6.S965
Digital Systems Laboratory II

Lecture 12

Administrative Stuff

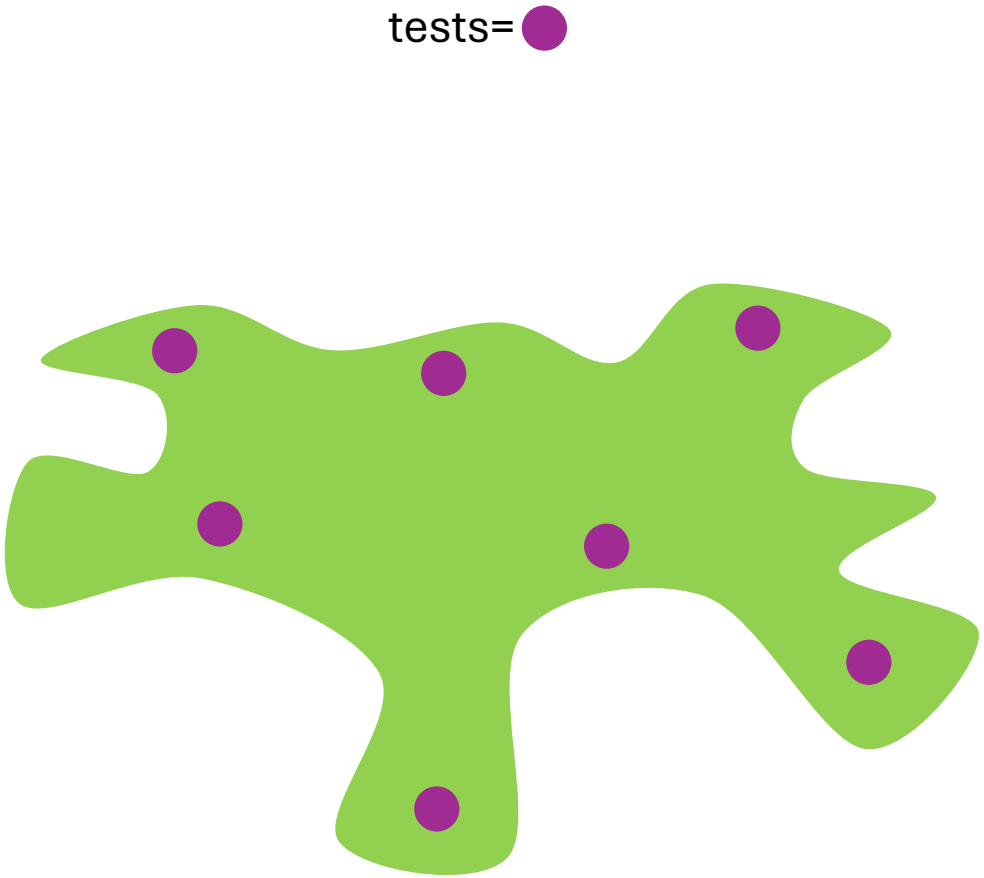
- Last week of stuff (week 6) content was released Friday
- Try to do by the end of the week.
- Meant to have project survey out by last weekend but I fell behind. Plan:
  - If you have a team of 2/3 you want to work with, email me the team.
  - If you want to get partnered, email me with some ideas of what you want.

Coverage

What is it? What does it mean?

Coverage

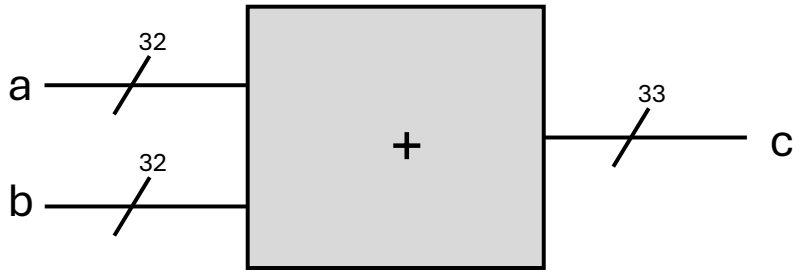
- Is concerned with how much have you tested a DUT



Some n-dimensional blob of possible module existence

The issue…

- Consider a device that adds two 32 bit numbers.



- There are 1.84×10^{19} input possibilities, each with a correct output.
- If you verified 1 billion input/output combinations per second it would take ~600 years to fully verify the design
- And this is just a simple adder…

And this gets astronomically worse as modules get more complicated

- …especially as they get more stateful
- …and with more inputs
- …and with multiple sets of ports and things

Can anything ever be fully covered?

- Some modules should be able to be almost fully covered
- Others maybe not, so you have to structure what you're looking for and zero in on important edge cases like:
  - Max/min values, edge cases, overflow cases,

What do you ”cover”?

- If a module has clearly defined states, you should check to see those
- Maybe check to see how those states transition?
- Maybe check to see different sequences of input and/or output signals
- Check certain output signals against input signals
- Check sequences of inputs
- Check combinations of things listed above.

Coverage is not necessarily about the verification of correct results

- I mean it is an adjacent topic…

- But really the notion of coverage is meant to say *how* much was tested…with the assumption that it tested correctly.

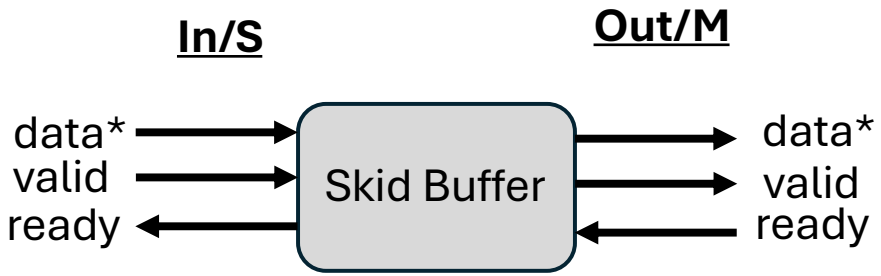
- It is also about exploring what/where your design can get to and can’t get to.

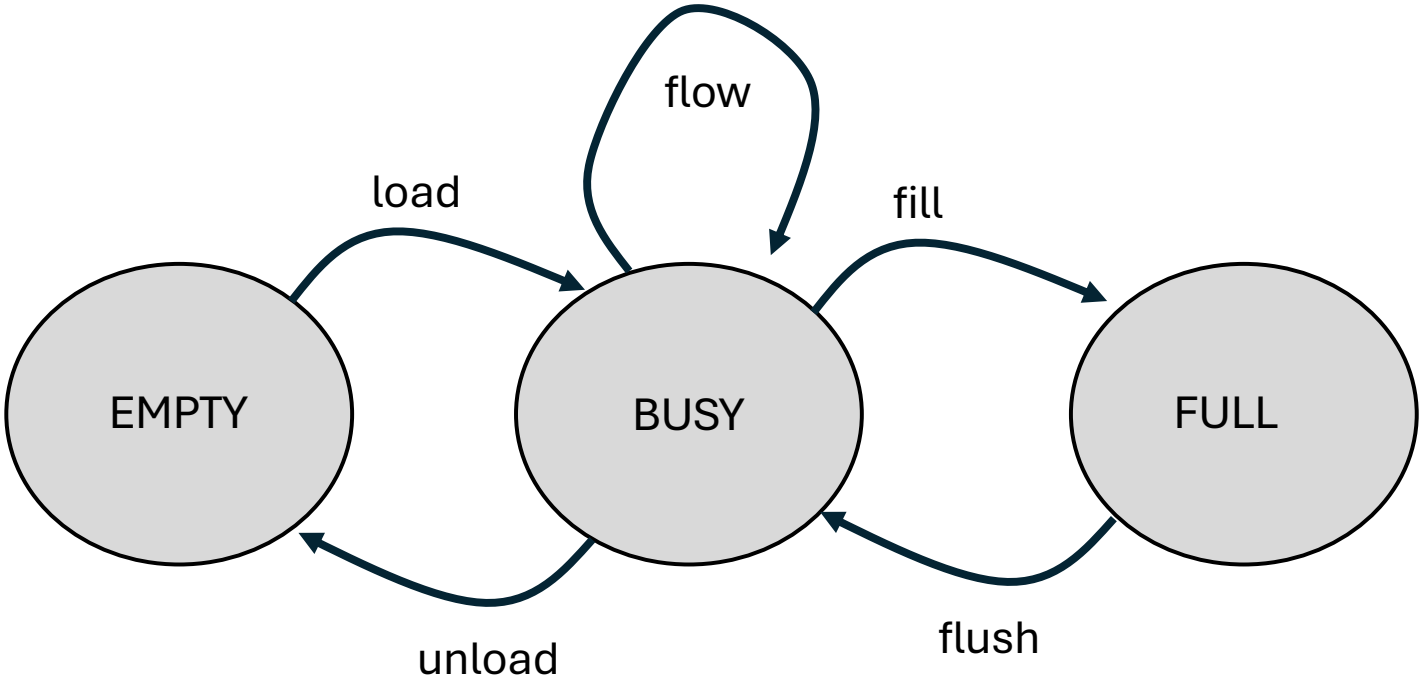
So let’s look at an example…

- We’ll revisit the issue of TREADY propagation and build a module to handle that properly.

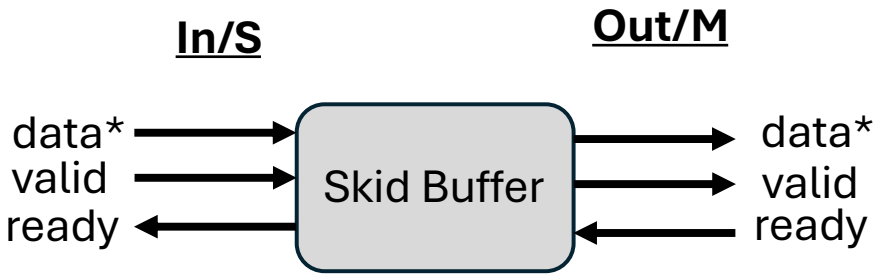
- The skid buffer fixes this…most of you are working on this right now.

