

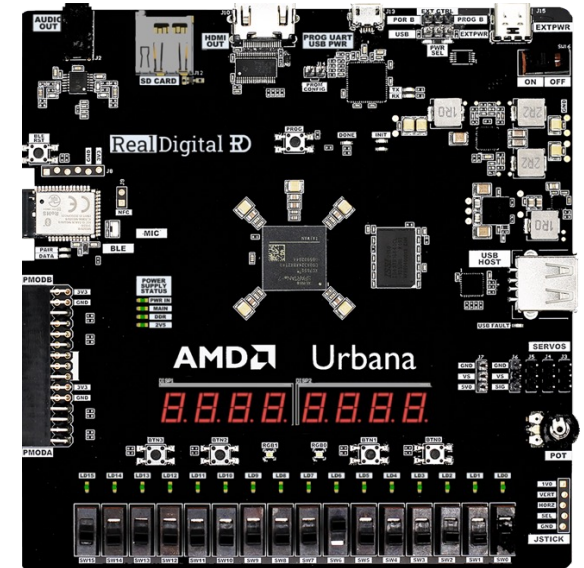
6.S965

Digital Systems Laboratory II

Lecture 7:
DRAM,
Models and Drivers and Other Things

6.205 FPGA

- Spartan 7 (xc7s50csga324-ish):
 - **2.7 Mb of BRAM**
 - 120 DSP slices
 - 52K logic cells*
- Dev Board also has 128 MB of DRAM

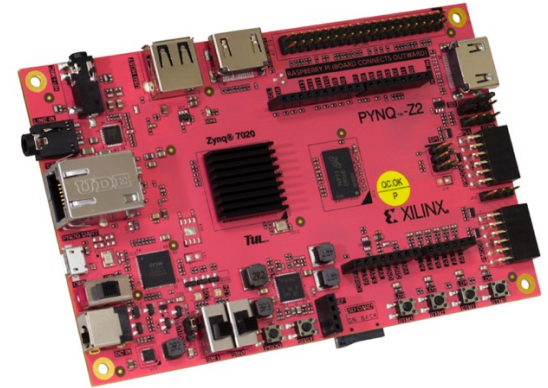


**"logic cell" is a vague term used to compare Xilinx/AMD FPGAs to other vendors. There actually is no such thing as a "logic" cell in Xilinx architecture*

https://docs.amd.com/v/u/en-US/ds180_7Series_Overview

6.S965 Zynq 7000

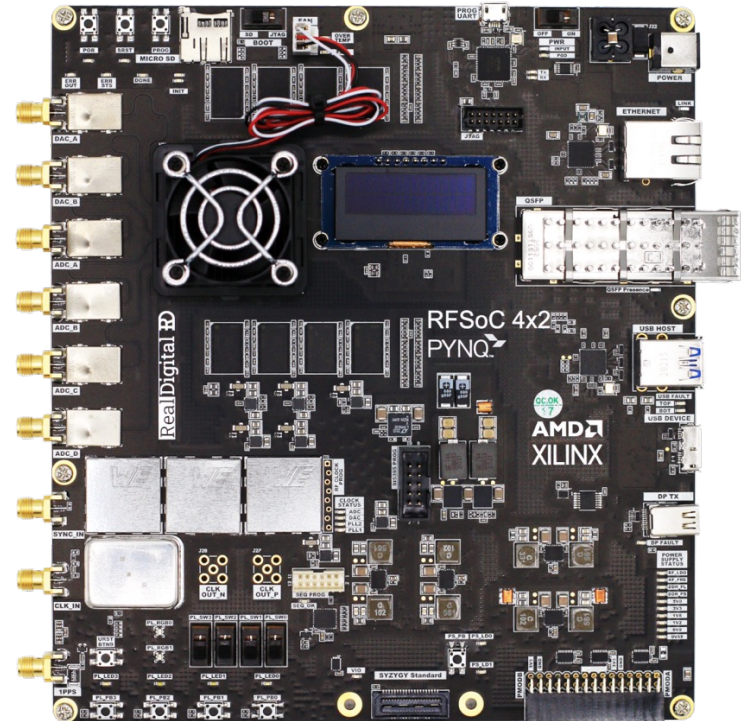
- Series 7000 XC7Z020:
 - **5.04 Mb of BRAM**
 - 220 DSP slices
 - 85K logic cells
 - Two 650 MHz A9 ARM processors
 - High-speed interconnects between two resources
- Board has 512 MB of DDR3



<https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html>

6.S965 RFSoc

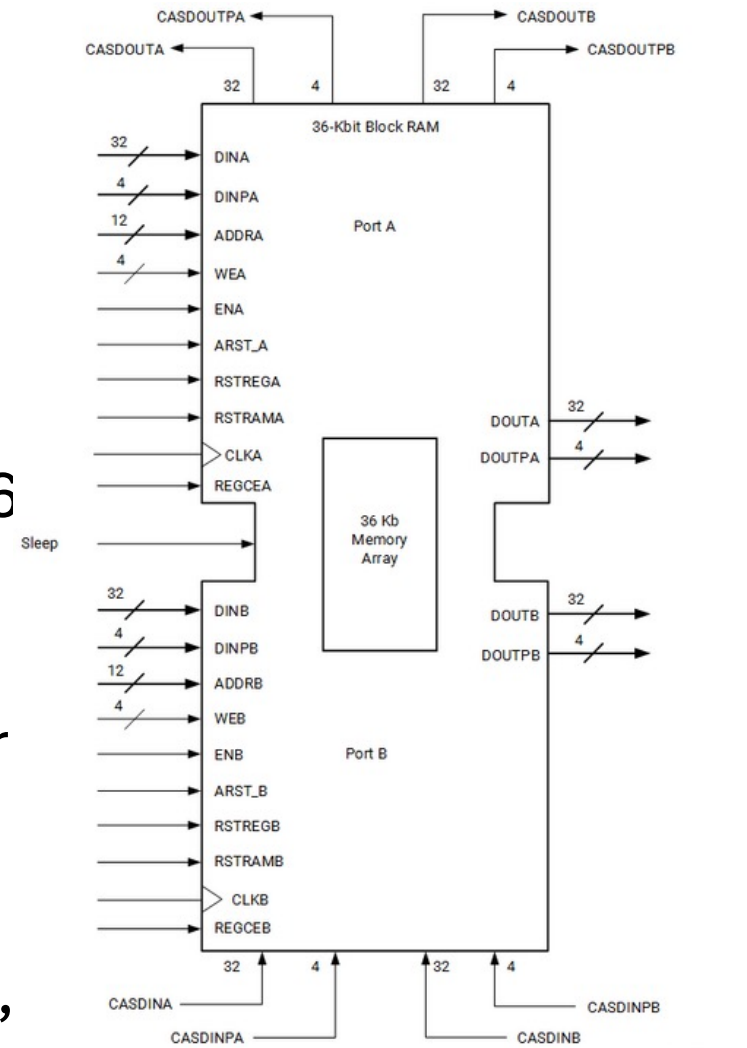
- UltraScale+ ZU48DR:
 - **38 Mb of BRAM**
 - **+22Mb of UltraRAM**
 - 4272 DSP slices
 - 930,000 Logic Cells
 - Four 5-Gsps 14 bit ADCs
 - Two 10-Gsps 14 bit DACs
 - Four 1.3 GHz ARM 53 processors
 - Two Real-time 533 MHz ARM processors
- Board has 4GB of DDR4 for FPGA portion ("PL") and 4 GB of DDR4 for processors ("PS")



<https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-rfsoc.html#tabs-b3ecea84f1-item-e96607e53b-tab>

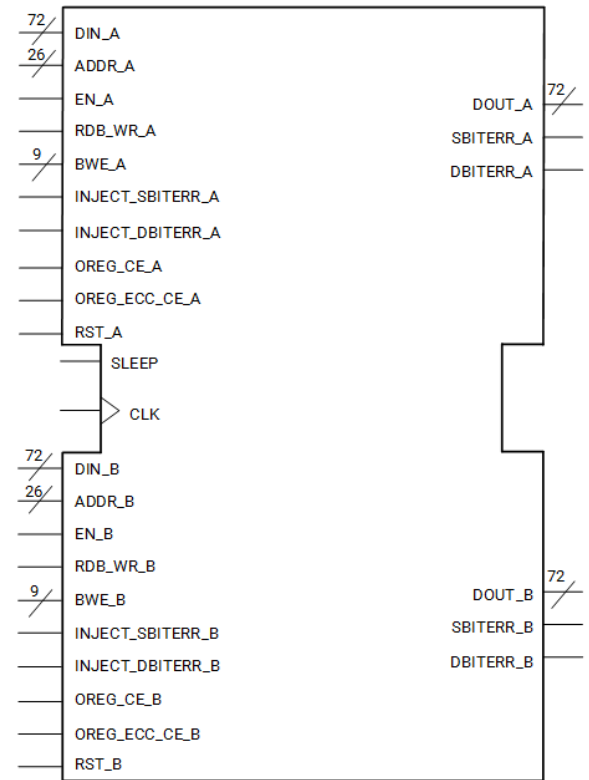
BRAM

- Block RAMs are hard primitives that exist in the FPGA
- Each one is two 18kbit or one 36 kbit dual-port memory with two clocks
- Form the basis of FIFOs, regular memory, lots of things
- Can be stacked together to make larger memory structures, albeit with performance hit



What is UltraRAM?

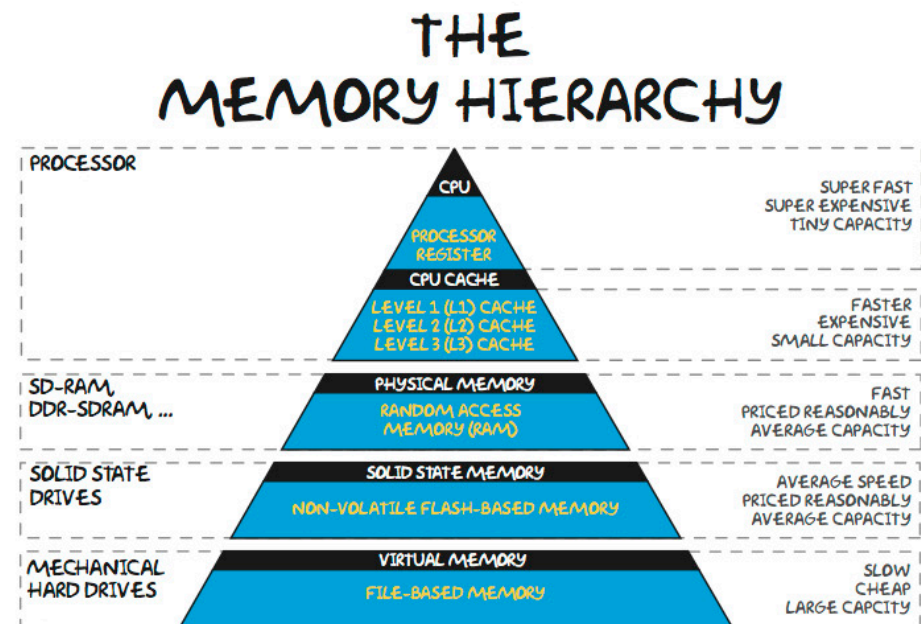
- Like a larger BRAM.
- Actually *less* flexible than BRAM, but its primitive size is ~8 times larger than BRAM
- So for mid-scale memory requirements, it can surpass linked together BRAMs in performance



X21505-101518

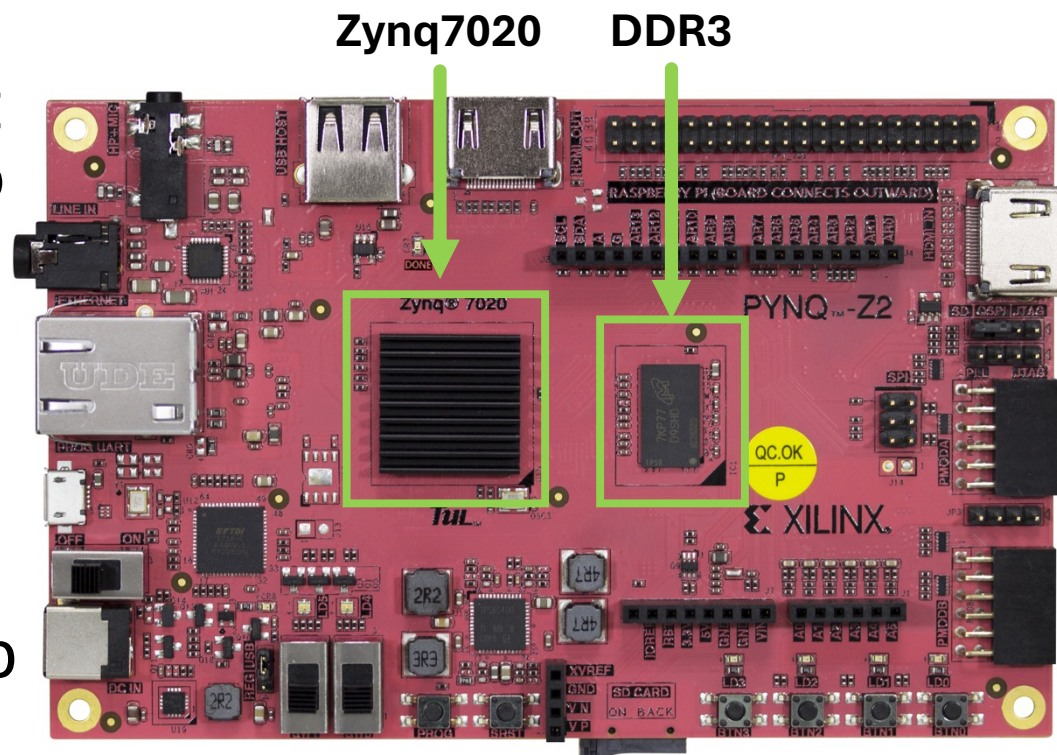
Reflective of Situation in all devices (not just FPGAs/SOCs)

- Quick-to-access memory is desirable
 - BRAM
 - URAM
- As far as memories go they take up a lot of space
- To have more memory you have to go off-chip



Off-Chip Memory Resources

- On-chip memory is always hard and expensive to make (it has gotten better, but still nowhere near what is needed)
- DRAM has proven to be the way to get lots of memory into a small spot
- But to make those massive, small-in-size designs, uses different fab tech
- Have to put off-chip as a result



DRAM

- Extremely dense array of transistor/capacitor "cells"
- So dense and tiny that read is destructive since you've stolen all charge from cap in the process....have to write back to it.
- Also So dense and tiny that the cells lose their information after about 100 ms due to parasitics naturally
- so need to be constantly read-out/rewritten, even when not using else you'll lose your info (called a refresh)

DRAM is pretty wild (aside)



~1980

8 KB

- MOSTEK developed the modern form of DRAM
- The MK4564 was the first widely successful DRAM chip
- 64 Kbits of RAM organized into 256 rows and 256 columns of one bit.
- They got *destroyed* by Japanese competition in the 1980s and closed up shop
- I found about 1100 of these chips (and variants) in the EECS stockroom.

