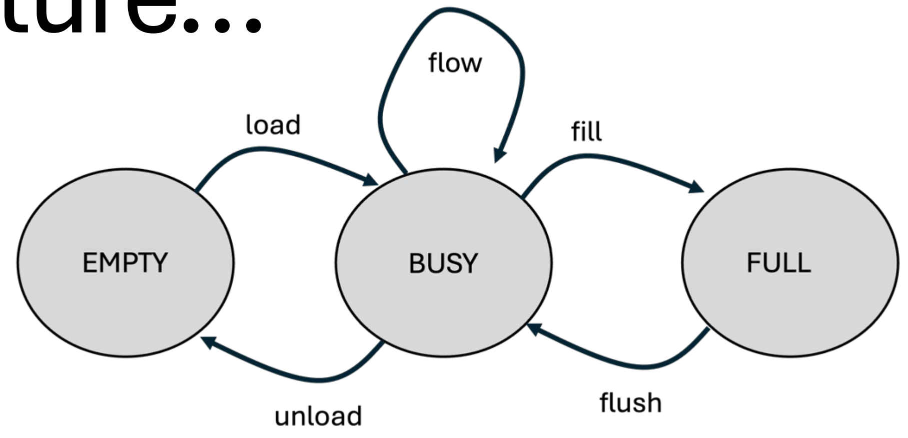
Lecture 12

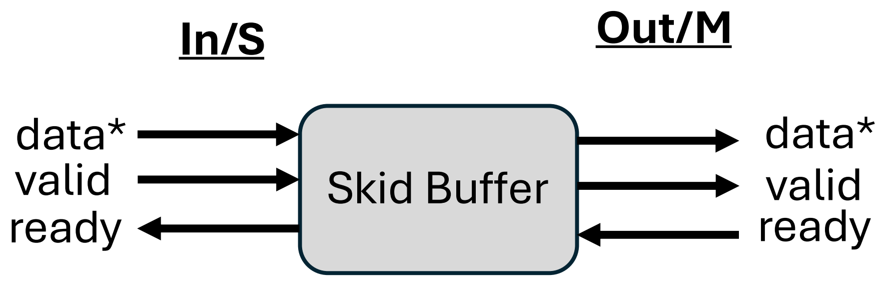
Administrative

- Last Lecture
- Finish Week 6’s material (the RFSoC stuff) by Friday please.
- Then projects (we’ll be having meetings)

Left off in last lecture…

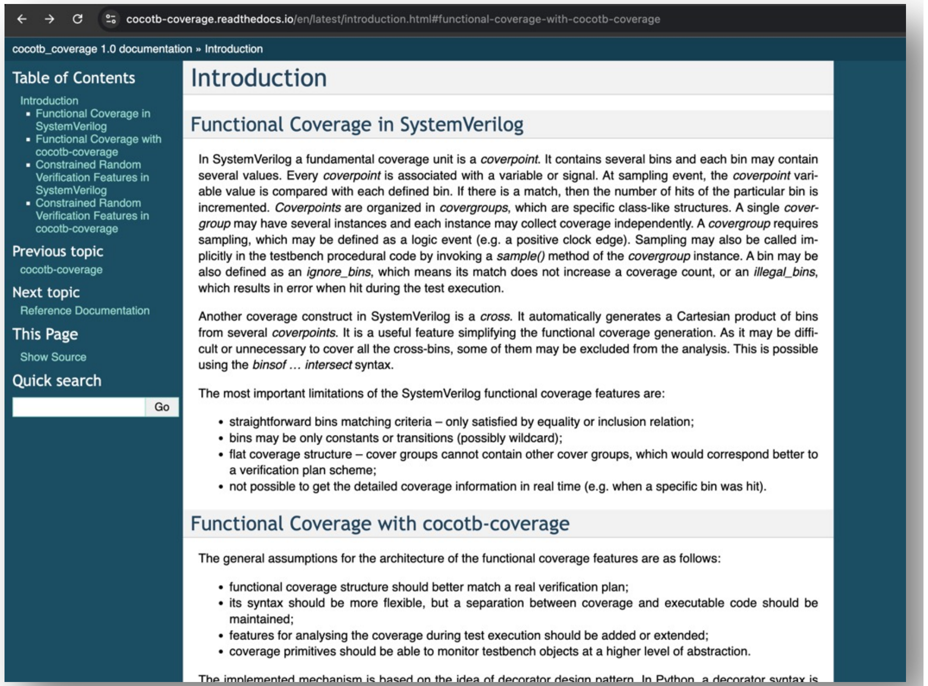
- Wrote a skid buffer and then were trying to test it with a variety of inputs,
- But also get a sense of how well we were testing it





Cocotb Coverage

- Cocotb variant Library
- Provides some structure and tracking for coverage


cocotb-coverage.readthedocs.io/en/latest/introduction.html#functional-coverage-with-cocotb-coverage
cocotb_coverage 1.0 documentation » Introduction
Table of Contents
Introduction
Functional Coverage in SystemVerilog
Functional Coverage with cocotb-coverage
Constrained Random Verification Features in SystemVerilog
Constrained Random Verification Features in cocotb-coverage
Previous topic
cocotb-coverage
Next topic
Reference Documentation
This Page
Show Source
Quick search
Go

Introduction

Functional Coverage in SystemVerilog

In SystemVerilog a fundamental coverage unit is a *coverpoint*. It contains several bins and each bin may contain several values. Every *coverpoint* is associated with a variable or signal. At sampling event, the *coverpoint* variable value is compared with each defined bin. If there is a match, then the number of hits of the particular bin is incremented. *Coverpoints* are organized in *covergroups*, which are specific class-like structures. A single *covergroup* may have several instances and each instance may collect coverage independently. A *covergroup* requires sampling, which may be defined as a logic event (e.g. a positive clock edge). Sampling may also be called implicitly in the testbench procedural code by invoking a *sample()* method of the *covergroup* instance. A bin may be also defined as an *ignore_bins*, which means its match does not increase a coverage count, or an *illegal_bins*, which results in error when hit during the test execution.

Another coverage construct in SystemVerilog is a *cross*. It automatically generates a Cartesian product of bins from several *coverpoints*. It is a useful feature simplifying the functional coverage generation. As it may be difficult or unnecessary to cover all the cross-bins, some of them may be excluded from the analysis. This is possible using the *binsof … intersect* syntax.

The most important limitations of the SystemVerilog functional coverage features are:

- straightforward bins matching criteria – only satisfied by equality or inclusion relation;
- bins may be only constants or transitions (possibly wildcard);
- flat coverage structure – cover groups cannot contain other cover groups, which would correspond better to a verification plan scheme;
- not possible to get the detailed coverage information in real time (e.g. when a specific bin was hit).

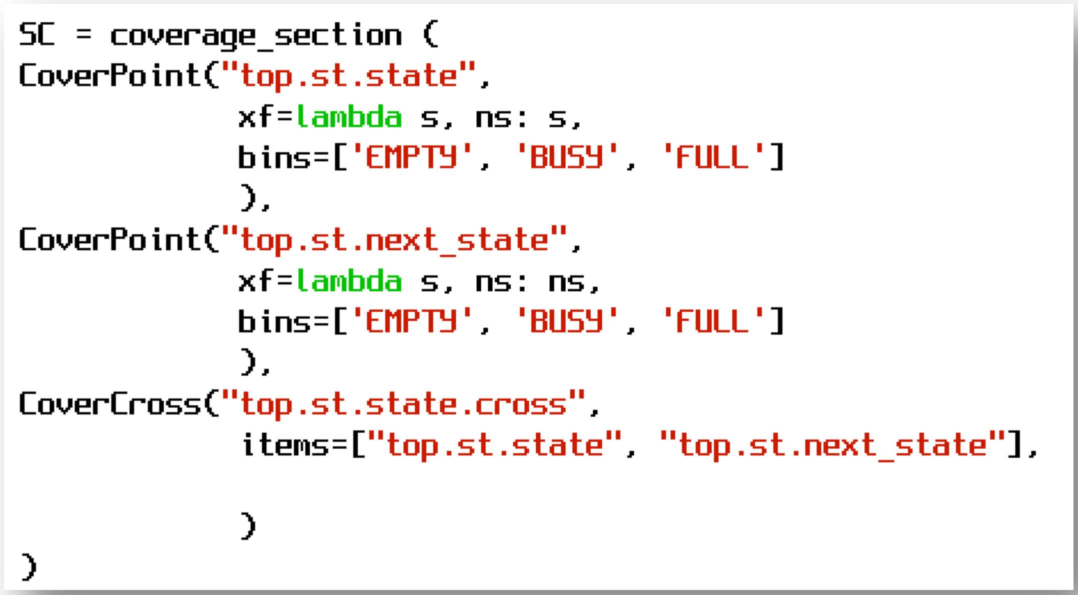
Functional Coverage with cocotb-coverage

The general assumptions for the architecture of the functional coverage features are as follows:

- functional coverage structure should better match a real verification plan;
- its syntax should be more flexible, but a separation between coverage and executable code should be maintained;
- features for analysing the coverage during test execution should be added or extended;
- coverage primitives should be able to monitor testbench objects at a higher level of abstraction.

The implemented mechanism is based on the idea of decorator design pattern. In Python, a decorator syntax is

CoverPoint and CoverCross and Cover… Group/Section

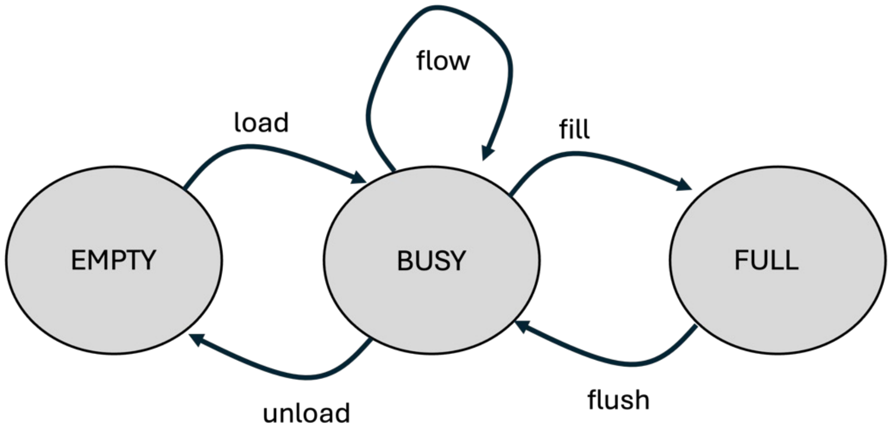


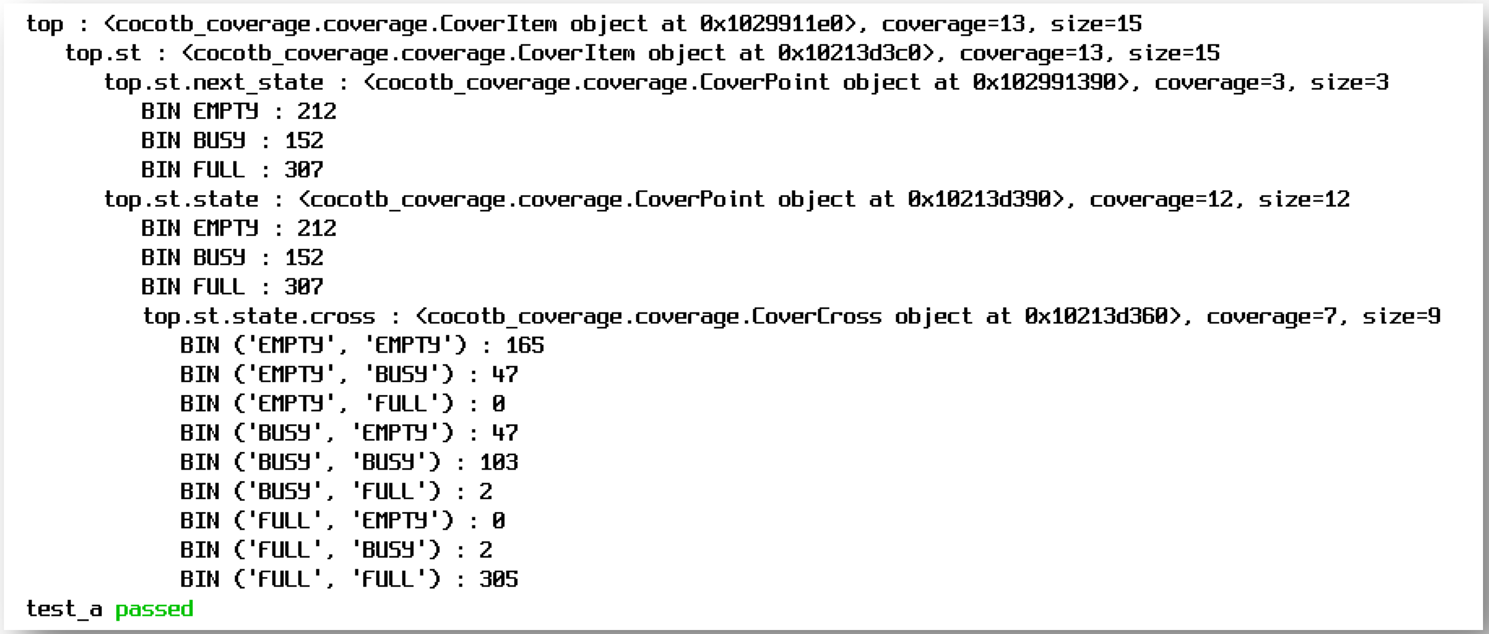
```
SC = coverage_section (
CoverPoint("top.st.state",
            xf=lambda s, ns: s,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverPoint("top.st.next_state",
            xf=lambda s, ns: ns,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverCross("top.st.state.cross",
            items=["top.st.state", "top.st.next_state"],

            )
)
```

- We started to classify existence into various bins of coverage. Each dimension was called a Coverpoint
- We started to lump them together.
- We at first looked at how well we were testing the FSM portion of the skid buffer and its state transitions

Results

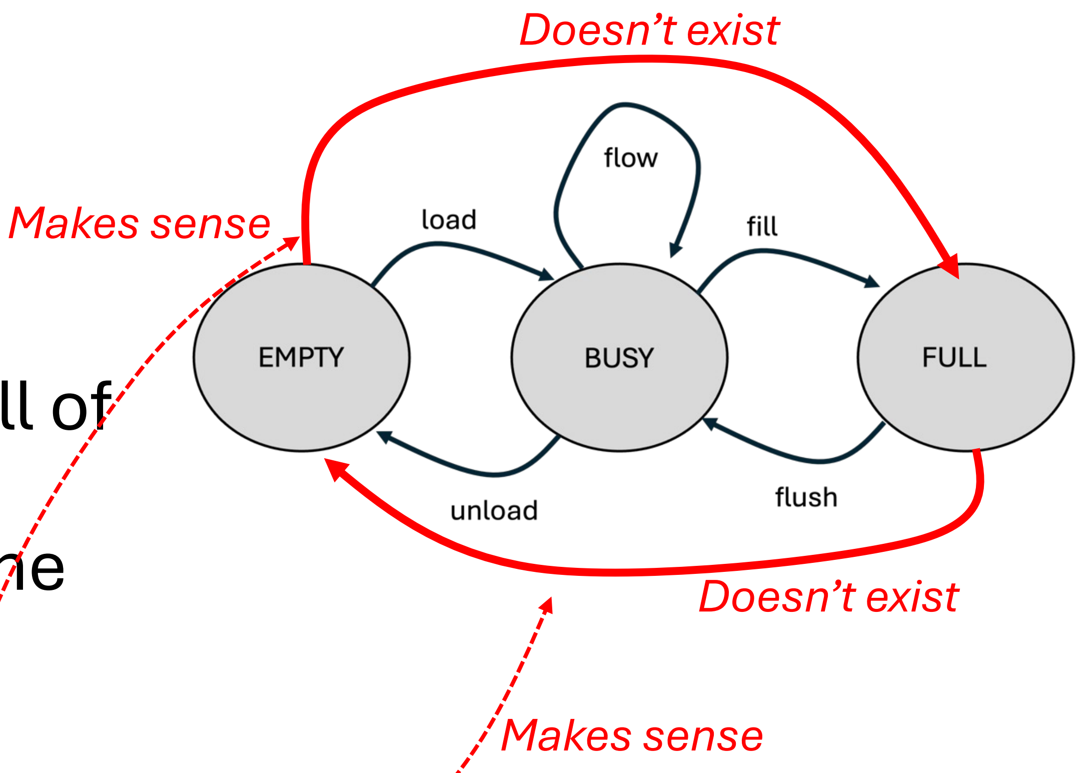
- The FSM was in all of its states pretty regularly during the test

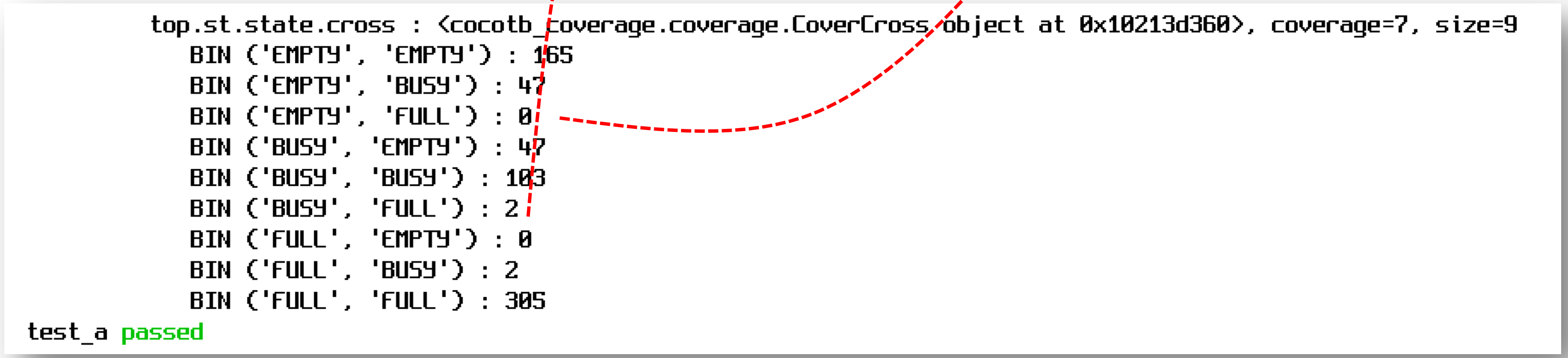



```
top : <cocotb_coverage.coverage.CoverItem object at 0x1029911e0>, coverage=13, size=15
   top.st : <cocotb_coverage.coverage.CoverItem object at 0x10213d3c0>, coverage=13, size=15
      top.st.next_state : <cocotb_coverage.coverage.CoverPoint object at 0x102991390>, coverage=3, size=3
         BIN EMPTY : 212
         BIN BUSY : 152
         BIN FULL : 307
      top.st.state : <cocotb_coverage.coverage.CoverPoint object at 0x10213d390>, coverage=12, size=12
         BIN EMPTY : 212
         BIN BUSY : 152
         BIN FULL : 307
         top.st.state.cross : <cocotb_coverage.coverage.CoverCross object at 0x10213d360>, coverage=7, size=9
            BIN ('EMPTY', 'EMPTY') : 165
            BIN ('EMPTY', 'BUSY') : 47
            BIN ('EMPTY', 'FULL') : 0
            BIN ('BUSY', 'EMPTY') : 47
            BIN ('BUSY', 'BUSY') : 103
            BIN ('BUSY', 'FULL') : 2
            BIN ('FULL', 'EMPTY') : 0
            BIN ('FULL', 'BUSY') : 2
            BIN ('FULL', 'FULL') : 305
test_a passed
```

Results

- The FSM was in all of its states pretty regularly during the test





```
        top.st.state.cross : <cocotb_coverage.coverage.CoverCross object at 0x10213d360>, coverage=7, size=9
           BIN ('EMPTY', 'EMPTY') : 165
           BIN ('EMPTY', 'BUSY') : 47
           BIN ('EMPTY', 'FULL') : 0
           BIN ('BUSY', 'EMPTY') : 47
           BIN ('BUSY', 'BUSY') : 103
           BIN ('BUSY', 'FULL') : 2
           BIN ('FULL', 'EMPTY') : 0
           BIN ('FULL', 'BUSY') : 2
           BIN ('FULL', 'FULL') : 305
test_a passed
```

