6.S965
Digital Systems Laboratory II

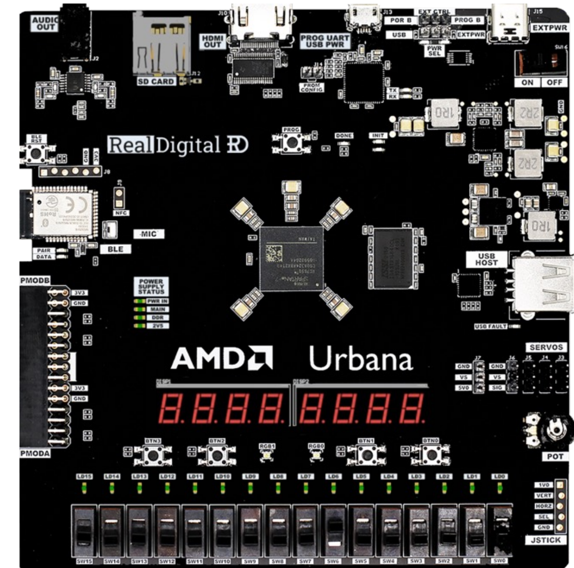
Lecture 5:

DMA,

Streaming AXI,

Monitors, Drivers, Scoreboards

6.205 FPGA



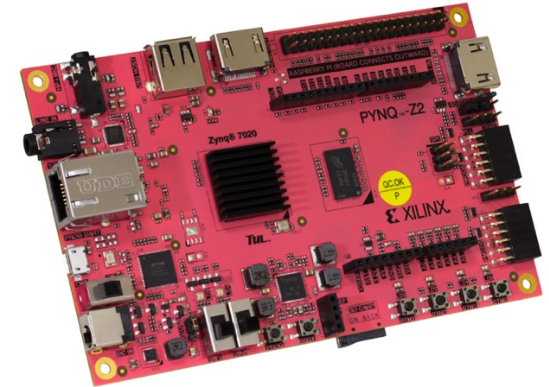
- Spartan 7 (xc7s50csga324-ish):
  - **2.7 Mb of BRAM**
  - 120 DSP slices
  - 52K logic cells*
- Dev Board also has 128 MB of DRAM