DRAM is pretty wild (aside)



~2024
2GB

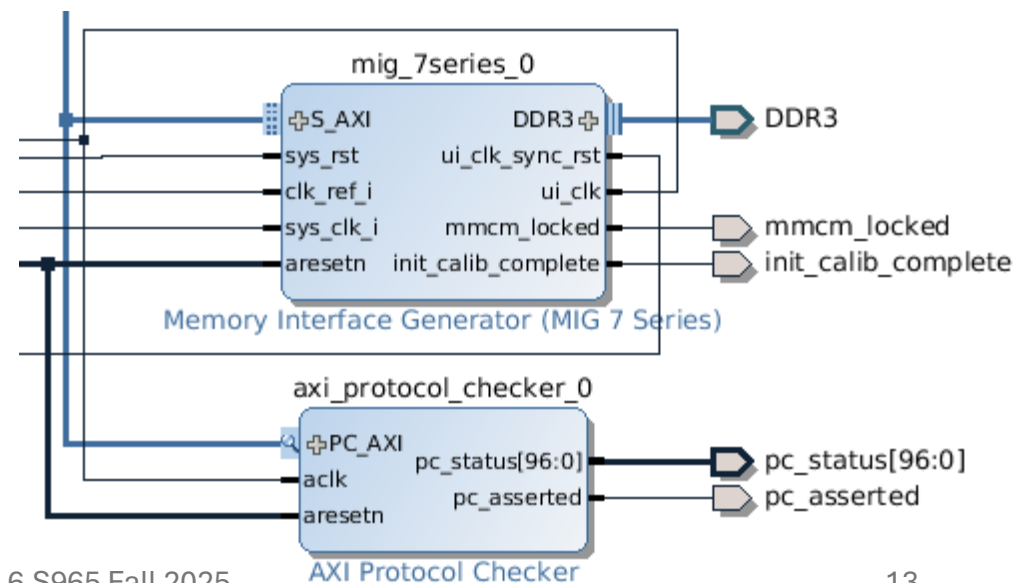
- Today you can buy 2GB DRAM variants for about $\frac{1}{4}$ the cost of what the 1980 version cost and you get:
 - 250,000 times the storage the 1980 version
 - About 10,000 the throughput the 1980 version

DRAM

- The constant need for refreshing means getting info into and out of the DRAM is not an easy task...
- Even more complicated in modern devices because they'll have different banks/channels/buffers
- Requires something to handle all the needs for refreshes and balancing them with requests for reads/writes, etc...
- This is the job of a Memory Interface/Controller

Xilinx Series 7 FPGAs

- The FPGAs used in 6.205 (series 7...Spartan or Artix) had no “hard” memory controller.
- Instead you’d use a Memory Interface Generator (“MIG”) to synthesize all the control logic
- Downside of this is it takes up a ton of your FPGA resources
- Also generally uses an AXI flow



Open Memory Controllers

- You can write your own controller, but it is a lot
- Even a simple controller for ~1980 DRAM is nothing to sneeze at.
- Modern DRAM needs to be periodically recalibrated in addition to all the refreshing, and many other things.

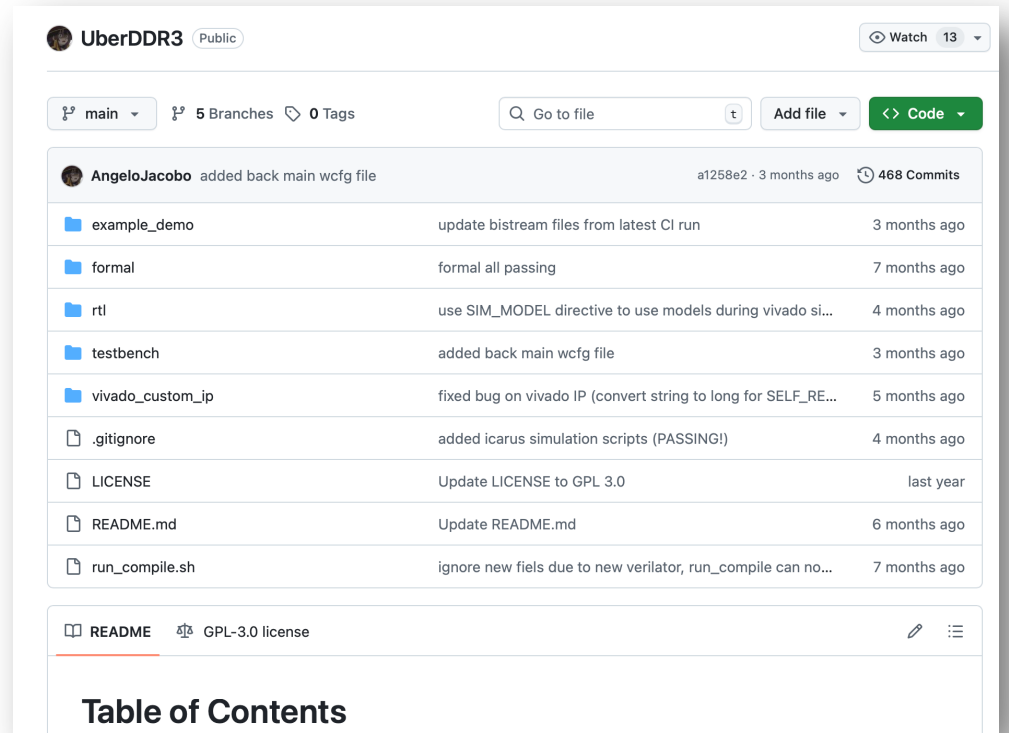


UberDDR3

- Really well-written pure SV/Verilog memory controller that I've gotten working well on a number of series 7 boards:

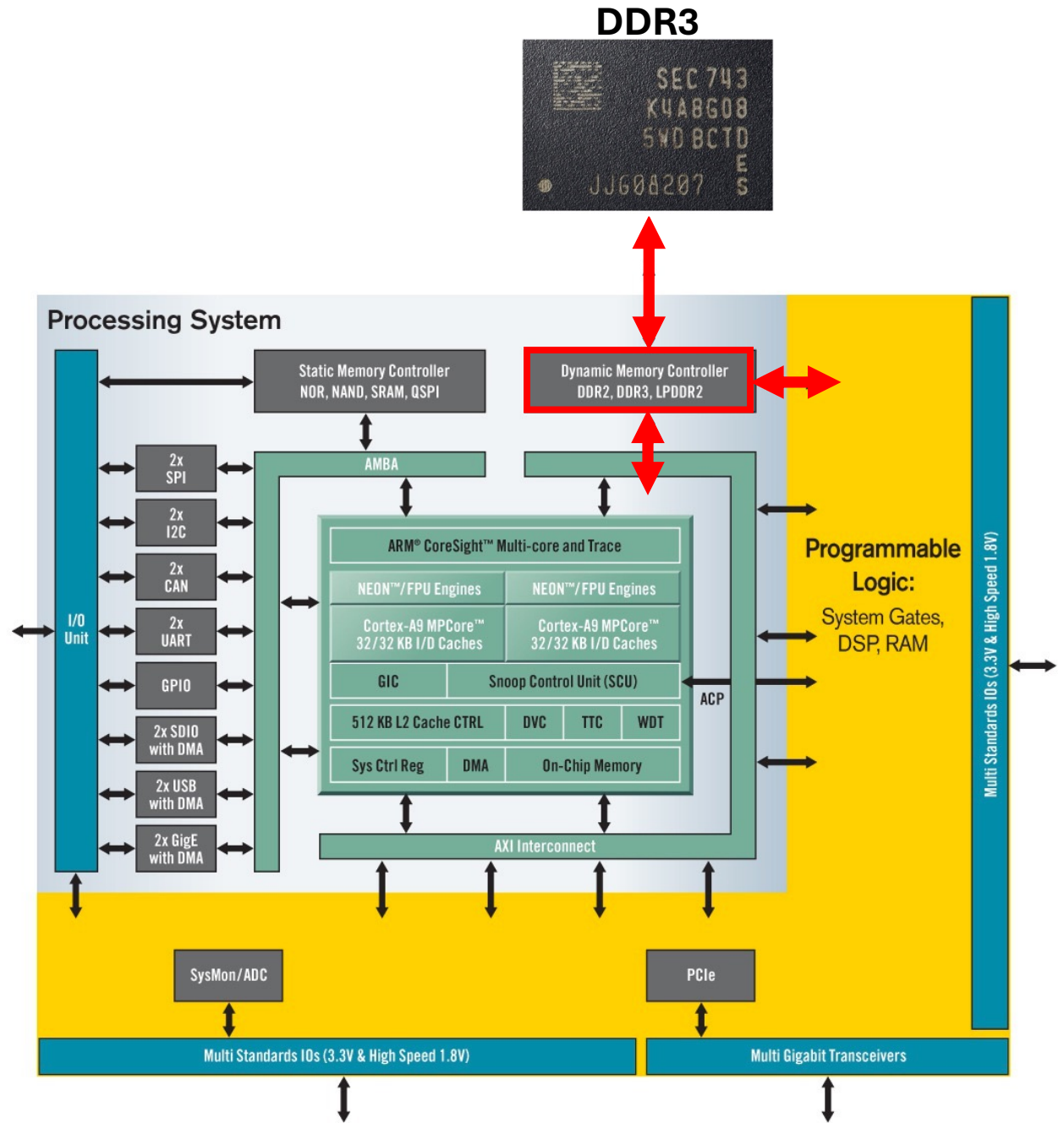
<https://github.com/AngeloJacobo/UberDDR3>

*uses fewer resources and is more performant (in my own testing) than Xilinx's own MIG



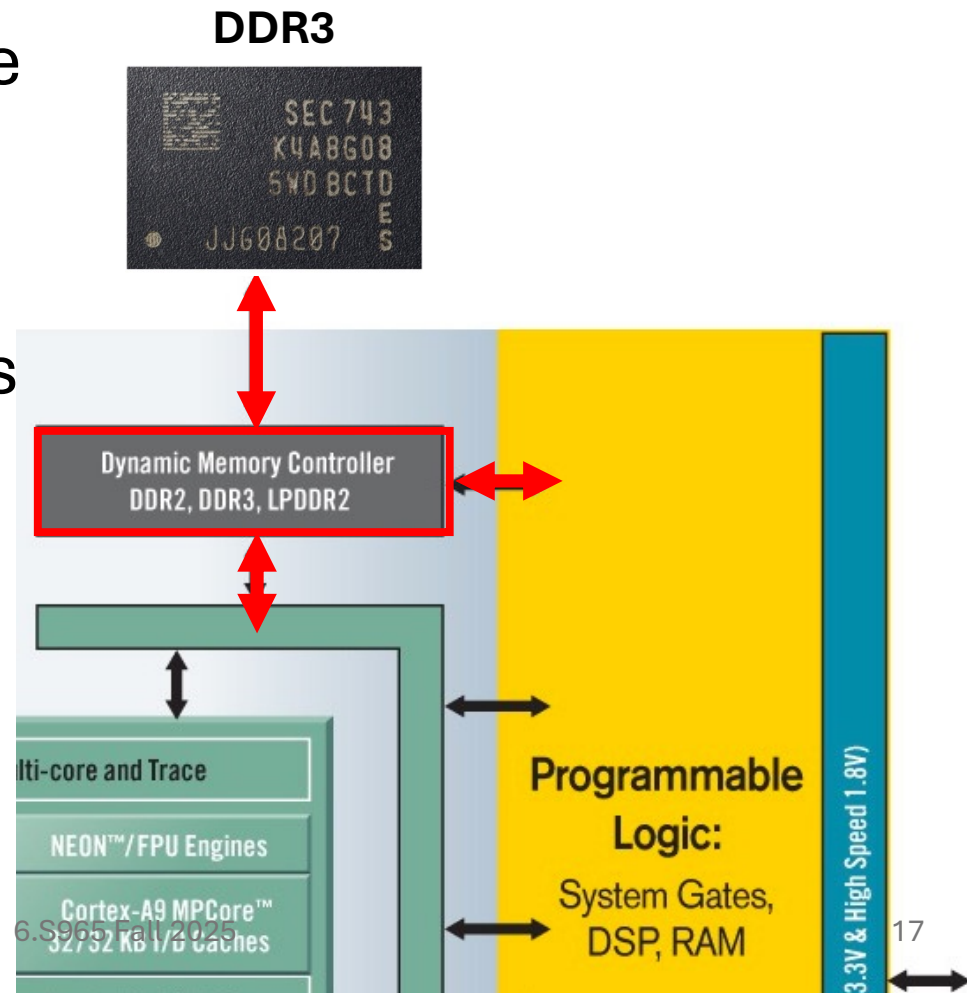
Zynq 7000

- On the Zynq-7000 chips, the DRAM is connected to “PS” pins
- But not directly to the ARM cores themselves



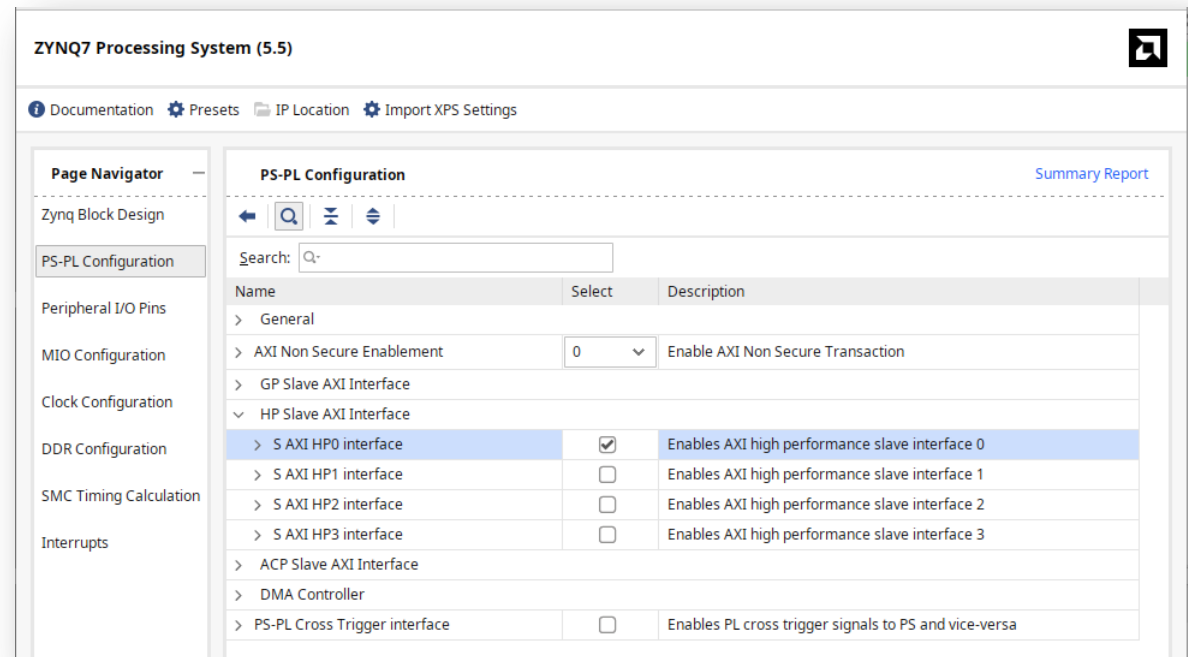
Direct-Memory-Access

- The Memory controller does have interfaces to both the ARM cores and the PL
- For the PL, this gives it “Direct Memory Access” or “DMA”
- As opposed to MA only through the processor



Interfacing to the Dynamic Memory Controller

- Can do it yourself.
- It is a solid amount of work, though.