If you know things shouldn't happen

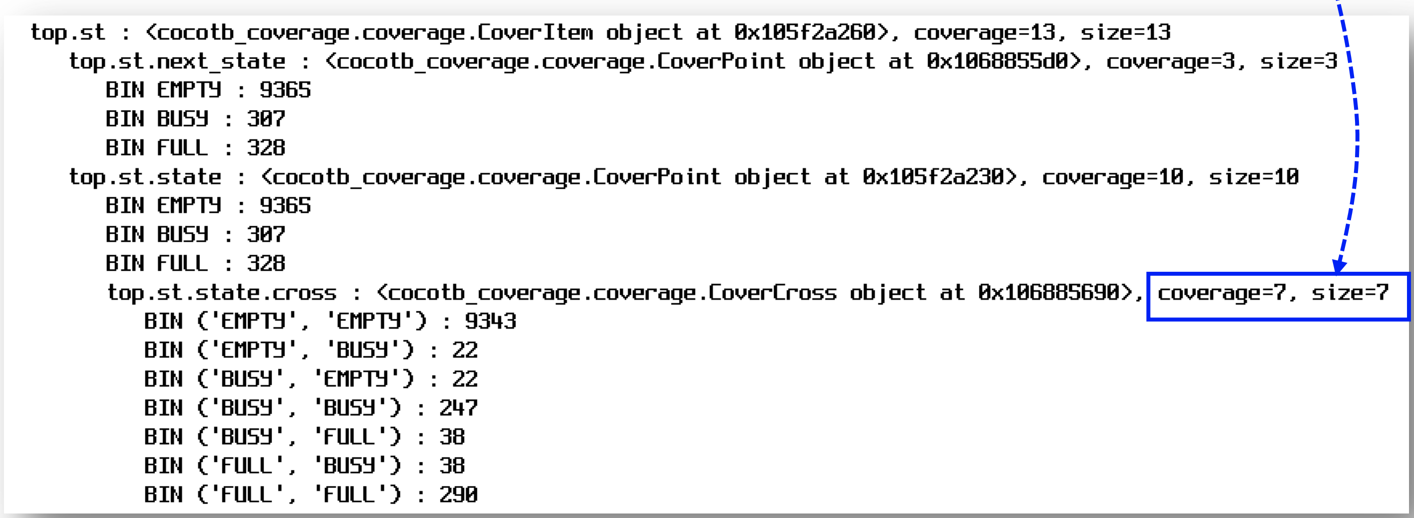
ignore_bins

```
SC = coverage_section (
CoverPoint("top.st.state",
            xf=lambda s, ns: s,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverPoint("top.st.next_state",
            xf=lambda s, ns: ns,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverCross("top.st.state.cross",
            items=["top.st.state", "top.st.next_state"],
            ign_bins=[('FULL','EMPTY'), ('EMPTY','FULL')])
)
```

Can now target 100% coverage

- If you can prove through some mechanism or another which bins should be reachable and which are false or unachievable, then you can view your coverage more as a milestone

nice

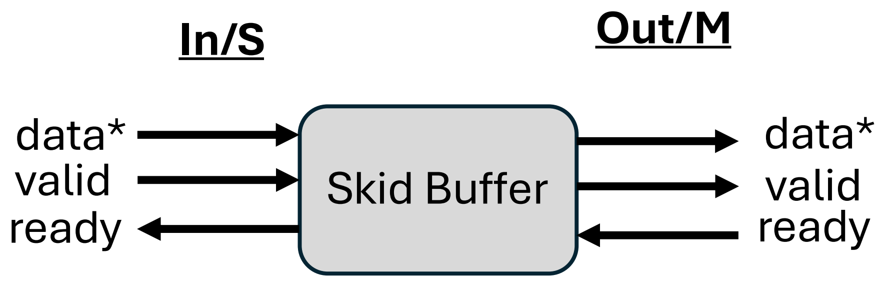


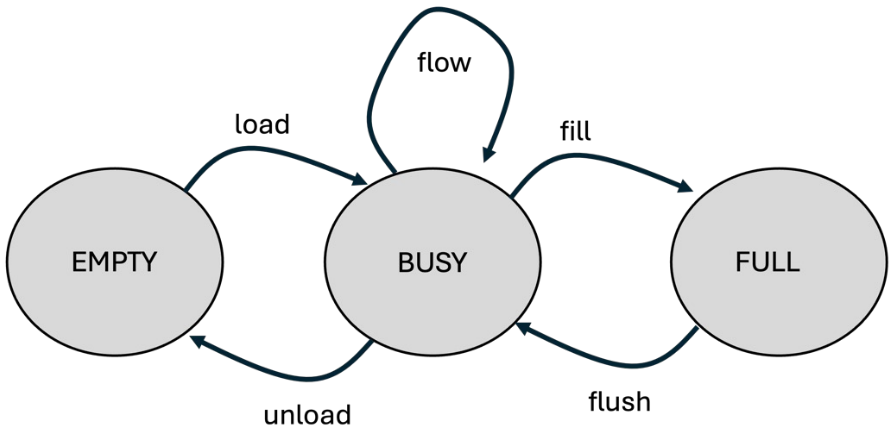
```
top.st : <cocotb_coverage.coverage.CoverItem object at 0x105f2a260>, coverage=13, size=13
   top.st.next_state : <cocotb_coverage.coverage.CoverPoint object at 0x1068855d0>, coverage=3, size=3
      BIN EMPTY : 9365
      BIN BUSY : 307
      BIN FULL : 328
   top.st.state : <cocotb_coverage.coverage.CoverPoint object at 0x105f2a230>, coverage=10, size=10
      BIN EMPTY : 9365
      BIN BUSY : 307
      BIN FULL : 328
      top.st.state.cross : <cocotb_coverage.coverage.CoverCross object at 0x106885690>, coverage=7, size=7
         BIN ('EMPTY', 'EMPTY') : 9343
         BIN ('EMPTY', 'BUSY') : 22
         BIN ('BUSY', 'EMPTY') : 22
         BIN ('BUSY', 'BUSY') : 247
         BIN ('BUSY', 'FULL') : 38
         BIN ('FULL', 'BUSY') : 38
         BIN ('FULL', 'FULL') : 290
```

Different tests but still you can see we got 100% coverage

Further Pushing on this System

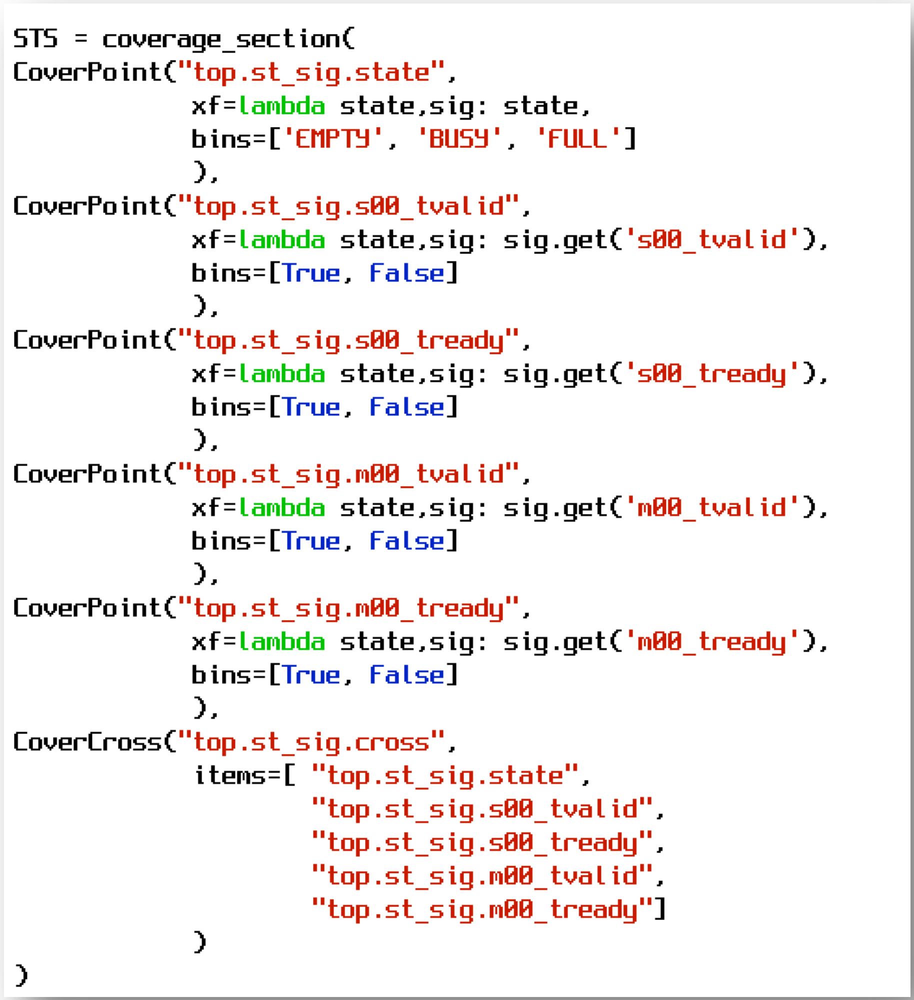
This simple FSM description…glossed over the potential complexity of the implementation: 3 states, each connected to 2 signals (valid/ready) per interface, for a total of 16 possible transitions out of each state, or 48 possible state transitions total.





So let's do state and input

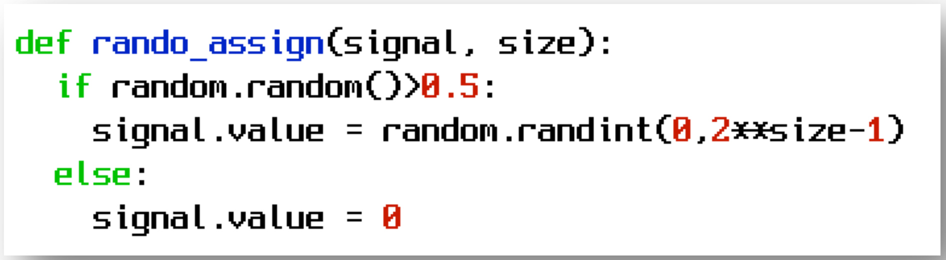
- Come up with STS covergroup (State and Signals)
- I want to look at the different states of my module as well as its exposure to different signal combinations on both S00 and M00 side



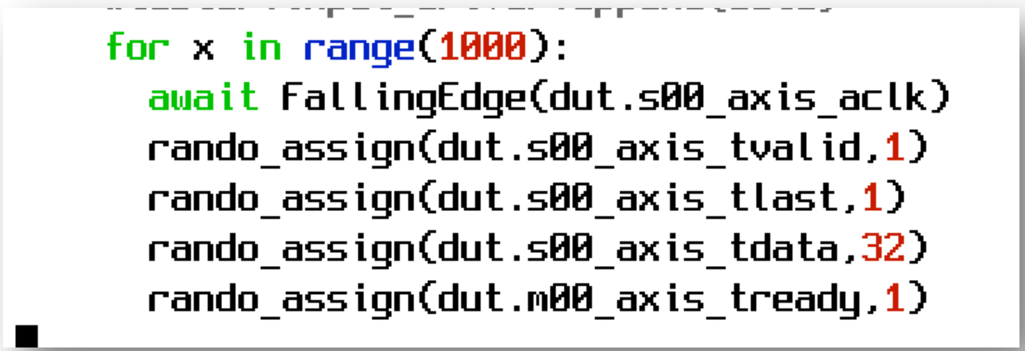
```
STS = coverage_section(
CoverPoint("top.st_sig.state",
            xf=lambda state,sig: state,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverPoint("top.st_sig.s00_tvalid",
            xf=lambda state,sig: sig.get('s00_tvalid'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.s00_tready",
            xf=lambda state,sig: sig.get('s00_tready'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.m00_tvalid",
            xf=lambda state,sig: sig.get('m00_tvalid'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.m00_tready",
            xf=lambda state,sig: sig.get('m00_tready'),
            bins=[True, False]
            ),
CoverCross("top.st_sig.cross",
            items=[ "top.st_sig.state",
                    "top.st_sig.s00_tvalid",
                    "top.st_sig.s00_tready",
                    "top.st_sig.m00_tvalid",
                    "top.st_sig.m00_tready"]
            )
)
```

Just Start Throwing Stuff at it...

- Depart for a moment and just start using random numbers to set values on these four lines and see what patterns emerge

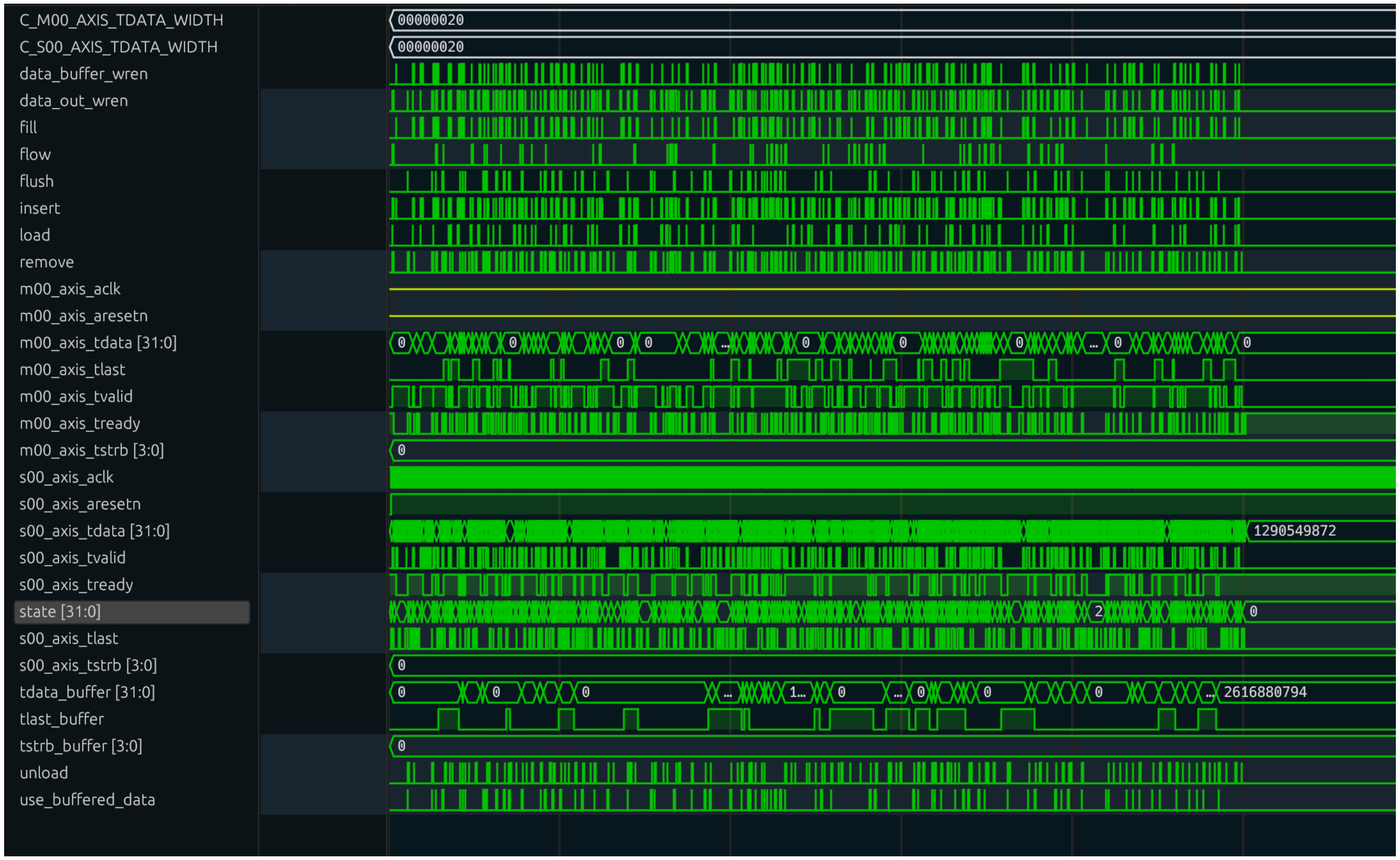


```
def rando_assign(signal, size):
  if random.random()>0.5:
    signal.value = random.randint(0,2**size-1)
  else:
    signal.value = 0
```



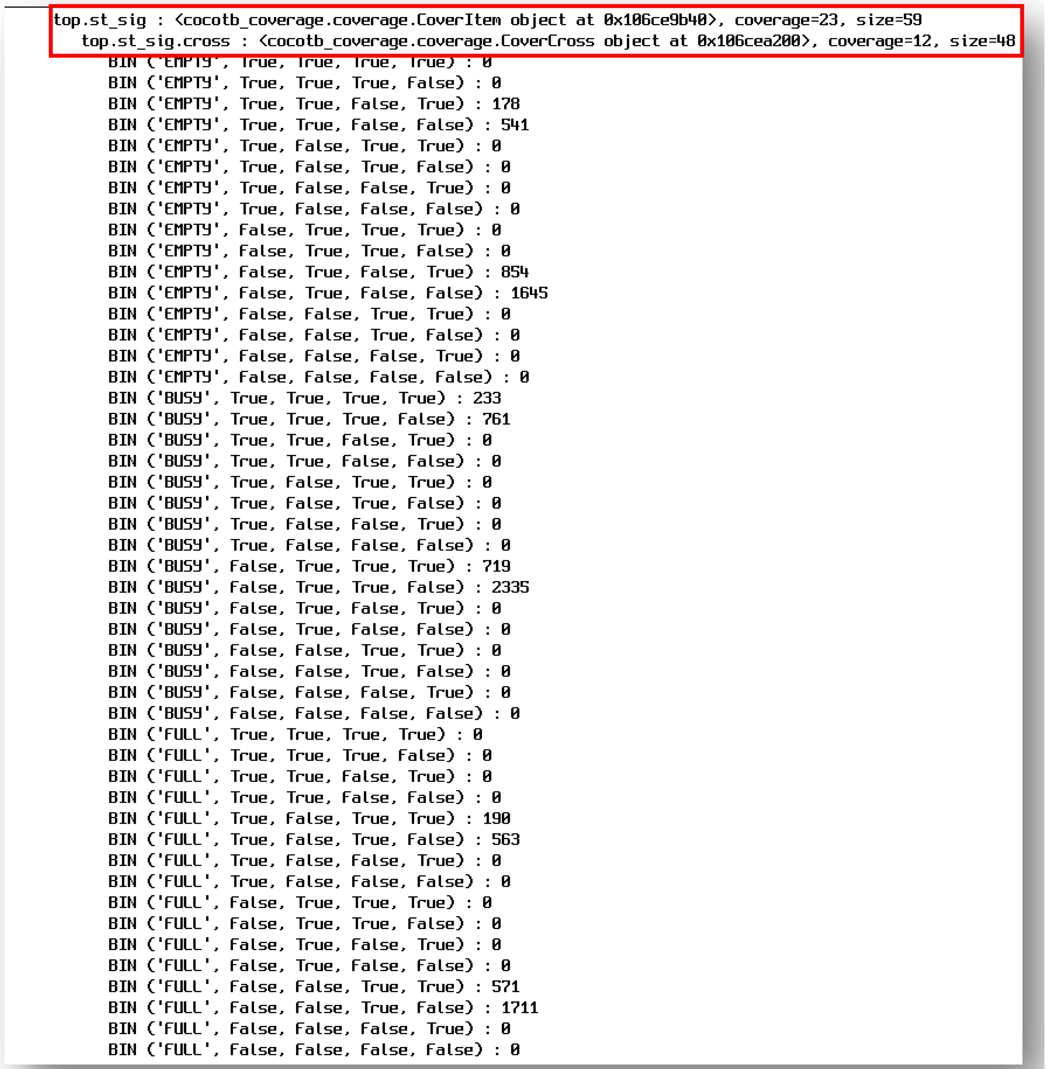
```
    for x in range(1000):
        await FallingEdge(dut.s00_axis_aclk)
        rando_assign(dut.s00_axis_tvalid,1)
        rando_assign(dut.s00_axis_tlast,1)
        rando_assign(dut.s00_axis_tdata,32)
        rando_assign(dut.m00_axis_tready,1)
```

Resulting Waveform



Results

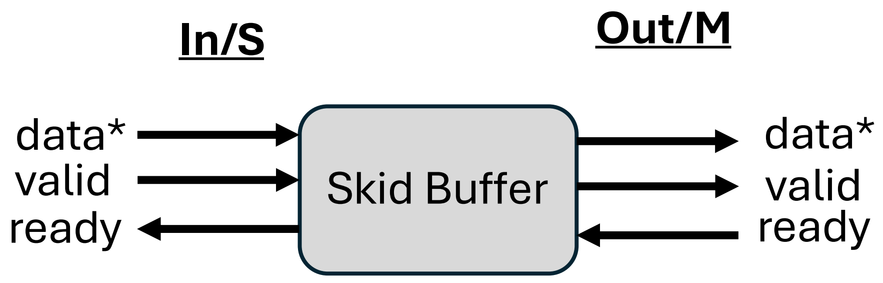
- It does seem to have some "coverage" of the input space
- but how much of that is relevant or not relevant.