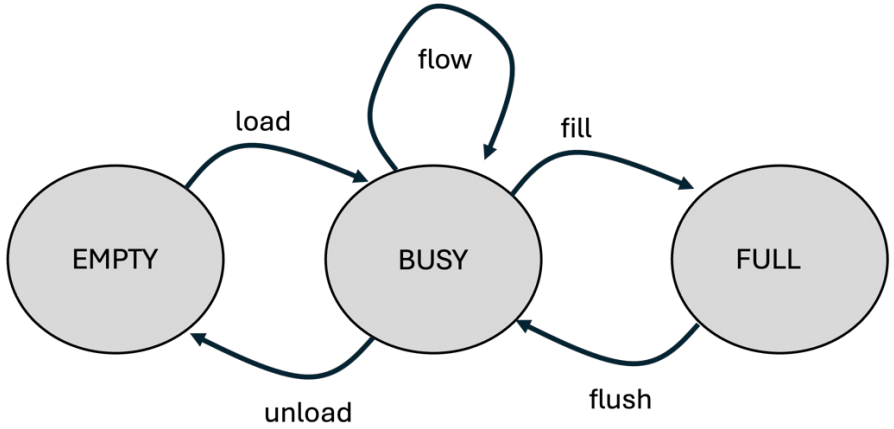
Example: Skid Buffer





Skid Buffer





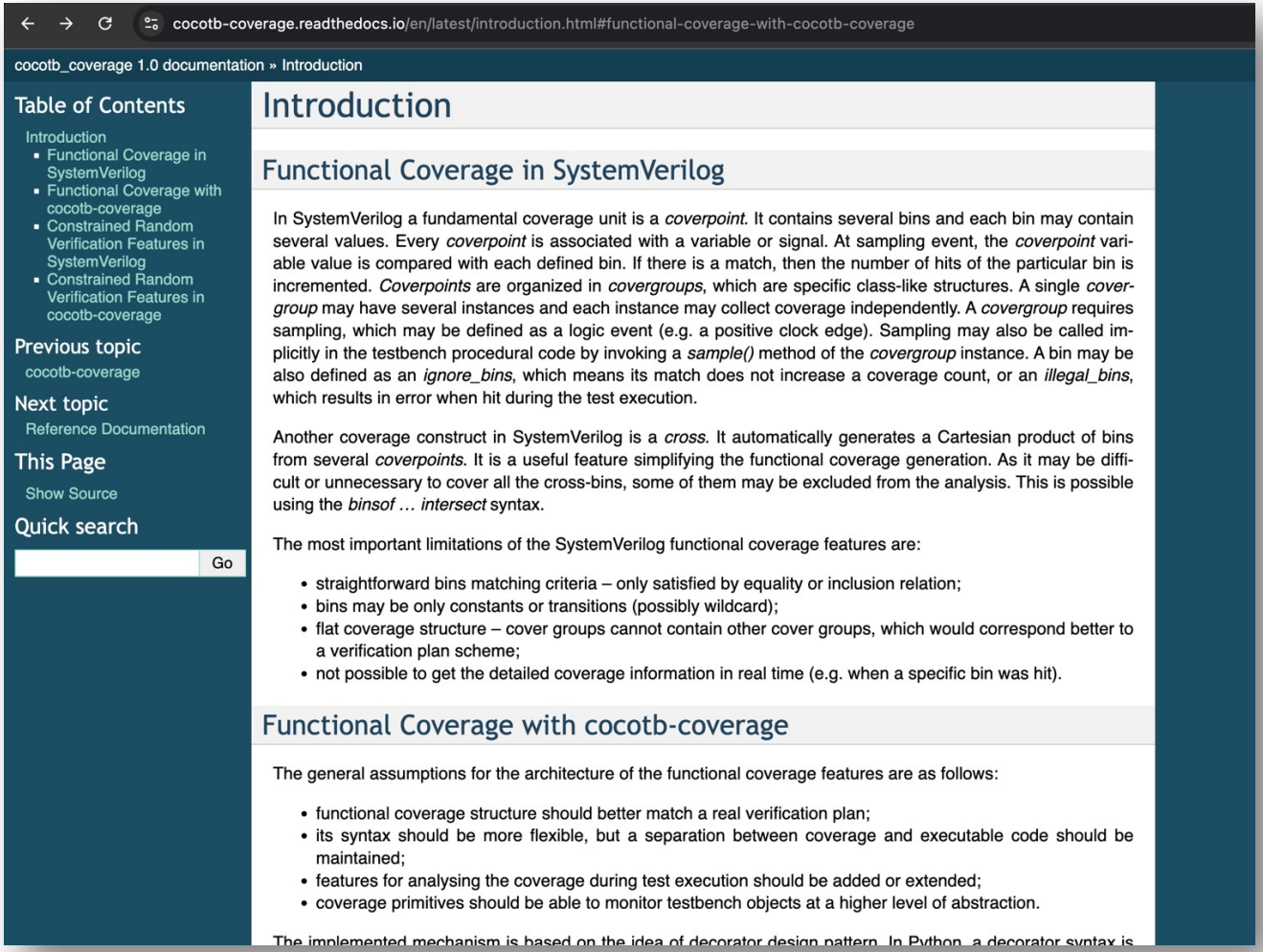
- BUSY is normal operation where data is coming in and out.
- If there's a hiccup on the output side, go to FULL and stall pipeline (s00_tready → 0)
- If there's a hiccup on the input side, go to EMPTY and stall pipeline (m00_tvalid → 0)

Skid Buffer

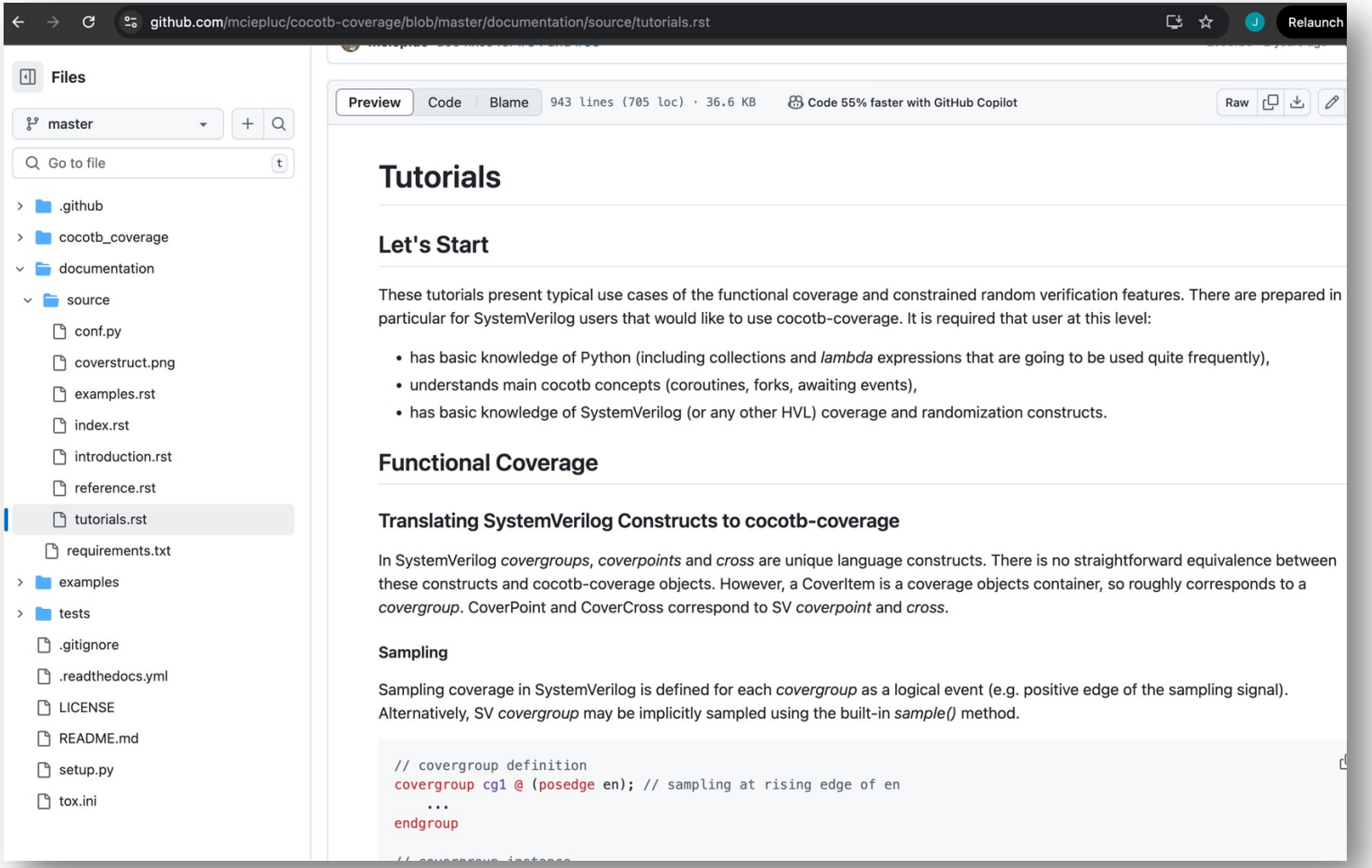
This simple FSM description…glossed over the potential complexity of the implementation: 3 states, each connected to 2 signals (valid/ready) per interface, for a total of 16 possible transitions out of each state, or 48 possible state transitions total.

Cocotb Coverage (version 1.2)

Install only version 1.2
They updated to support cocotb v 2.0



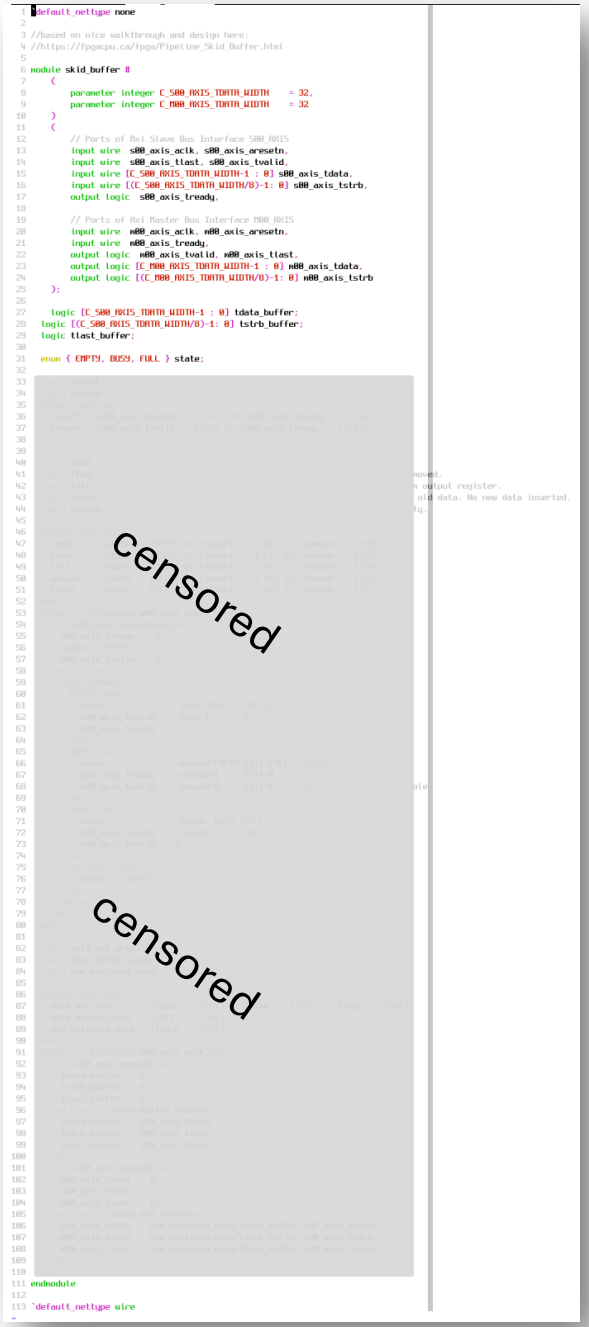
Another library with ok docs and source code



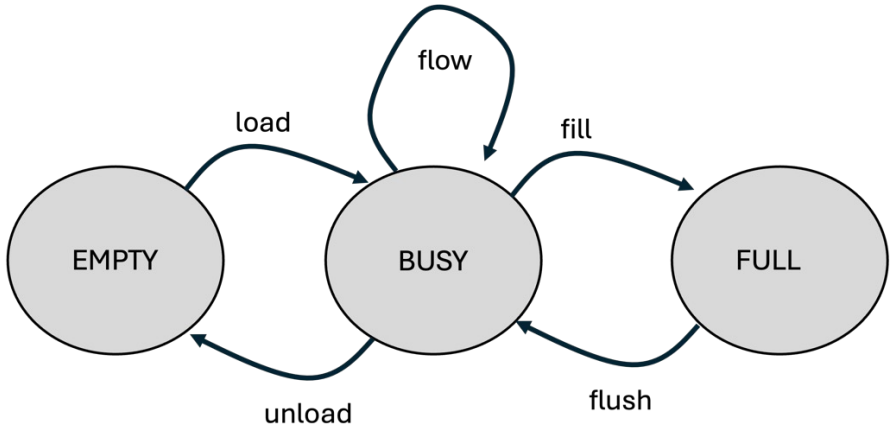
My Skid Buffer

- Most is hidden from you, but one thing to point out is there is an internal state variable.

```
enum { EMPTY, BUSY, FULL } state;
```