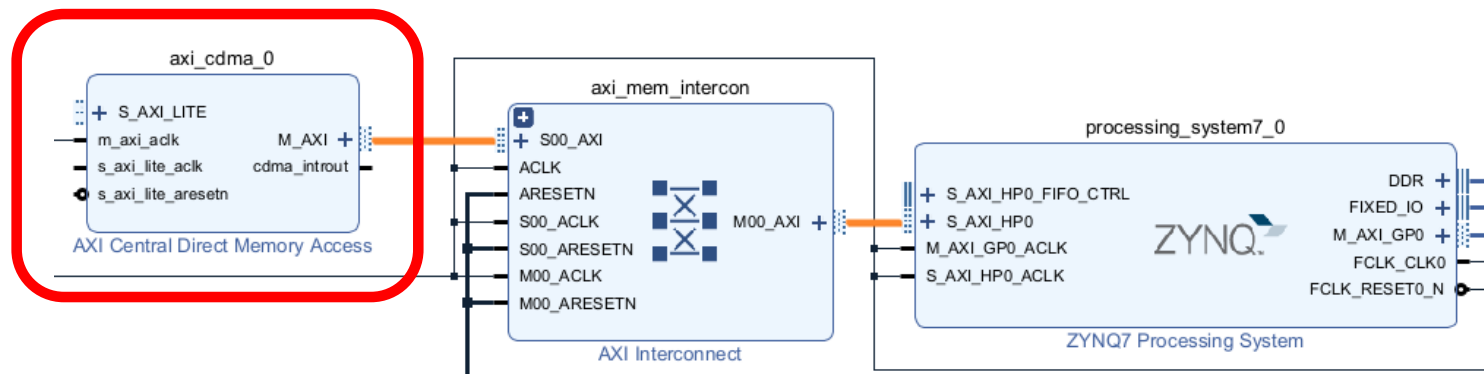
- Open up an HP AXI port from the PL into the PS and directly talk to it via memory map reads/writes

A few pieces of IP can facilitate the connection

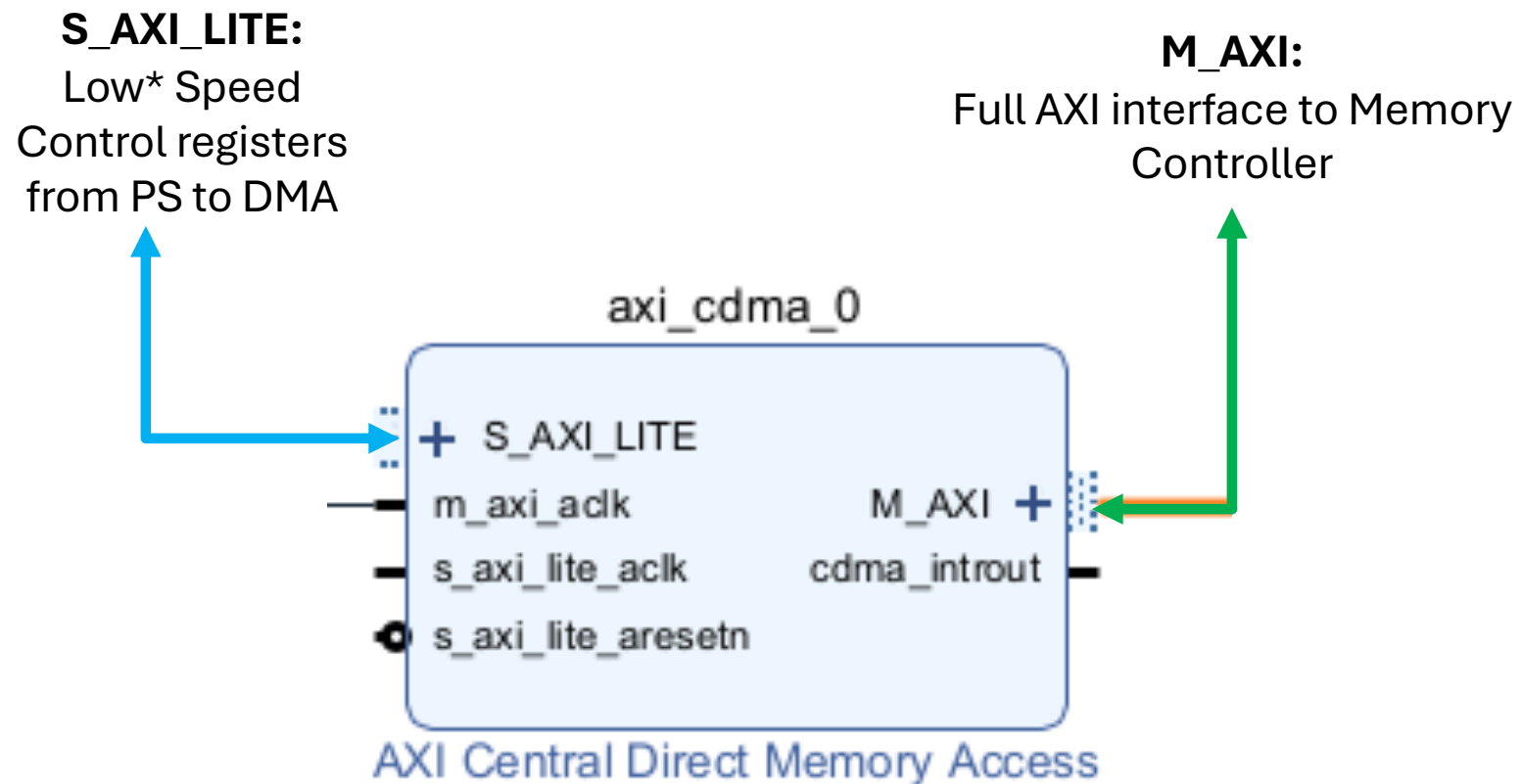
- There are some pre-packaged pieces of IP that take care of a lot of the boilerplate needed to talk to the memory controller.
- Two big options:
 - AXI Central Direct Memory Access
 - AXI Direct Memory Access

AXI Central Direct Memory Access

- Exposes the majority of the DDR3 memory space as an memory mapped object



Provides AXI-Lite-level memory interface

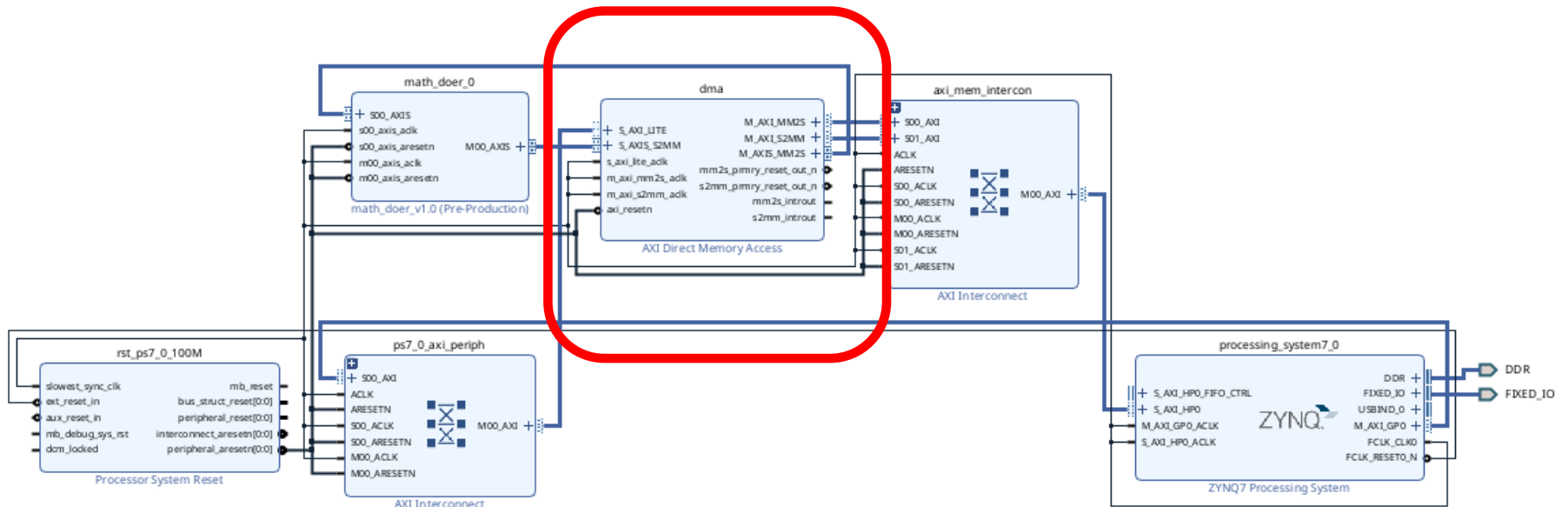


Downside with it when interfacing with PS-side programming?

- You have to know all your memory addresses in order to effectively move data between the two sides.
- Python doesn't really make that an easy thing to do.
- Even C when you're running on an OS isn't super thrilled with you using hardened constant memory locations
- Have to share data about where in memory we're looking

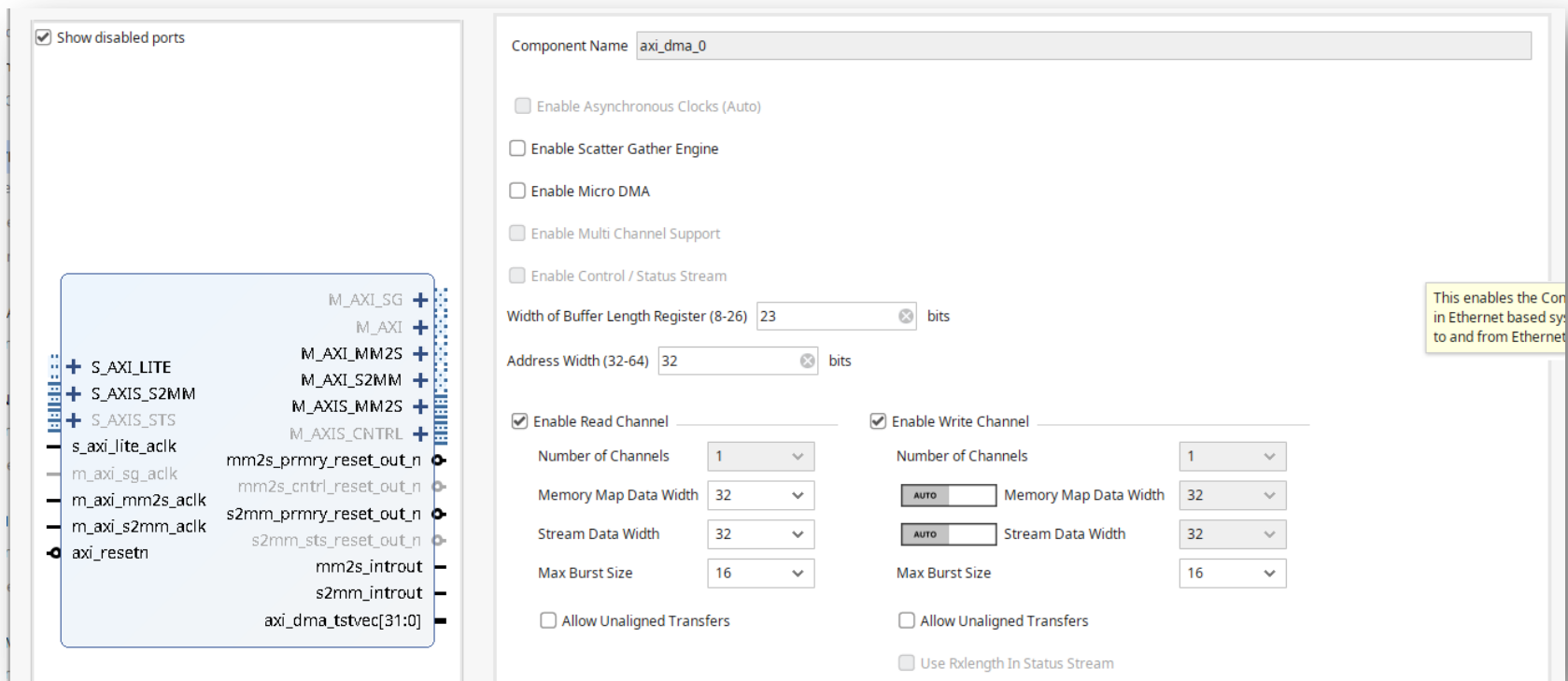
AXI Direct Memory Access

- We'll use this in week 4 and in future weeks



AXI Direct Memory Access

- Provides streaming input and output interfaces to the memory



Three General Flavors of AXI4

- **AXI4 (Full AXI):** For memory-mapped links. Provides highest performance.
 1. Address is supplied
 2. Then a data burst transfer of up to 256 data words
- **AXI4 Lite:** A memory-mapped simplified link supporting only one data transfer per connection (no bursts). (also restricted to 32 bit addr/data)
 1. Address is supplied
 2. One data transfer
- **AXI4 Stream:** Meant for high-speed streaming data
 - Can do burst transfers of unrestricted size
 - No addressing
 - Meant to stream data from one device to another quickly on its own direct connection

Streaming means: no Addressing!

- Data flows unidirectionally
- Data's “place” is where it is in the chain. It doesn't have a address it is supposed to live at
- For values that are independent of one another this is pretty much all that's needed

All the AXIs Showing up in axi_dma

S2MM = “Stream to Memory Map”

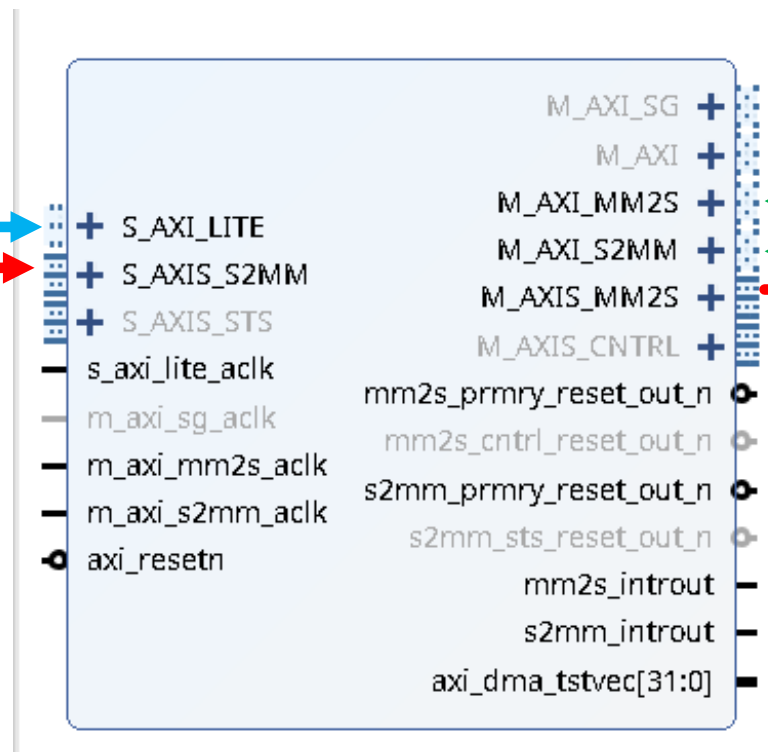
MM2S = “Memory Map to Stream”

S_AXI_LITE:
Low* Speed
Control registers
from PS to DMA

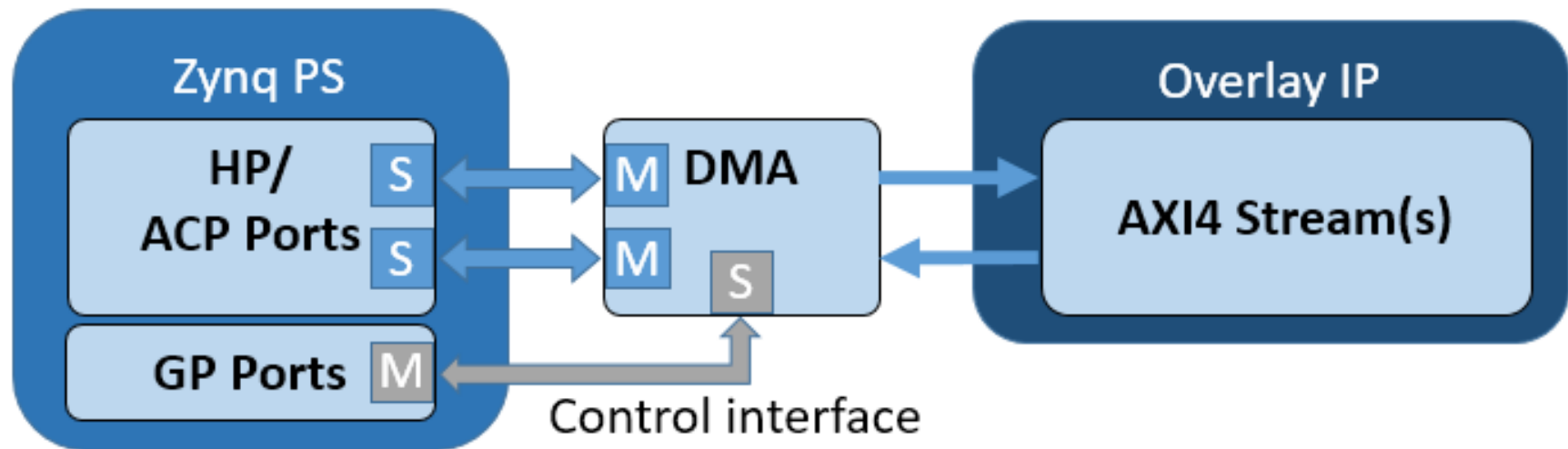
M_AXI_MM2S/S2MM:
High-Speed Full AXI interface
to Memory Controller

S_AXIS_S2MM:
Write data into
memory from PL

M_AXIS_MM2S:
Read data from
memory into PL



Block Diagram of Usage



https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries/dma.html

On the Python (processor) side...

- Use allocate to dynamically shift memory

```
from pynq import PL
PL.reset()
from pynq import Overlay #import the overlay module
ol = Overlay('./design_1_wrapper.bit') #locate to the bit file
dma = ol.dma # GRAB THE DMA
from pynq import allocate
import numpy as np
# Allocate buffers for the input and output signals
n = 1000000
in_buffer = allocate(shape=(n,), dtype=np.int32)
out_buffer = allocate(shape=(n,), dtype=np.int32)
# Copy the samples to the in_buffer
np.copyto(in_buffer, samples) #samples come from somewhere
# Trigger the DMA transfer and wait for the result
dma.sendchannel.transfer(in_buffer) #send data out into memory
dma.recvchannel.transfer(out_buffer) #wait for data to appear
dma.sendchannel.wait()
dma.recvchannel.wait()
```

Pynq Allocate

- Basically is like a malloc compatible with the PL side.