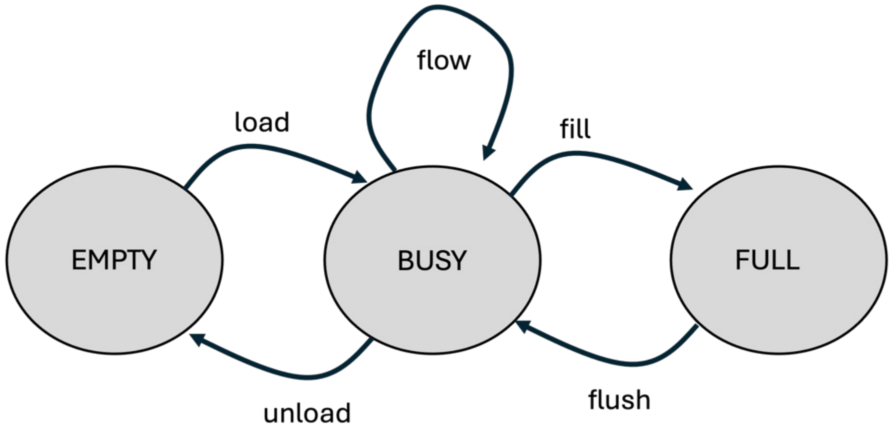


```
top.st_sig : <cocotb_coverage.coverage.CoverItem object at 0x106ce9b40>, coverage=23, size=59
   top.st_sig.cross : <cocotb_coverage.coverage.CoverCross object at 0x106cea200>, coverage=12, size=48
      BIN ('EMPTY', True, True, True, True) : 0
      BIN ('EMPTY', True, True, True, False) : 0
      BIN ('EMPTY', True, True, False, True) : 178
      BIN ('EMPTY', True, True, False, False) : 541
      BIN ('EMPTY', True, False, True, True) : 0
      BIN ('EMPTY', True, False, True, False) : 0
      BIN ('EMPTY', True, False, False, True) : 0
      BIN ('EMPTY', True, False, False, False) : 0
      BIN ('EMPTY', False, True, True, True) : 0
      BIN ('EMPTY', False, True, True, False) : 0
      BIN ('EMPTY', False, True, False, True) : 854
      BIN ('EMPTY', False, True, False, False) : 1645
      BIN ('EMPTY', False, False, True, True) : 0
      BIN ('EMPTY', False, False, True, False) : 0
      BIN ('EMPTY', False, False, False, True) : 0
      BIN ('EMPTY', False, False, False, False) : 0
      BIN ('BUSY', True, True, True, True) : 233
      BIN ('BUSY', True, True, True, False) : 761
      BIN ('BUSY', True, True, False, True) : 0
      BIN ('BUSY', True, True, False, False) : 0
      BIN ('BUSY', True, False, True, True) : 0
      BIN ('BUSY', True, False, True, False) : 0
      BIN ('BUSY', True, False, False, True) : 0
      BIN ('BUSY', True, False, False, False) : 0
      BIN ('BUSY', False, True, True, True) : 719
      BIN ('BUSY', False, True, True, False) : 2335
      BIN ('BUSY', False, True, False, True) : 0
      BIN ('BUSY', False, True, False, False) : 0
      BIN ('BUSY', False, False, True, True) : 0
      BIN ('BUSY', False, False, True, False) : 0
      BIN ('BUSY', False, False, False, True) : 0
      BIN ('BUSY', False, False, False, False) : 0
      BIN ('FULL', True, True, True, True) : 0
      BIN ('FULL', True, True, True, False) : 0
      BIN ('FULL', True, True, False, True) : 0
      BIN ('FULL', True, True, False, False) : 0
      BIN ('FULL', True, False, True, True) : 190
      BIN ('FULL', True, False, True, False) : 563
      BIN ('FULL', True, False, False, True) : 0
      BIN ('FULL', True, False, False, False) : 0
      BIN ('FULL', False, True, True, True) : 0
      BIN ('FULL', False, True, True, False) : 0
      BIN ('FULL', False, True, False, True) : 0
      BIN ('FULL', False, True, False, False) : 0
      BIN ('FULL', False, False, True, True) : 571
      BIN ('FULL', False, False, True, False) : 1711
      BIN ('FULL', False, False, False, True) : 0
      BIN ('FULL', False, False, False, False) : 0
```

At the naïve level…

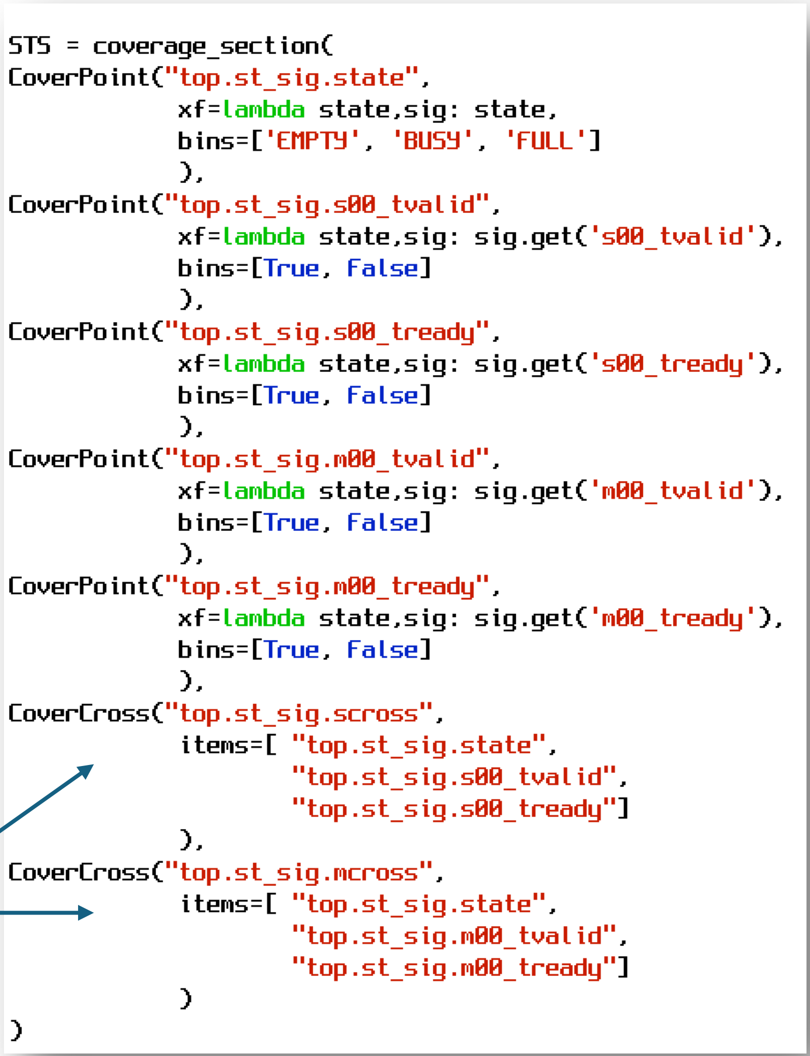
- Yes there are 48 possible state transitions and things, but as Jordan confidently pointed out on Monday, the state controls some of these signals, so that seems maybe a little excessive.





Change the Crosses

- There’s likely no reason (at least at this point) to have the signals on both sides mixed together in one large coverage cross



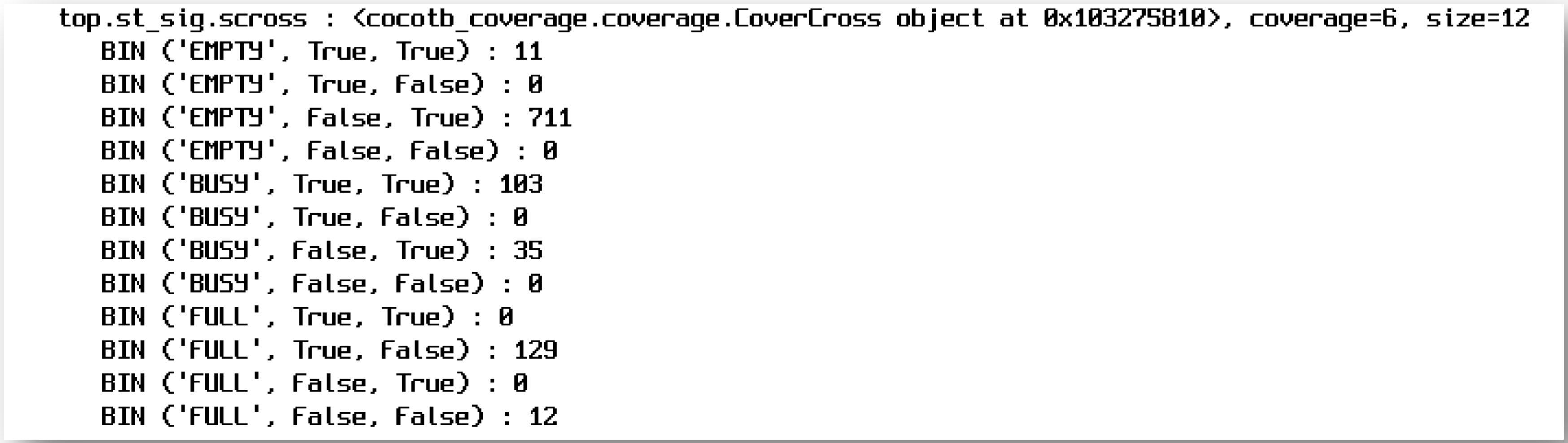
```
STS = coverage_section(
CoverPoint("top.st_sig.state",
            xf=lambda state,sig: state,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverPoint("top.st_sig.s00_tvalid",
            xf=lambda state,sig: sig.get('s00_tvalid'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.s00_tready",
            xf=lambda state,sig: sig.get('s00_tready'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.m00_tvalid",
            xf=lambda state,sig: sig.get('m00_tvalid'),
            bins=[True, False]
            ),
CoverPoint("top.st_sig.m00_tready",
            xf=lambda state,sig: sig.get('m00_tready'),
            bins=[True, False]
            ),
CoverCross("top.st_sig.scross",
            items=[ "top.st_sig.state",
                    "top.st_sig.s00_tvalid",
                    "top.st_sig.s00_tready"]
            ),
CoverCross("top.st_sig.mcross",
            items=[ "top.st_sig.state",
                    "top.st_sig.m00_tvalid",
                    "top.st_sig.m00_tready"]
            )
)
```

Only cross the state and values at each interface

Result

Slave Cross:

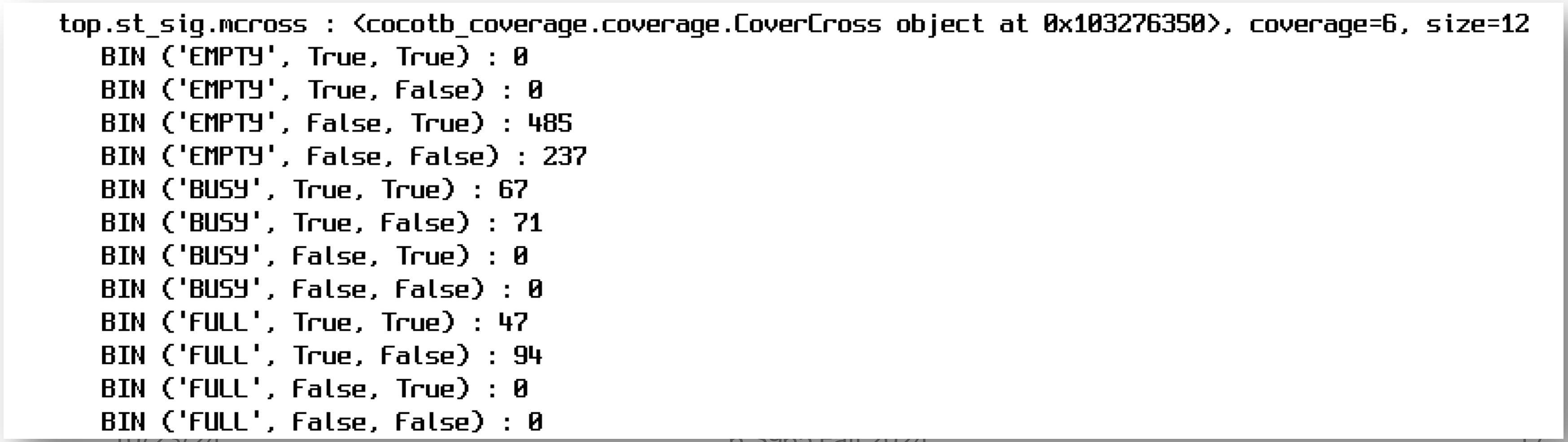
(STATE, VALID, READY)



```
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x103275810>, coverage=6, size=12
   BIN ('EMPTY', True, True) : 11
   BIN ('EMPTY', True, False) : 0
   BIN ('EMPTY', False, True) : 711
   BIN ('EMPTY', False, False) : 0
   BIN ('BUSY', True, True) : 103
   BIN ('BUSY', True, False) : 0
   BIN ('BUSY', False, True) : 35
   BIN ('BUSY', False, False) : 0
   BIN ('FULL', True, True) : 0
   BIN ('FULL', True, False) : 129
   BIN ('FULL', False, True) : 0
   BIN ('FULL', False, False) : 12
```

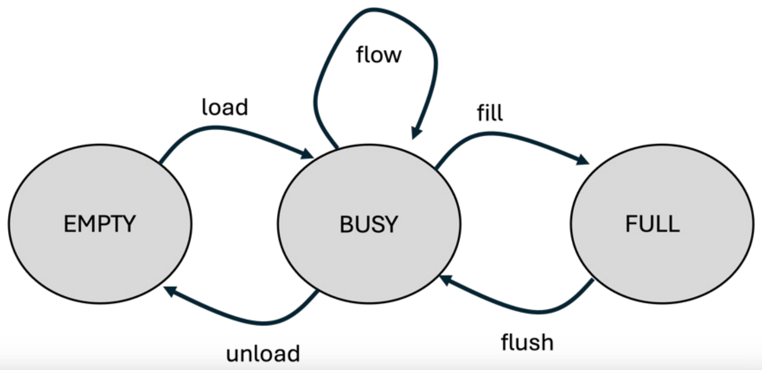
Master Cross:

(STATE, VALID, READY)

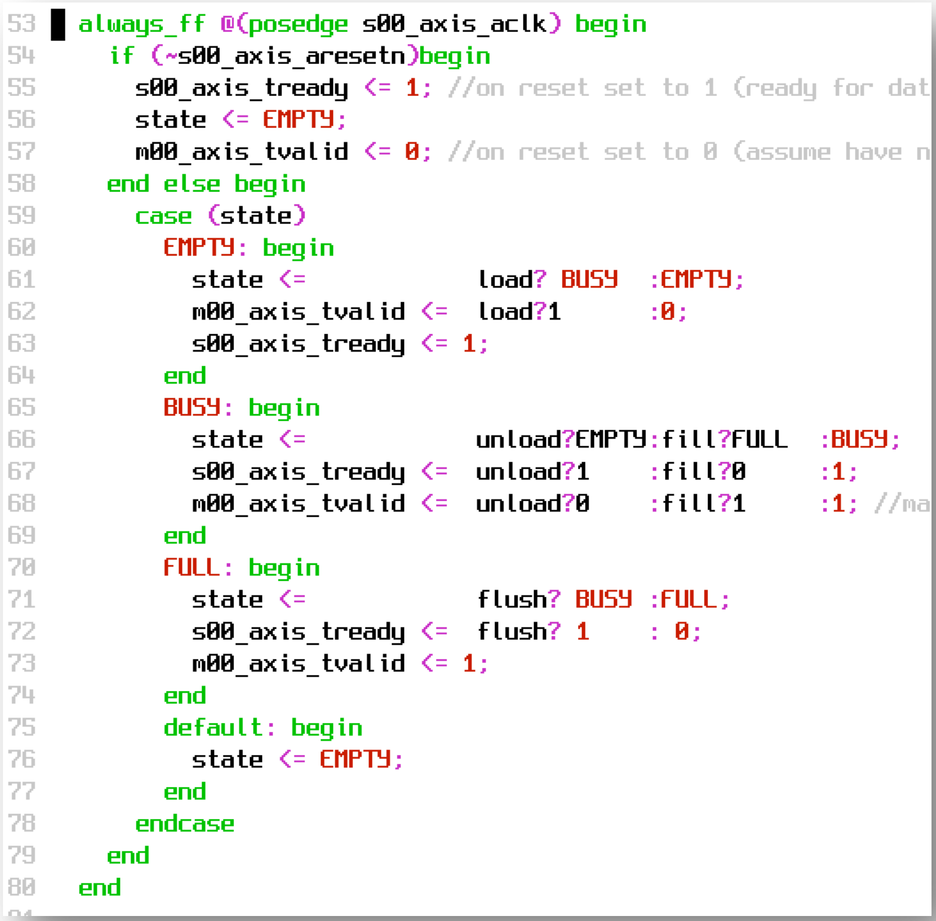


```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x103276350>, coverage=6, size=12
   BIN ('EMPTY', True, True) : 0
   BIN ('EMPTY', True, False) : 0
   BIN ('EMPTY', False, True) : 485
   BIN ('EMPTY', False, False) : 237
   BIN ('BUSY', True, True) : 67
   BIN ('BUSY', True, False) : 71
   BIN ('BUSY', False, True) : 0
   BIN ('BUSY', False, False) : 0
   BIN ('FULL', True, True) : 47
   BIN ('FULL', True, False) : 94
   BIN ('FULL', False, True) : 0
   BIN ('FULL', False, False) : 0
```

Look at our design



- Some of these cross values should not be achieved :
  - s00_axis_tready never 0 in EMPTY
  - m00_axis_tvalid never 0 in FULL



```
always_ff @(posedge s00_axis_aclk) begin
  if (~s00_axis_aresetn)begin
    s00_axis_tready <= 1; //on reset set to 1 (ready for dat
    state <= EMPTY;
    m00_axis_tvalid <= 0; //on reset set to 0 (assume have n
  end else begin
    case (state)
      EMPTY: begin
        state <=               load? BUSY   :EMPTY;
        m00_axis_tvalid <=     load?1       :0;
        s00_axis_tready <= 1;
      end
      BUSY: begin
        state <=               unload?EMPTY:fill?FULL   :BUSY;
        s00_axis_tready <=     unload?1    :fill?0      :1;
        m00_axis_tvalid <=     unload?0    :fill?1      :1; //ma
      end
      FULL: begin
        state <=               flush? BUSY :FULL;
        s00_axis_tready <=     flush? 1    : 0;
        m00_axis_tvalid <= 1;
      end
      default: begin
        state <= EMPTY;
      end
    endcase
  end
end
```

Result

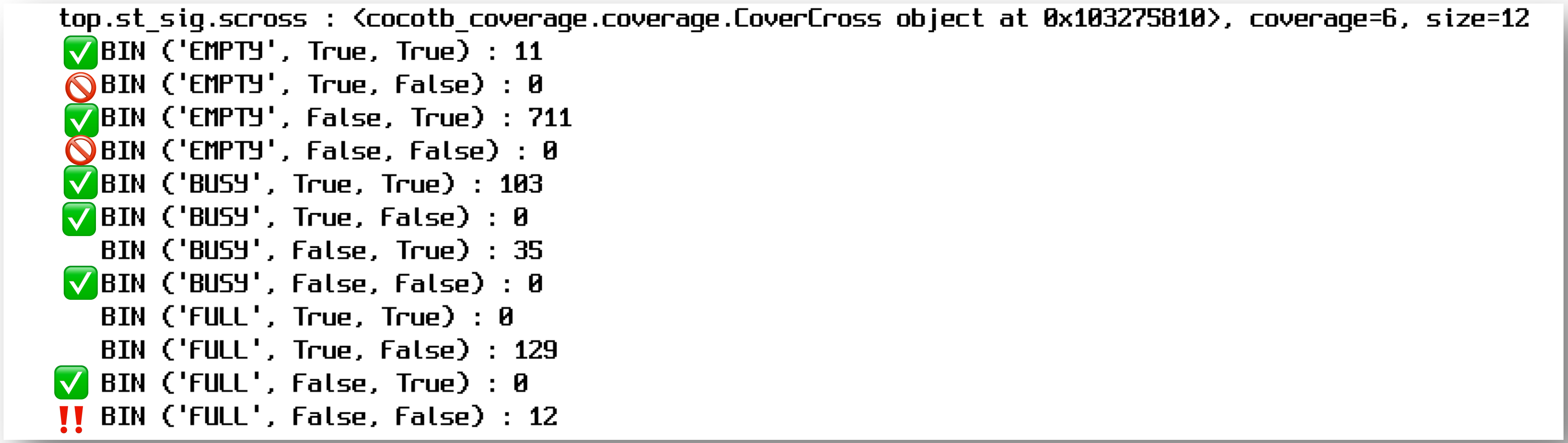
Slave Cross:

Legit/Might Occur:✅

Should Not Occur:🚫

s00_axis_tready never 0 in EMPTY
m00_axis_tvalid never 0 in FULL

(STATE, VALID, READY)