**"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

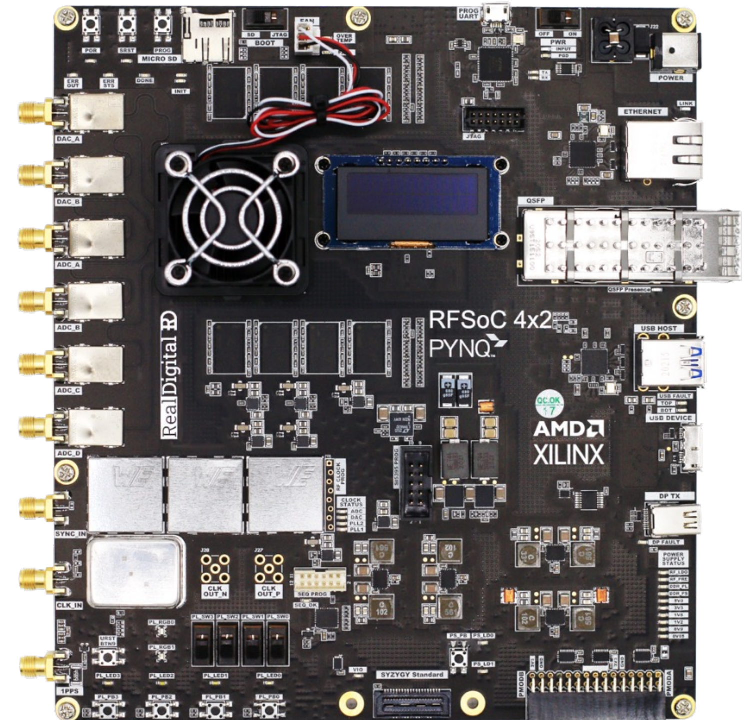
6.S965 Zynq 7000

- 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

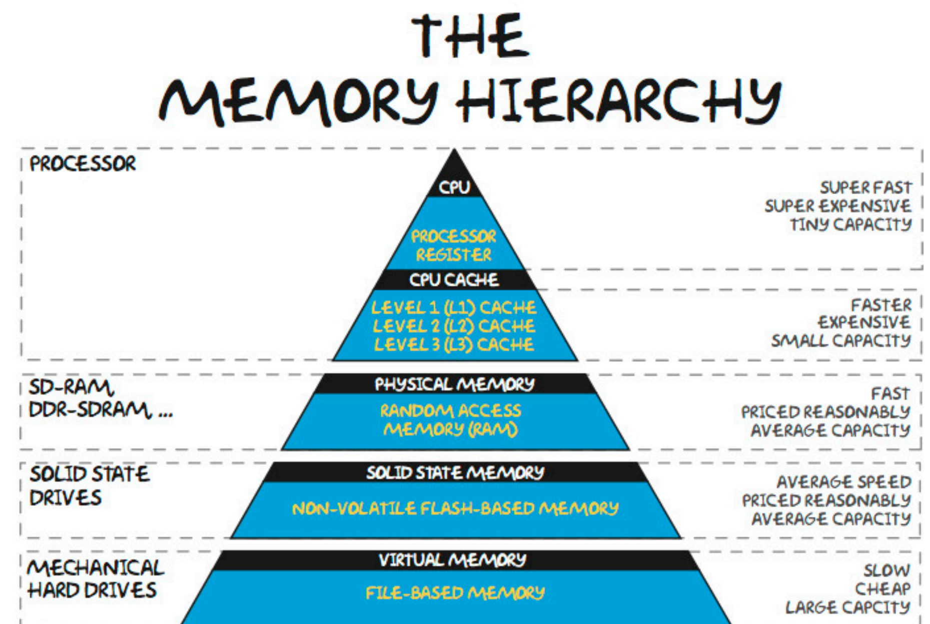


- UltraScale+ ZU48DR:
  - **38 Mb of BRAM**
  - **+22Mb of UltraRAM**
  - 4272 DSP slices
  - 930,000 Logic Cells
  - Four 5-Gsps 14 bit ADCs
  - Two 10-Gsps 14 bit DACs
  - Four 1.3 GHz ARM 53 processors
  - Two Real-time 533 MHz ARM processors
- Board has 4GB of DDR4 for FPGA portion (”PL”) and 4 GB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

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

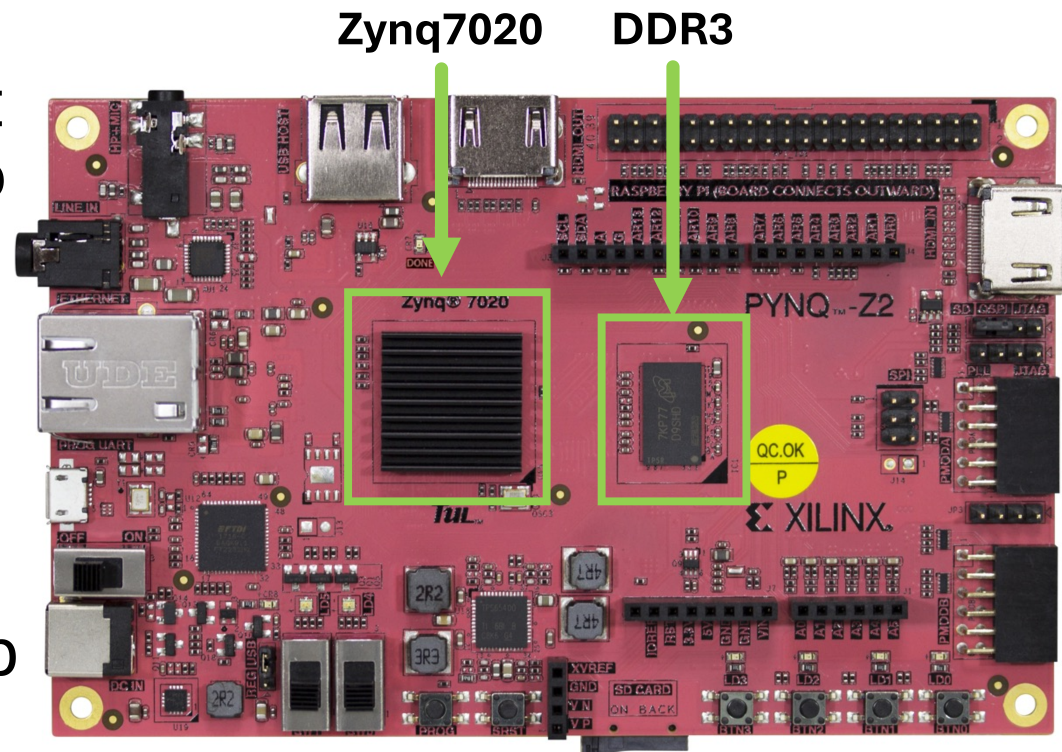
- Quick-to-access memory is desirable
  - BRAM
  - URAM
- As far as memories go they take up a lot of space
- To have more memory you have to go off-chip