Cocotb_coverage




```
from cocotb_coverage.coverage import CoverCross, CoverPoint, coverage_db, coverage_section
import constraint
```

- Let's first focus on how we could measure the states that our system exists in?
- This thing has a very clearly defined state machine design and only certain states will connect to certain states

First step is to define some coverage that we care about

- Let's look at current state of our FSM and next/upcoming state of our FSM

```
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"],

            )
)
```

CoverPoint

```
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"],

            )
)
```

- Object thar represents coverage. Concerned with a signal or combination of signals or state of being.
- Has a name (which you organize in a hierarchical fashion)
- Is used with a function you define
- Qualifies the inputs as one of the values specified in its `bins` argument

CoverCross

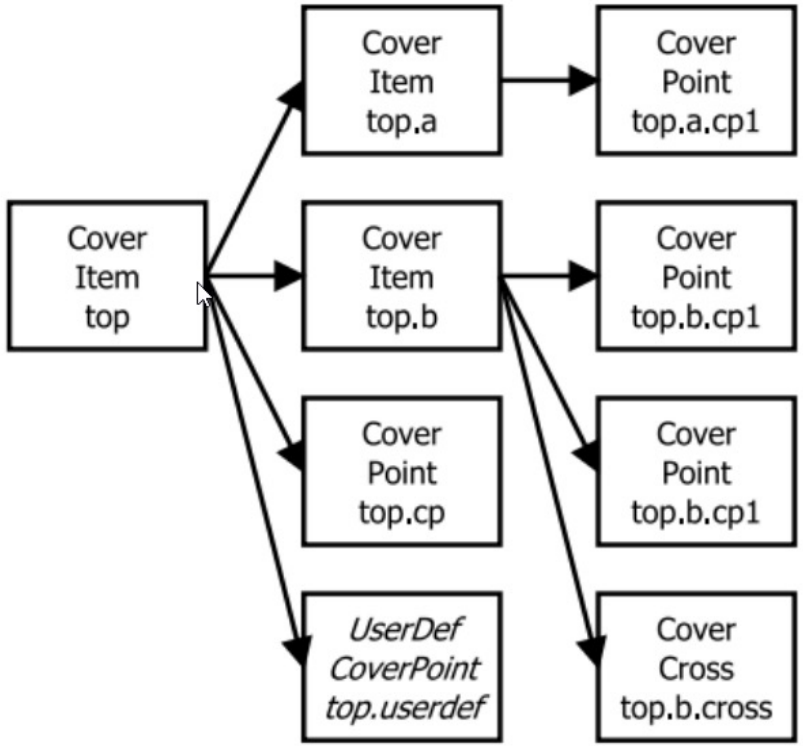
```
SC = coverage_section (
CoverPoint("top.st.state",
            xf=lambda s, ns: s,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverPoint("top.st.next_state",
            xf=lambda s, ns: ns,
            bins=['EMPTY', 'BUSY', 'FULL']
            ),
CoverCross("top.st.state.cross",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

            )
)
```

- CoverCross generates the Cartesian Product of multiple CoverPoints
- The CoverCross shown here will have how many possible bins?

Coverage_section

- Is another object that represents a collection of coverpoints (and any related crosses)

- The idea is to hierarchically organize the things you care about



Must Sample/interface with the actual DUT

- Write a sampling function (just a passthrough here)
- That is then called repeatedly in a monitor that is studying the state/next state on the rising clock edge

Decorator links to coverage_section by name…this is the function that is used by the cover points for analysis

```
@SC
def sampling_function(s,ns):
  pass

async def state_monitor(dut):
  states = {0:'EMPTY', 1:'BUSY', 2:'FULL'}
  read_only = ReadOnly() #This is
  falling_edge = FallingEdge(dut.s00_axis_aclk)
  rising_edge = RisingEdge(dut.s00_axis_aclk)
  await read_only
  old_state = dut.state.value
  while True:
    await rising_edge #when module would change
    await read_only
    state = dut.state.value
    sampling_function(states[old_state], states[state])
    old_state = state
```

Then run…

- Launch state monitor here:

```
359 @cocotb.test()
360 async def test_a(dut):
361     """cocotb test for averager controller"""
362     inm = AXIS_Monitor(dut,'s00',dut.s00_axis_aclk,callback=j_math_model)
363     outm = AXIS_Monitor(dut,'m00',dut.s00_axis_aclk,callback=lambda x: sig_out_a
364     ind = M_AXIS_Driver(dut,'s00',dut.s00_axis_aclk) #M driver for S port
365     outd = S_AXIS_Driver(dut,'m00',dut.s00_axis_aclk) #S driver for M port
366
367     # Create a scoreboard on the stream_out bus
368     scoreboard = Scoreboard(dut,fail_immediately=False)
369     scoreboard.add_interface(outm, sig_out_exp)
370     cocotb.start_soon(Clock(dut.s00_axis_aclk, 10, units="ns").start())
371
372     cocotb.start_soon(state_monitor(dut))
```

- At end of test…report it out using coverage_db.report_coverage

```
390     coverage_db.report_coverage(cocotb.log.info, bins=True)
391     coverage_file = os.path.join(os.getenv('sim_result', "./"), 'coverage.xml')
392     coverage_db.export_to_xml(filename=coverage_file)
393     assert inm.transactions==outm.transactions, f"Transaction Count doesn't match! :/"
```

The result

```
top : <cocotb_coverage.coverage.CoverItem object at 0x107301d10>, coverage=13, size=15
   top.st : <cocotb_coverage.coverage.CoverItem object at 0x10725e900>, coverage=13, size=15
      top.st.next_state : <cocotb_coverage.coverage.CoverPoint object at 0x107301e50>, coverage=3, size=3
         BIN EMPTY : 207
         BIN BUSY : 162
         BIN FULL : 131
      top.st.state : <cocotb_coverage.coverage.CoverPoint object at 0x10725e7b0>, coverage=12, size=12
         BIN EMPTY : 207
         BIN BUSY : 162
         BIN FULL : 131
         top.st.state.cross : <cocotb_coverage.coverage.CoverCross object at 0x10725ea50>, coverage=7, size=9
            BIN ('EMPTY', 'EMPTY') : 158
            BIN ('EMPTY', 'BUSY') : 49
            BIN ('EMPTY', 'FULL') : 0
            BIN ('BUSY', 'EMPTY') : 49
            BIN ('BUSY', 'BUSY') : 90
            BIN ('BUSY', 'FULL') : 23
            BIN ('FULL', 'EMPTY') : 0
            BIN ('FULL', 'BUSY') : 23
            BIN ('FULL', 'FULL') : 108
test_a passed
**************************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************************
** test_skid_buffer.test_a        PASS        5010.00           0.05     101040.37  **
**************************************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0               5010.00           0.09      56264.42  **
**************************************************************************************************
```

Or if you prefer to read xml

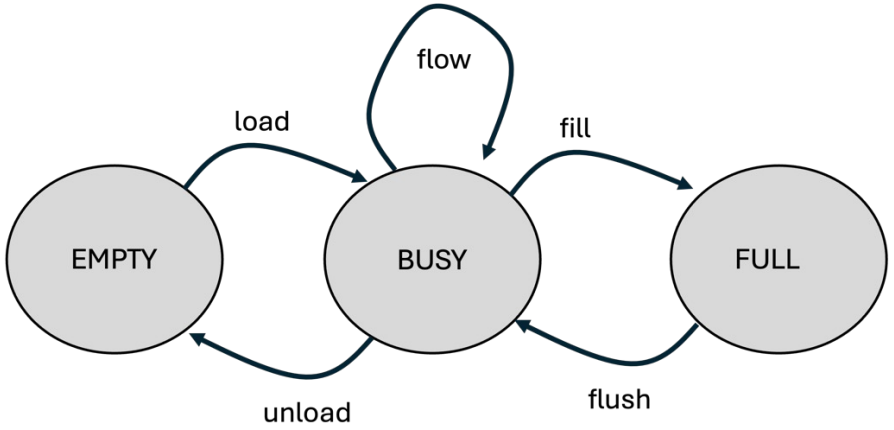
- I guess

This XML file does not appear to have any style information associated with it. The document tree is shown below.


```
▼<top abs_name="top" size="15" coverage="13" cover_percentage="86.67">
  ▼<st size="15" coverage="13" cover_percentage="86.67" abs_name="top.st">
    ▼<state size="12" coverage="12" cover_percentage="100.0" abs_name="top.st.state" weight="1" at_least="1">
       <bin0 bin="EMPTY" hits="207" abs_name="top.st.state.bin0"/>
       <bin1 bin="BUSY" hits="162" abs_name="top.st.state.bin1"/>
       <bin2 bin="FULL" hits="131" abs_name="top.st.state.bin2"/>
      ▼<cross size="9" coverage="7" cover_percentage="77.78" abs_name="top.st.state.cross" weight="1" at_least="1">
         <bin0 bin="('EMPTY', 'EMPTY')" hits="158" abs_name="top.st.state.cross.bin0"/>
         <bin1 bin="('EMPTY', 'BUSY')" hits="49" abs_name="top.st.state.cross.bin1"/>
         <bin2 bin="('EMPTY', 'FULL')" hits="0" abs_name="top.st.state.cross.bin2"/>
         <bin3 bin="('BUSY', 'EMPTY')" hits="49" abs_name="top.st.state.cross.bin3"/>
         <bin4 bin="('BUSY', 'BUSY')" hits="90" abs_name="top.st.state.cross.bin4"/>
         <bin5 bin="('BUSY', 'FULL')" hits="23" abs_name="top.st.state.cross.bin5"/>
         <bin6 bin="('FULL', 'EMPTY')" hits="0" abs_name="top.st.state.cross.bin6"/>
         <bin7 bin="('FULL', 'BUSY')" hits="23" abs_name="top.st.state.cross.bin7"/>
         <bin8 bin="('FULL', 'FULL')" hits="108" abs_name="top.st.state.cross.bin8"/>
       </cross>
     </state>
    ▼<next_state size="3" coverage="3" cover_percentage="100.0" abs_name="top.st.next_state" weight="1" at_least="1">
       <bin0 bin="EMPTY" hits="207" abs_name="top.st.next_state.bin0"/>
       <bin1 bin="BUSY" hits="162" abs_name="top.st.next_state.bin1"/>
       <bin2 bin="FULL" hits="131" abs_name="top.st.next_state.bin2"/>
     </next_state>
   </st>
 </top>
```

Results

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

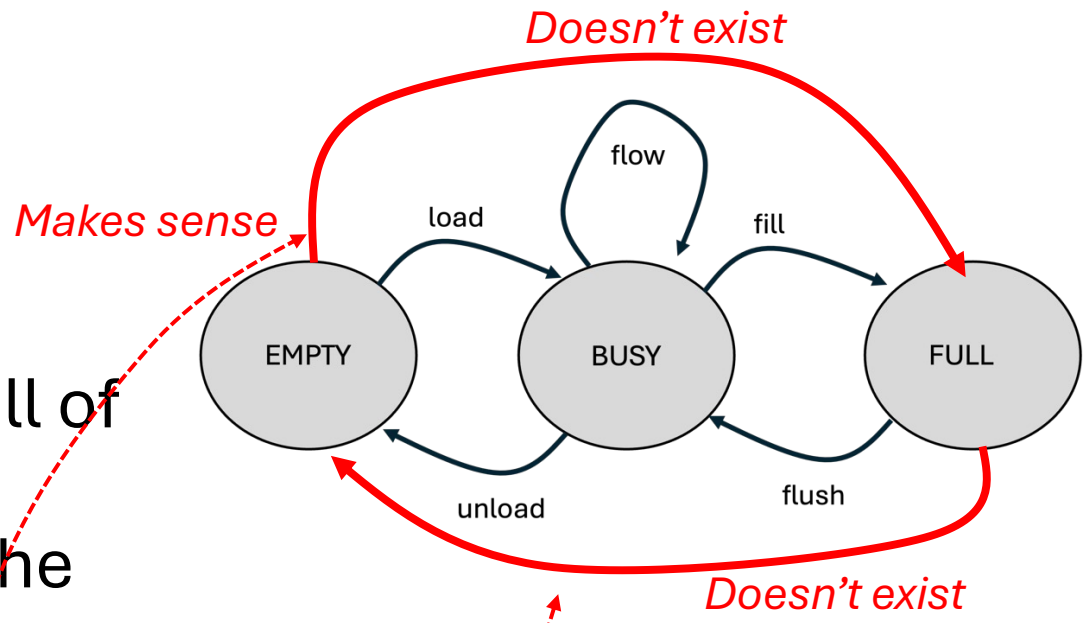



```
top : <cocotb_coverage.coverage.CoverItem object at 0x1029911e0>, coverage=13, size=15
   top.st : <cocotb_coverage.coverage.CoverItem object at 0x10213d3c0>, coverage=13, size=15
      top.st.next_state : <cocotb_coverage.coverage.CoverPoint object at 0x102991390>, coverage=3, size=3
         BIN 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
```

Potential issue?