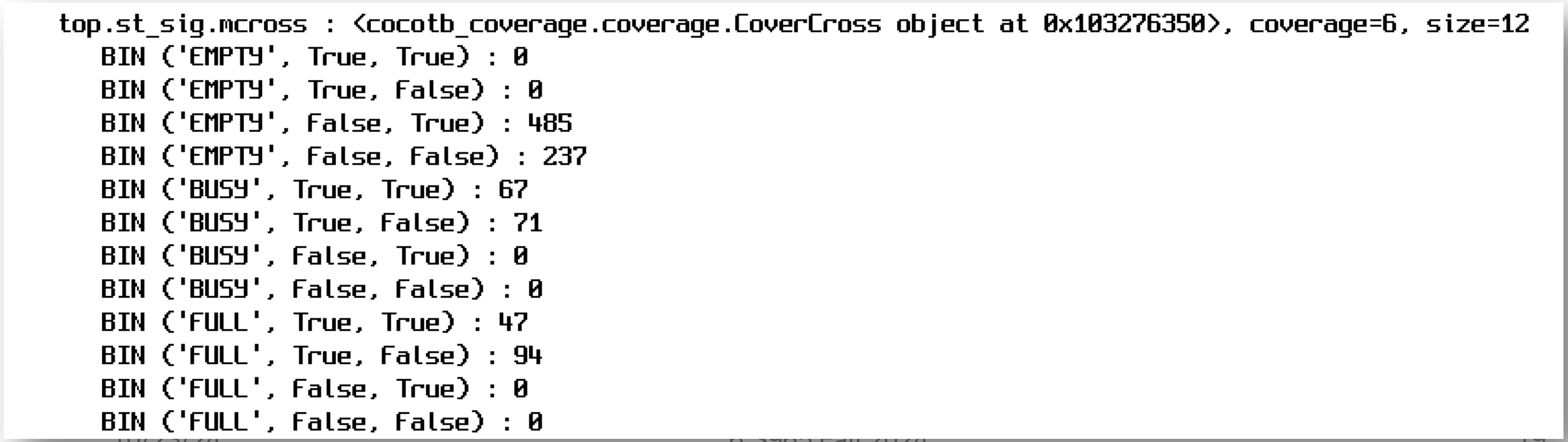


```
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x103275810>, coverage=6, size=12
✅BIN ('EMPTY', True, True) : 11
🚫BIN ('EMPTY', True, False) : 0
✅BIN ('EMPTY', False, True) : 711
🚫BIN ('EMPTY', False, False) : 0
✅BIN ('BUSY', True, True) : 103
✅BIN ('BUSY', True, False) : 0
  BIN ('BUSY', False, True) : 35
✅BIN ('BUSY', False, False) : 0
  BIN ('FULL', True, True) : 0
  BIN ('FULL', True, False) : 129
✅ BIN ('FULL', False, True) : 0
‼ BIN ('FULL', False, False) : 12
```

wtf

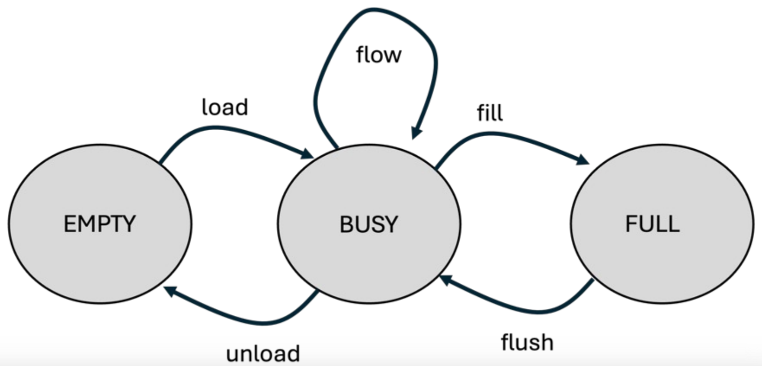
Master Cross:

(STATE, VALID, READY)

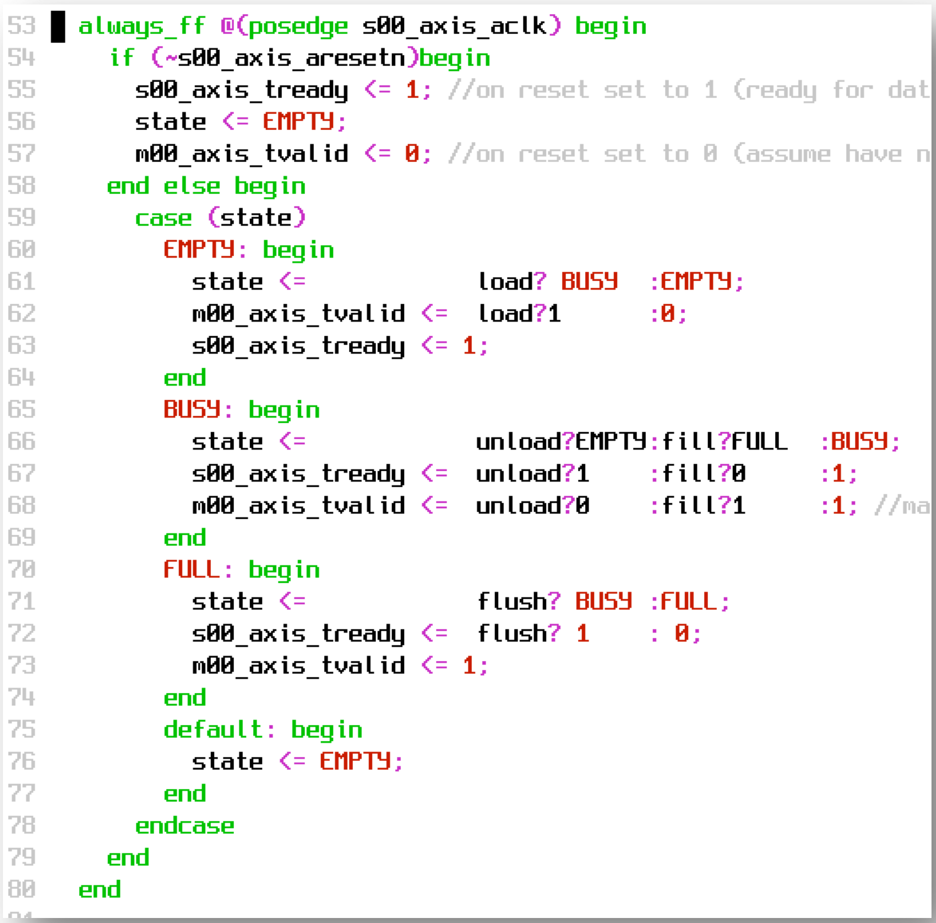


```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x103276350>, coverage=6, size=12
   BIN ('EMPTY', True, True) : 0
   BIN ('EMPTY', True, False) : 0
   BIN ('EMPTY', False, True) : 485
   BIN ('EMPTY', False, False) : 237
   BIN ('BUSY', True, True) : 67
   BIN ('BUSY', True, False) : 71
   BIN ('BUSY', False, True) : 0
   BIN ('BUSY', False, False) : 0
   BIN ('FULL', True, True) : 47
   BIN ('FULL', True, False) : 94
   BIN ('FULL', False, True) : 0
   BIN ('FULL', False, False) : 0
```

Look at our design



- Some of these cross values should not be achieved :
  - s00_axis_tready never 0 ***when was EMPTY***
  - m00_axis_tvalid never 0 ***when was FULL***



```
always_ff @(posedge s00_axis_aclk) begin
  if (~s00_axis_aresetn)begin
    s00_axis_tready <= 1; //on reset set to 1 (ready for dat
    state <= EMPTY;
    m00_axis_tvalid <= 0; //on reset set to 0 (assume have n
  end else begin
    case (state)
      EMPTY: begin
        state <=           load? BUSY  :EMPTY;
        m00_axis_tvalid <= load?1      :0;
        s00_axis_tready <= 1;
      end
      BUSY: begin
        state <=           unload?EMPTY:fill?FULL  :BUSY;
        s00_axis_tready <= unload?1    :fill?0     :1;
        m00_axis_tvalid <= unload?0    :fill?1     :1; //ma
      end
      FULL: begin
        state <=           flush? BUSY :FULL;
        s00_axis_tready <= flush? 1    : 0;
        m00_axis_tvalid <= 1;
      end
      default: begin
        state <= EMPTY;
      end
    endcase
  end
end
```

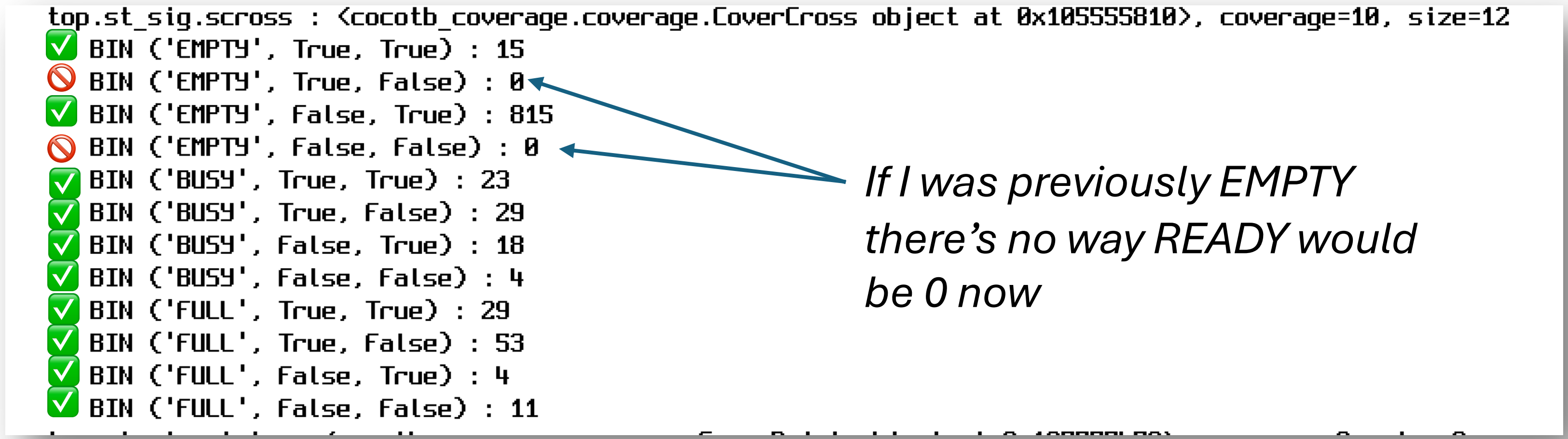
Should these be achievable?

Legit/Might Occur:✅

Should Not Occur:🚫

Slave Cross:

(OLD_STATE, VALID, READY)

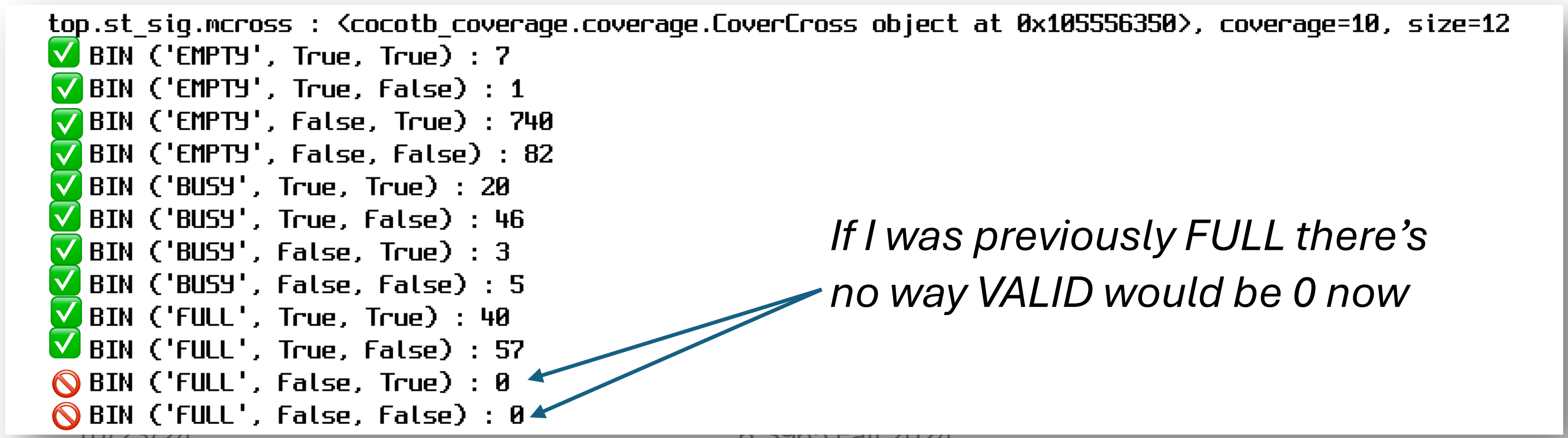


```
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x105555810>, coverage=10, size=12
✅ BIN ('EMPTY', True, True) : 15
🚫 BIN ('EMPTY', True, False) : 0
✅ BIN ('EMPTY', False, True) : 815
🚫 BIN ('EMPTY', False, False) : 0
✅ BIN ('BUSY', True, True) : 23
✅ BIN ('BUSY', True, False) : 29
✅ BIN ('BUSY', False, True) : 18
✅ BIN ('BUSY', False, False) : 4
✅ BIN ('FULL', True, True) : 29
✅ BIN ('FULL', True, False) : 53
✅ BIN ('FULL', False, True) : 4
✅ BIN ('FULL', False, False) : 11
```

If I was previously EMPTY there's no way READY would be 0 now

Master Cross:

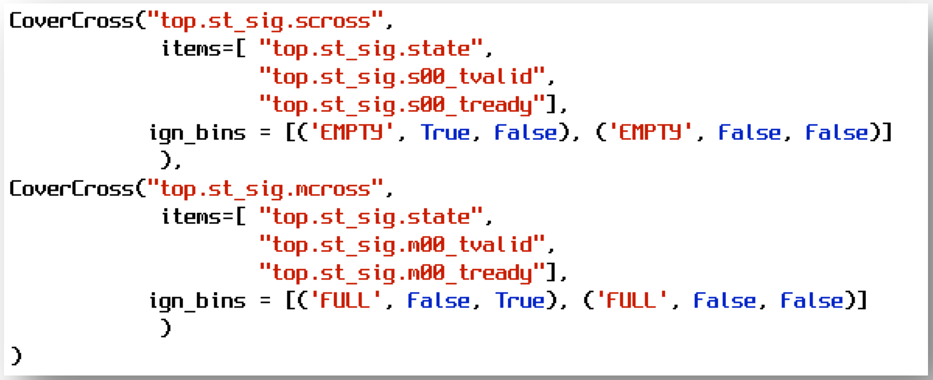
(OLD_STATE, VALID, READY)



```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x105556350>, coverage=10, size=12
✅ BIN ('EMPTY', True, True) : 7
✅ BIN ('EMPTY', True, False) : 1
✅ BIN ('EMPTY', False, True) : 740
✅ BIN ('EMPTY', False, False) : 82
✅ BIN ('BUSY', True, True) : 20
✅ BIN ('BUSY', True, False) : 46
✅ BIN ('BUSY', False, True) : 3
✅ BIN ('BUSY', False, False) : 5
✅ BIN ('FULL', True, True) : 40
✅ BIN ('FULL', True, False) : 57
🚫 BIN ('FULL', False, True) : 0
🚫 BIN ('FULL', False, False) : 0
```

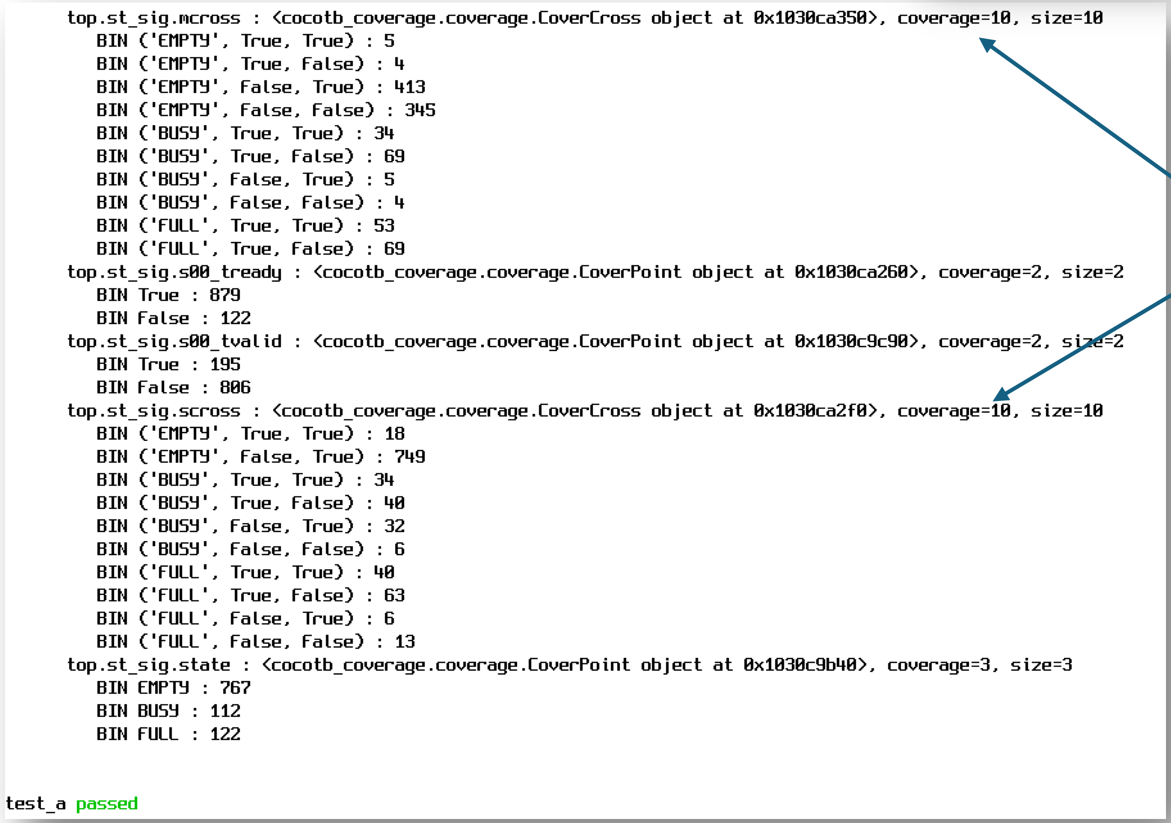
If I was previously FULL there's no way VALID would be 0 now

Ignore those...



```
CoverCross("top.st_sig.scross",
              items=[ "top.st_sig.state",
                       "top.st_sig.s00_tvalid",
                       "top.st_sig.s00_tready"],
             ign_bins = [('EMPTY', True, False), ('EMPTY', False, False)]
              ),
CoverCross("top.st_sig.mcross",
              items=[ "top.st_sig.state",
                       "top.st_sig.m00_tvalid",
                       "top.st_sig.m00_tready"],
             ign_bins = [('FULL', False, True), ('FULL', False, False)]
              )
)
```

- Run again:



```
      top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x1030ca350>, coverage=10, size=10
         BIN ('EMPTY', True, True) : 5
         BIN ('EMPTY', True, False) : 4
         BIN ('EMPTY', False, True) : 413
         BIN ('EMPTY', False, False) : 345
         BIN ('BUSY', True, True) : 34
         BIN ('BUSY', True, False) : 69
         BIN ('BUSY', False, True) : 5
         BIN ('BUSY', False, False) : 4
         BIN ('FULL', True, True) : 53
         BIN ('FULL', True, False) : 69
      top.st_sig.s00_tready : <cocotb_coverage.coverage.CoverPoint object at 0x1030ca260>, coverage=2, size=2
         BIN True : 879
         BIN False : 122
      top.st_sig.s00_tvalid : <cocotb_coverage.coverage.CoverPoint object at 0x1030c9c90>, coverage=2, size=2
         BIN True : 195
         BIN False : 806
      top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x1030ca2f0>, coverage=10, size=10
         BIN ('EMPTY', True, True) : 18
         BIN ('EMPTY', False, True) : 749
         BIN ('BUSY', True, True) : 34
         BIN ('BUSY', True, False) : 40
         BIN ('BUSY', False, True) : 32
         BIN ('BUSY', False, False) : 6
         BIN ('FULL', True, True) : 40
         BIN ('FULL', True, False) : 63
         BIN ('FULL', False, True) : 6
         BIN ('FULL', False, False) : 13
      top.st_sig.state : <cocotb_coverage.coverage.CoverPoint object at 0x1030c9b40>, coverage=3, size=3
         BIN EMPTY : 767
         BIN BUSY : 112
         BIN FULL : 122


test_a passed
```

Tests are doing 100% of coverage now

Another Big Issue

- AXI is about more than just the value at any point in time.
- As pointed out in class on Monday, AXI as a protocol has rules and those are rules are inherently stateful.
- Just throwing random values at the busses with no regard for history/meaning could be wrong:
  - Giving it illegal values
  - Wasting cycles testing stuff that shouldn’t be tested

Generalized Transaction

- All Channel Interactions follow same high-level structure
- Data is handed off IF AND ONLY IF VALID and READY are high on the rising edge of the clock
- If that happens, both parties must realize that data transfer has happened

Keep in mind this could be 64 parallel wires of 1's and 0's of info or 8 bytes for example…
Or it could be something else

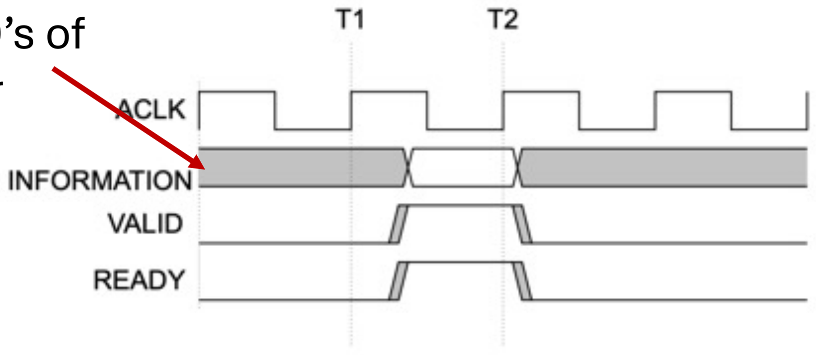


Figure A3-4 VALID with READY handshake

VALID then READY

- Valid can be high first
- Then ready can show up later
- Only when both are high is data exchanged