https://computerscience.chemeketa.edu/cs160Reader/ComputerArchitecture/MemoryHeirarchy.html

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 devices 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

- This constant need for refreshing means getting info into and out of the DRAM is not an easy task…
- 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

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…we're hopefully going to write a controller for them in 6.205 this year since they are so slow

DRAM is pretty wild (aside)

~2024
2GB

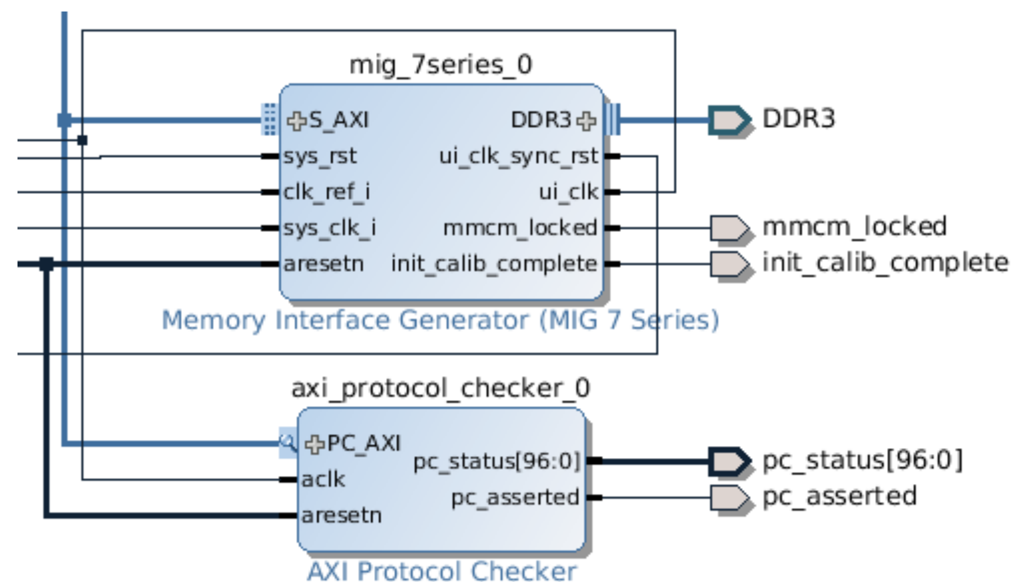
- Today you can buy 2GB DRAM variants for about ¼ the cost of what the 1980 version cost and you get:
  - 250,000 times the storage
  - About 10,000 the throughput