Allocate

The `pynq.allocate` function is used to allocate memory that will be used by IP in the programmable logic.

IP connected to the AXI Master (HP or ACP ports) has access to PS DRAM. Before IP in the PL accesses DRAM, some memory must first be allocated (reserved) for the IP to use and the size, and address of the memory passed to the IP. An array in Python, or Numpy, will be allocated somewhere in virtual memory. The physical memory address of the allocated memory must be provided to IP in the PL.

`pynq.allocate` allocates memory which is physically contiguous and returns a `pynq.Buffer` object representing the allocated buffer. The buffer is a numpy array for use with other Python libraries and also provides a `.device_address` property which contains the physical address for use with IP. For backwards compatibility a `.physical_address` property is also provided

The allocate function uses the same signature as `numpy.ndarray` allowing for any shape and data-type supported by numpy to be used with PYNQ.

https://pynq.readthedocs.io/en/latest/pynq_libraries/allocate.html#allocate

Address Space

The screenshot shows the 'Address Editor' window with a tree view on the left and a table of assigned addresses on the right. The tree view shows a hierarchy of address spaces, including Network 0, Network 1, Network 2, and Network 3. Network 2 is expanded, showing its sub-components. Network 3 is also expanded, showing its sub-components. The table on the right lists the assigned addresses for the selected components.

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
> Network 0 (/axi_cdma_0/Data)					
> Network 1 (/axi_cdma_0/Data_SG)					
▼ Network 2 (/axi_dma_0/Data_S2MM)					
▼ /axi_dma_0					
▼ /axi_dma_0/Data_S2MM (32 address bits : 4G)					
/processing_system7_0/S_AXI_HP0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0	512M	0x1FFF_FFFF
▼ Network 3 (/processing_system7_0/Data)					
▼ /processing_system7_0					
▼ /processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_dma_0/S_AXI_LITE	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
/fir_interface_0/S00_AXI	S00_AXI	S00_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF

Result/Speed:

- In lab this week, you'll send down 2 million 32 bit integers into the PL fabric and then run some filters on them and put the results back up into the DRAM for processor consumption
- Timing it this takes about 0.021 seconds.
- That ends up being ~ 380 MBps processing speed in a semi-sustained manner which is nothing to sneeze at.

Speed

- *Peak* throughput of the DDR3 on the Pynq Z2 board:
 - $16 \text{ bits} * 525 \text{ MHz} * 2 = \mathbf{2.1 \text{ GBps}}$ peak bandwidth
- The 380 MBps actually involves moving data *into* and *out of* the memory so really we're getting 760MBps data movement (~1/3 of peak bandwidth *sustained*)
- Also that is largely based on the fact that the AXIS streaming system you'll build is clocked at 100 MHz and moving data on a 32 bit bus (about 400 MBps throughput)
- Clocking faster and doing some other things should be able to increase this if needed.

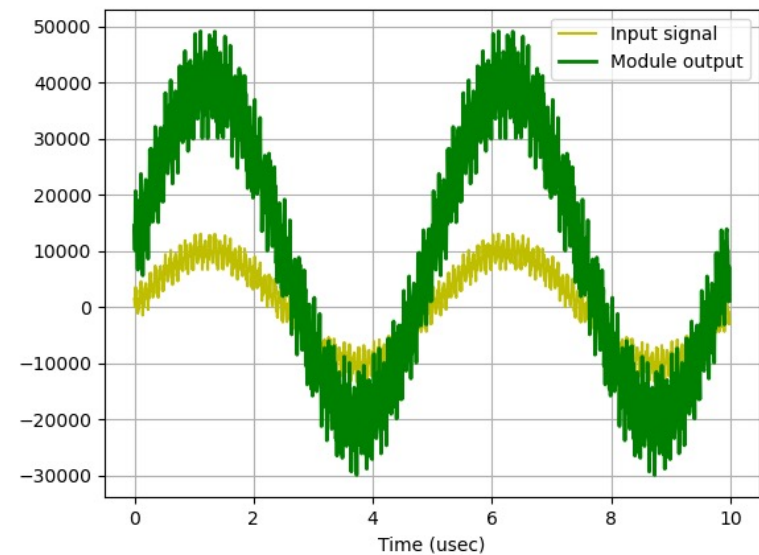
Week 4: Part 1

- Use allocate to dynamically shift memory

```
from pynq import PL
PL.reset()
from pynq import Overlay #import the overlay module
ol = Overlay('./design_1_wrapper.bit') #locate to the bit file
dma = ol.dma # GRAB THE DMA
from pynq import allocate
import numpy as np
# Allocate buffers for the input and output signals
n = 1000000
in_buffer = allocate(shape=(n,), dtype=np.int32)
out_buffer = allocate(shape=(n,), dtype=np.int32)
# Copy the samples to the in_buffer
np.copyto(in_buffer, samples) #samples come from somewhere
# Trigger the DMA transfer and wait for the result
dma.sendchannel.transfer(in_buffer) #send data out into memory
dma.recvchannel.transfer(out_buffer) #wait for data to appear
dma.sendchannel.wait()
dma.recvchannel.wait()
```

Week 4: Part 1

- AXI-fy your FIR filter
- Compare its speed to scipy on the Pynq board directly by sending data down to PL and then back up using DMA



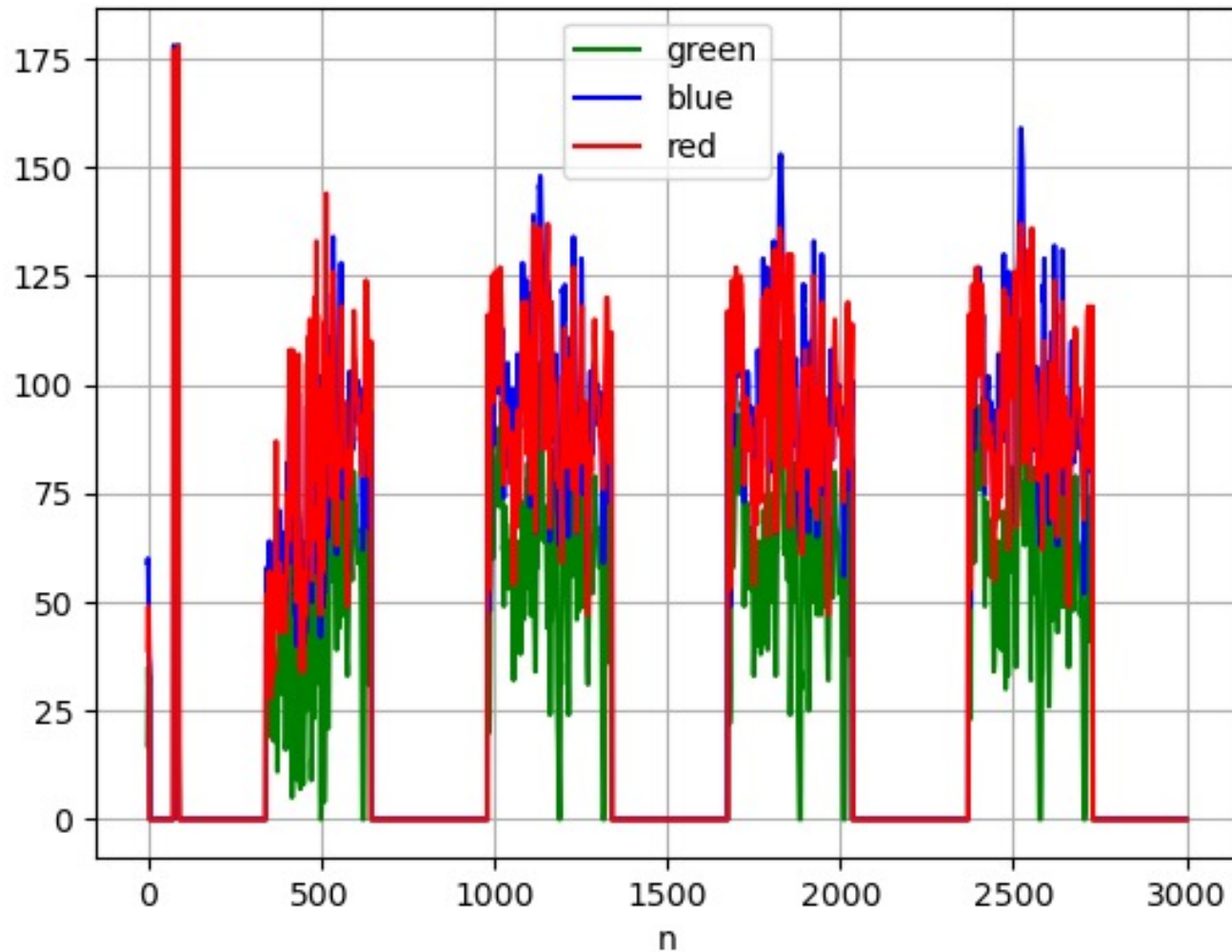
Week 4: Part 2

- Allocate a spot in memory and then write lines of video to it from the PL

```
n = 65536
out_buffer = allocate(shape=(n,), dtype=np.int32)
# Copy the samples to the in_buffer
# Trigger the DMA transfer and wait for the result
dma.recvchannel.transfer(out_buffer) #wait for data to appear
dma.recvchannel.wait()
```

Week 4: Part 2

Individual lines of video data
plotted in python:



More Cocotb

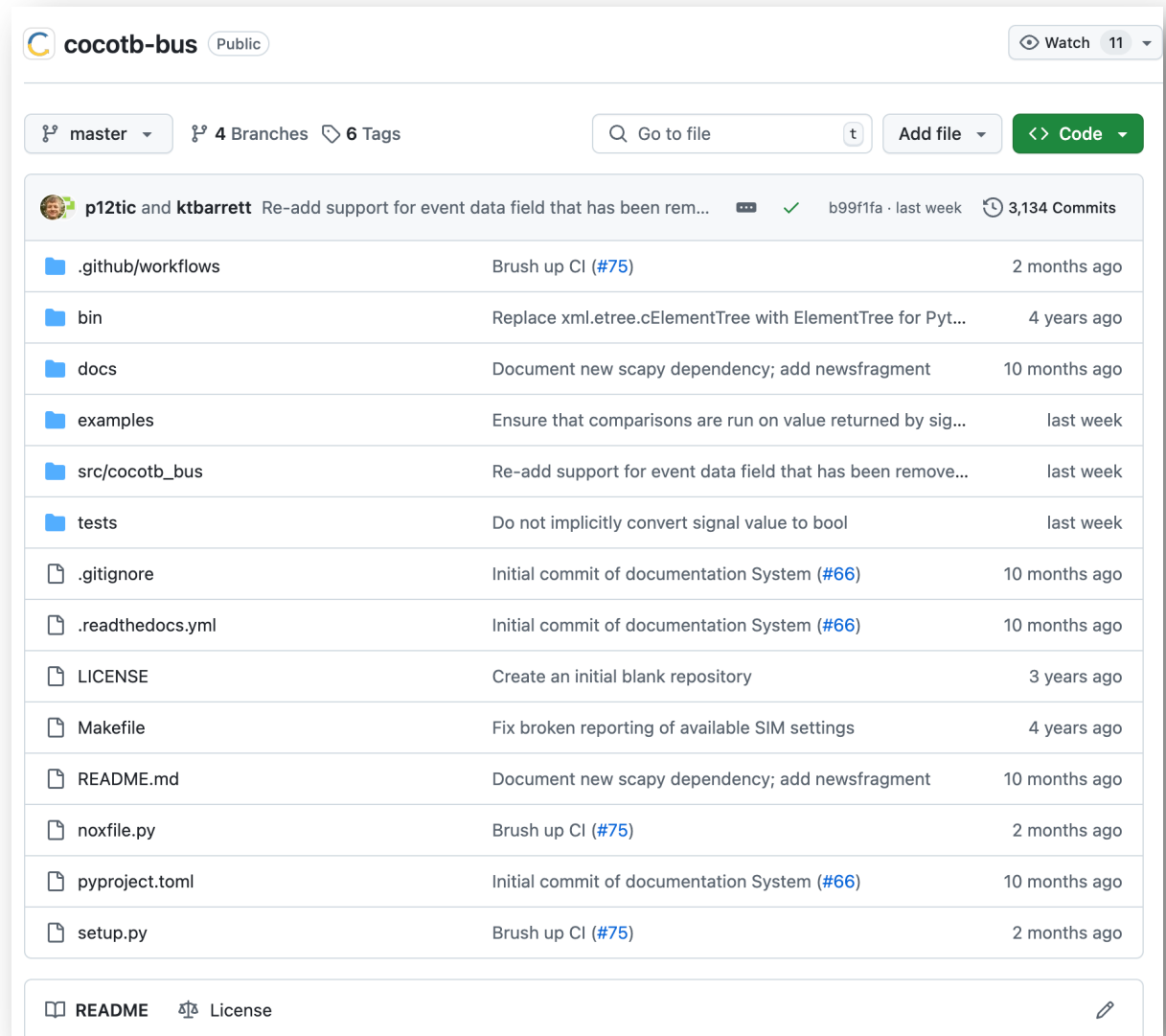
I'm coocoo for Cocotb

Adding Layers to Cocotb

- So we've been kinda building up some loose testing modules in Python using Cocotb.
- What we'd like to do is start to add some structure and reusability to this.
- To help with this, we'll start using the `cocotb_bus` library

cocotb_bus

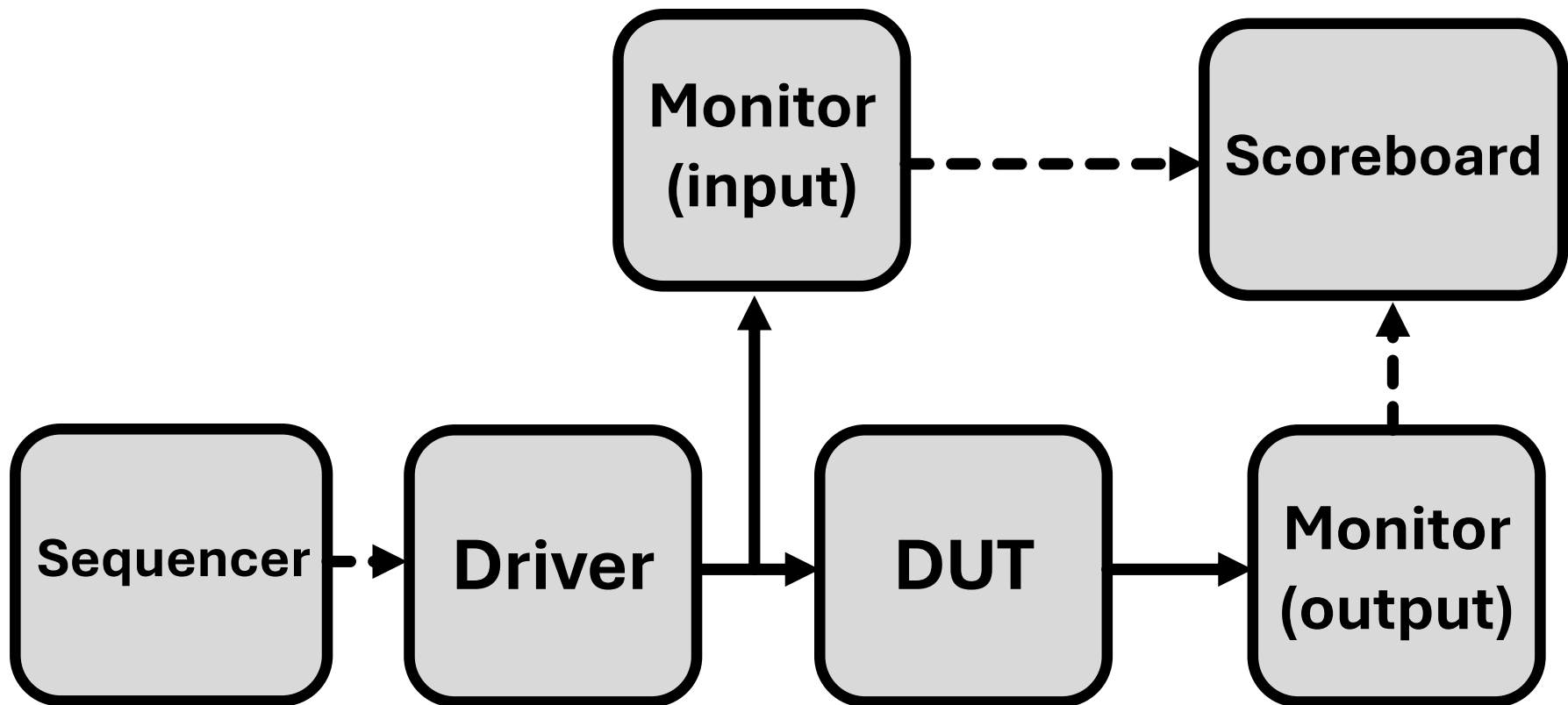
- I think this was originally part of cocotb but was split off
- Not sure why. I don't think it was a bad thing like happened with Rust or Node or RethinkDB



OOPiness Ahead

- Part of the job of any verification framework is to organize.
- And one of the (few) strengths of Object Oriented Programming is organization.
- Cocotb_bus is very OOPy just like UVM in SystemVerilog is very OOPy.
- The intention is to minimize code reuse and make things portable.