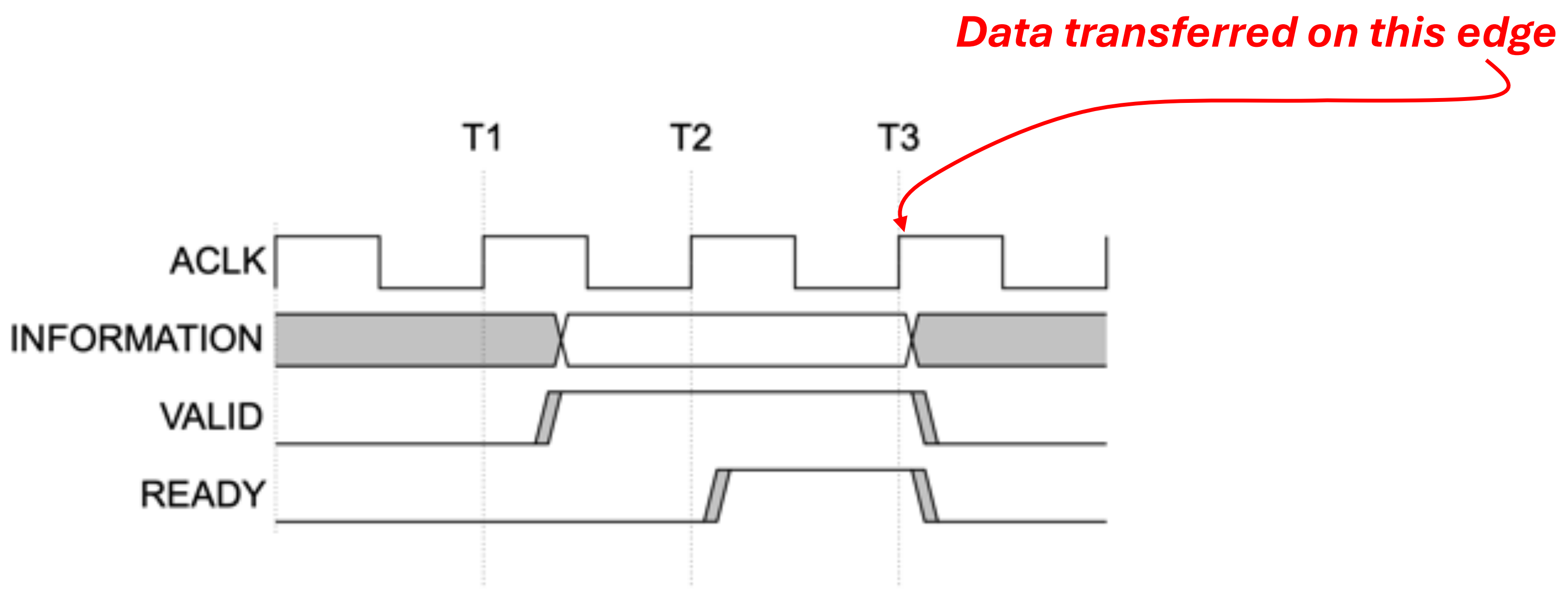


Figure A3-2 VALID before READY handshake

READY then VALID

- Ready can be high first
- Then Valid can show up later
- Only when both are high is data exchanged

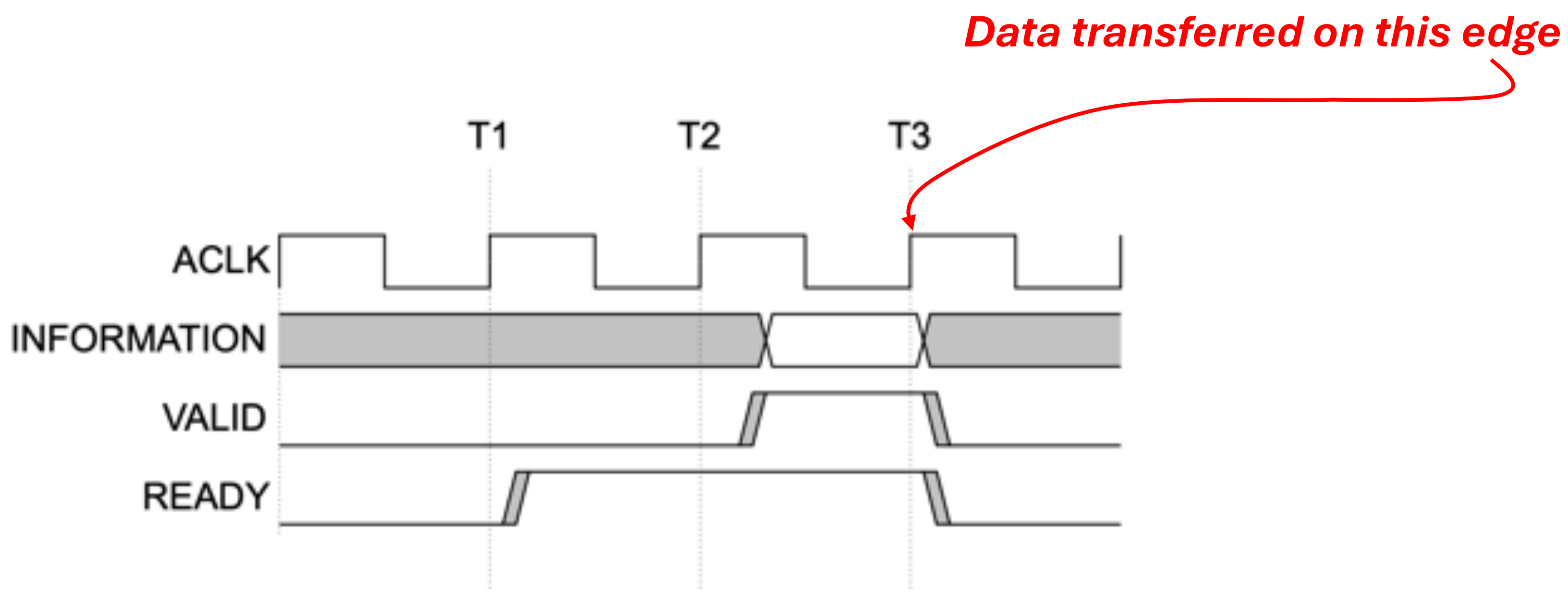


Figure A3-3 READY before VALID handshake

READY WITH VALID

- Ready and Valid come high at the same time
- Totally allowed
- Data is exchanged on that clock edge

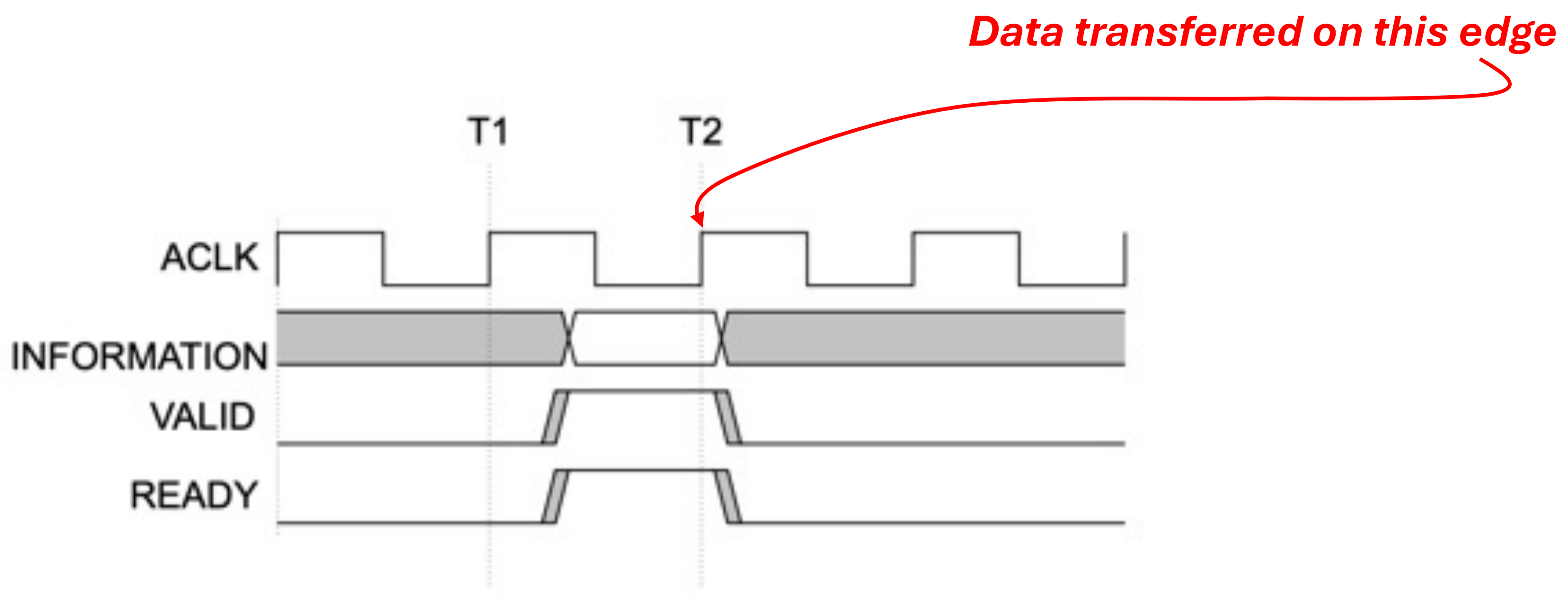
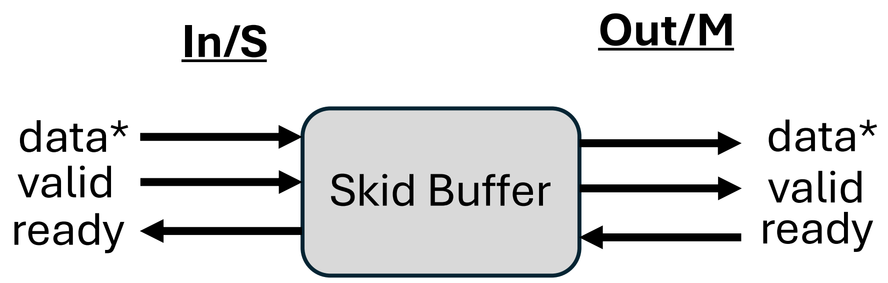


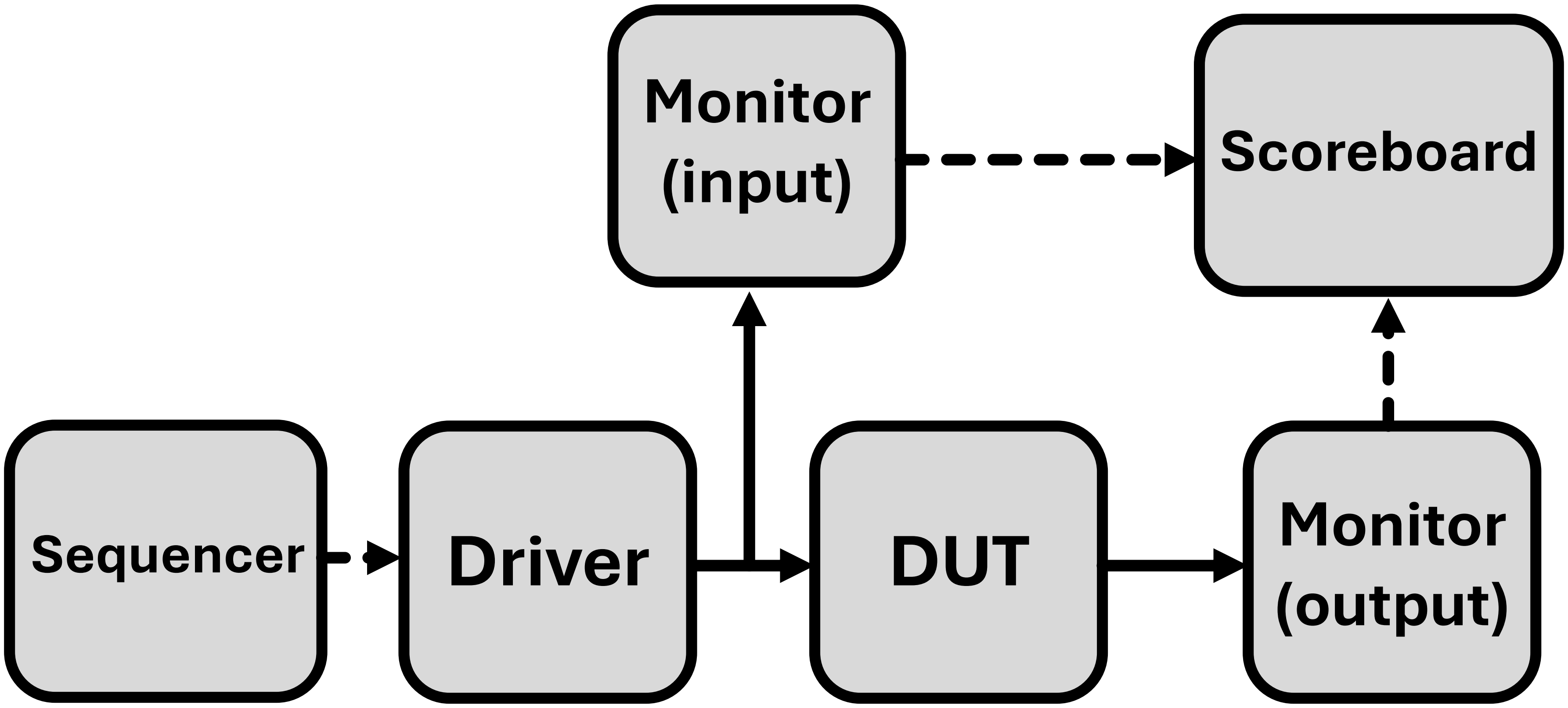
Figure A3-4 VALID with READY handshake

IMPORTANT

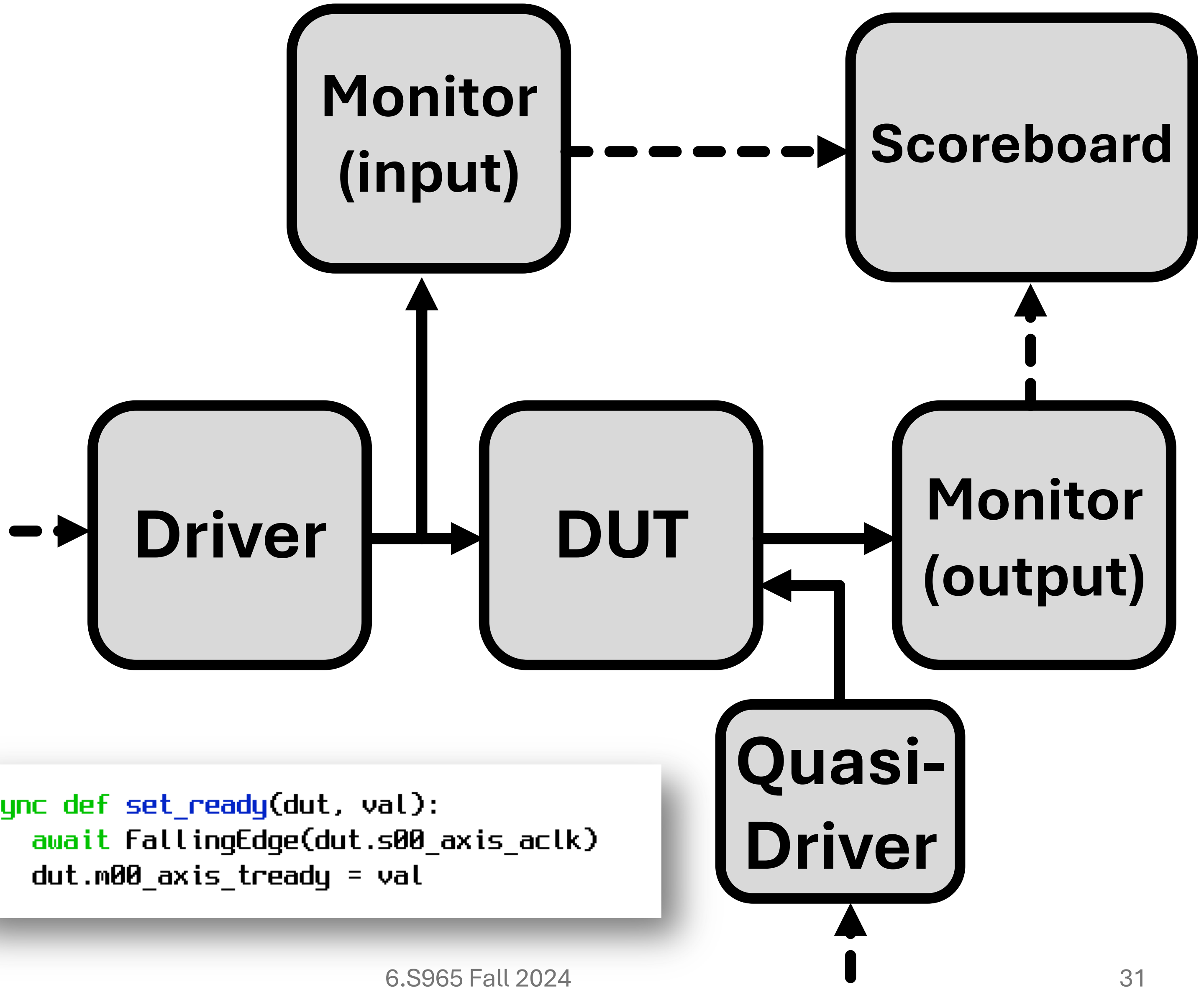
- the **VALID** signal of the AXI interface sending information *must not be dependent* on the **READY** signal of the AXI interface receiving that information
    - an AXI interface that is receiving information *may* wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.
    - In other words **READY** can depend on **VALID**, but not the other way around.
- Once **VALID** is asserted, it cannot be deasserted until **READY** has also been asserted for at least one cycle


In/S
Out/M
data*
valid
ready
Skid Buffer
data*
valid
ready

Standard Testing Framework



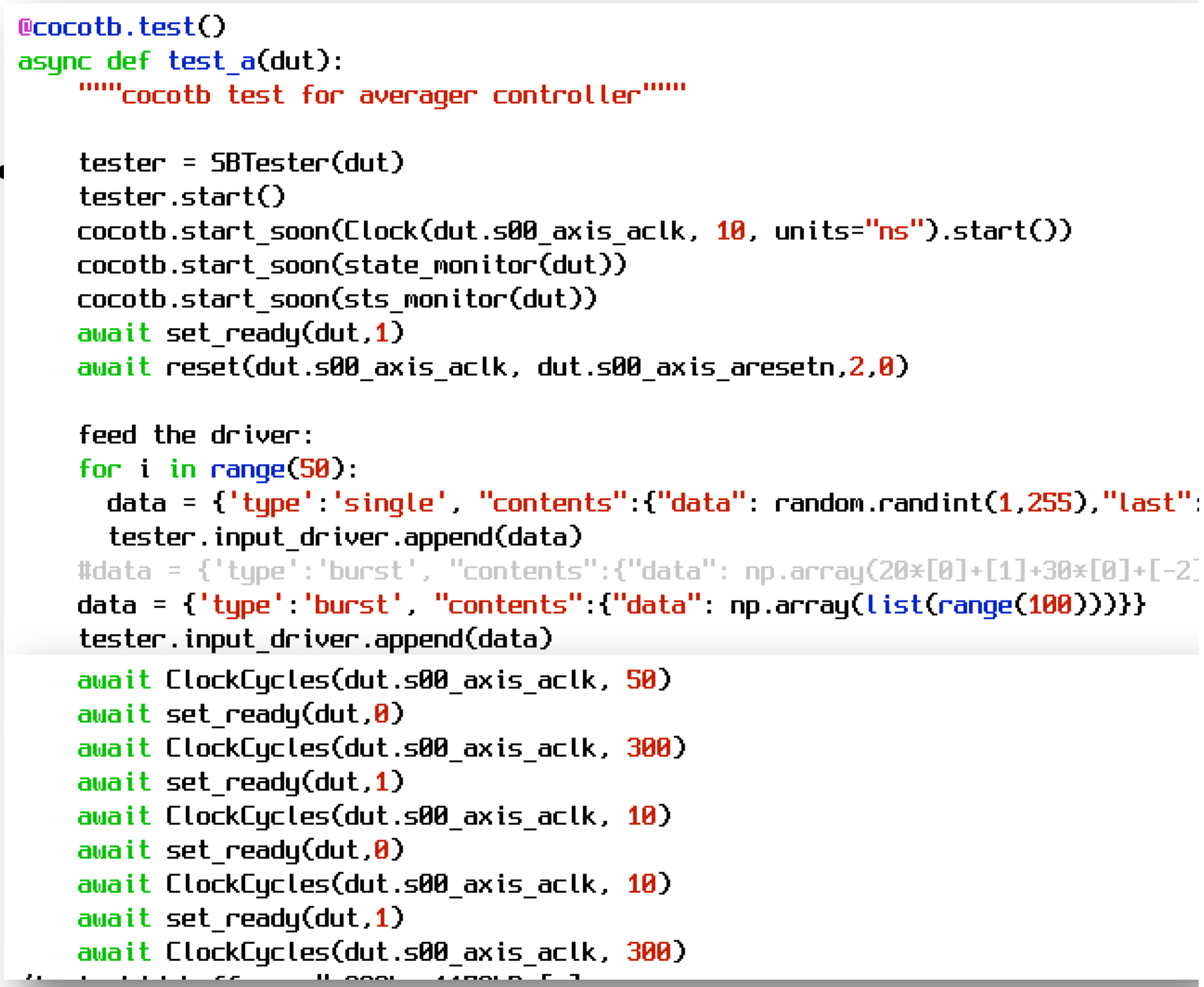
Our Current Testing Framework



```
async def set_ready(dut, val):
    await FallingEdge(dut.s00_axis_aclk)
    dut.m00_axis_tready = val
```