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

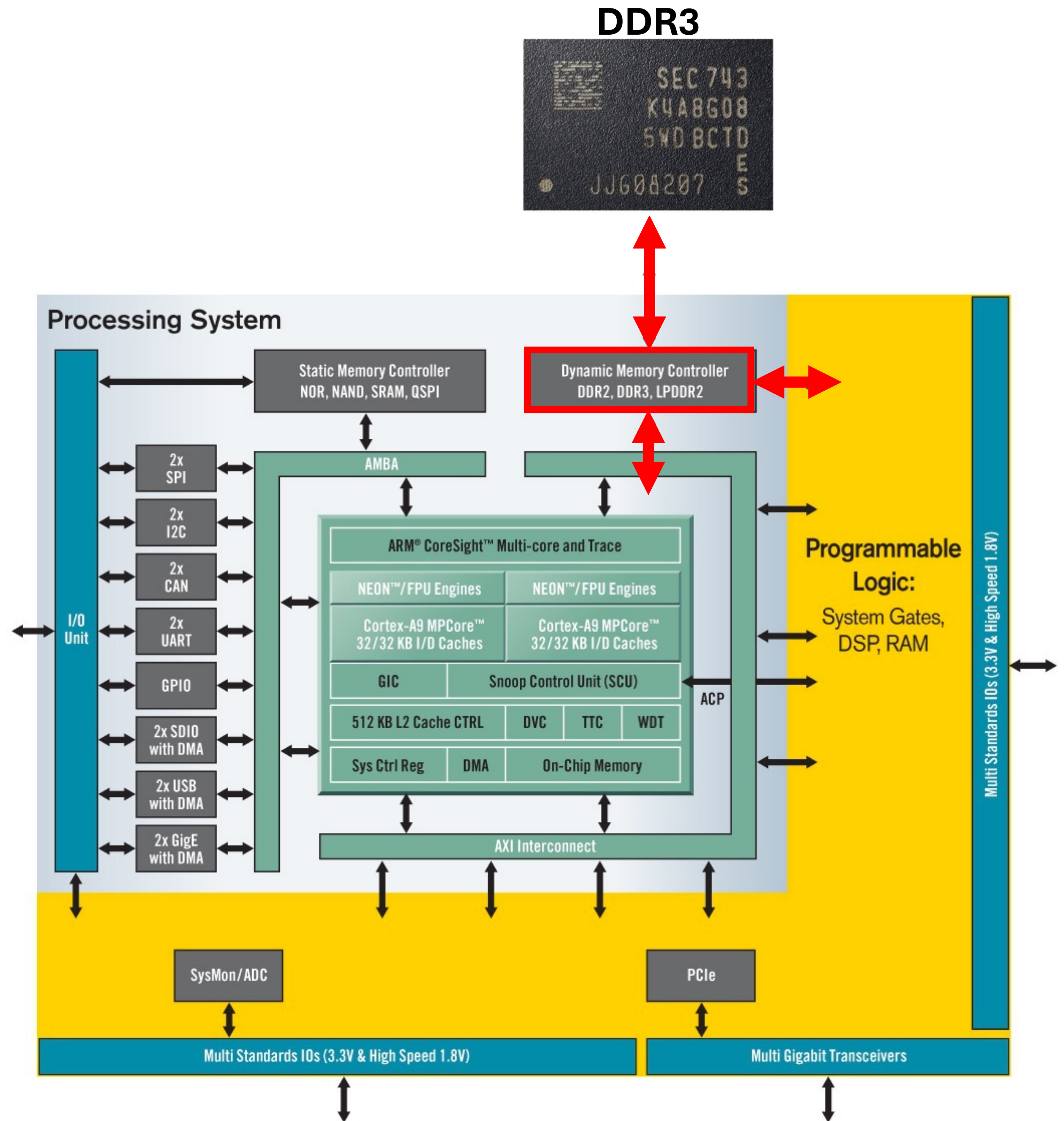
Xilinx Series 7 FPGAs

- 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



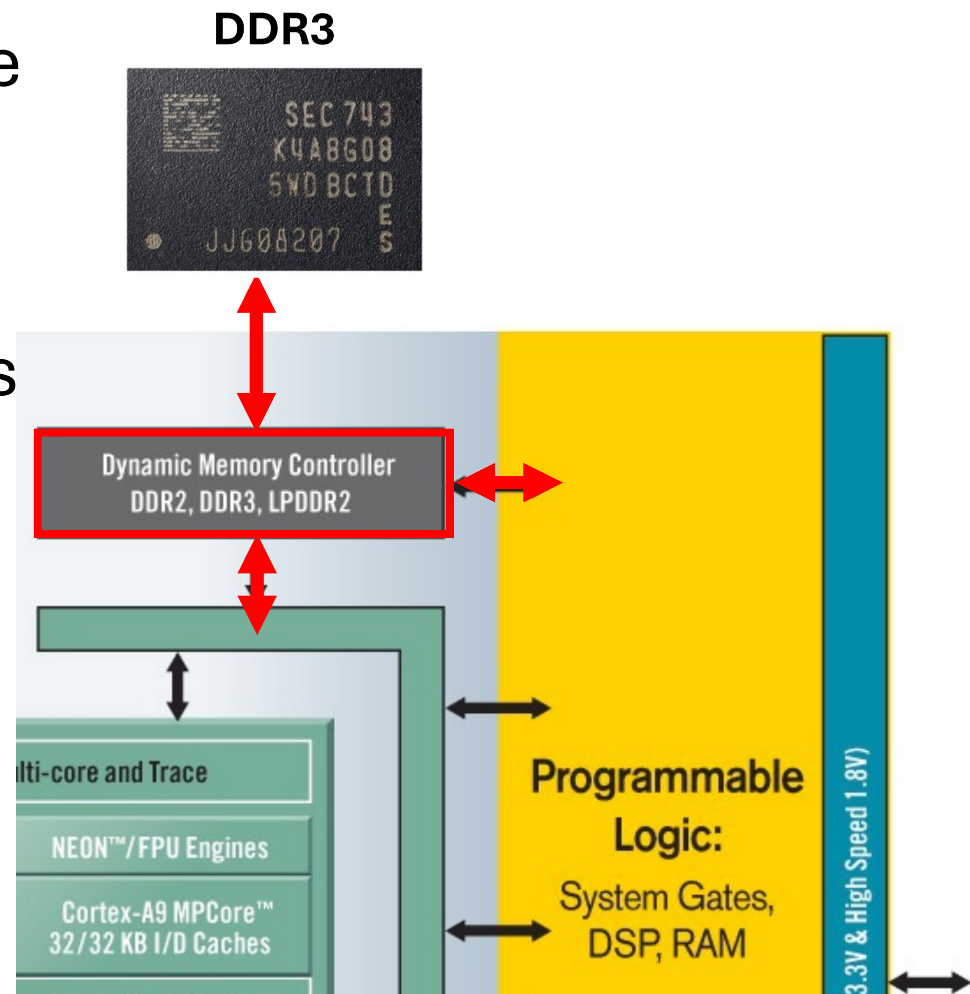
Zynq 7000

- 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



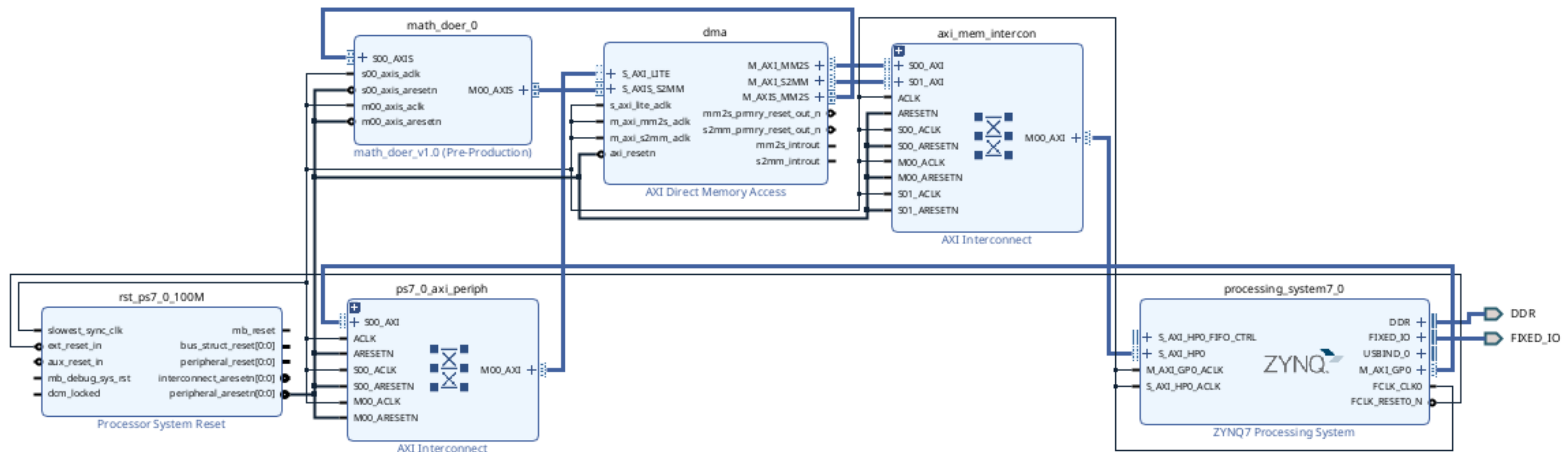
On the Python (processor) side…

- Use allocate

```
from pynq import PL
PL.reset()
from pynq import Overlay #import the overlay module
ol = Overlay('./design_1_wrapper.bit') #locate/point to the bit file
import pprint
pprint.pprint(ol.ip_dict)
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()
```