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>, co
            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')])
)
```

This does not mean you should ignore things that don't make sense…just things you've convinced yourself should not happen

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

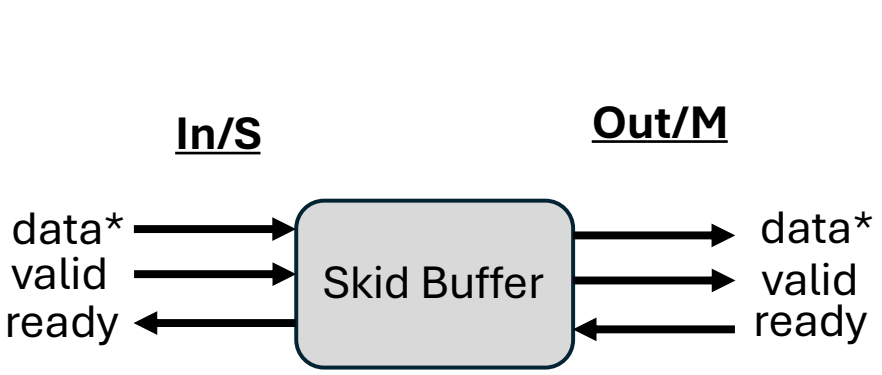
nice

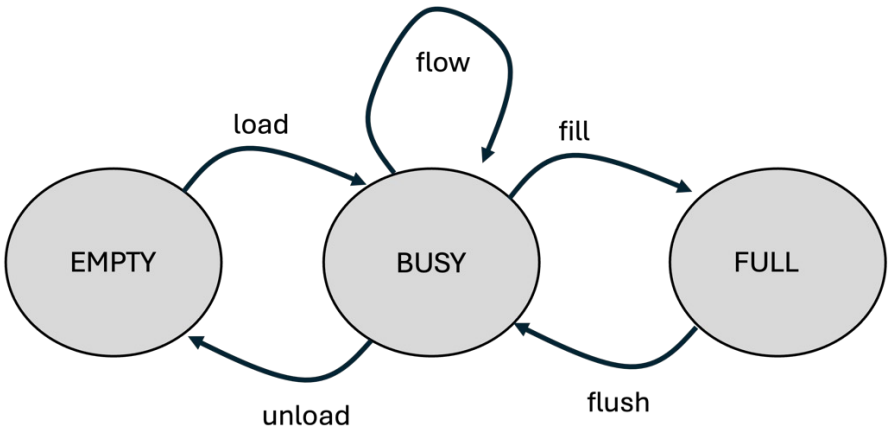
```
top : <cocotb_coverage.coverage.CoverItem object at 0x104db5d10>, coverage=13, size=13
   top.st : <cocotb_coverage.coverage.CoverItem object at 0x104d16900>, coverage=13, size=13
      top.st.next_state : <cocotb_coverage.coverage.CoverPoint object at 0x104db5e50>, coverage=3, size=3
         BIN EMPTY : 208
         BIN BUSY : 163
         BIN FULL : 129
      top.st.state : <cocotb_coverage.coverage.CoverPoint object at 0x104d167b0>, coverage=10, size=10
         BIN EMPTY : 208
         BIN BUSY : 163
         BIN FULL : 129
         top.st.state.cross : <cocotb_coverage.coverage.CoverCross object at 0x104d16a50>, coverage=7, size=7
            BIN ('EMPTY', 'EMPTY') : 163
            BIN ('EMPTY', 'BUSY') : 45
            BIN ('BUSY', 'EMPTY') : 45
            BIN ('BUSY', 'BUSY') : 94
            BIN ('BUSY', 'FULL') : 24
            BIN ('FULL', 'BUSY') : 24
            BIN ('FULL', 'FULL') : 105
test_a passed
```

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

How many bins will this have?

```
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"]
           )
)
```

Write a quick monitor for it…

- Instead of just feeding in state and old state now feed in state and all four valid/ready signals

```
@STS
def sts_sampling_function(state,sig):
  pass

async def state_and_input_monitor(dut):
    states = {0:'EMPTY', 1:'BUSY', 2:'FULL'}
    read_only = ReadOnly()
    falling_edge = FallingEdge(dut.s00_axis_aclk)
    rising_edge = RisingEdge(dut.s00_axis_aclk)
    await read_only
    old_state = dut.state.value

    while True:
        await falling_edge #when module would change
        await read_only
        state = dut.state.value
        sig = { '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
        }
        sts_sampling_function(states[old_state],sig)
        old_state = state
```

Can run along side other coverage monitors

```
cocotb.start_soon(state_monitor(dut))
cocotb.start_soon(state_and_input_monitor(dut))
```

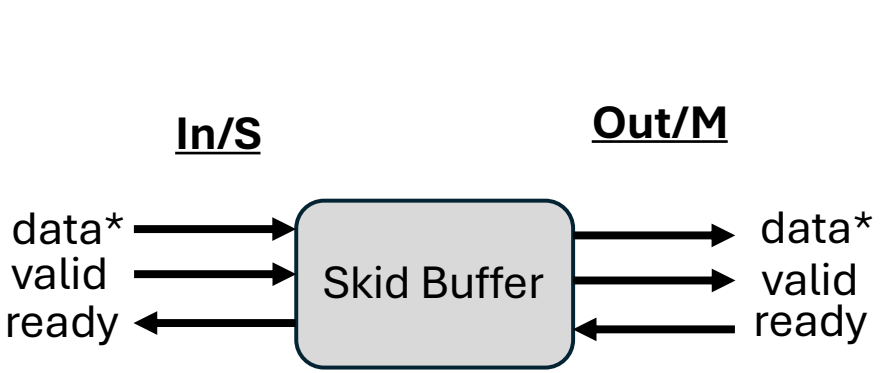
And Run It…

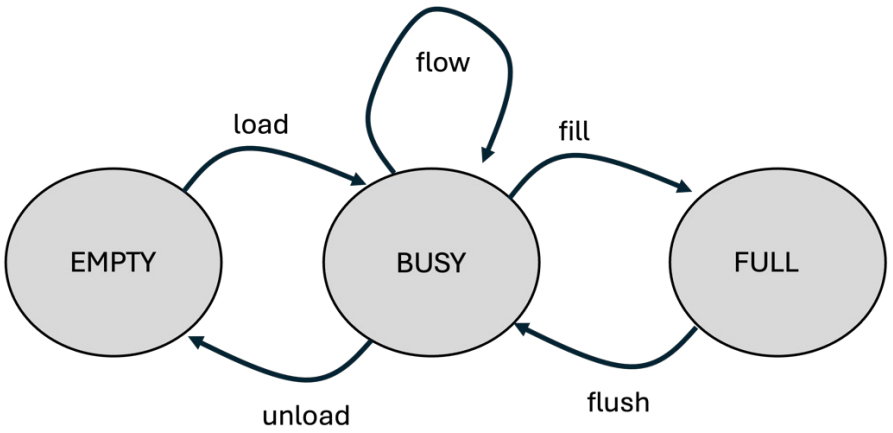
- Coverage is:
  - 26/48 total

```
top.st_sig.cross : <cocotb_coverage.coverage.CoverCross object at 0x10700e350>, coverage=26, size=48
   BIN ('EMPTY', True, True, True, True) : 1
   BIN ('EMPTY', True, True, True, False) : 0
   BIN ('EMPTY', True, True, False, True) : 32
   BIN ('EMPTY', True, True, False, False) : 2
   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) : 36
   BIN ('EMPTY', False, True, True, False) : 6
   BIN ('EMPTY', False, True, False, True) : 140
   BIN ('EMPTY', False, True, False, False) : 3
   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) : 63
   BIN ('BUSY', True, True, True, False) : 23
   BIN ('BUSY', True, True, False, True) : 4
   BIN ('BUSY', True, True, False, False) : 5
   BIN ('BUSY', True, False, True, True) : 1
   BIN ('BUSY', True, False, True, False) : 17
   BIN ('BUSY', True, False, False, True) : 0
   BIN ('BUSY', True, False, False, False) : 0
   BIN ('BUSY', False, True, True, True) : 3
   BIN ('BUSY', False, True, True, False) : 11
   BIN ('BUSY', False, True, False, True) : 29
   BIN ('BUSY', False, True, False, False) : 5
   BIN ('BUSY', False, False, True, True) : 1
   BIN ('BUSY', False, False, True, False) : 5
   BIN ('BUSY', False, False, False, True) : 0
   BIN ('BUSY', False, False, False, False) : 0
   BIN ('FULL', True, True, True, True) : 19
   BIN ('FULL', True, True, True, False) : 1
   BIN ('FULL', True, True, False, True) : 0
   BIN ('FULL', True, True, False, False) : 0
   BIN ('FULL', True, False, True, True) : 19
   BIN ('FULL', True, False, True, False) : 64
   BIN ('FULL', True, False, False, True) : 0
   BIN ('FULL', True, False, False, False) : 0
   BIN ('FULL', False, True, True, True) : 4
   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) : 3
   BIN ('FULL', False, False, True, False) : 4
   BIN ('FULL', False, False, False, True) : 0
   BIN ('FULL', False, False, False, False) : 0
```

At the naïve level…

- Yes there are 48 possible state transition and input combinations, but 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

(STATE, VALID, READY)

Slave Cross:

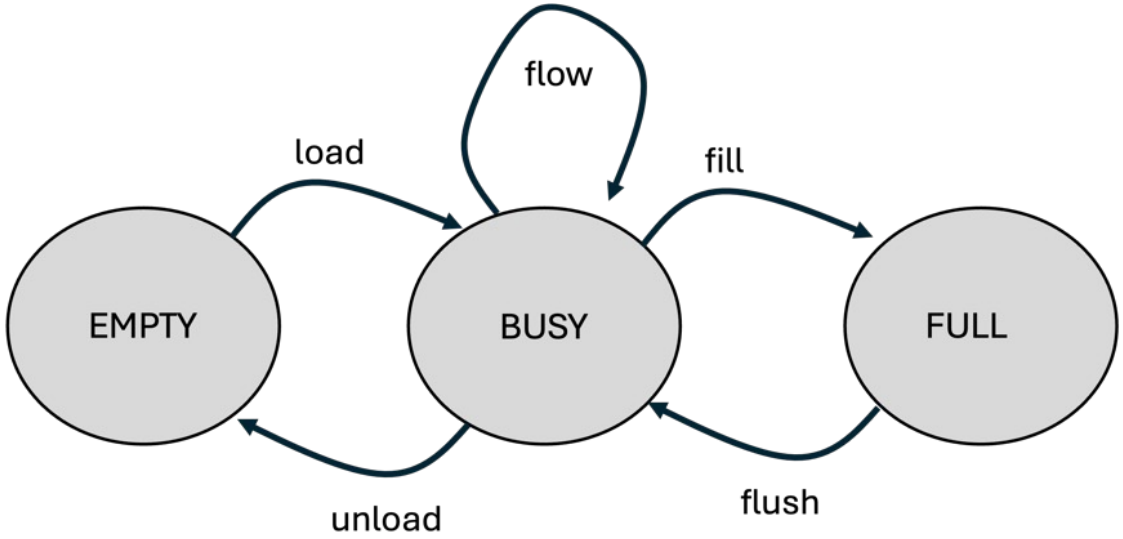
```
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x10752e350>, coverage=10, size=12
   BIN ('EMPTY', True, True) : 39
   BIN ('EMPTY', True, False) : 0
   BIN ('EMPTY', False, True) : 206
   BIN ('EMPTY', False, False) : 0
   BIN ('BUSY', True, True) : 92
   BIN ('BUSY', True, False) : 18
   BIN ('BUSY', False, True) : 51
   BIN ('BUSY', False, False) : 3
   BIN ('FULL', True, True) : 19
   BIN ('FULL', True, False) : 68
   BIN ('FULL', False, True) : 2
   BIN ('FULL', False, False) : 3
```

(STATE, VALID, READY)

Master Cross:

```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x10752e490>, coverage=10, size=12
   BIN ('EMPTY', True, True) : 41
   BIN ('EMPTY', True, False) : 5
   BIN ('EMPTY', False, True) : 182
   BIN ('EMPTY', False, False) : 17
   BIN ('BUSY', True, True) : 73
   BIN ('BUSY', True, False) : 45
   BIN ('BUSY', False, True) : 38
   BIN ('BUSY', False, False) : 8
   BIN ('FULL', True, True) : 36
   BIN ('FULL', True, False) : 56
   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 be 0 in EMPTY
  - m00_axis_tvalid never be 0 in FULL

Result

Legit/Might Occur: ✅

Should Not Occur: 🚫

s00_axis_tready never 0 in EMPTY
m00_axis_tvalid never 0 in FULL

Slave Cross:

(STATE, VALID, READY)

```
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x10752e350>, coverage=10, size=12
✅ BIN ('EMPTY', True, True) : 39
🚫 BIN ('EMPTY', True, False) : 0
✅ BIN ('EMPTY', False, True) : 206
🚫 BIN ('EMPTY', False, False) : 0
✅ BIN ('BUSY', True, True) : 92
✅ BIN ('BUSY', True, False) : 18
✅ BIN ('BUSY', False, True) : 51
✅ BIN ('BUSY', False, False) : 3
✅ BIN ('FULL', True, True) : 19
✅ BIN ('FULL', True, False) : 68
✅ BIN ('FULL', False, True) : 2
✅ BIN ('FULL', False, False) : 3
```

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

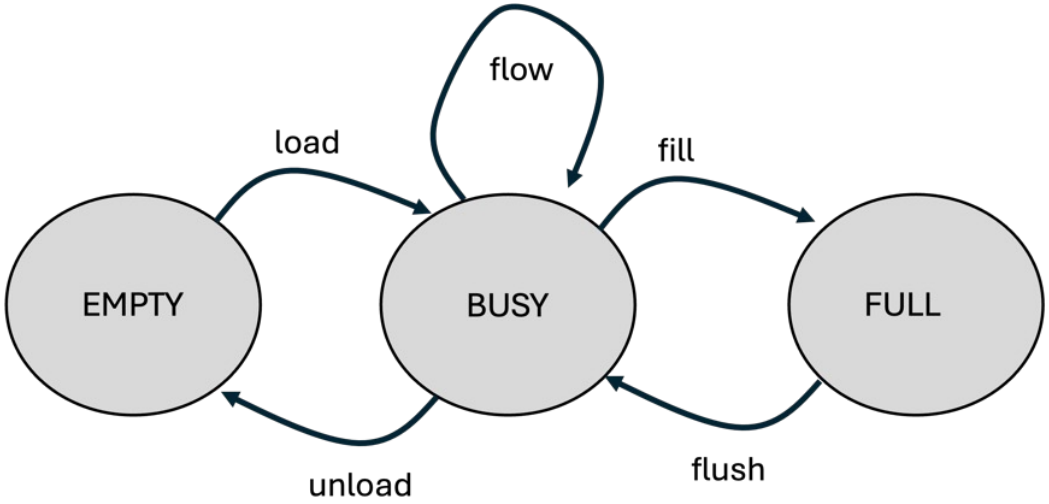
Master Cross:

(STATE, VALID, READY)

```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x10752e490>, coverage=10, size=12
✅ BIN ('EMPTY', True, True) : 41
✅ BIN ('EMPTY', True, False) : 5
✅ BIN ('EMPTY', False, True) : 182
✅ BIN ('EMPTY', False, False) : 17
✅ BIN ('BUSY', True, True) : 73
✅ BIN ('BUSY', True, False) : 45
✅ BIN ('BUSY', False, True) : 38
✅ BIN ('BUSY', False, False) : 8
✅ BIN ('FULL', True, True) : 36
✅ BIN ('FULL', True, False) : 56
🚫 BIN ('FULL', False, True) : 0
🚫 BIN ('FULL', False, False) : 0
```

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

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***

Point of Clarity...

- Monitor uses previous state in combination with all four valid/ready signals

```
@STS
def sts_sampling_function(state,sig):
  pass

async def state_and_input_monitor(dut):
    states = {0:'EMPTY', 1:'BUSY', 2:'FULL'}
    read_only = ReadOnly()
    falling_edge = FallingEdge(dut.s00_axis_aclk)
    rising_edge = RisingEdge(dut.s00_axis_aclk)
    await read_only
    old_state = dut.state.value

    while True:
        await falling_edge #when module would change
        await read_only
        state = dut.state.value
        sig = { '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
        }
        sts_sampling_function(states[old_state],sig)
        old_state = state
```

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…

- Run again:

```
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)]
           )
)
```

```
top.st_sig.mcross : <cocotb_coverage.coverage.CoverCross object at 0x106ffe490>, coverage=10, size=10
   BIN ('EMPTY', True, True) : 42
   BIN ('EMPTY', True, False) : 4
   BIN ('EMPTY', False, True) : 165
   BIN ('EMPTY', False, False) : 13
   BIN ('BUSY', True, True) : 72
   BIN ('BUSY', True, False) : 40
   BIN ('BUSY', False, True) : 39
   BIN ('BUSY', False, False) : 7
   BIN ('FULL', True, True) : 36
   BIN ('FULL', True, False) : 83
top.st_sig.s00_tready : <cocotb_coverage.coverage.CoverPoint object at 0x106bffe10>, coverage=2, size=2
   BIN True : 382
   BIN False : 119
top.st_sig.s00_tvalid : <cocotb_coverage.coverage.CoverPoint object at 0x106bfe3f0>, coverage=2, size=2
   BIN True : 262
   BIN False : 239
top.st_sig.scross : <cocotb_coverage.coverage.CoverCross object at 0x106ffe350>, coverage=10, size=10
   BIN ('EMPTY', True, True) : 38
   BIN ('EMPTY', False, True) : 186
   BIN ('BUSY', True, True) : 93
   BIN ('BUSY', True, False) : 18
   BIN ('BUSY', False, True) : 45
   BIN ('BUSY', False, False) : 2
   BIN ('FULL', True, True) : 19
   BIN ('FULL', True, False) : 94
   BIN ('FULL', False, True) : 1
   BIN ('FULL', False, False) : 5
```