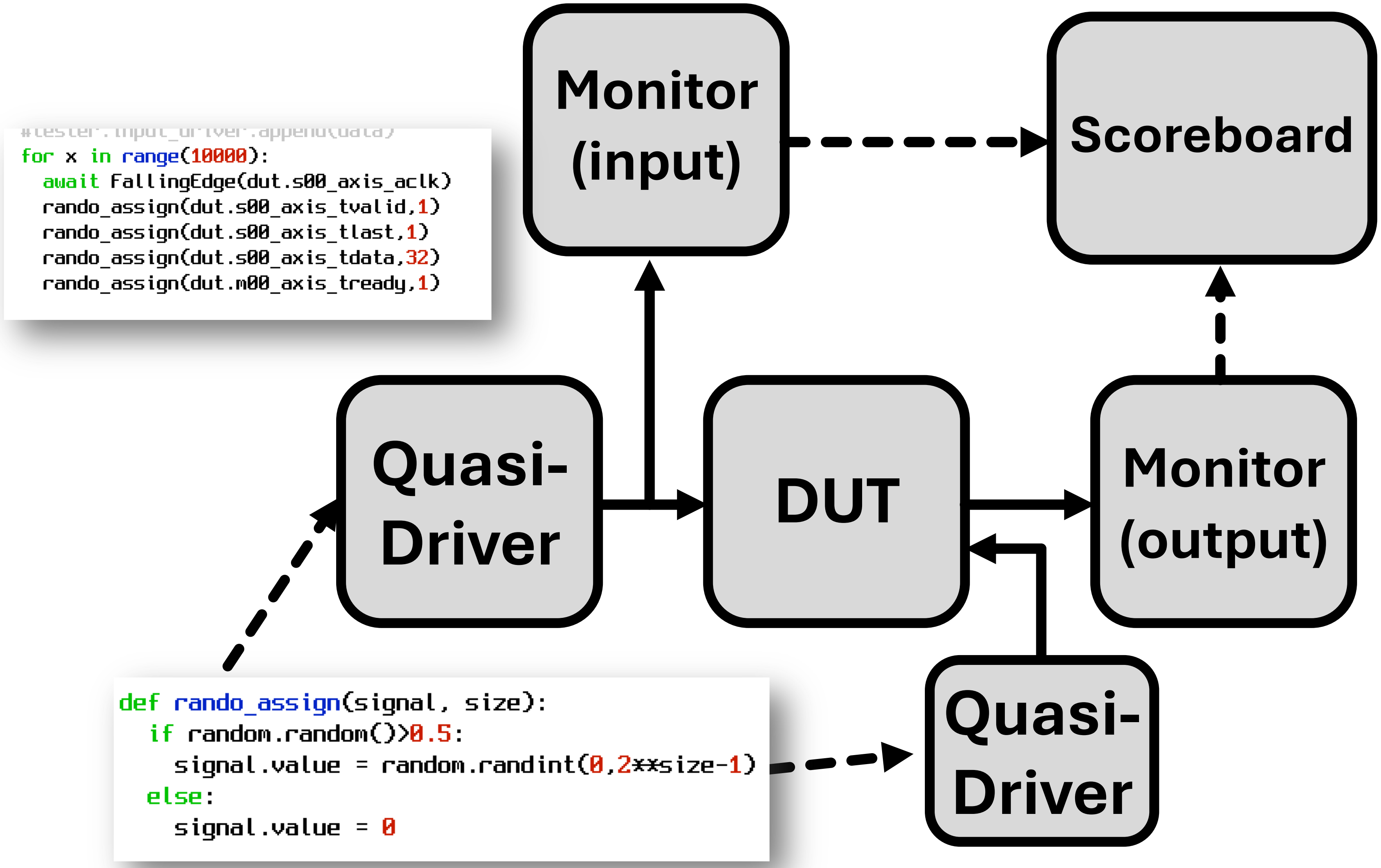
Right now…

- Kind just fudging the ready signal, but really we should try to more intelligently probe this thing



```
@cocotb.test()
async def test_a(dut):
    """cocotb test for averager controller"""

    tester = SBTester(dut)
    tester.start()
    cocotb.start_soon(Clock(dut.s00_axis_aclk, 10, units="ns").start())
    cocotb.start_soon(state_monitor(dut))
    cocotb.start_soon(sts_monitor(dut))
    await set_ready(dut,1)
    await reset(dut.s00_axis_aclk, dut.s00_axis_aresetn,2,0)

    feed the driver:
    for i in range(50):
      data = {'type':'single', "contents":{"data": random.randint(1,255),"last":
      tester.input_driver.append(data)
    #data = {'type':'burst', "contents":{"data": np.array(20*[0]+[1]+30*[0]+[-2
    data = {'type':'burst', "contents":{"data": np.array(list(range(100)))}}
    tester.input_driver.append(data)
    await ClockCycles(dut.s00_axis_aclk, 50)
    await set_ready(dut,0)
    await ClockCycles(dut.s00_axis_aclk, 300)
    await set_ready(dut,1)
    await ClockCycles(dut.s00_axis_aclk, 10)
    await set_ready(dut,0)
    await ClockCycles(dut.s00_axis_aclk, 10)
    await set_ready(dut,1)
    await ClockCycles(dut.s00_axis_aclk, 300)
```

Our Current Testing Framework

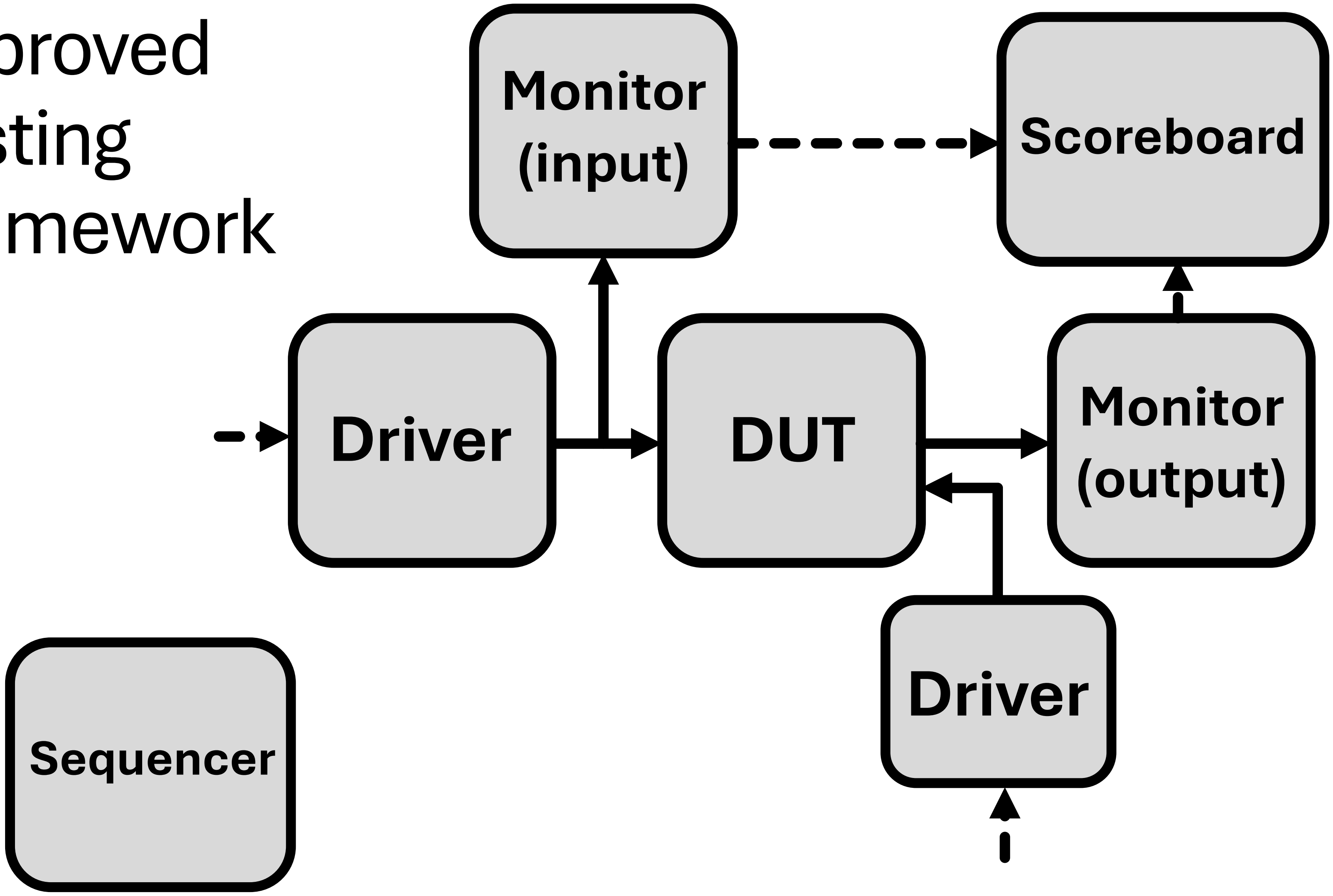


```python
#tester.input_driver.append(data)
for x in range(10000):
  await FallingEdge(dut.s00_axis_aclk)
  rando_assign(dut.s00_axis_tvalid,1)
  rando_assign(dut.s00_axis_tlast,1)
  rando_assign(dut.s00_axis_tdata,32)
  rando_assign(dut.m00_axis_tready,1)
```

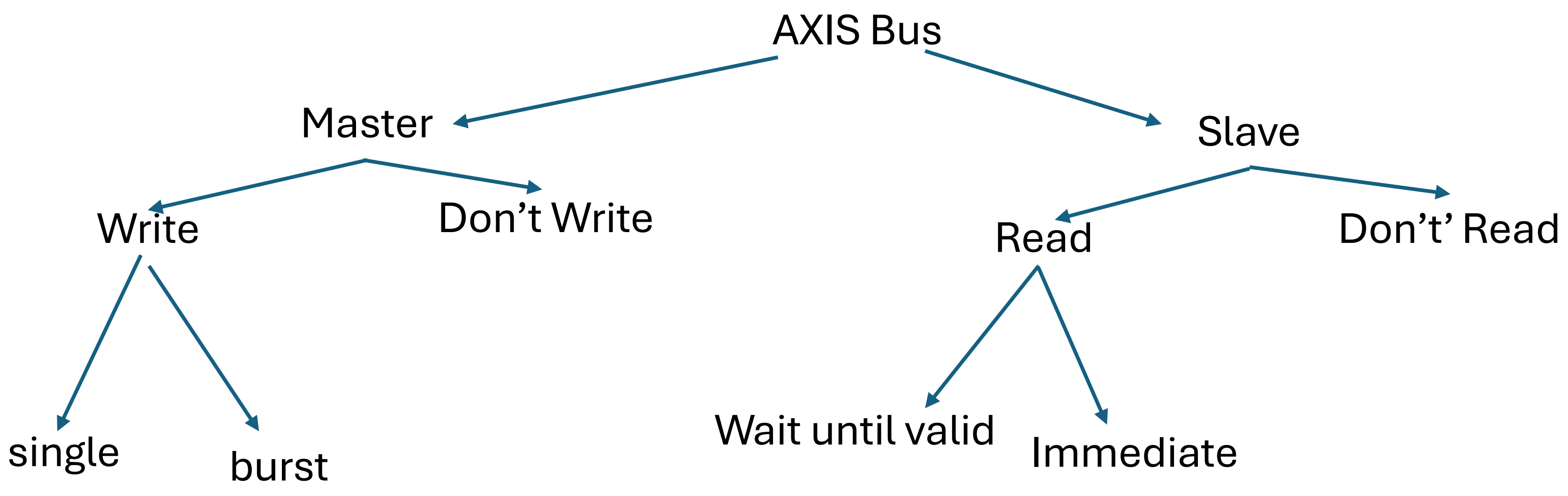
```python
def rando_assign(signal, size):
  if random.random()>0.5:
    signal.value = random.randint(0,2**size-1)
  else:
    signal.value = 0
```

What we’d really like is something to coordinate

Improved Testing Framework

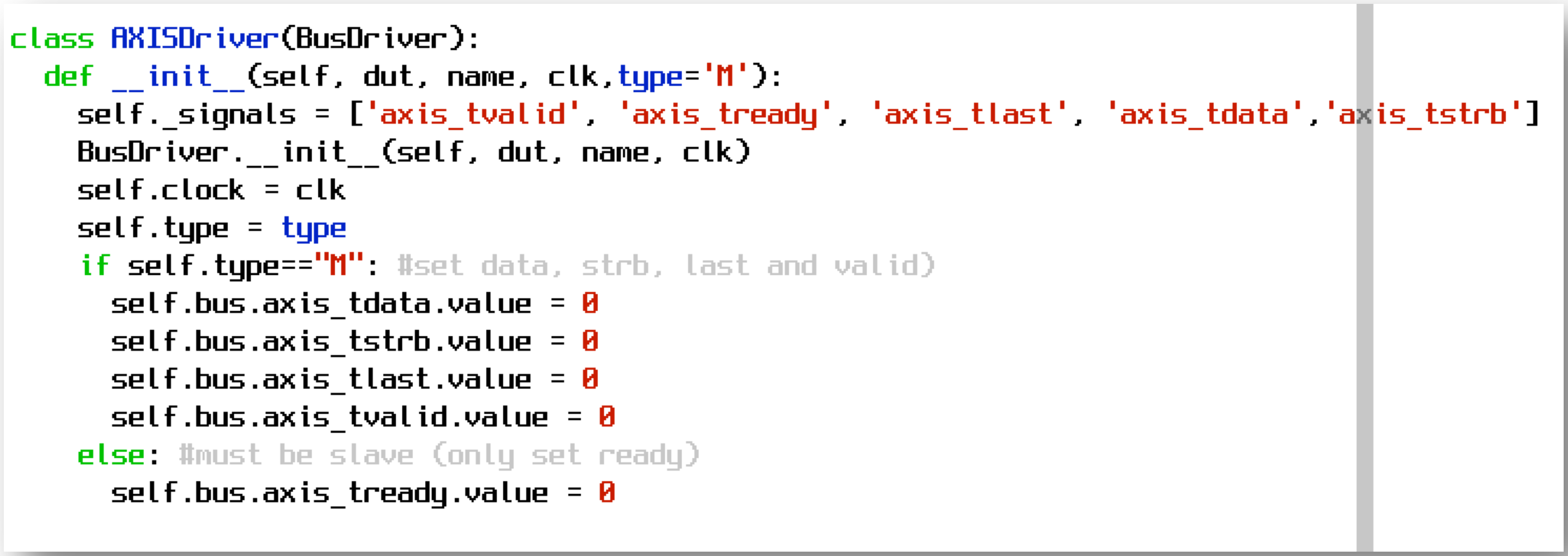


Protocol Tree



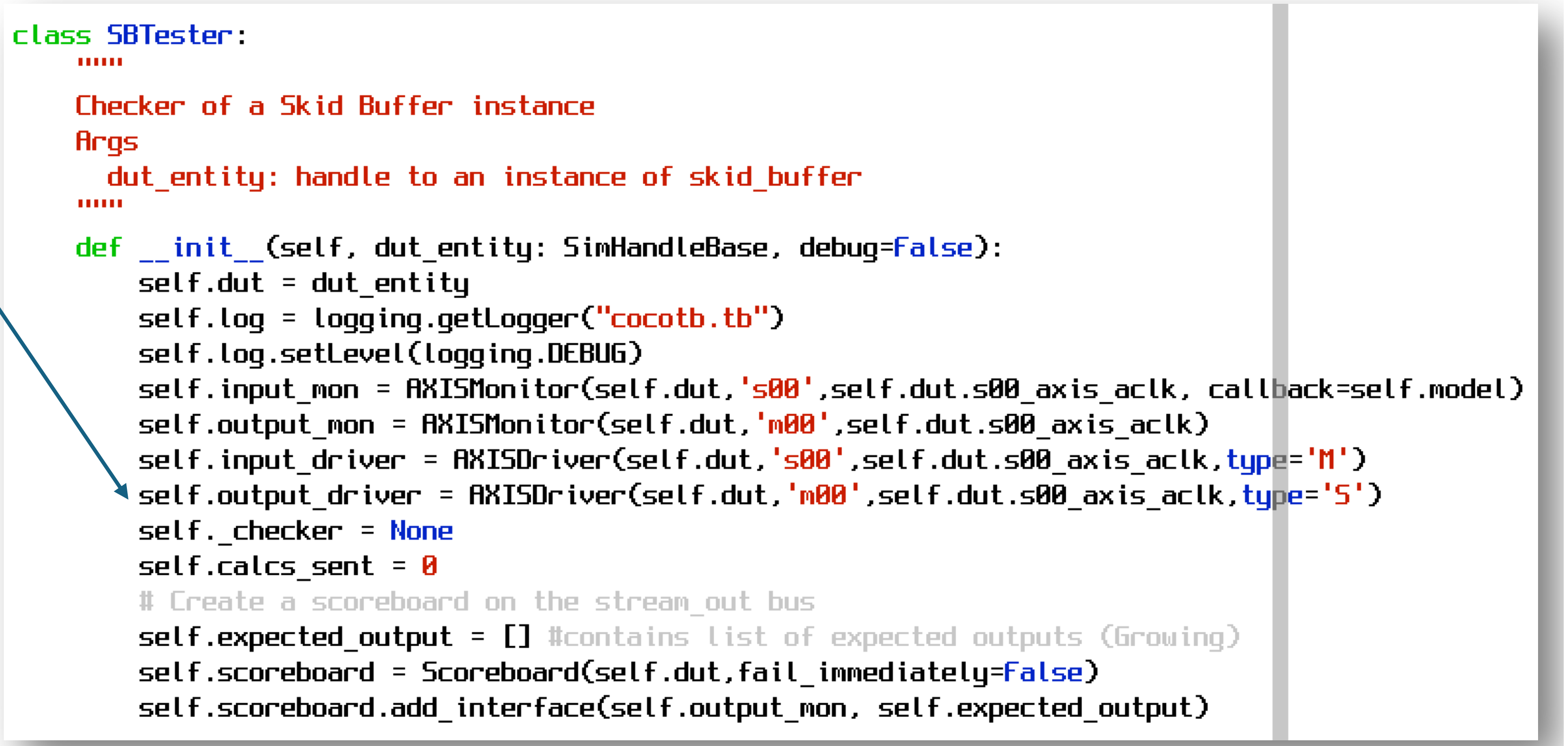
So let’s maybe rewrite our Driver

- Driver can now be for a Master or a Slave



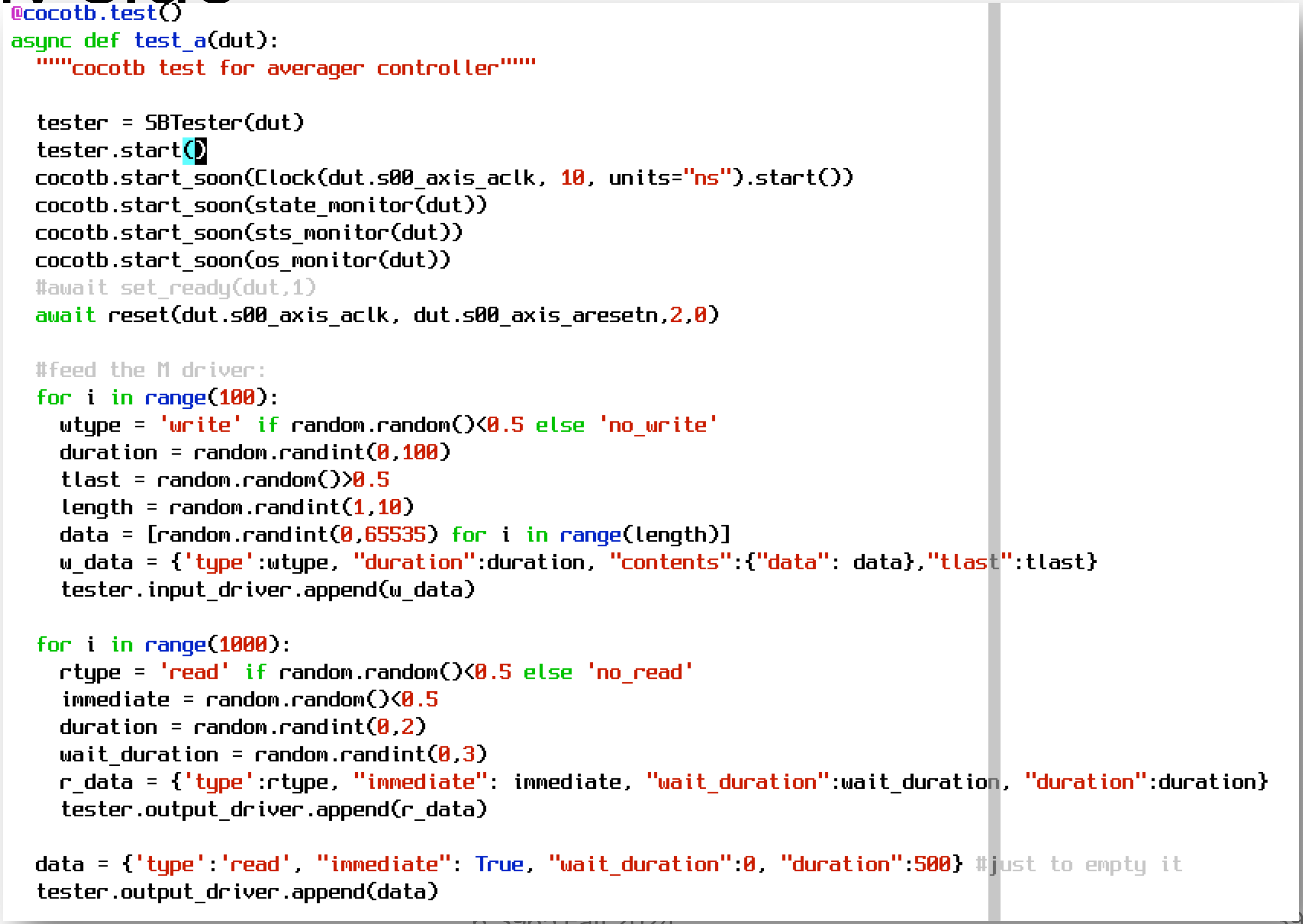
```
class AXISDriver(BusDriver):
  def __init__(self, dut, name, clk,type='M'):
    self._signals = ['axis_tvalid', 'axis_tready', 'axis_tlast', 'axis_tdata','axis_tstrb']
    BusDriver.__init__(self, dut, name, clk)
    self.clock = clk
    self.type = type
    if self.type=="M": #set data, strb, last and valid)
      self.bus.axis_tdata.value = 0
      self.bus.axis_tstrb.value = 0
      self.bus.axis_tlast.value = 0
      self.bus.axis_tvalid.value = 0
    else: #must be slave (only set ready)
      self.bus.axis_tready.value = 0
```