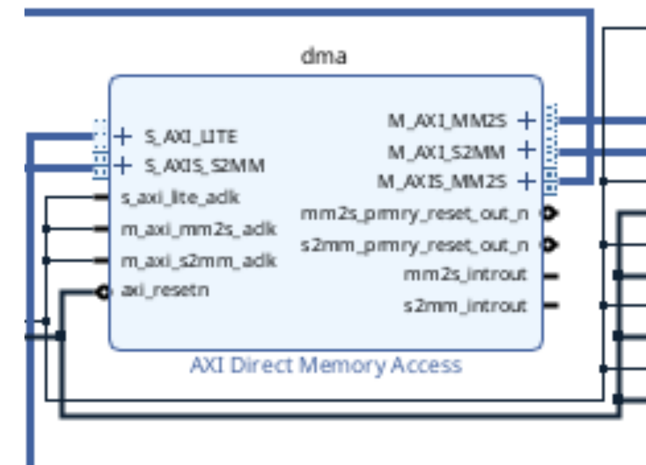
On the PL (FPGA) side…

- There's a piece of IP



On the PL (FPGA) side…

- There's a piece of IP:
  - Has several different flavors of AXI port.
  - Data comes into the PL and gets put back into the DMA through AXI Streaming (AXIS) ports
  - The other AXI ports are for control largely
- It actually isn't too bad to build tbh.



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 about 380 MBps which is nothing to sneeze at.

Speed

- DDR3: 32 bits * 1066 MHz * 2 = **4.3 GBps**
- So the 380 MBps is actually kinda low
- But 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.

AXI Streaming

- All of this does meant though to get access to all that memory and to be able to exchange information in large volumes quickly between the two environments, we will need to use AXI-Streaming