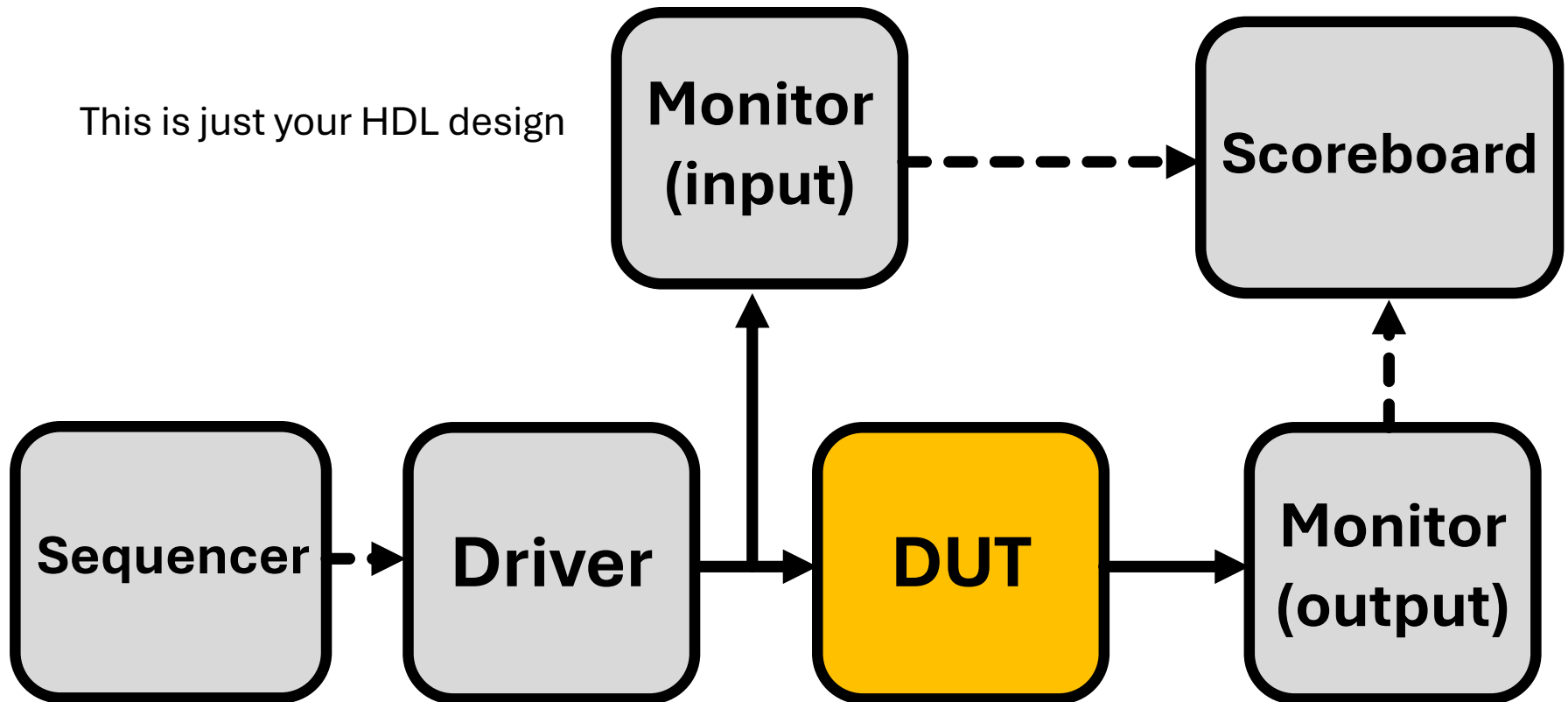
Standard Testing Framework



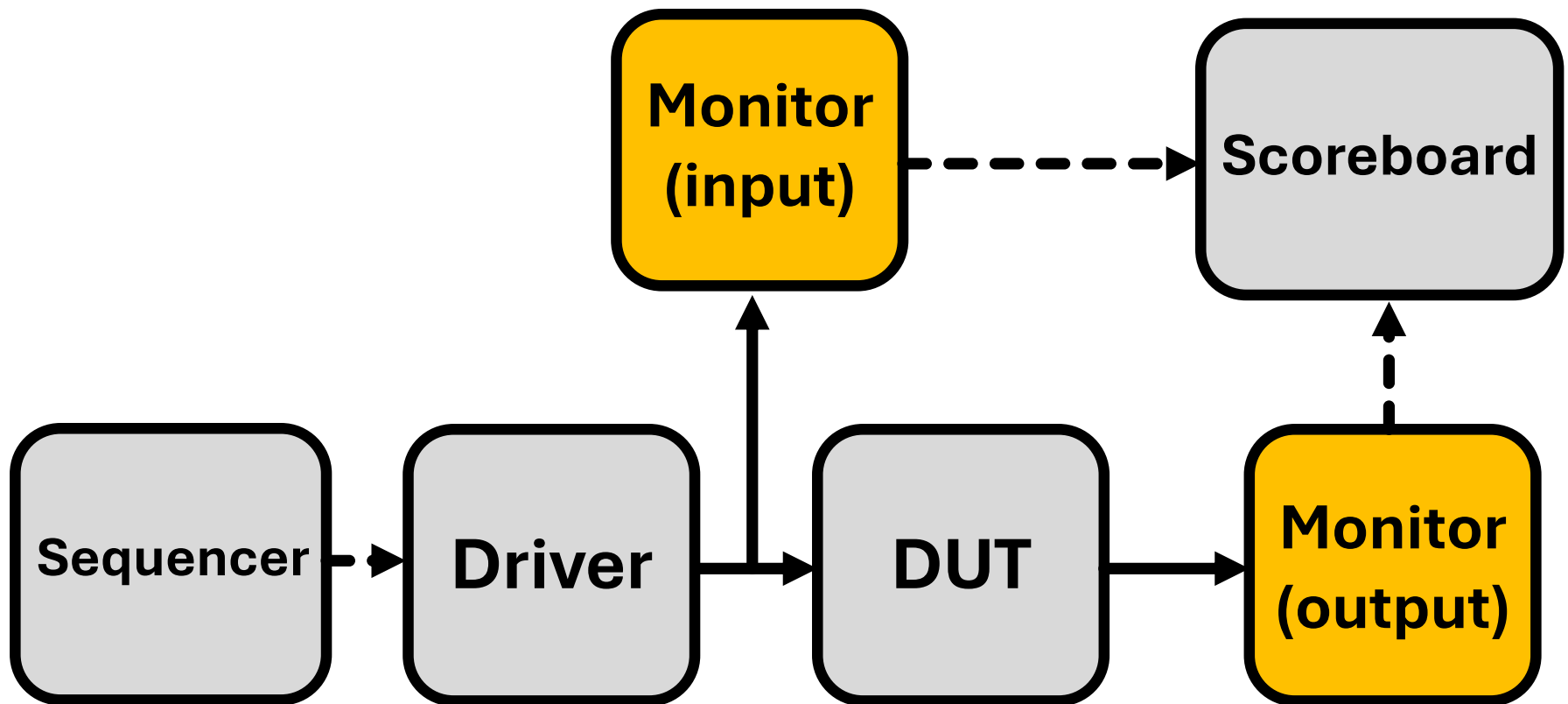
DUT

Design unit under test

This is just your HDL design



Monitors



Monitor/Bus Monitor

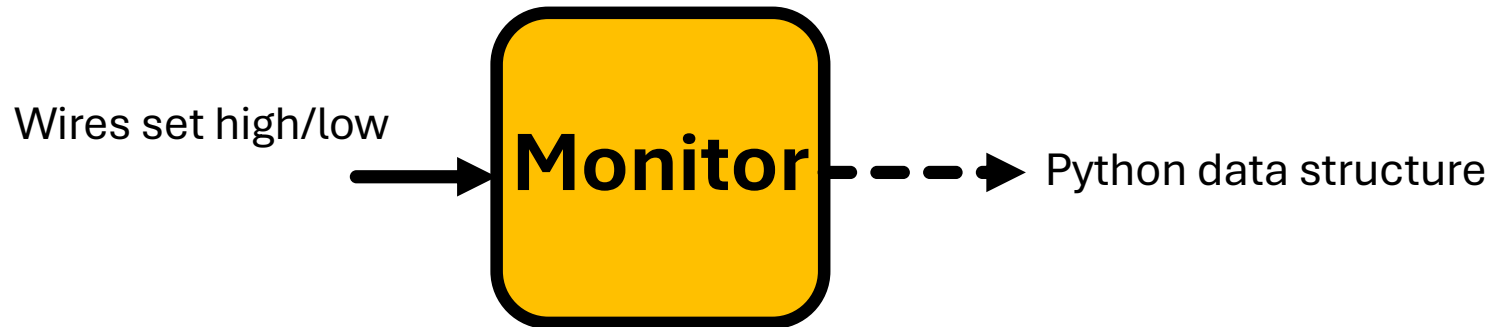
From cocotb source

```
30  ✓ class Monitor:
31      """Base class for Monitor objects.
32
33      Monitors are passive 'listening' objects that monitor pins going in or out of a DUT.
34      This class should not be used directly,
35      but should be sub-classed and the internal :meth:`_monitor_recv` method should be override
36      This :meth:`_monitor_recv` method should capture some behavior of the pins, form a transact
37      and pass this transaction to the internal :meth:`_recv` method.
38      The :meth:`_monitor_recv` method is added to the cocotb scheduler during the ``__init__`` p
39      so it should not be awaited anywhere.
40
41      The primary use of a Monitor is as an interface for a :class:`~cocotb.scoreboard.Scoreboard
42
43      Args:
44          callback (callable): Callback to be called with each recovered transaction
45                              as the argument. If the callback isn't used, received transactions will
46                              be placed on a queue and the event used to notify any consumers.
47          event (cocotb.triggers.Event): Event that will be called when a transaction
48                                         is received through the internal :meth:`_recv` method.
49                                         `Event.data` is set to the received transaction.
50      """
```

Monitors

- Monitors should listen to signals on a bus and log transactions as they detect them for processing by another party.
- A single Monitor is relatively useless
- Multiple monitors, however can generate lists of transactions and together these can be used to assess what the DUT is generating.

At a high level...



- Monitors should be completely passive entities that simply report what they see and not affect the system
- Keeps it simple

AXIS Monitor

```
14 class AXISMonitor(BusMonitor):
15     """
16     monitors axi streaming bus
17     """
18     transactions = 0
19     def __init__(self, dut, name, clk):
20         self._signals = ['axis_tvalid', 'axis_tready', 'axis_tlast', 'axis_tdata', 'axis_tstrb']
21         BusMonitor.__init__(self, dut, name, clk)
22         self.clock = clk
23         self.transactions = 0
24     async def _monitor_rcv(self):
25         """
26         Monitor receiver
27         """
28         rising_edge = RisingEdge(self.clock) # make these coroutines once and reuse
29         falling_edge = FallingEdge(self.clock)
30         read_only = ReadOnly() #This is
31         while True:
32             await rising_edge
33             await falling_edge #sometimes see in AXI shit
34             await read_only #readonly (the postline)
35             valid = self.bus.axis_tvalid.value
36             ready = self.bus.axis_tready.value
37             last = self.bus.axis_tlast.value
38             data = self.bus.axis_tdata.value #.signed_integer
39             if valid and ready:
40                 self.transactions+=1
41                 thing = dict(data=data, last=last, name=self.name, count=self.transactions, time=gs
42                 print(thing)
43                 self._rcv(thing)
```

- **Start to develop this week!**
- Monitors an AXIS bus
- If a valid/ready transaction occurs, do something with it.

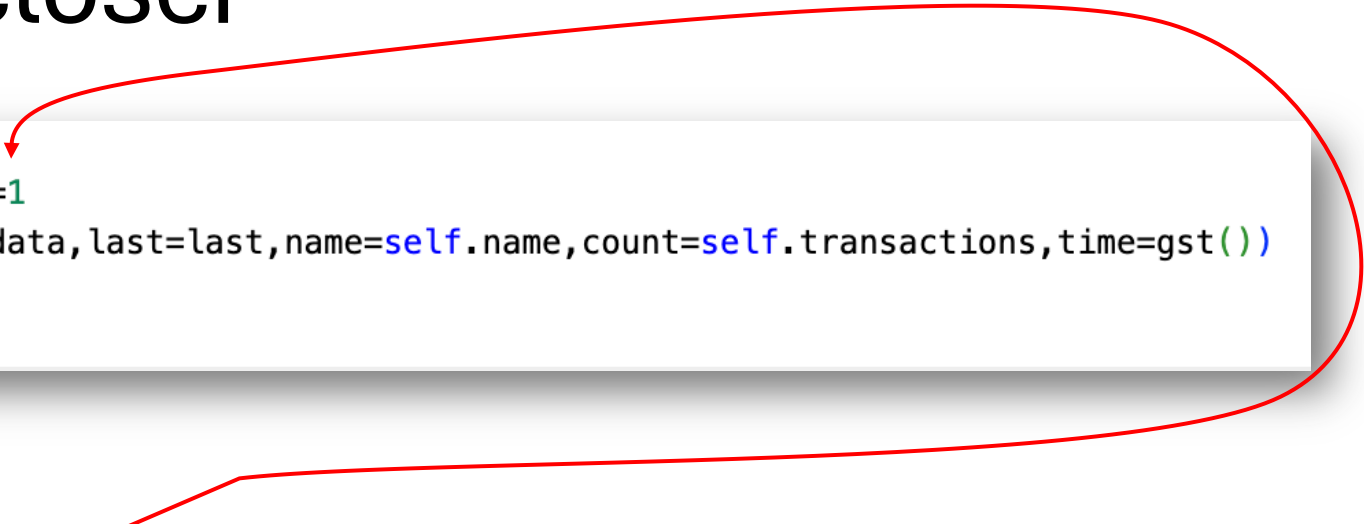
AXIS Monitor

```
24     async def _monitor_recv(self):
25         """
26         Monitor receiver
27         """
28         rising_edge = RisingEdge(self.clock) # make these coroutines once and reuse
29         falling_edge = FallingEdge(self.clock)
30         read_only = ReadOnly() #This is
31         while True:
32             await rising_edge
33             await falling_edge #sometimes see in AXI shit
34             await read_only #readonly (the postline)
35             valid = self.bus.axis_tvalid.value
36             ready = self.bus.axis_tready.value
37             last = self.bus.axis_tlast.value
38             data = self.bus.axis_tdata.value #.signed_integer
39             if valid and ready:
40                 self.transactions+=1
41                 thing = dict(data=data, last=last, name=self.name, count=self.transactions, time=gs
42                 print(thing)
43                 self._recv(thing)
```

Notice it isn't concerned with setting the values or not...it just watches the bus as a separate party

A little closer

```
if valid and ready:
    self.transactions+=1
    thing = dict(data=data, last=last, name=self.name, count=self.transactions, time=gst())
    print(thing)
    self._recv(thing)
```



- Make a internal variable to keep track of the number of valid/ready things we saw happen on the bus
- Then created a Python data structure from it (for easy human interpretability) and reported it.