Add an output_driver to our Tester class



```python
class SBTester:
    """
    Checker of a Skid Buffer instance
    Args
      dut_entity: handle to an instance of skid_buffer
    """
    def __init__(self, dut_entity: SimHandleBase, debug=False):
        self.dut = dut_entity
        self.log = logging.getLogger("cocotb.tb")
        self.log.setLevel(logging.DEBUG)
        self.input_mon = AXISMonitor(self.dut,'s00',self.dut.s00_axis_aclk, callback=self.model)
        self.output_mon = AXISMonitor(self.dut,'m00',self.dut.s00_axis_aclk)
        self.input_driver = AXISDriver(self.dut,'s00',self.dut.s00_axis_aclk,type='M')
        self.output_driver = AXISDriver(self.dut,'m00',self.dut.s00_axis_aclk,type='S')
        self._checker = None
        self.calcs_sent = 0
        # Create a scoreboard on the stream_out bus
        self.expected_output = [] #contains list of expected outputs (Growing)
        self.scoreboard = Scoreboard(self.dut,fail_immediately=False)
        self.scoreboard.add_interface(self.output_mon, self.expected_output)
```

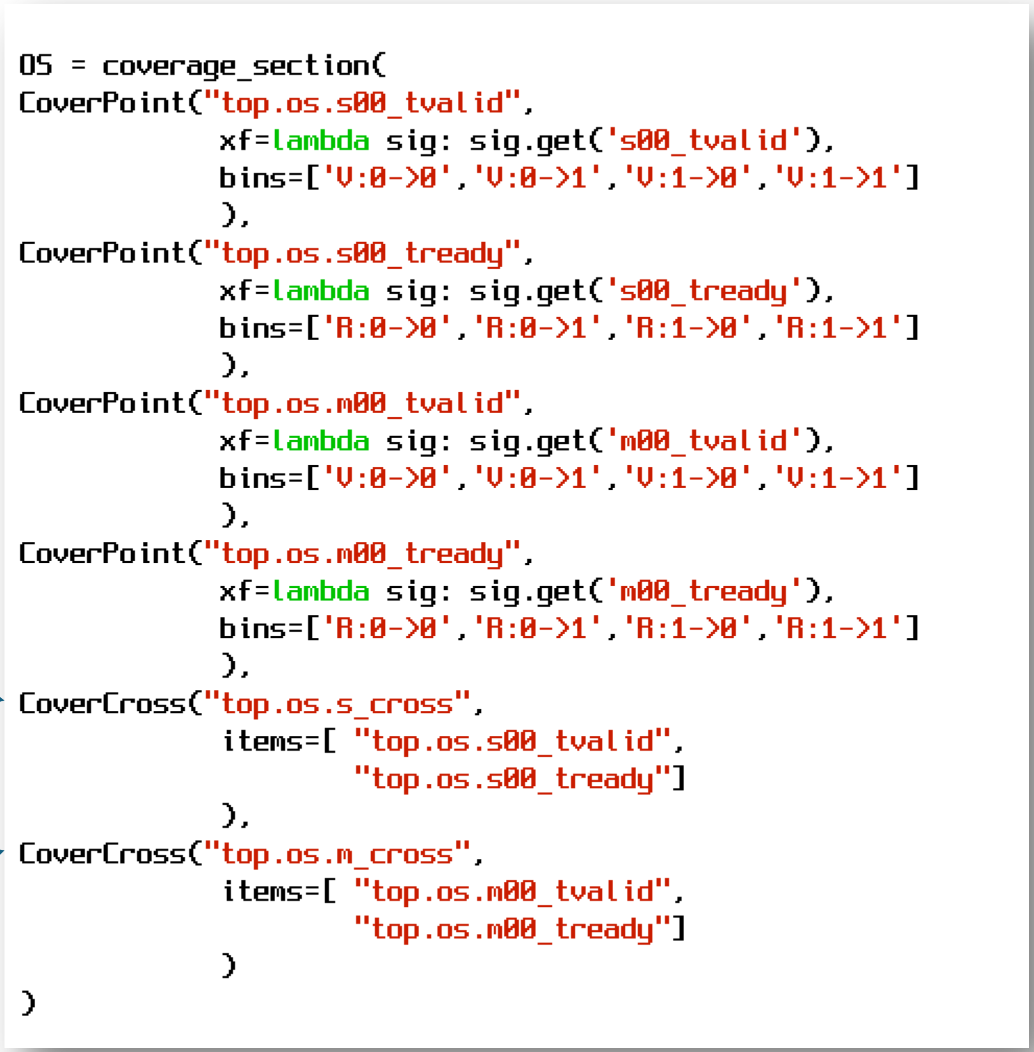
Now feed in random, legal transactions to both the valid and ready side



```
@cocotb.test()
async def test_a(dut):
  """cocotb test for averager controller"""

  tester = SBTester(dut)
  tester.start()
  cocotb.start_soon(Clock(dut.s00_axis_aclk, 10, units="ns").start())
  cocotb.start_soon(state_monitor(dut))
  cocotb.start_soon(sts_monitor(dut))
  cocotb.start_soon(os_monitor(dut))
  #await set_ready(dut,1)
  await reset(dut.s00_axis_aclk, dut.s00_axis_aresetn,2,0)

  #feed the M driver:
  for i in range(100):
    wtype = 'write' if random.random()<0.5 else 'no_write'
    duration = random.randint(0,100)
    tlast = random.random()>0.5
    length = random.randint(1,10)
    data = [random.randint(0,65535) for i in range(length)]
    w_data = {'type':wtype, "duration":duration, "contents":{"data": data},"tlast":tlast}
    tester.input_driver.append(w_data)

  for i in range(1000):
    rtype = 'read' if random.random()<0.5 else 'no_read'
    immediate = random.random()<0.5
    duration = random.randint(0,2)
    wait_duration = random.randint(0,3)
    r_data = {'type':rtype, "immediate": immediate, "wait_duration":wait_duration, "duration":duration}
    tester.output_driver.append(r_data)

  data = {'type':'read', "immediate": True, "wait_duration":0, "duration":500} #just to empty it
  tester.output_driver.append(data)
```

Make a New "higher level" Cover section

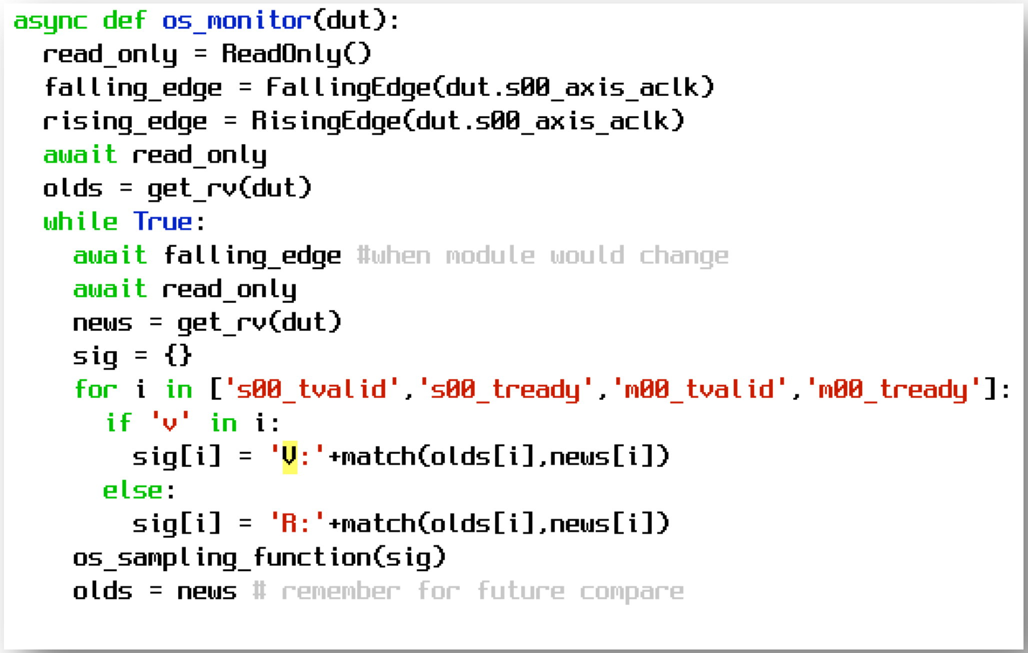
- This one will track cycle-to-cycle transitions of the valid and ready signals on both ports
- No reason to combine the two ports really…there's nothing about the spec anyways



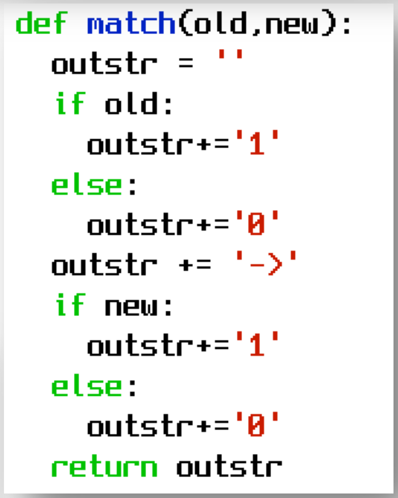
```
OS = coverage_section(
CoverPoint("top.os.s00_tvalid",
          xf=lambda sig: sig.get('s00_tvalid'),
          bins=['V:0->0','V:0->1','V:1->0','V:1->1']
          ),
CoverPoint("top.os.s00_tready",
          xf=lambda sig: sig.get('s00_tready'),
          bins=['R:0->0','R:0->1','R:1->0','R:1->1']
          ),
CoverPoint("top.os.m00_tvalid",
          xf=lambda sig: sig.get('m00_tvalid'),
          bins=['V:0->0','V:0->1','V:1->0','V:1->1']
          ),
CoverPoint("top.os.m00_tready",
          xf=lambda sig: sig.get('m00_tready'),
          bins=['R:0->0','R:0->1','R:1->0','R:1->1']
          ),
CoverCross("top.os.s_cross",
          items=[ "top.os.s00_tvalid",
                 "top.os.s00_tready"]
          ),
CoverCross("top.os.m_cross",
          items=[ "top.os.m00_tvalid",
                 "top.os.m00_tready"]
          )
)
```

Make support functions

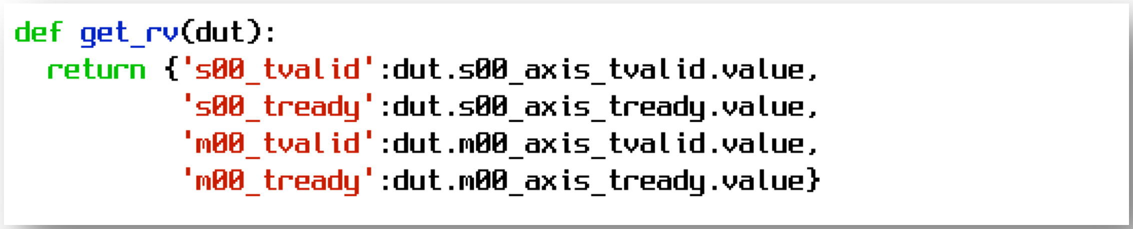
- Track and Label transitions of all four signals over time.



```python
async def os_monitor(dut):
  read_only = ReadOnly()
  falling_edge = FallingEdge(dut.s00_axis_aclk)
  rising_edge = RisingEdge(dut.s00_axis_aclk)
  await read_only
  olds = get_rv(dut)
  while True:
    await falling_edge #when module would change
    await read_only
    news = get_rv(dut)
    sig = {}
    for i in ['s00_tvalid','s00_tready','m00_tvalid','m00_tready']:
      if 'v' in i:
        sig[i] = 'V:'+match(olds[i],news[i])
      else:
        sig[i] = 'R:'+match(olds[i],news[i])
    os_sampling_function(sig)
    olds = news # remember for future compare
```

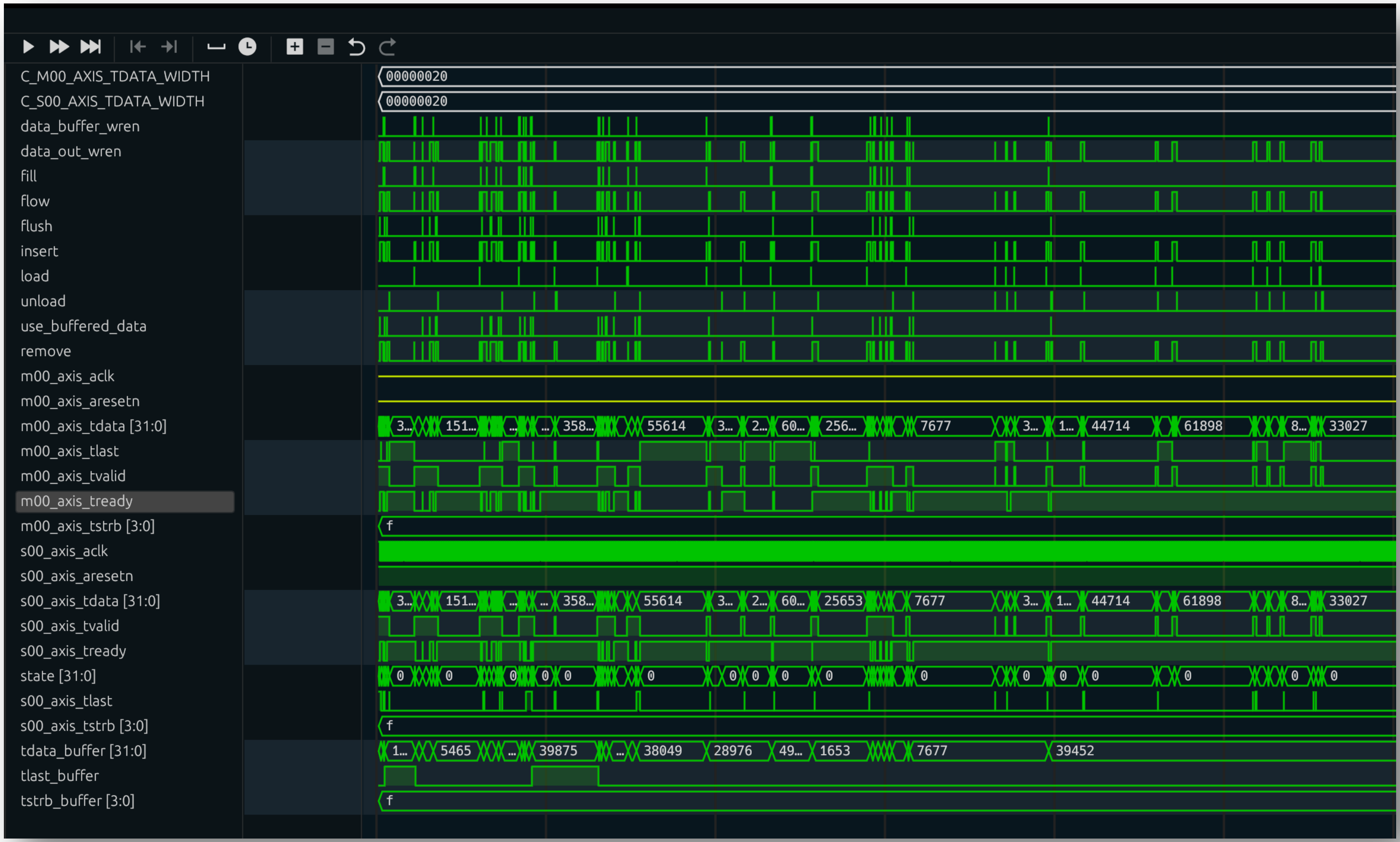


```python
def match(old,new):
  outstr = ''
  if old:
    outstr+='1'
  else:
    outstr+='0'
  outstr += '->'
  if new:
    outstr+='1'
  else:
    outstr+='0'
  return outstr
```

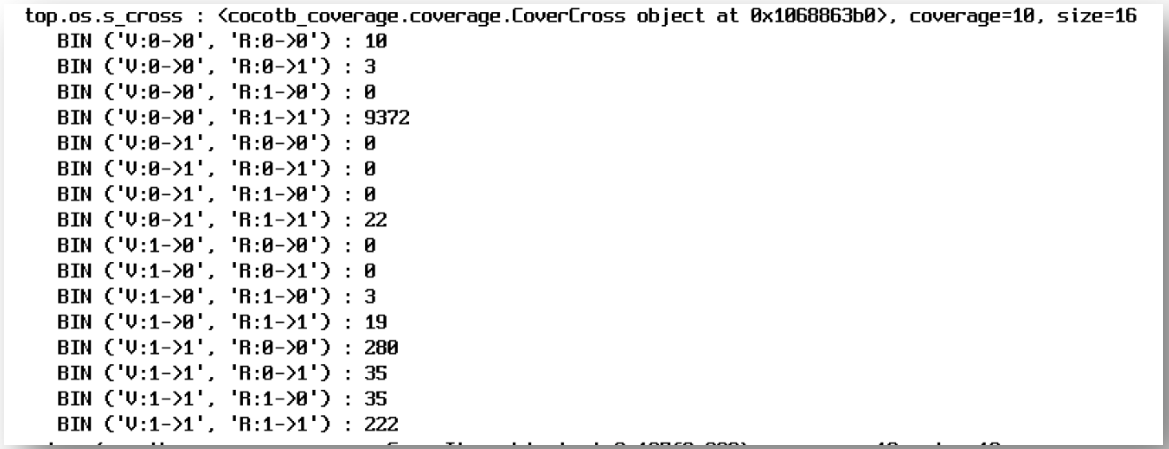


```python
def get_rv(dut):
  return {'s00_tvalid':dut.s00_axis_tvalid.value,
          's00_tready':dut.s00_axis_tready.value,
          'm00_tvalid':dut.m00_axis_tvalid.value,
          'm00_tready':dut.m00_axis_tready.value}
```