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

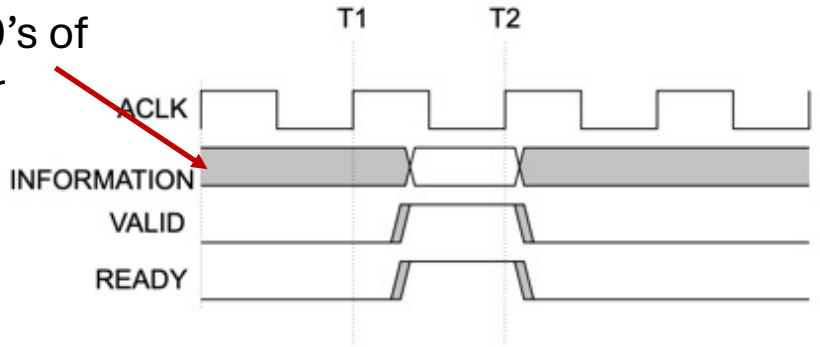


Figure A3-4 VALID with READY handshake

VALID then READY

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

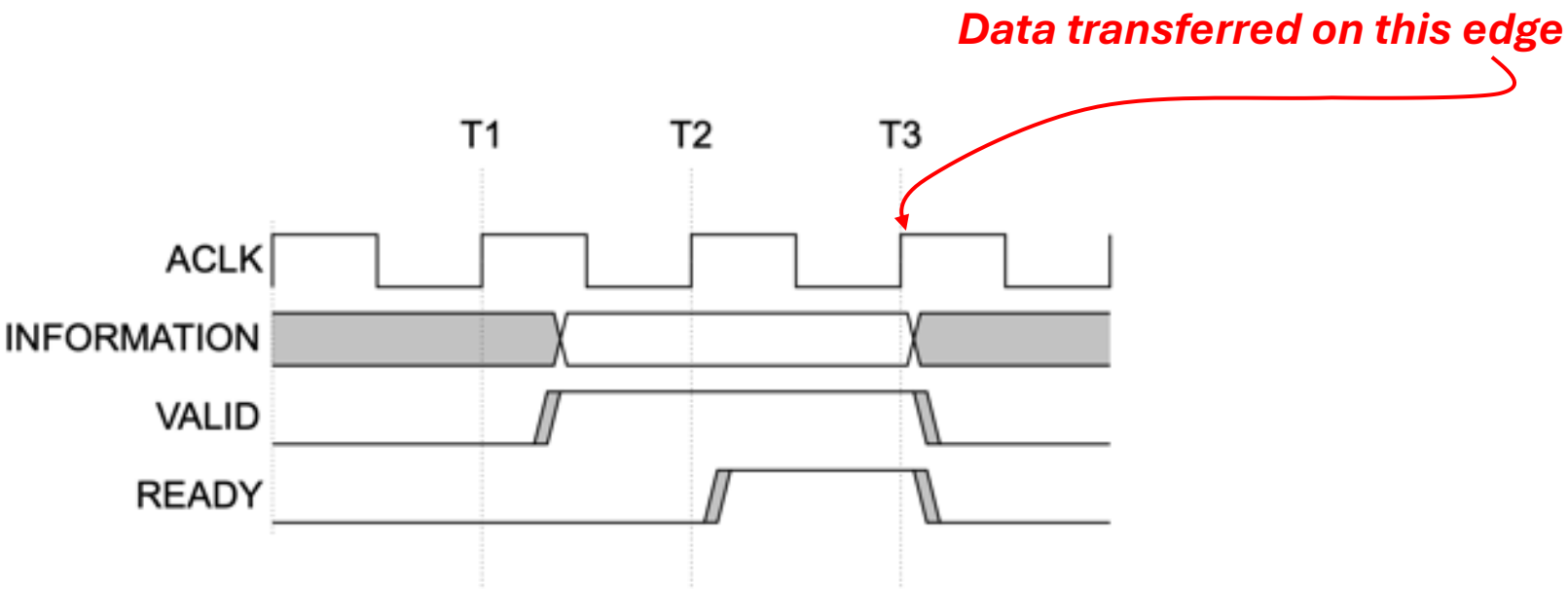


Figure A3-2 VALID before READY handshake

READY then VALID

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

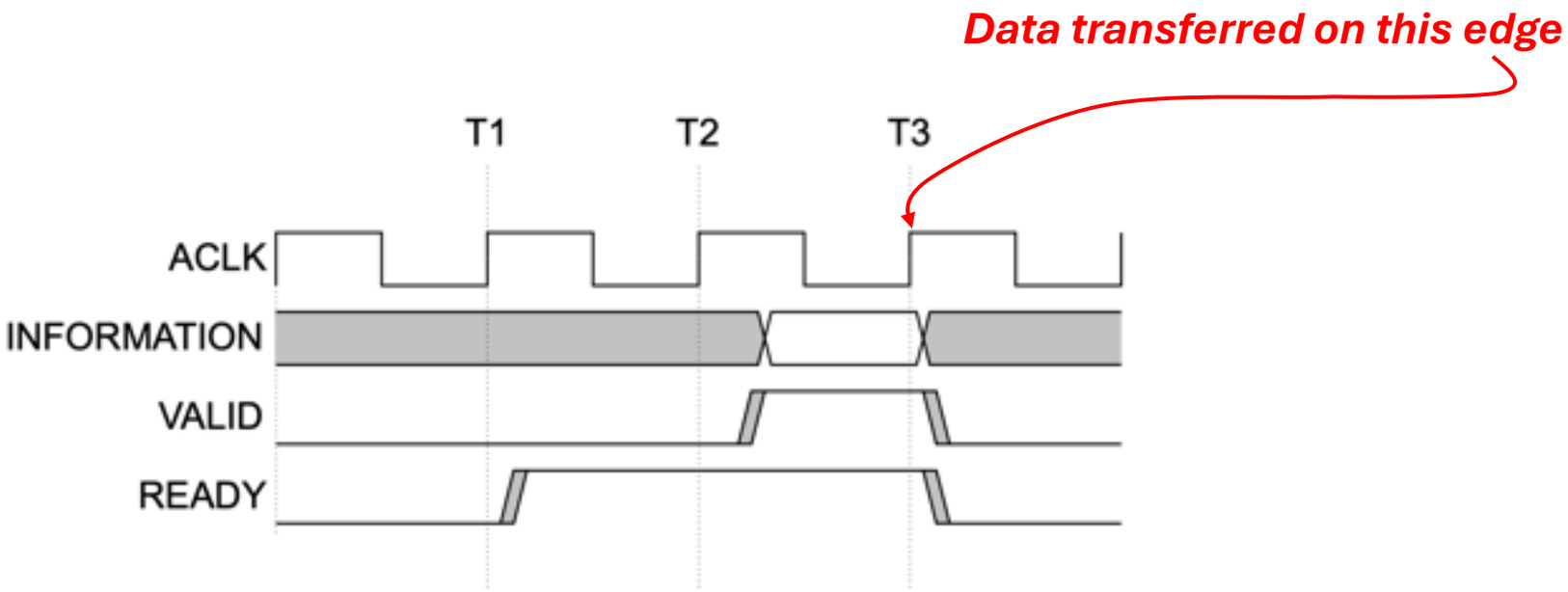


Figure A3-3 READY before VALID handshake

READY WITH VALID

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

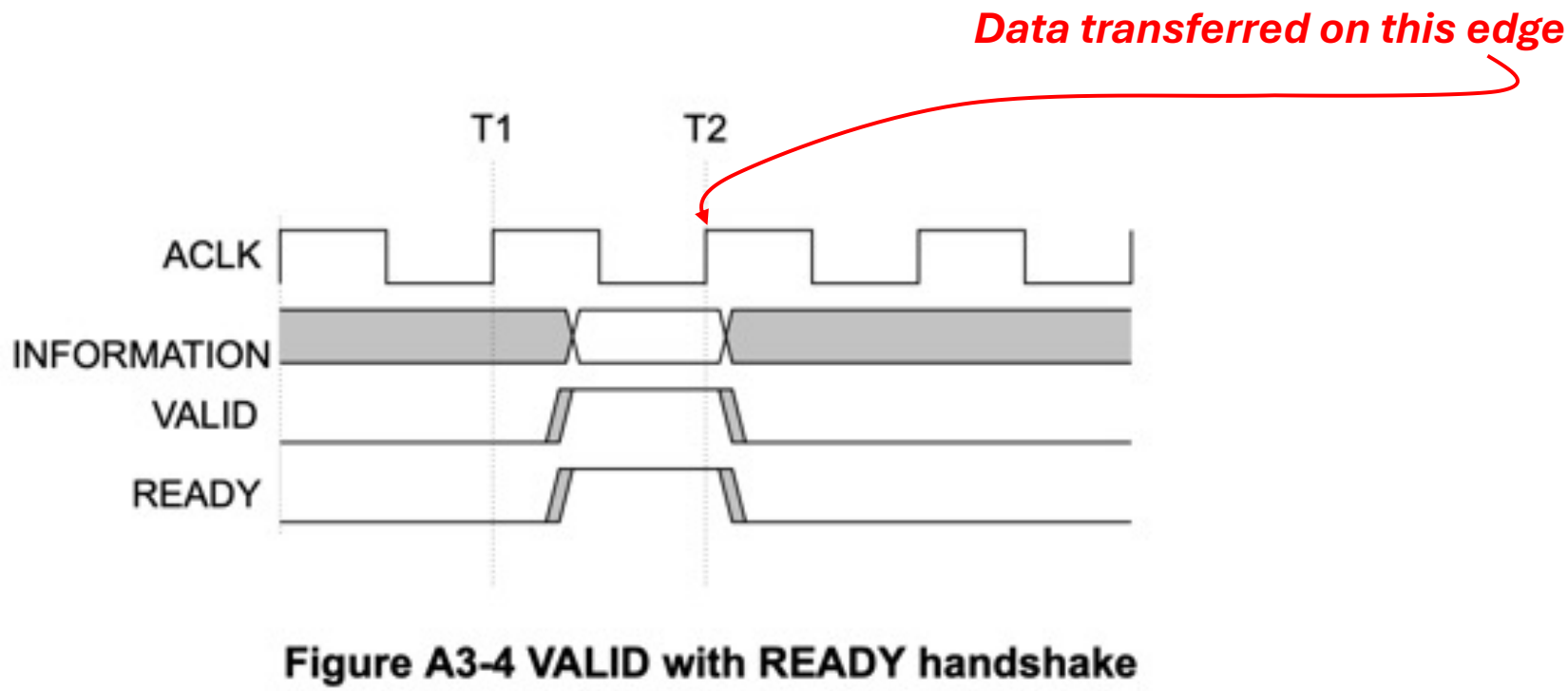


Figure A3-4 VALID with READY handshake

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

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

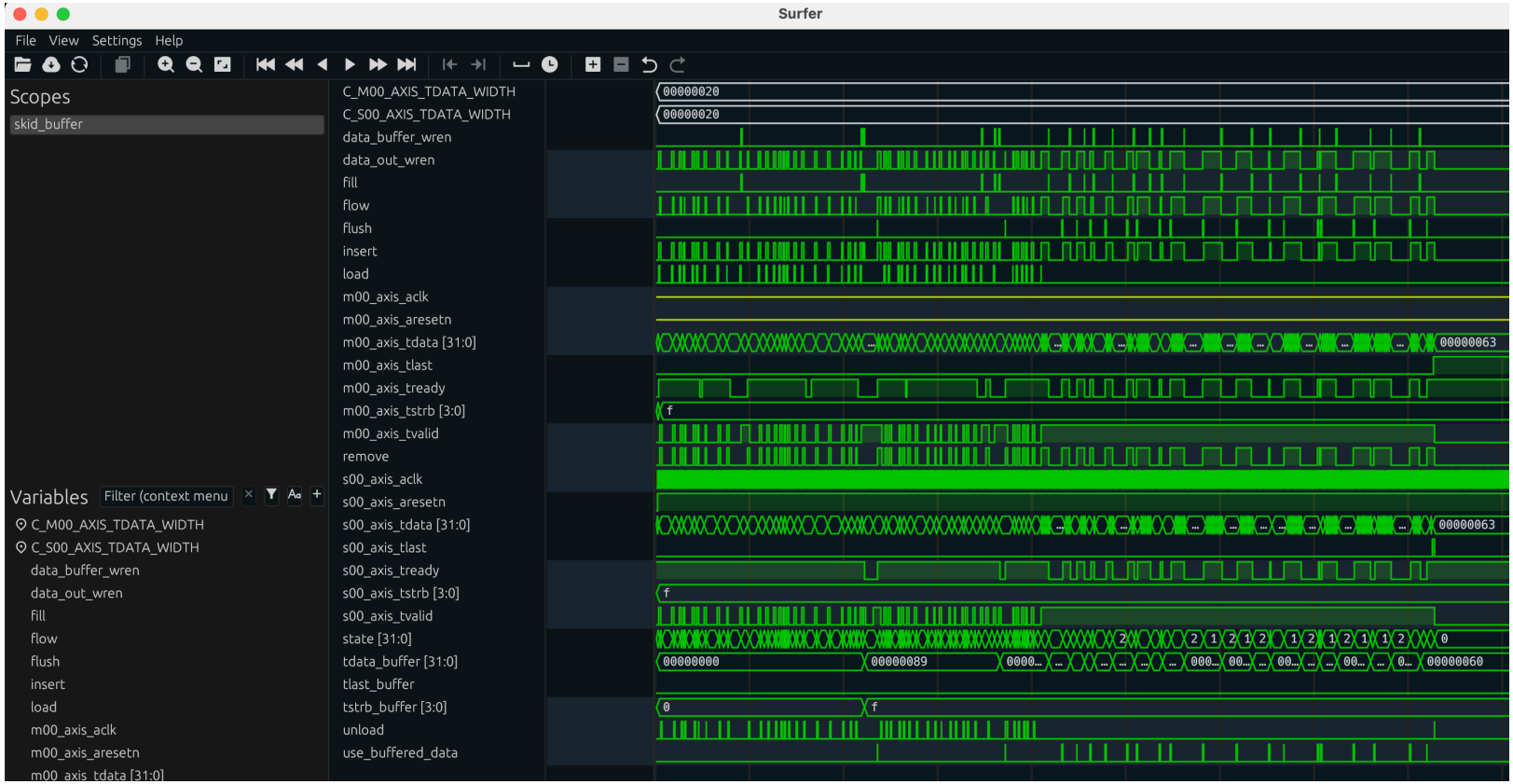
- Track and Label transitions of all four signals over time.

```
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
```

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

```
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


Surfer
File View Settings Help
Scopes
skid_buffer
Variables
Filter (context menu
C_M00_AXIS_TDATA_WIDTH
C_S00_AXIS_TDATA_WIDTH
data_buffer_wren
data_out_wren
fill
flow
flush
insert
load
m00_axis_aclk
m00_axis_aresetn
m00_axis_tdata [31:0]
m00_axis_tlast
m00_axis_tready
m00_axis_tstrb [3:0]
m00_axis_tvalid
remove
s00_axis_aclk
s00_axis_aresetn
s00_axis_tdata [31:0]
s00_axis_tlast
s00_axis_tready
s00_axis_tstrb [3:0]
s00_axis_tvalid
state [31:0]
tdata_buffer [31:0]
tlast_buffer
tstrb_buffer [3:0]
unload
use_buffered_data
00000020
00000063
00000089
00000060

Run it and you get…