The `_recv` method does a few things:

`__init__` of Monitor Class:

```
def __init__(self, callback=None, event=None):
    self._event = event
    if self._event is not None:
        self._event.data = None # FIXME: This
    self._wait_event = Event()
    self._wait_event.data = None
    self._recvQ = deque()
    self._callbacks = []
```

```
126 ✓ def _recv(self, transaction):
127     """Common handling of a received transaction."""
128
129     self.stats.received_transactions += 1
130
131     # either callback based consumer
132     for callback in self._callbacks:
133         callback(transaction)
134
135     # Or queued with a notification
136     if not self._callbacks:
137         self._recvQ.append(transaction)
138
139     if self._event is not None:
140         set_event(self._event, transaction)
141
142     # If anyone was waiting then let them know
143     if self._wait_event is not None:
144         set_event(self._wait_event, transaction)
145         self._wait_event.clear()
```

Call function with transaction!

Or shove it into a queue

Or trigger various events

What would the monitor/callbacks be useful for?

- maybe printing stuff and verify by eye

```
{'data': 000000000000000000010100000110011, 'last': 0, 'name': 'm00', 'count': 150, 'time': 5105000}
{'data': 000000000000000000000000000001100011, 'last': 1, 'name': 's00', 'count': 150, 'time': 5115000}
{'data': 000000000000000000000000010100000110110, 'last': 0, 'name': 'm00', 'count': 151, 'time': 5115000}
{'data': 000000000000000000000000010100000111001, 'last': 1, 'name': 'm00', 'count': 152, 'time': 5125000}
```

- But it would be nice to verify the data more robustly, for example... That's where maybe a callback could come in

One useful callback might be a model

- You wrote a model in week 1 when “verifying” that crappy divider I gave you in week 1
- A callback to a model might be useful if attached to an input monitor.
- Every time an input to DUT is observed, you trigger the model to compute what to expect of it.

With these modifications...

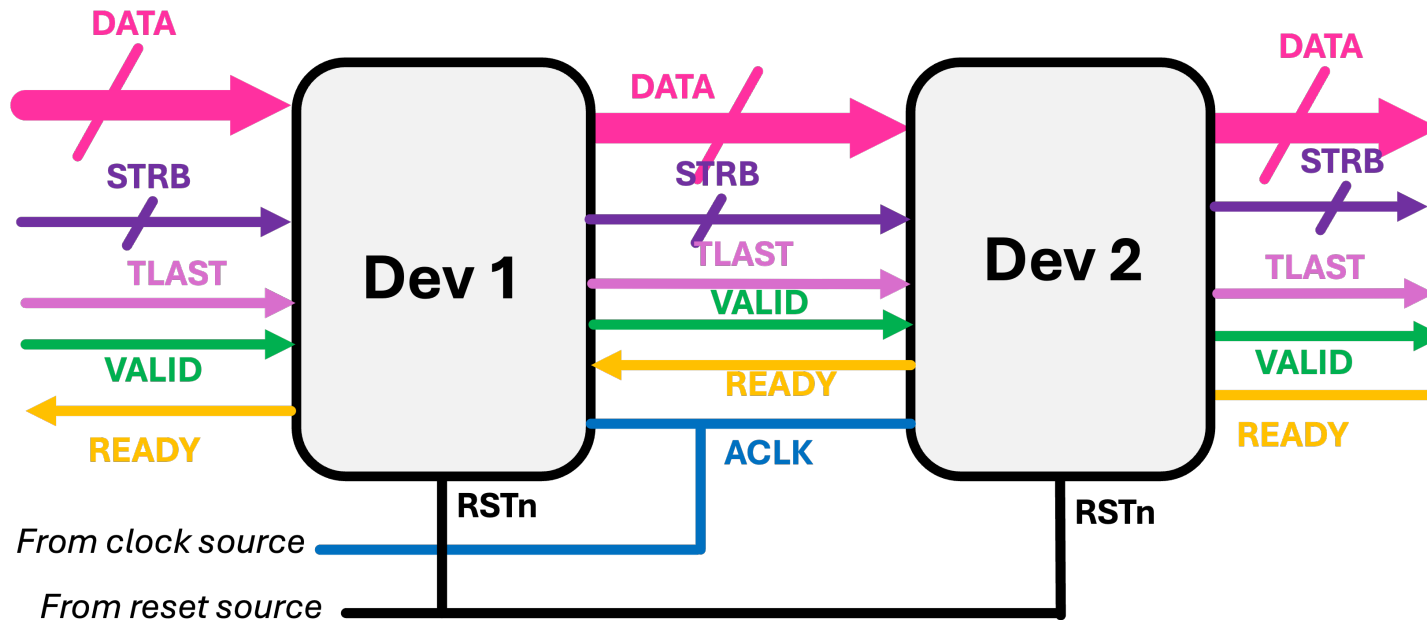
Proven software model

callback

```
65
66  mq = [] #list for holding output of model
67
68  def model(transaction):
69      #val = transaction.get('data')
70      val = transaction
71      print(val)
72      mq.append(3*val+10000) #gold standard model
73
74
75  @cocotb.test()
76  async def test_a(dut):
77
78      inm = axismonitor(dut, 's00', dut.s00_axis_aclk, callback=model)
79      outm = axismonitor(dut, 'm00', dut.s00_axis_aclk)
```

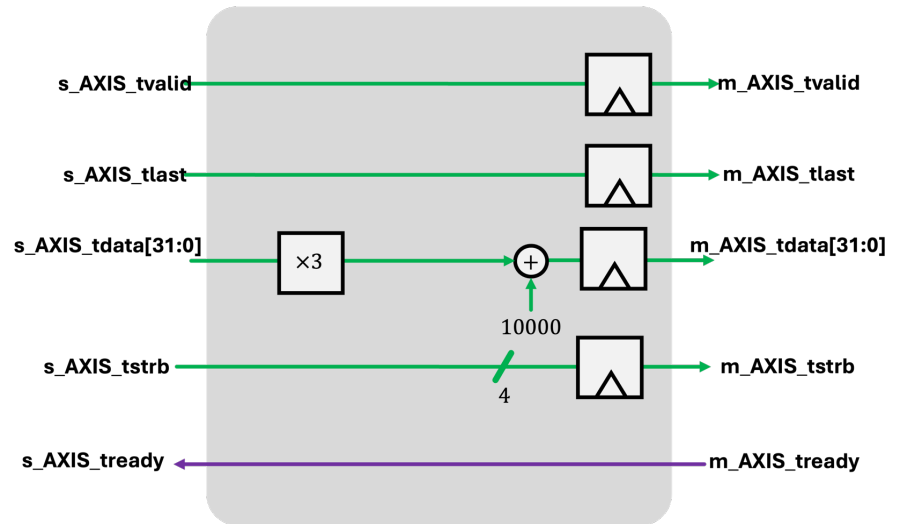
Code Reusability!

- You'll be building a few AXI-Streaming modules this week



AXI Streamers

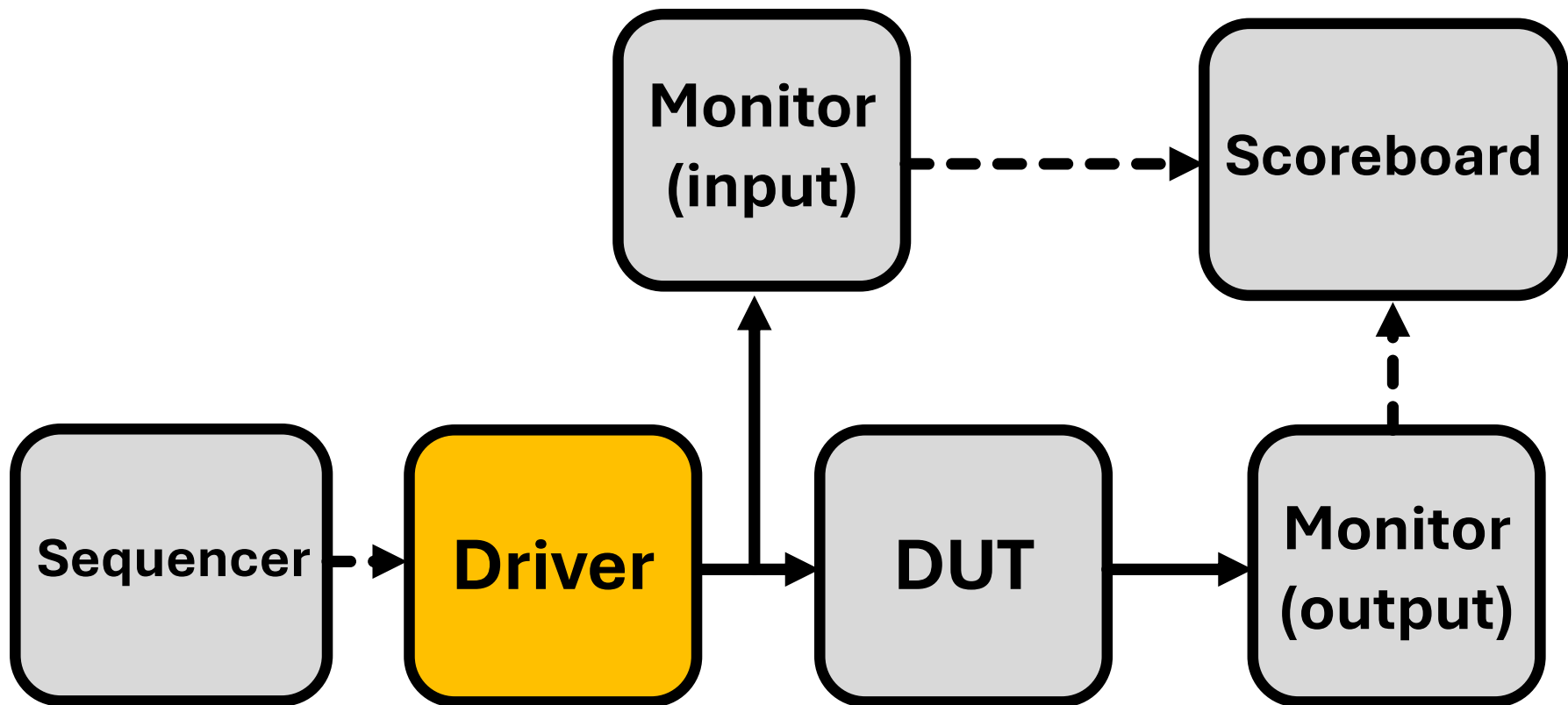
- Many of our modules will start to have multiple standardized busses
- Being able to reuse the monitors will be super nice.



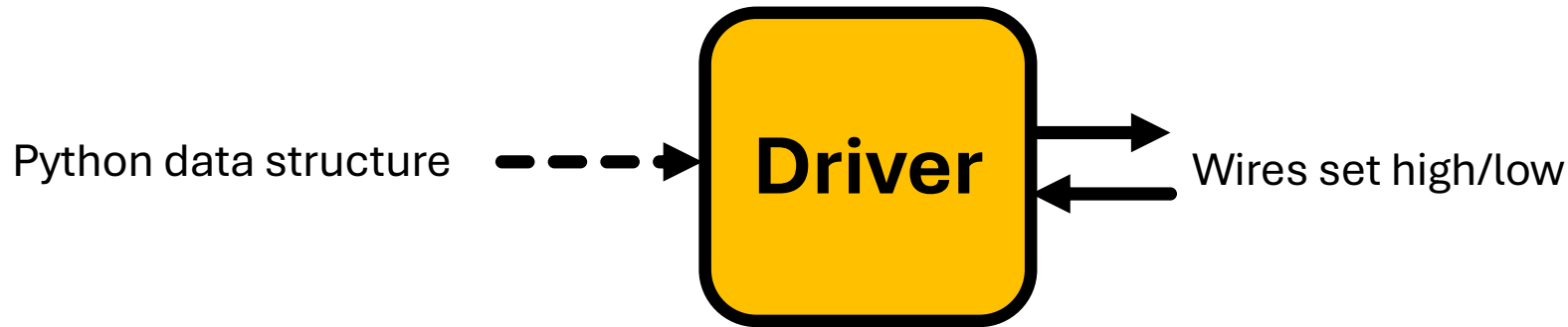
```

75 @cocotb.test()
76 async def test_a(dut):
77
78     inm = axismonitor(dut, 's00', dut.s00_axis_aclk, callback=model)
79     outm = axismonitor(dut, 'm00', dut.s00_axis_aclk)
  
```

Standard Testing Framework



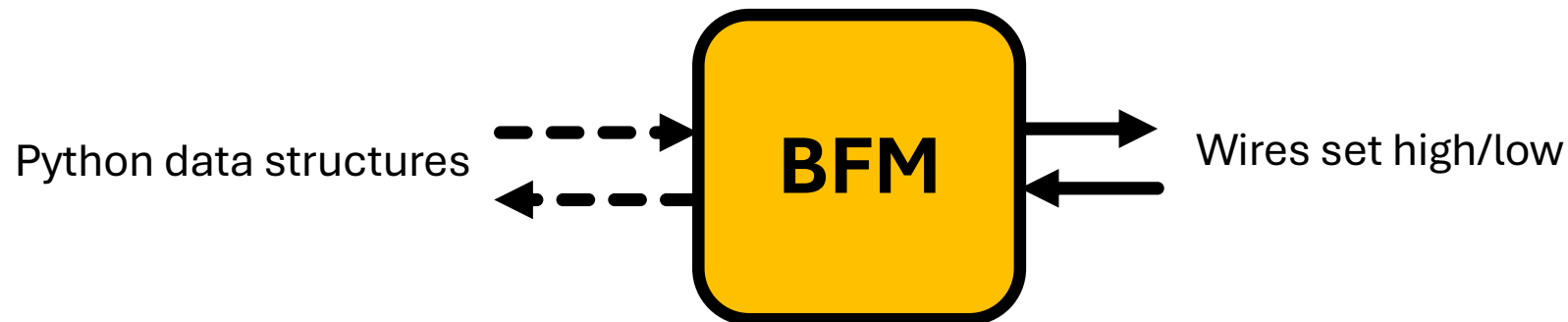
At a high level...



- Ideally, we want to just be able to issue high-level commands of things to send and not have to worry about:
 - Turning signals on/off
 - Waiting for signals to be on/off (e.g. READY)
- Drivers are almost always more complicated than Monitors

Bus Functional Model: “BFM”

- You will find this term thrown around a lot and it is kinda/basically the same thing we’re doing here.
- A BFM is a programming construct that allows interfacing of testing languages/frameworks to work with the simulated digital designs



Drivers

- Several different classes and subclasses

[cocotb-bus / src / cocotb_bus / drivers / __init__.py](#) 

```
21
22  ✓ class BitDriver:
23     """Drives a signal onto a single bit.
24
25     Useful for exercising ready/valid flags.
26     """
27
```

```
72
73  ✓ class Driver:
74     """Class defining the standard interface for a driver within a testbench.
75
76     The driver is responsible for serializing transactions onto the physical
77     pins of the interface. This may consume simulation time.
78     """
```

```
205
206  ✓ class BusDriver(Driver):
207     """Wrapper around common functionality for buses which have:
208
209     * a list of :attr:`_signals` (class attribute)
210     * a list of :attr:`_optional_signals` (class attribute)
211     * a clock
212     * a name
```

```
286
287  ✓ class ValidatedBusDriver(BusDriver):
288     """Same as a :class:`BusDriver` except we support an optional generator
289     to control which cycles are valid.
290
```

The Driver Base Class

```
72
73 ✓ class Driver:
74     """Class defining the standard interface for a driver within a testbench.
75
76     The driver is responsible for serializing transactions onto the physical
77     pins of the interface. This may consume simulation time.
78     """
79
80 ✓ def __init__(self):
81     """Constructor for a driver instance."""
82     self._pending = Event(name="Driver._pending")
83     self._sendQ = deque()
84     self.busy_event = Event("Driver._busy")
85     self.busy = False
86
87     # Sub-classes may already set up logging
88     if not hasattr(self, "log"):
89         self.log = logging.getLogger("cocotb.driver.%s" % (type(self).__qualname__))
90
91     # Create an independent coroutine which can send stuff
92     self._thread = cocotb.start_soon(self._send_thread())
93
94 ✓ async def _acquire_lock(self):
95     if self.busy:
96         await self.busy_event.wait()
97     self.busy_event.clear()
98     self.busy = True
99
```

Drivers

- In week 4's stuff we're creating (and then using) a Bus Driver that you write like the following:

```
ind = AXISDriver(dut, 's00', dut.s00_axis_aclk)

for i in range(50):
    data = {'type': 'single', "contents": {"data": random.randint(1, 255), "last": 0, "strb": 15}}
    ind.append(data)

data = {'type': 'burst', "contents": {"data": np.array(list(range(100)))}}
ind.append(data)
```

- What is this “append” method?