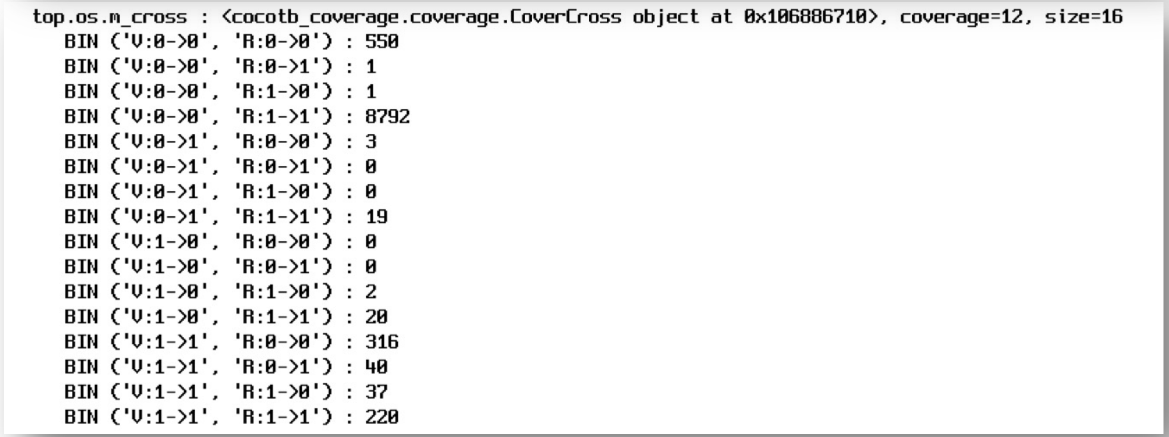
Run it


C_M00_AXIS_TDATA_WIDTH
C_S00_AXIS_TDATA_WIDTH
data_buffer_wren
data_out_wren
fill
flow
flush
insert
load
unload
use_buffered_data
remove
m00_axis_aclk
m00_axis_aresetn
m00_axis_tdata [31:0]
m00_axis_tlast
m00_axis_tvalid
m00_axis_tready
m00_axis_tstrb [3:0]
s00_axis_aclk
s00_axis_aresetn
s00_axis_tdata [31:0]
s00_axis_tvalid
s00_axis_tready
state [31:0]
s00_axis_tlast
s00_axis_tstrb [3:0]
tdata_buffer [31:0]
tlast_buffer
tstrb_buffer [3:0]
00000020
00000020
55614
7677
44714
61898
33027
25653
39875
38049
28976
1653
39452
f

Run it and you get…



```
top.os.s_cross : <cocotb_coverage.coverage.CoverCross object at 0x1068863b0>, coverage=10, size=16
   BIN ('V:0->0', 'R:0->0') : 10
   BIN ('V:0->0', 'R:0->1') : 3
   BIN ('V:0->0', 'R:1->0') : 0
   BIN ('V:0->0', 'R:1->1') : 9372
   BIN ('V:0->1', 'R:0->0') : 0
   BIN ('V:0->1', 'R:0->1') : 0
   BIN ('V:0->1', 'R:1->0') : 0
   BIN ('V:0->1', 'R:1->1') : 22
   BIN ('V:1->0', 'R:0->0') : 0
   BIN ('V:1->0', 'R:0->1') : 0
   BIN ('V:1->0', 'R:1->0') : 3
   BIN ('V:1->0', 'R:1->1') : 19
   BIN ('V:1->1', 'R:0->0') : 280
   BIN ('V:1->1', 'R:0->1') : 35
   BIN ('V:1->1', 'R:1->0') : 35
   BIN ('V:1->1', 'R:1->1') : 222
```

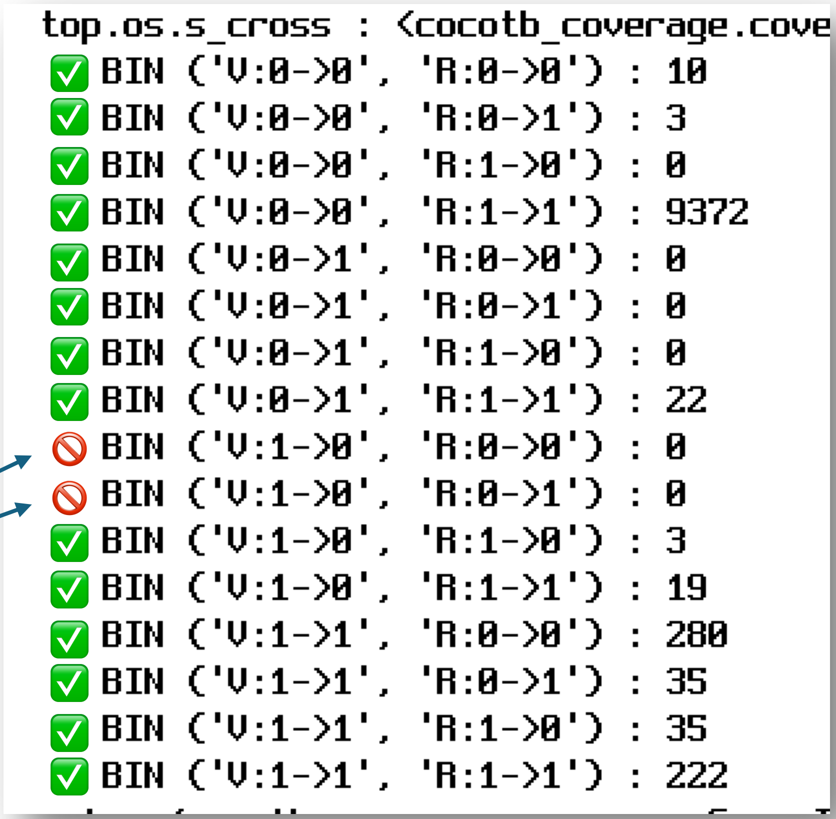


```
top.os.m_cross : <cocotb_coverage.coverage.CoverCross object at 0x106886710>, coverage=12, size=16
   BIN ('V:0->0', 'R:0->0') : 550
   BIN ('V:0->0', 'R:0->1') : 1
   BIN ('V:0->0', 'R:1->0') : 1
   BIN ('V:0->0', 'R:1->1') : 8792
   BIN ('V:0->1', 'R:0->0') : 3
   BIN ('V:0->1', 'R:0->1') : 0
   BIN ('V:0->1', 'R:1->0') : 0
   BIN ('V:0->1', 'R:1->1') : 19
   BIN ('V:1->0', 'R:0->0') : 0
   BIN ('V:1->0', 'R:0->1') : 0
   BIN ('V:1->0', 'R:1->0') : 2
   BIN ('V:1->0', 'R:1->1') : 20
   BIN ('V:1->1', 'R:0->0') : 316
   BIN ('V:1->1', 'R:0->1') : 40
   BIN ('V:1->1', 'R:1->0') : 37
   BIN ('V:1->1', 'R:1->1') : 220
```

Let’s Consider Slave Side

Legit/Might Occur:✅

Should Not Occur:🚫



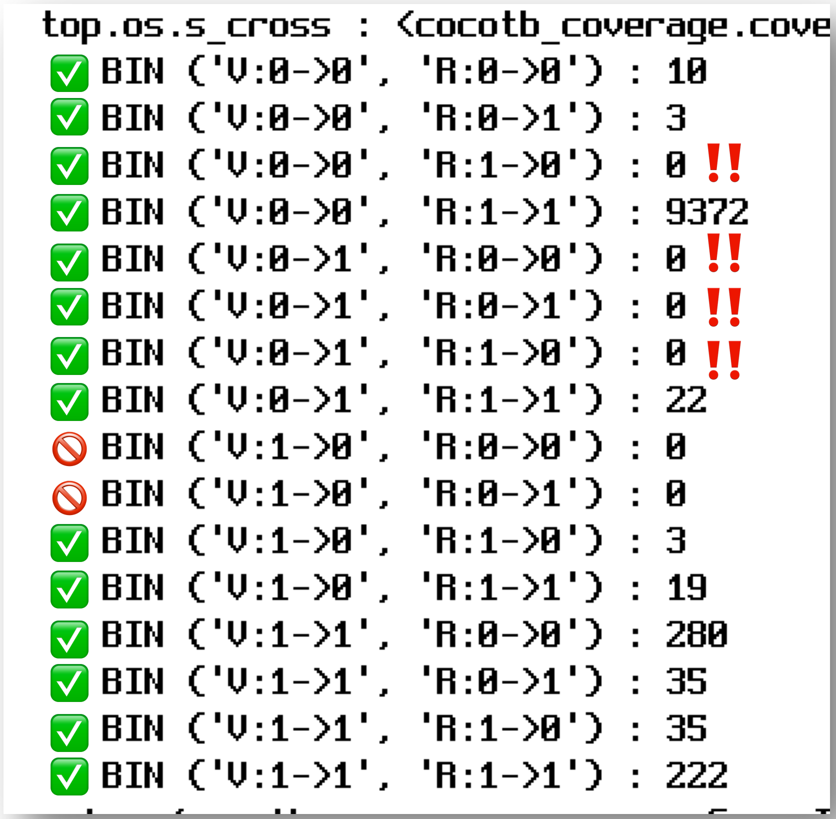
```
top.os.s_cross : <cocotb_coverage.cove
✅ BIN ('V:0->0', 'R:0->0') : 10
✅ BIN ('V:0->0', 'R:0->1') : 3
✅ BIN ('V:0->0', 'R:1->0') : 0
✅ BIN ('V:0->0', 'R:1->1') : 9372
✅ BIN ('V:0->1', 'R:0->0') : 0
✅ BIN ('V:0->1', 'R:0->1') : 0
✅ BIN ('V:0->1', 'R:1->0') : 0
✅ BIN ('V:0->1', 'R:1->1') : 22
🚫 BIN ('V:1->0', 'R:0->0') : 0
🚫 BIN ('V:1->0', 'R:0->1') : 0
✅ BIN ('V:1->0', 'R:1->0') : 3
✅ BIN ('V:1->0', 'R:1->1') : 19
✅ BIN ('V:1->1', 'R:0->0') : 280
✅ BIN ('V:1->1', 'R:0->1') : 35
✅ BIN ('V:1->1', 'R:1->0') : 35
✅ BIN ('V:1->1', 'R:1->1') : 222
```

Both these are situations where the Valid is de-asserting before a handshake occurred

So what should we be concerned about?

Legit/Might Occur:✅

Should Not Occur:🚫



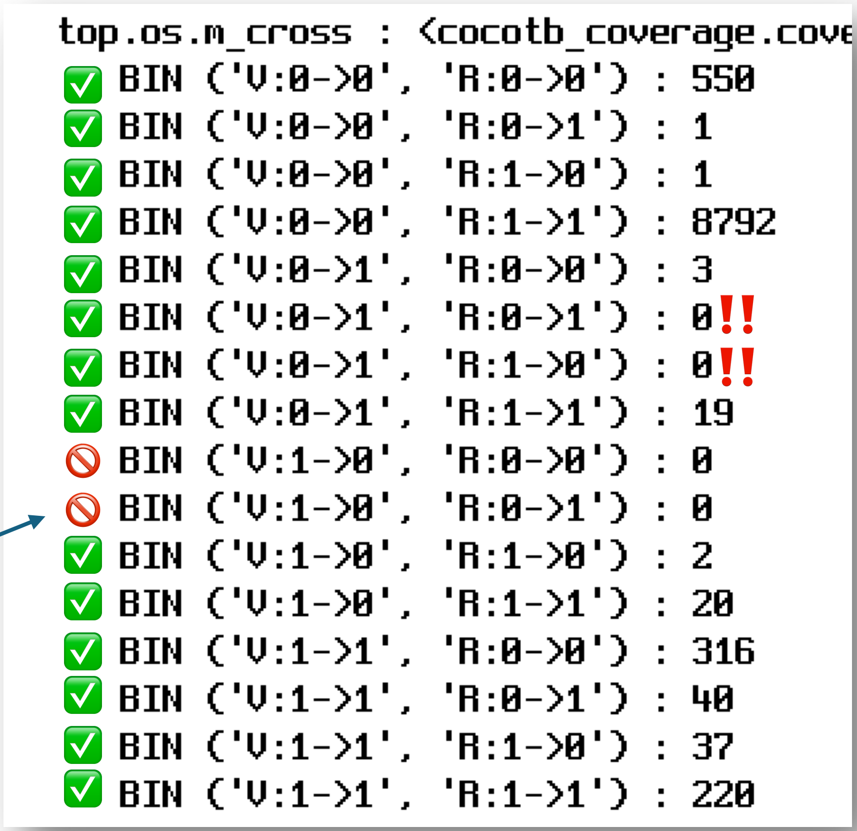
```
top.os.s_cross : <cocotb_coverage.cove
✅ BIN ('V:0->0', 'R:0->0') : 10
✅ BIN ('V:0->0', 'R:0->1') : 3
✅ BIN ('V:0->0', 'R:1->0') : 0 ‼
✅ BIN ('V:0->0', 'R:1->1') : 9372
✅ BIN ('V:0->1', 'R:0->0') : 0 ‼
✅ BIN ('V:0->1', 'R:0->1') : 0 ‼
✅ BIN ('V:0->1', 'R:1->0') : 0 ‼
✅ BIN ('V:0->1', 'R:1->1') : 22
🚫 BIN ('V:1->0', 'R:0->0') : 0
🚫 BIN ('V:1->0', 'R:0->1') : 0
✅ BIN ('V:1->0', 'R:1->0') : 3
✅ BIN ('V:1->0', 'R:1->1') : 19
✅ BIN ('V:1->1', 'R:0->0') : 280
✅ BIN ('V:1->1', 'R:0->1') : 35
✅ BIN ('V:1->1', 'R:1->0') : 35
✅ BIN ('V:1->1', 'R:1->1') : 222
```

Similarly on Master Side:

Legit/Might Occur:✅

Should Not Occur:🚫

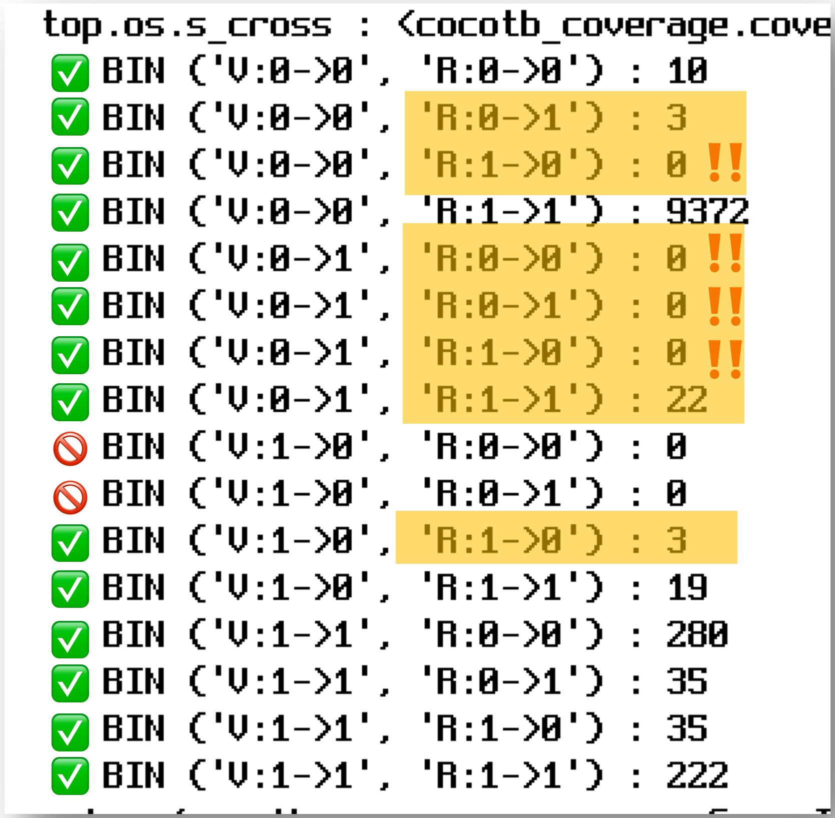
This is actually pretty reassuring since our DUT would be the device that would actually be causing these violations



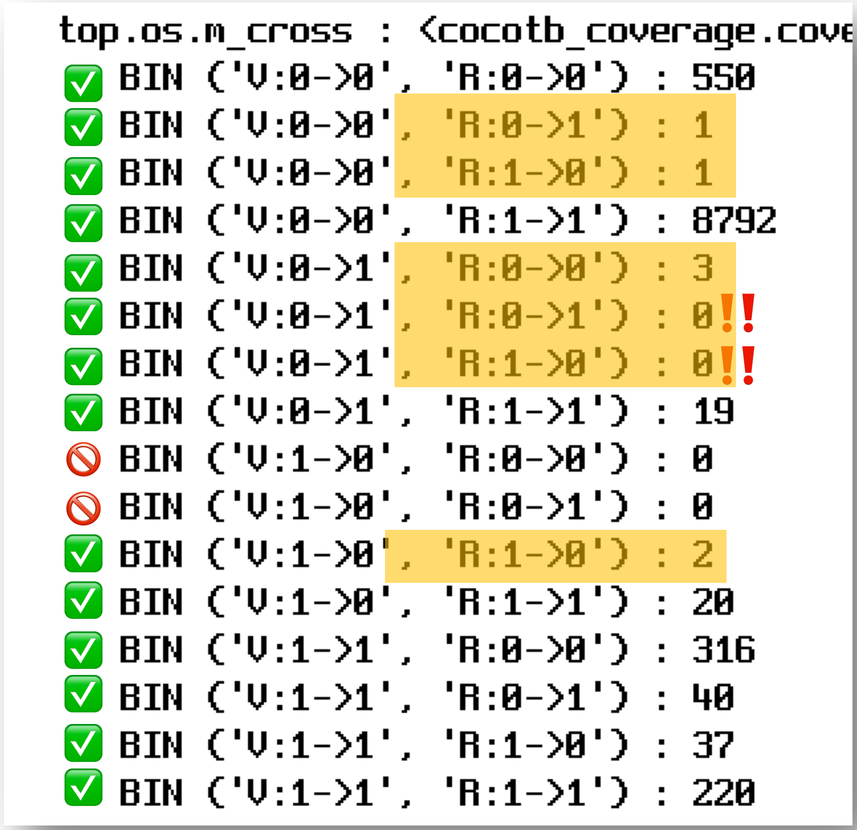
```
top.os.m_cross : <cocotb_coverage.cove
✅ BIN ('V:0->0', 'R:0->0') : 550
✅ BIN ('V:0->0', 'R:0->1') : 1
✅ BIN ('V:0->0', 'R:1->0') : 1
✅ BIN ('V:0->0', 'R:1->1') : 8792
✅ BIN ('V:0->1', 'R:0->0') : 3
✅ BIN ('V:0->1', 'R:0->1') : 0‼
✅ BIN ('V:0->1', 'R:1->0') : 0‼
✅ BIN ('V:0->1', 'R:1->1') : 19
🚫 BIN ('V:1->0', 'R:0->0') : 0
🚫 BIN ('V:1->0', 'R:0->1') : 0
✅ BIN ('V:1->0', 'R:1->0') : 2
✅ BIN ('V:1->0', 'R:1->1') : 20
✅ BIN ('V:1->1', 'R:0->0') : 316
✅ BIN ('V:1->1', 'R:0->1') : 40
✅ BIN ('V:1->1', 'R:1->0') : 37
✅ BIN ('V:1->1', 'R:1->1') : 220
```

Conclusions?

So probably more read toggling in our testbench would be good to be honest.



```
top.os.s_cross : <cocotb_coverage.cove
✅ BIN ('V:0->0', 'R:0->0') : 10
✅ BIN ('V:0->0', 'R:0->1') : 3
✅ BIN ('V:0->0', 'R:1->0') : 0 ‼
✅ BIN ('V:0->0', 'R:1->1') : 9372
✅ BIN ('V:0->1', 'R:0->0') : 0 ‼
✅ BIN ('V:0->1', 'R:0->1') : 0 ‼
✅ BIN ('V:0->1', 'R:1->0') : 0 ‼
✅ BIN ('V:0->1', 'R:1->1') : 22
🚫 BIN ('V:1->0', 'R:0->0') : 0
🚫 BIN ('V:1->0', 'R:0->1') : 0
✅ BIN ('V:1->0', 'R:1->0') : 3
✅ BIN ('V:1->0', 'R:1->1') : 19
✅ BIN ('V:1->1', 'R:0->0') : 280
✅ BIN ('V:1->1', 'R:0->1') : 35
✅ BIN ('V:1->1', 'R:1->0') : 35
✅ BIN ('V:1->1', 'R:1->1') : 222
```



```
top.os.m_cross : <cocotb_coverage.cove
✅ BIN ('V:0->0', 'R:0->0') : 550
✅ BIN ('V:0->0', 'R:0->1') : 1
✅ BIN ('V:0->0', 'R:1->0') : 1
✅ BIN ('V:0->0', 'R:1->1') : 8792
✅ BIN ('V:0->1', 'R:0->0') : 3
✅ BIN ('V:0->1', 'R:0->1') : 0‼
✅ BIN ('V:0->1', 'R:1->0') : 0‼
✅ BIN ('V:0->1', 'R:1->1') : 19
🚫 BIN ('V:1->0', 'R:0->0') : 0
🚫 BIN ('V:1->0', 'R:0->1') : 0
✅ BIN ('V:1->0', 'R:1->0') : 2
✅ BIN ('V:1->0', 'R:1->1') : 20
✅ BIN ('V:1->1', 'R:0->0') : 316
✅ BIN ('V:1->1', 'R:0->1') : 40
✅ BIN ('V:1->1', 'R:1->0') : 37
✅ BIN ('V:1->1', 'R:1->1') : 220
```