```
top.os.s_cross : <cocotb_coverage.coverage.CoverCross object at 0x10306c2b0>, coverage=10, size=16
   BIN ('V:0->0', 'R:0->0') : 0
   BIN ('V:0->0', 'R:0->1') : 1
   BIN ('V:0->0', 'R:1->0') : 0
   BIN ('V:0->0', 'R:1->1') : 198
   BIN ('V:0->1', 'R:0->0') : 2
   BIN ('V:0->1', 'R:0->1') : 0
   BIN ('V:0->1', 'R:1->0') : 0
   BIN ('V:0->1', 'R:1->1') : 49
   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') : 48
   BIN ('V:1->1', 'R:0->0') : 83
   BIN ('V:1->1', 'R:0->1') : 18
   BIN ('V:1->1', 'R:1->0') : 16
   BIN ('V:1->1', 'R:1->1') : 83
```

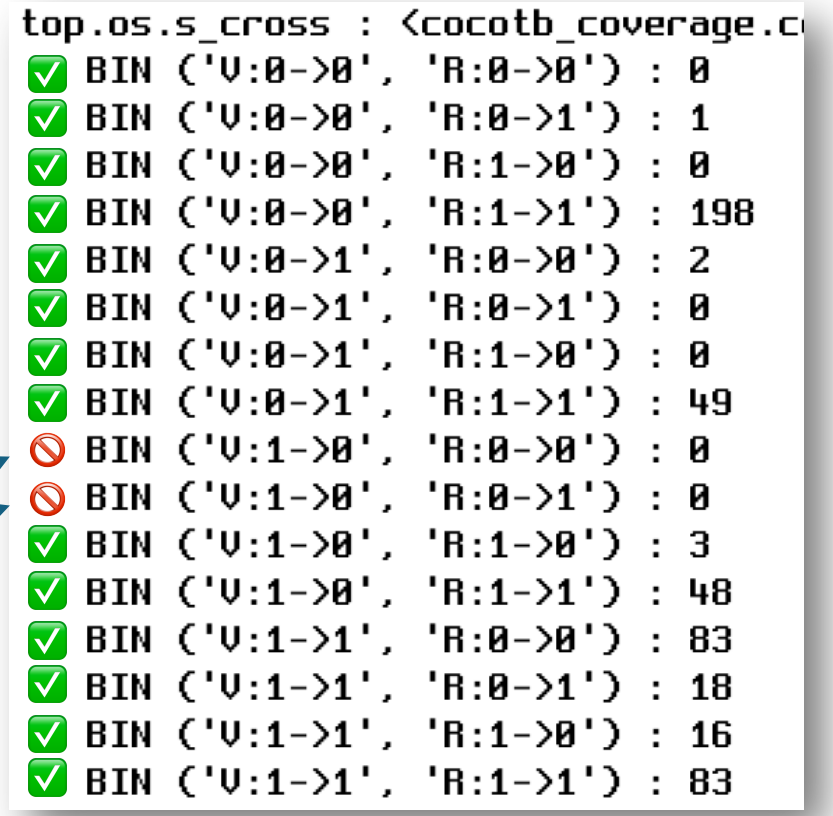
```
top.os.m_cross : <cocotb_coverage.coverage.CoverCross object at 0x10306c3e0>, coverage=12, size=16
   BIN ('V:0->0', 'R:0->0') : 22
   BIN ('V:0->0', 'R:0->1') : 4
   BIN ('V:0->0', 'R:1->0') : 0
   BIN ('V:0->0', 'R:1->1') : 161
   BIN ('V:0->1', 'R:0->0') : 4
   BIN ('V:0->1', 'R:0->1') : 2
   BIN ('V:0->1', 'R:1->0') : 0
   BIN ('V:0->1', 'R:1->1') : 39
   BIN ('V:1->0', 'R:0->0') : 0
   BIN ('V:1->0', 'R:0->1') : 0
   BIN ('V:1->0', 'R:1->0') : 9
   BIN ('V:1->0', 'R:1->1') : 36
   BIN ('V:1->1', 'R:0->0') : 99
   BIN ('V:1->1', 'R:0->1') : 20
   BIN ('V:1->1', 'R:1->0') : 16
   BIN ('V:1->1', 'R:1->1') : 89
```

Let’s Consider Slave Side

Legit/Might Occur:✅

Should Not Occur:🚫

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:🚫

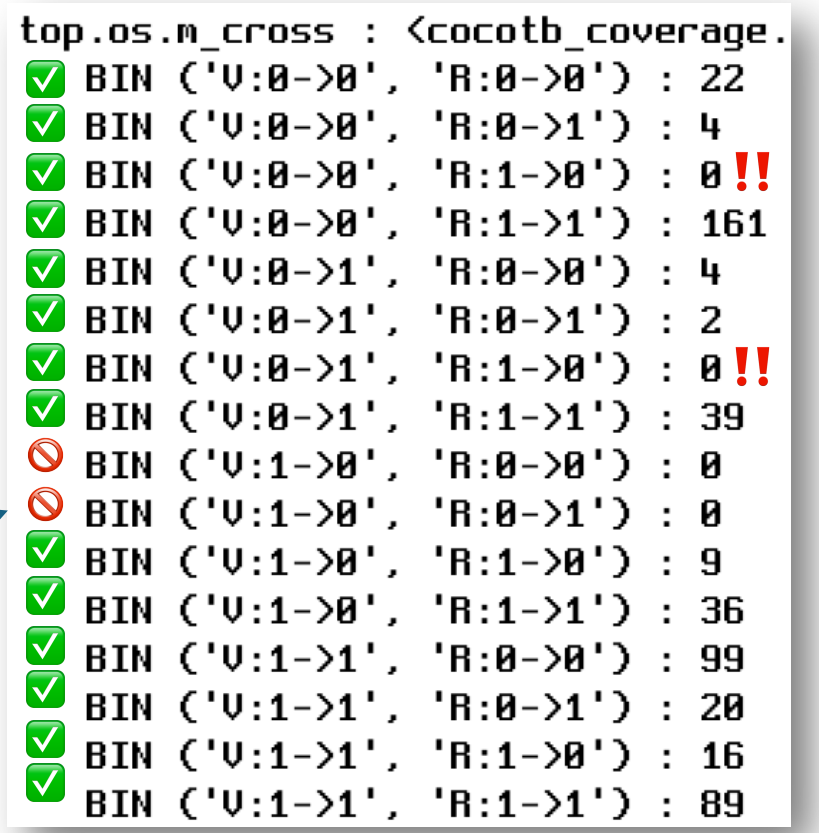
```
top.os.s_cross : <cocotb_coverage.c
✅ BIN ('V:0->0', 'R:0->0') : 0
✅ BIN ('V:0->0', 'R:0->1') : 1
✅ BIN ('V:0->0', 'R:1->0') : 0 !!
✅ BIN ('V:0->0', 'R:1->1') : 198
✅ BIN ('V:0->1', 'R:0->0') : 2
✅ BIN ('V:0->1', 'R:0->1') : 0 !!
✅ BIN ('V:0->1', 'R:1->0') : 0 !!
✅ BIN ('V:0->1', 'R:1->1') : 49
🚫 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') : 48
✅ BIN ('V:1->1', 'R:0->0') : 83
✅ BIN ('V:1->1', 'R:0->1') : 18
✅ BIN ('V:1->1', 'R:1->0') : 16
✅ BIN ('V:1->1', 'R:1->1') : 83
```

Similarly on Master Side:

Legit/Might Occur:✅

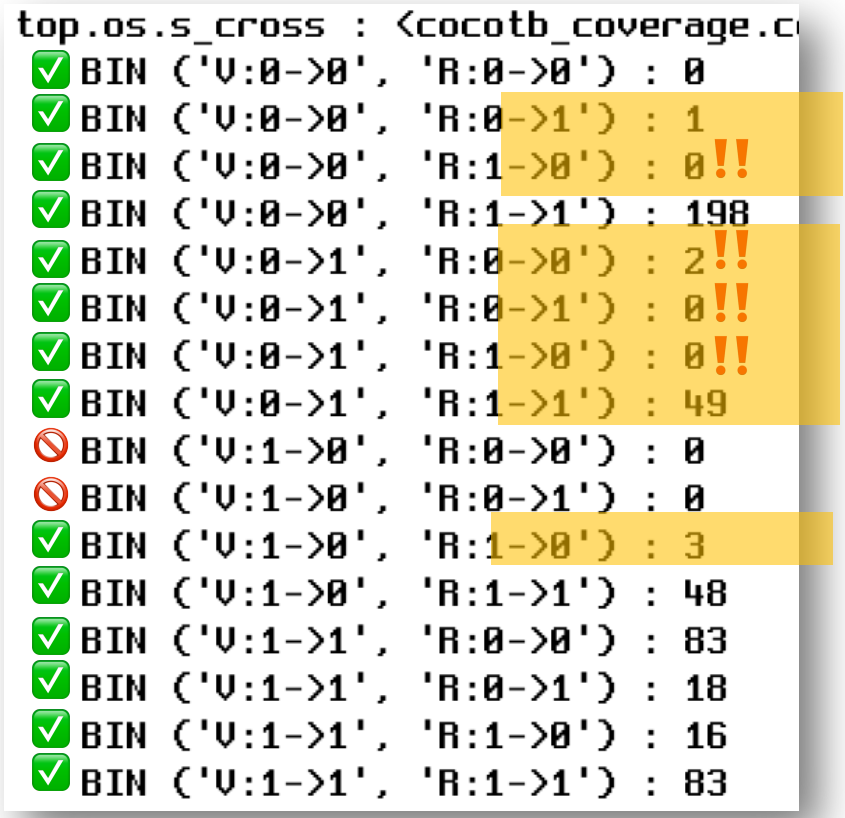
Should Not Occur:🚫

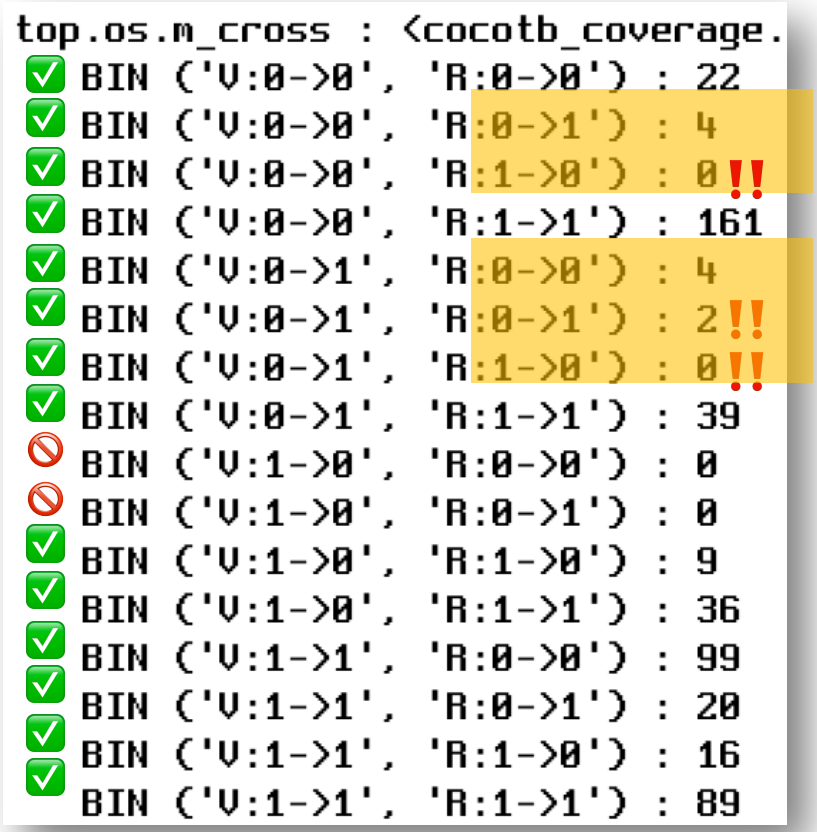
This is actually pretty reassuring since our DUT would be the device that would actually be causing these violations



Conclusions?

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



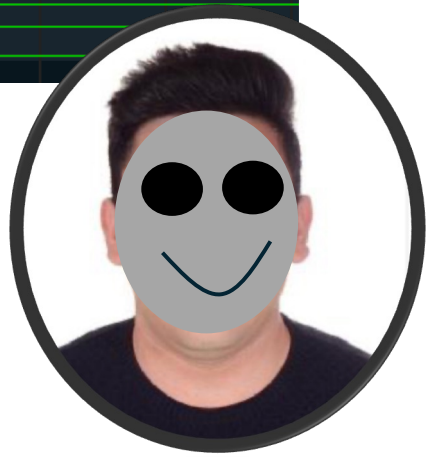


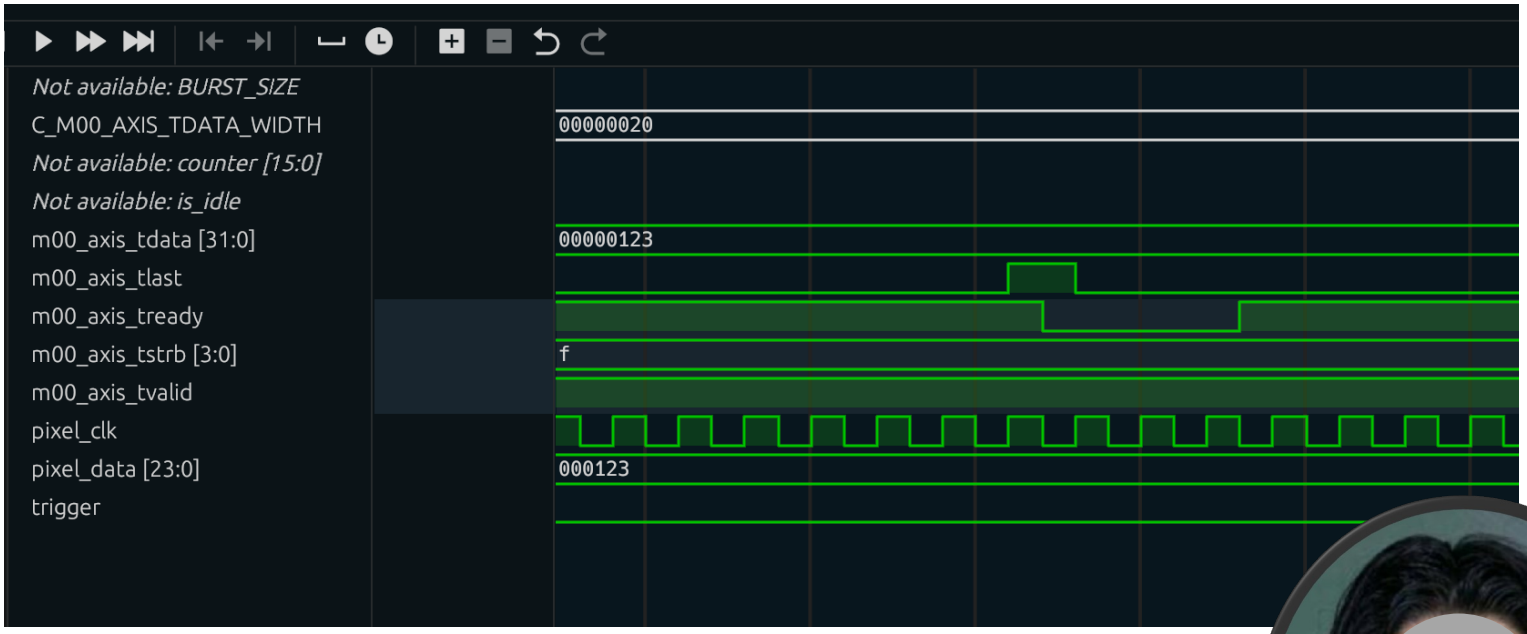
The TLAST Issue