The append method

```
110  def append(  
111      self, transaction: Any, callback: Callable[[Any], Any] = None,  
112      event: Event = None, **kwargs: Any  
113  ) -> None:  
114      """Queue up a transaction to be sent over the bus.  
115  
116      Mechanisms are provided to permit the caller to know when the  
117      transaction is processed.  
118  
119      Args:  
120          transaction: The transaction to be sent.  
121          callback: Optional function to be called  
122                  when the transaction has been sent.  
123          event: :class:`~cocotb.triggers.Event` to be set  
124                when the transaction has been sent.  
125          **kwargs: Any additional arguments used in child class'  
126                   :any:`_driver_send` method.  
127      """  
128      self._sendQ.append((transaction, callback, event, kwargs))  
129      self._pending.set()
```


What is self._sendQ ?

```
72
73 ✓ class Driver:
74     """Class defining the standard interface for a driver within a testbench.
75
76     The driver is responsible for serializing transactions onto the physical
77     pins of the interface. This may consume simulation time.
78     """
79
80 ✓ def __init__(self):
81     """Constructor for a driver instance."""
82     self._pending = Event(name="Driver._pending")
83     self._sendQ = deque()
84     self._busy_event = Event("Driver._busy")
85     self._busy = False
86
87     # Sub-classes may already set up logging
88     if not hasattr(self, "log"):
89         self.log = logging.getLogger("cocotb.driver.%s" % (type(self).__qualname__))
90
91     # Create an independent coroutine which can send stuff
92     self._thread = cocotb.start_soon(self._send_thread())
93
```

Double Ended Queue

geeksforgeeks.org/deque-in-python/

Courses ▾Tutorials ▾Jobs ▾Practice ▾Contests ▾

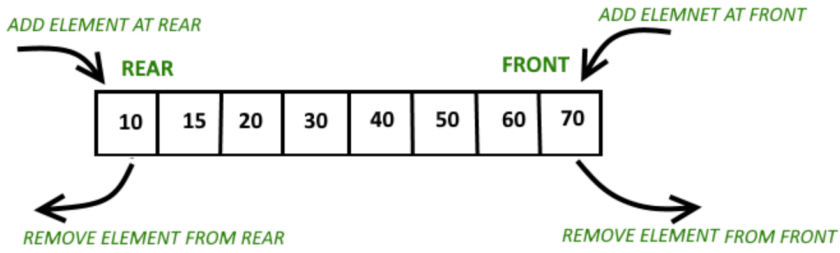


Python CoursePython BasicsInterview QuestionsPython QuizPopular PackagesPython ProjectsPractice PythonAI With PythonLearn Python3Python Automate

Deque in Python

Last Updated : 20 Jun, 2024

Deque (Doubly Ended Queue) in [Python](#) is implemented using the [module "collections"](#). Deque is preferred over a [list](#) in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an **O(1)** time complexity for append and pop operations as compared to a list that provides O(n) time complexity.



The diagram illustrates a deque as a horizontal array of eight boxes containing the numbers 10, 15, 20, 30, 40, 50, 60, and 70. Above the first box (10), the word 'REAR' is written in green, with an arrow pointing to it from the text 'ADD ELEMENT AT REAR' above. Above the last box (70), the word 'FRONT' is written in green, with an arrow pointing to it from the text 'ADD ELEMNET AT FRONT' above. Below the first box (10), an arrow points away from it towards the left, labeled 'REMOVE ELEMENT FROM REAR'. Below the last box (70), an arrow points away from it towards the right, labeled 'REMOVE ELEMENT FROM FRONT'.

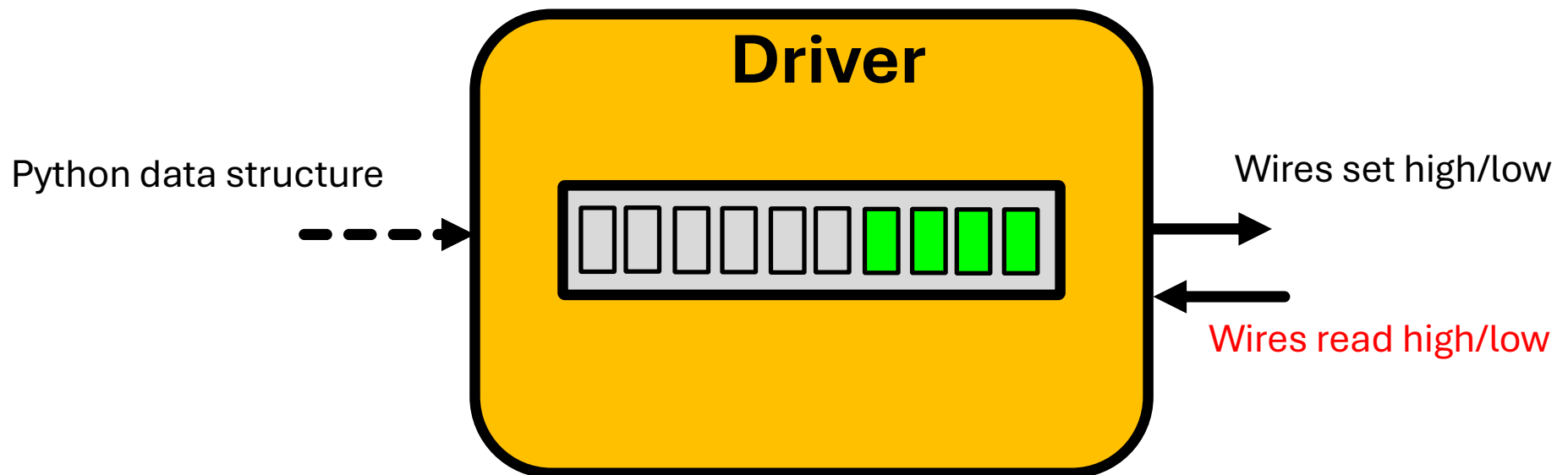
Types of Restricted Deque Input

- **Input Restricted Deque:** Input is limited at one end while deletion is permitted at both ends.
- **Output Restricted Deque:** output is limited at one end but insertion is permitted at both ends.

Example: Python code to demonstrate deque

An internal FIFO of things to do

- Just like in hardware a FIFO/queue allows breathing room and a decoupling of commands from implementation

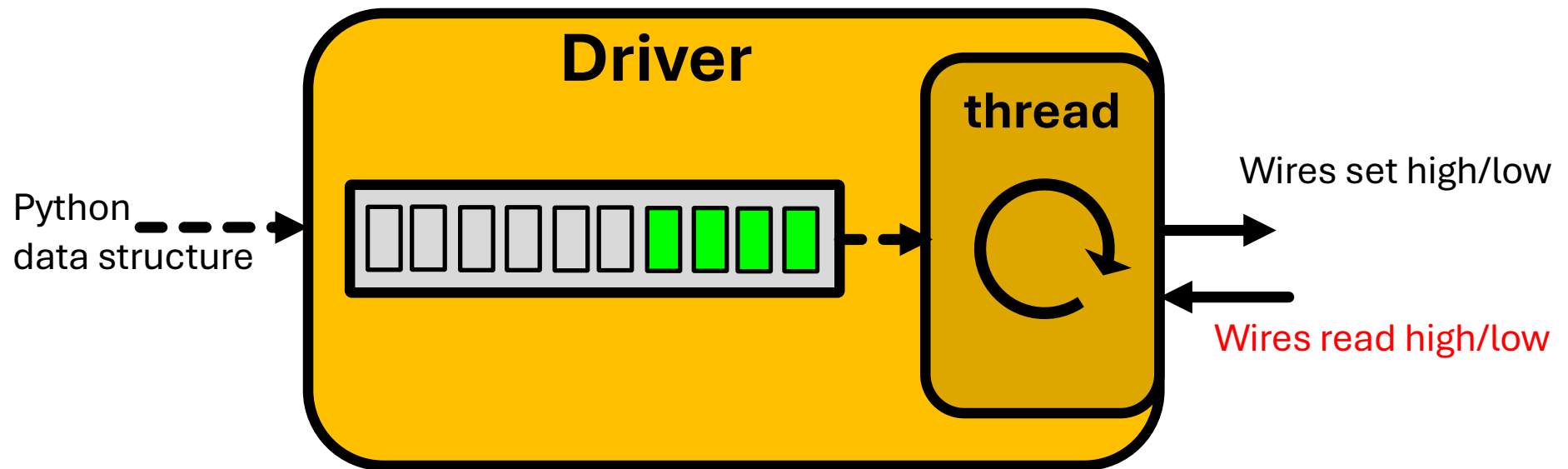


Launches the running process

```
72
73 ∨ class Driver:
74     """Class defining the standard interface for a driver within a testbench.
75
76     The driver is responsible for serializing transactions onto the physical
77     pins of the interface. This may consume simulation time.
78     """
79
80 ∨ def __init__(self):
81     """Constructor for a driver instance."""
82     self._pending = Event(name="Driver._pending")
83     self._sendQ = deque()
84     self.busy_event = Event("Driver._busy")
85     self.busy = False
86
87     # Sub-classes may already set up logging
88     if not hasattr(self, "log"):
89         self.log = logging.getLogger("cocotb.driver.%s" % (type(self).__qualname__))
90
91     # Create an independent coroutine which can send stuff
92     self._thread = cocotb.start_soon(self._send_thread())
93
```

An internal FIFO of things to do

- Just like in hardware a FIFO/queue allows breathing room and a decoupling of commands from implementation



`_send_thread` coroutine

- Tracks the queue...if stuff in it...calls the `_send` procedure

```
185
186  ✓  async def _send_thread(self):
187      while True:
188
189          # Sleep until we have something to send
190          while not self._sendQ:
191              self._pending.clear()
192              await self._pending.wait()
193
194          synchronised = False
195
196          # Send in all the queued packets,
197          # only synchronize on the first send
198          while self._sendQ:
199              transaction, callback, event, kwargs = self._sendQ.popleft()
200              self.log.debug("Sending queued packet...")
201              await self._send(transaction, callback, event,
202                              sync=not synchronised, **kwargs)
203              synchronised = True
204
```

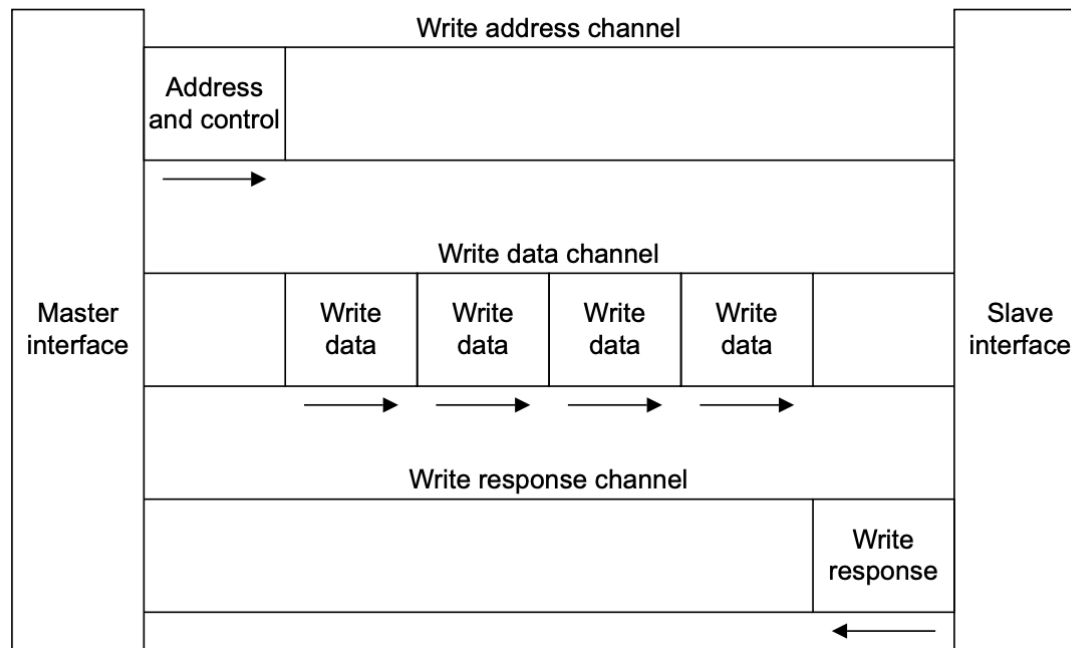
_send

- Finally this is getting to the `_driver_send` procedure which is a thing you'll need to write

*Notice this is awaiting it
And actually the last few
pages have been awaits*

```
163  ✓  async def _send(  
164      self, transaction: Any, callback: Callable[[Any], Any], event: Event,  
165      sync: bool = True, **kwargs  
166  ) -> None:  
167      """Send coroutine.  
168  
169      Args:  
170          transaction: The transaction to be sent.  
171          callback: Optional function to be called  
172                  when the transaction has been sent.  
173          event: event to be set when the transaction has been sent.  
174          sync: Synchronize the transfer by waiting for a rising edge.  
175          **kwargs: Any additional arguments used in child class'  
176                  :any: driver_send method.  
177      """  
178      await self._driver_send(transaction, sync=sync, **kwargs)  
179  
180      # Notify the world that this transaction is complete  
181      if event:  
182          event.set()  
183      if callback:  
184          callback(transaction)  
185
```

Usefulness? Think of the AXI LITE

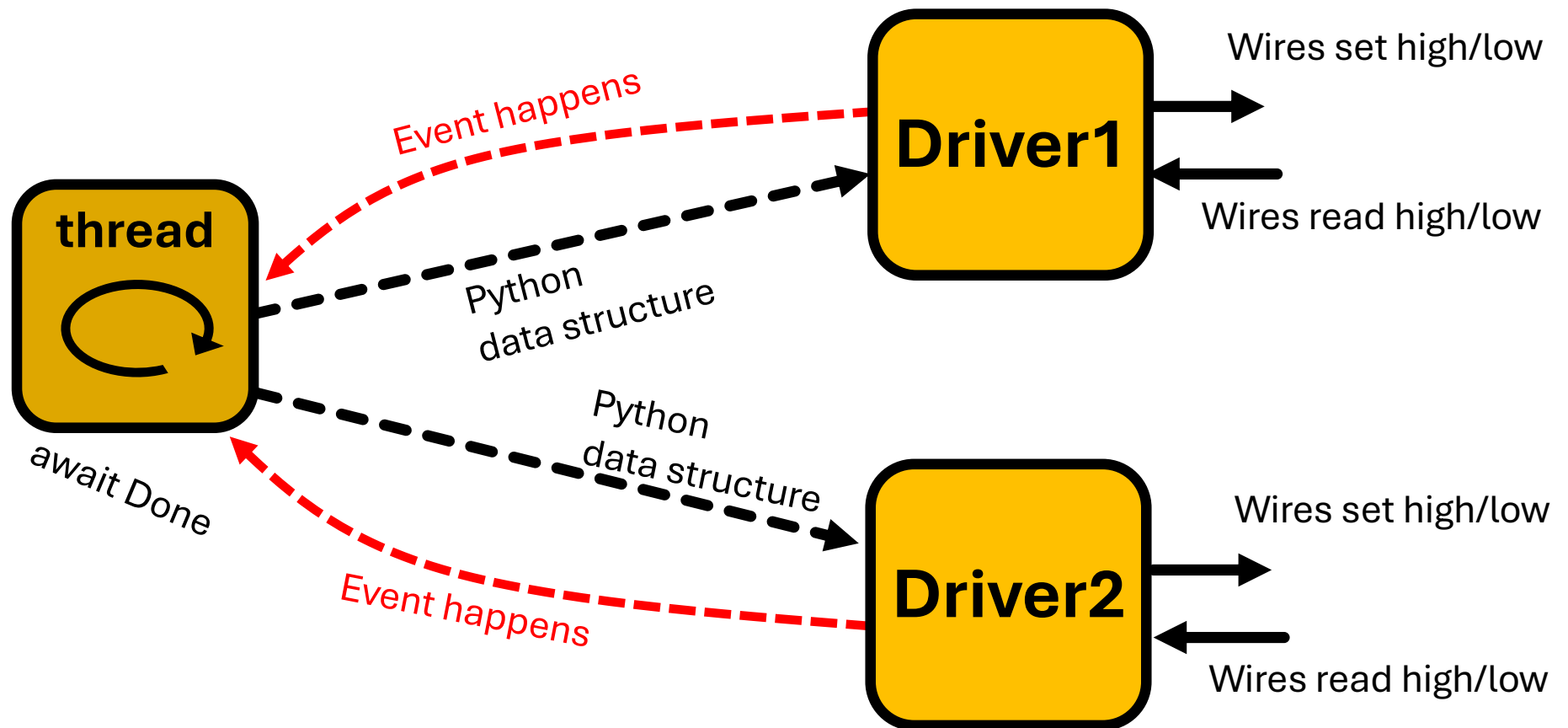


- One interface actually has three separate busses in it. Have Driver for Each Bus, but need to sync them...