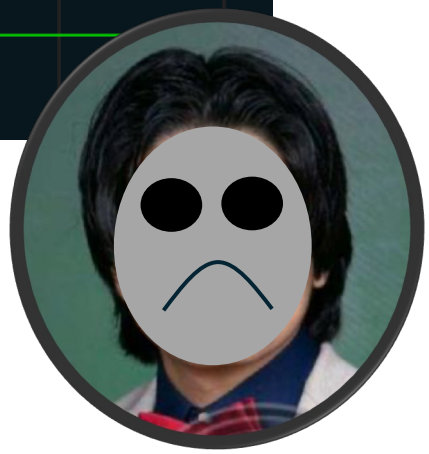
- I think in week 5, a decent number of you made data_framers that failed at the end
- I modified the S driver to deassert ready if last shows up

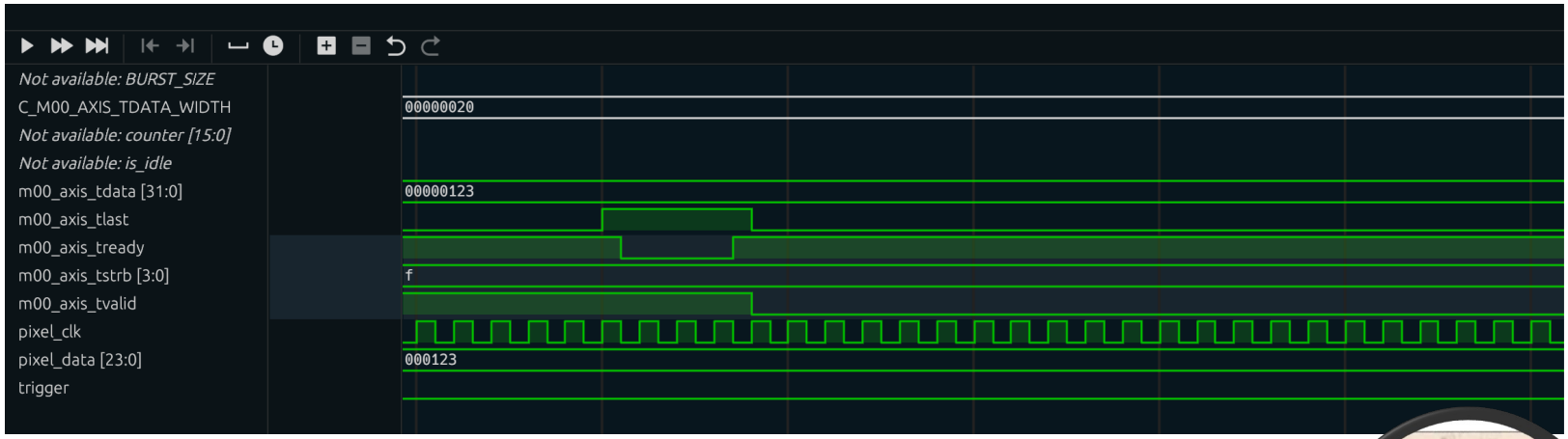
```
elif value.get("type") == "delay_last_read":
    for i in range(value.get("duration",1)):
        await falling_edge #wait until after a rising edge has passed.
        if self.bus.axis_tvalid.value == 1 and self.bus.axis_tlast.value==1:
            self.bus.axis_tready.value = 0 #set valid to be 1
            await rising_edge
            await rising_edge
            await rising_edge
            await falling_edge #wait until after a rising edge has passed.
            self.bus.axis_tready.value = 1 #set valid to be 1
            await read_only
            if self.bus.axis_tvalid.value == 0: #ifnot there...
                await RisingEdge(self.bus.axis_tvalid) #wait until it does go high
            await rising_edge
        else:
            self.bus.axis_tready.value = 1 #set valid to be 1
            await read_only
            if self.bus.axis_tvalid.value == 0: #ifnot there...
                await RisingEdge(self.bus.axis_tvalid) #wait until it does go high
            await rising_edge
    #self.bus.axis_tready.value = 0 #set to 0 and be done.
```

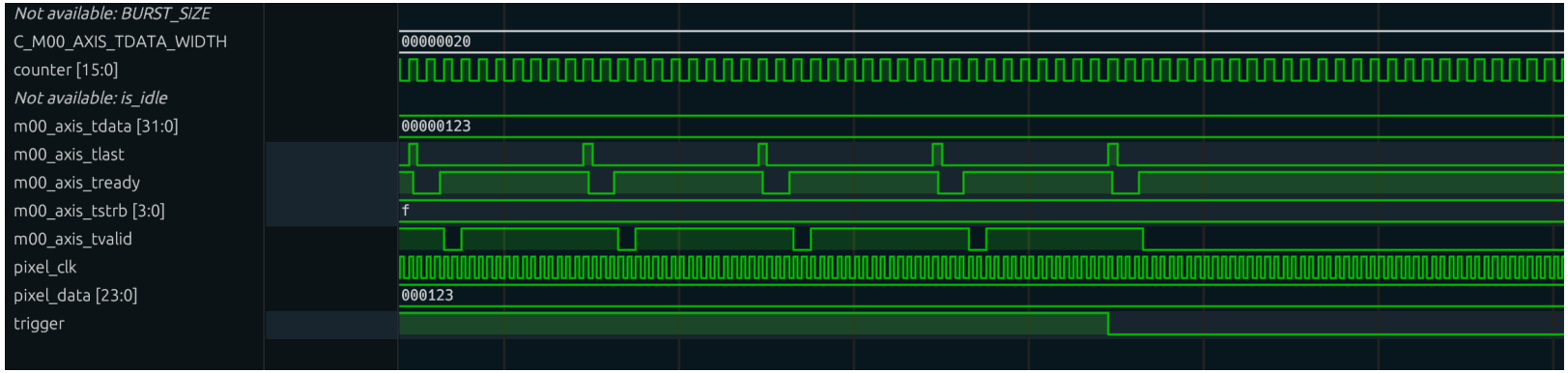

BURST_SIZE
C_M00_AXIS_TDATA_WIDTH
counter [15:0]
is_idle
m00_axis_tdata [31:0]
m00_axis_tlast
m00_axis_tready
m00_axis_tstrb [3:0]
m00_axis_tvalid
pixel_clk
pixel_data [23:0]
trigger
000000010000
00000020
fffb
fffc
fffd
fffe
ffff
0000
00000123
f
000123




Not available: BURST_SIZE
C_M00_AXIS_TDATA_WIDTH
00000020
Not available: counter [15:0]
Not available: is_idle
m00_axis_tdata [31:0]
00000123
m00_axis_tlast
m00_axis_tready
m00_axis_tstrb [3:0]
f
m00_axis_tvalid
pixel_clk
pixel_data [23:0]
000123
trigger




Not available: BURST_SIZE
C_M00_AXIS_TDATA_WIDTH
00000020
Not available: counter [15:0]
Not available: is_idle
m00_axis_tdata [31:0]
00000123
m00_axis_tlast
m00_axis_tready
m00_axis_tstrb [3:0]
f
m00_axis_tvalid
pixel_clk
pixel_data [23:0]
000123
trigger


Not available: BURST_SIZE
C_M00_AXIS_TDATA_WIDTH
00000020
counter [15:0]
Not available: is_idle
m00_axis_tdata [31:0]
00000123
m00_axis_tlast
m00_axis_tready
m00_axis_tstrb [3:0]
f
m00_axis_tvalid
pixel_clk
pixel_data [23:0]
000123
trigger