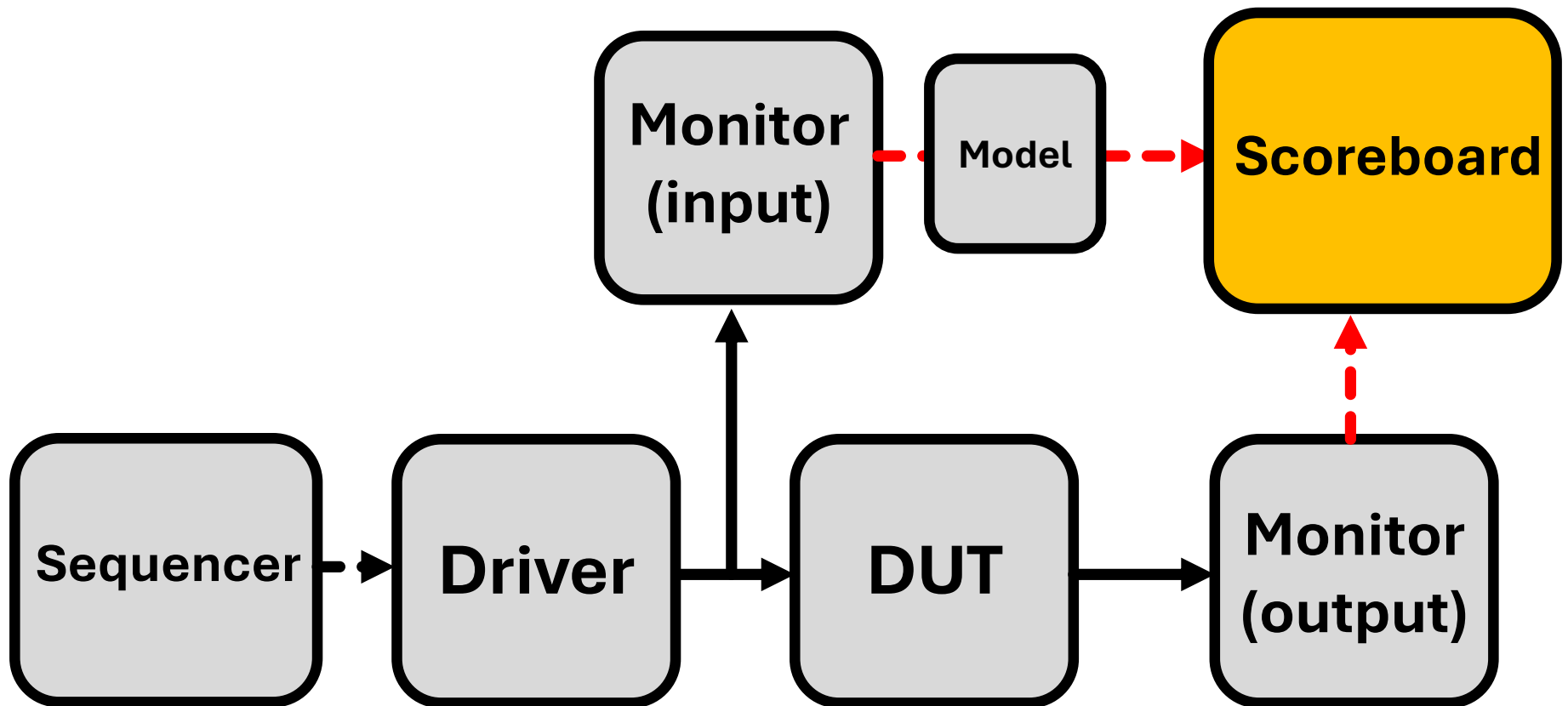
Events

Python data structure   Wires set high/low

- Events/Triggers could allow Driver 1 to only send after Driver 2 sent or vice versa or whatever



Plug an entity in to Compare these results



ScoreBoard

- There's a scoreboarding class that is designed to work with data streams that monitors will produce

```
10
17  ✓ class Scoreboard:
18      """Generic scoreboarding class.
19
20      We can add interfaces by providing a monitor and an expected output queue.
21
22      The expected output can either be a function which provides a transaction
23      or a simple list containing the expected output.
24
25      TODO:
26          Statistics for end-of-test summary etc.
27
28      Args:
29          dut (SimHandle): Handle to the DUT.
30          reorder_depth (int, optional): Consider up to `reorder_depth` elements
31          of the expected result list as passing matches.
32          Default is 0, meaning only the first element in the expected result list
33          is considered for a passing match.
34          fail_immediately (bool, optional): Raise :exc:`AssertionError`
35          immediately when something is wrong instead of just
36          recording an error. Default is ``True``.
37      """
38
39  ✓ def __init__(self, dut, reorder_depth=0, fail_immediately=True): # FIXME: reorder_depth ne
40      self.dut = dut
41      self.log = logging.getLogger("cocotb.scoreboard.%s" % self.dut._name)
42      self.errors = 0
43      self.expected = {}
44      self._imm = fail_immediately
45
46      @property
47  ✓ def result(self):
48      """Determine the test result, do we have any pending data remaining?
49
50      Raises:
```

Make a scoreboard

```
@cocotb.test()
async def test_a(dut):
    """cocotb test for seven segment controller"""
    inm = AXISMonitor(dut, 's00', dut.s00_axis_aclk, callback=model)
    outm = AXISMonitor(dut, 'm00', dut.s00_axis_aclk)
    ind = AXISDriver(dut, 's00', dut.s00_axis_aclk)
    scoreboard = Scoreboard(dut)
    scoreboard.add_interface(outm, mq)
```

Scoreboard instance

Thing for it to check...

actual, expected

Scoreboard Class

- Has all the stuff running to check/compare the actual/expected pairs as they come in.
- Can also override the compare to do whatever you want...ranges, whatever

```
def compare(self, got, exp, log, strict_type=True):
    """Common function for comparing two transactions.

    Can be re-implemented by a sub-class.

    Args:
        got: The received transaction.
        exp: The expected transaction.
        log: The logger for reporting messages.
        strict_type (bool, optional): Require transaction type to match
            exactly if ``True``, otherwise compare its string representation.

    Raises:
        :exc:`AssertionError`: If received transaction differed from
            expected transaction when :attr:`fail_immediately` is ``True``.
            If *strict_type* is ``True``,
            also the transaction type must match.
    """
```

Scoreboard sees failure and tells you

```
/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py:118: DeprecationWarning: Use `bv.integer` instead.
self._recv(data.value)
35.00ns ERROR cocotb.scoreboard.j_math.m00 Received transaction differed from expected output
35.00ns INFO cocotb.scoreboard.j_math.m00 Expected:
20312
35.00ns INFO cocotb.scoreboard.j_math.m00 Received:
10312
/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/scoreboard.py:140: DeprecationWarning: cocotb.utils.hexdiffs is deprecated. Use scapy.utils.hexdiff instead.
log.warning("Difference:\n%s" % hexdiffs(strexp, strgot))
/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/scoreboard.py:140: DeprecationWarning: Passing strings to hexdiffs is deprecated, pass bytes instead
log.warning("Difference:\n%s" % hexdiffs(strexp, strgot))
35.00ns WARNING cocotb.scoreboard.j_math.m00 Difference:
0000 3230333132 20312
0000 3130333132 10312
/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/scoreboard.py:142: DeprecationWarning: TestFailure is deprecated, use an ``assert`` statement instead
raise TestFailure("Received transaction differed from expected ")
35.00ns INFO ..Task 1.AXISMonitor._monitor_recv Test stopped by this forked coroutine
35.00ns INFO cocotb.regression test a failed
Traceback (most recent call last):
  File "/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py", line 118, in _monitor_recv
    self._recv(data.value)
  File "/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/monitors/__init__.py", line 130, in _recv
    callback(transaction)
  File "/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/scoreboard.py", line 227, in check_received_transaction
    self.compare(transaction, exp, log, strict_type=strict_type)
  File "/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/scoreboard.py", line 142, in compare
    raise TestFailure("Received transaction differed from expected ")
cocotb.result.TestFailure: Received transaction differed from expected transaction
35.00ns INFO cocotb.regression
*****
** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** test_fir.test_a FAIL 35.00 0.00 12019.39 **
*****
** TESTS=1 PASS=0 FAIL=1 SKIP=0 35.00 0.04 843.29 **
*****
```

Scoreboard see no error and you're good

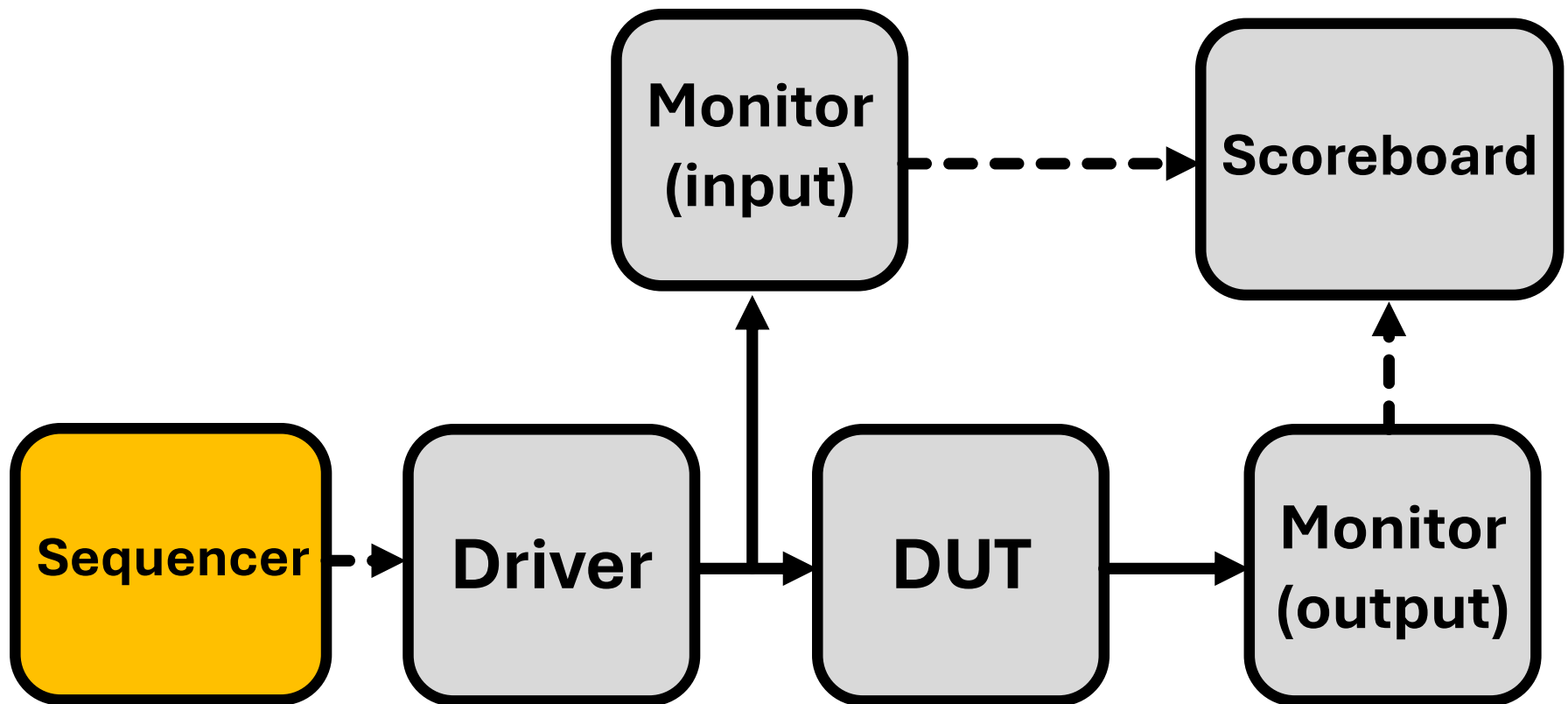
```

** test_fir.test_a                                FAIL                35.00                0.00                12019.39 **
*****
** TESTS=1 PASS=0 FAIL=1 SKIP=0                    35.00                0.04                843.29 **
*****

INFO: Results file: /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/results.xml
(6205_python) (base) DHCP-POOL-18-25-22-252:sin jodalyst$ python3 test_fir.py
/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py:12: UserWarning: Python runners and associated APIs are an experimental feature and subject to change.
  from cocotb.runner import get_runner
INFO: Running command iverilog -o /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/sin.vvp -D COCOTB_SIM=1 -s j_math -g2012 -Wall -s cocotb_iverilog_dump -f /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/cnds.f /Users/jodalyst/cocotb_development/fir_dev2/hdl/j_math.sv /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/cocotb_iverilog_dump.v in directory /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build
INFO: Running command vvp -M /Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb/libs -n libcocotbvpi_icarus /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/sin.vvp in directory /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build
--ns INFO      gpi                      ..nbed/gpi_embed.cpp:109   in set_program_name_in_venv      Using Python virtual environment interpreter at /Users/jodalyst/6205_python/bin/python
--ns INFO      gpi                      ../gpi/GpiCommon.cpp:101   in gpi_print_registered_impl  UPI registered
0.00ns INFO    cocotb                  Running on Icarus Verilog version 12.0 (stable)
0.00ns INFO    cocotb                  Running tests with cocotb v1.9.1 from /Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb
0.00ns INFO    cocotb                  Seeding Python random module with 1727293312
0.00ns INFO    cocotb.regression        pytest not found, install it to enable better AssertionError messages
/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py:12: UserWarning: Python runners and associated APIs are an experimental feature and subject to change.
  from cocotb.runner import get_runner
0.00ns INFO    cocotb.regression        Found test test_fir.test_a
0.00ns INFO    cocotb.regression        running test_a (1/1)
      cocotb test for seven segment controller
/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/monitors/__init__.py:67: DeprecationWarning: This method is now private.
  self._thread = cocotb.scheduler.add(self._monitor_recv())
/Users/jodalyst/6205_python/lib/python3.10/site-packages/cocotb_bus/drivers/__init__.py:92: DeprecationWarning: This method is now private.
  self._thread = cocotb.scheduler.add(self._send_thread())
0.00ns INFO    cocotb.scoreboard.j_math      Created with reorder_depth 0
VCD info: dumpfile /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/j_math.fst opened for output.
/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py:170: DeprecationWarning: Setting values on handles using the ``dut.handle = value`` syntax is deprecated. Instead use the ``handle.value = value`` syntax
  dut.n000_axis_tready = val
  dut.n000_axis_tready = val
/Users/jodalyst/cocotb_development/fir_dev2/sin/test_fir.py:118: DeprecationWarning: Use `bv.integer` instead.
  self._recv(data.value)
6720.00ns INFO    cocotb.regression        test_a passed
6720.00ns INFO    cocotb.regression        *****
** TEST                                     STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*****
** test_fir.test_a                          PASS                    6720.00                0.05                129322.58 **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0                    6720.00                0.09                72405.14 **
*****

INFO: Results file: /Users/jodalyst/cocotb_development/fir_dev2/sin/sin_build/results.xml
(6205_python) (base) DHCP-POOL-18-25-22-252:sin jodalyst$
```

Sequencer



Sequencer

- This is tied a bit more into *what* we test on the device so we'll cover it in the future.
- For now we'll be kinda kludging this part.

Tasks to Do

- Week 3 due on Friday 5pm
- Week 4 coming out Friday mid-day.
- That will likely be our last Pynq-board lab, then we'll move to the RFSoc in Week 5.