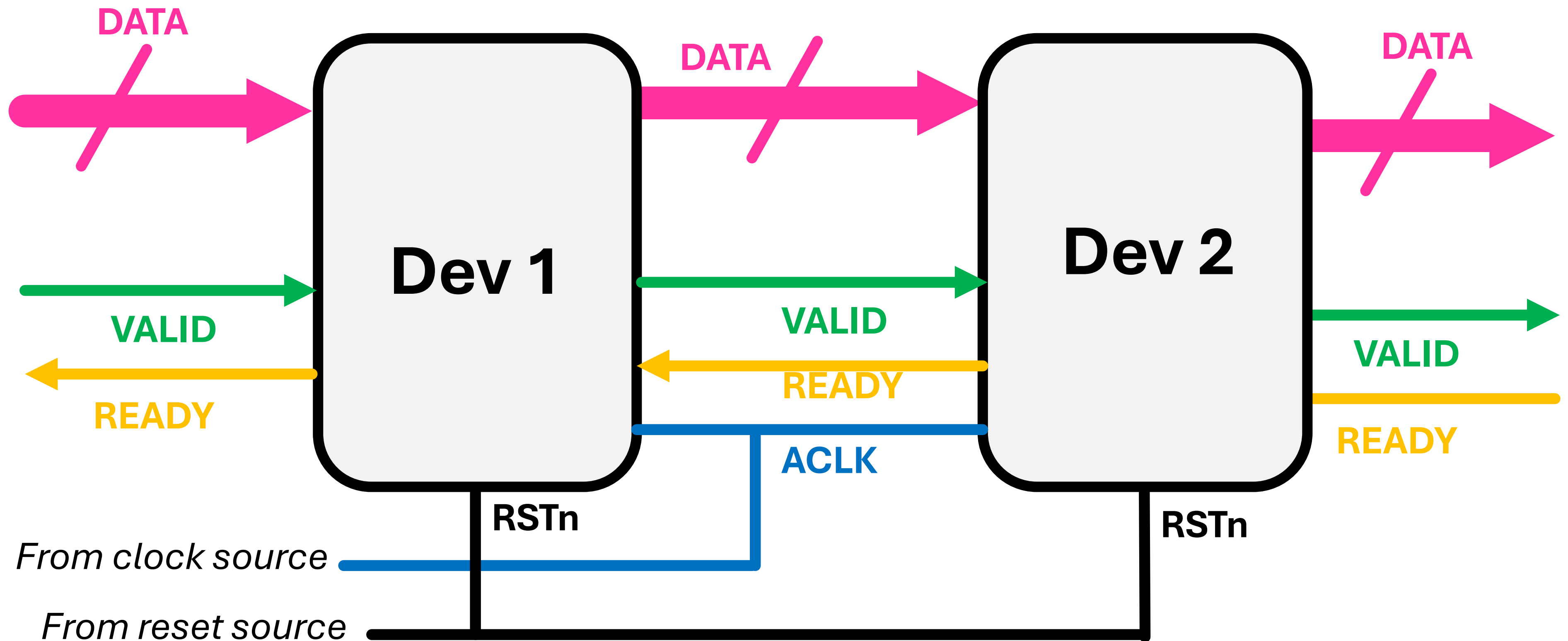
Three General Flavors of AXI4

- **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

From the Zynq Book

Good News about AXIS

- It is the least complicated of the AXI protocols

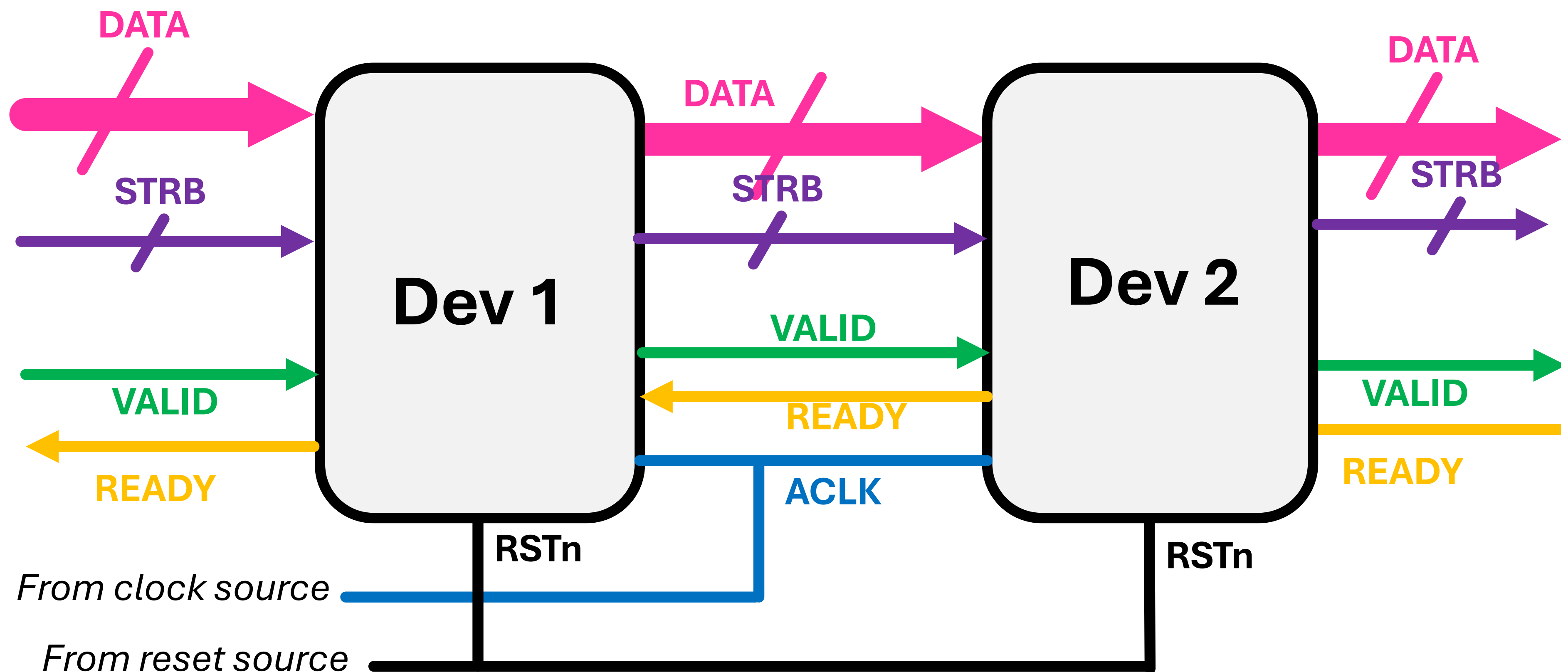


No Addressing!

- 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
- Often a few other signals…

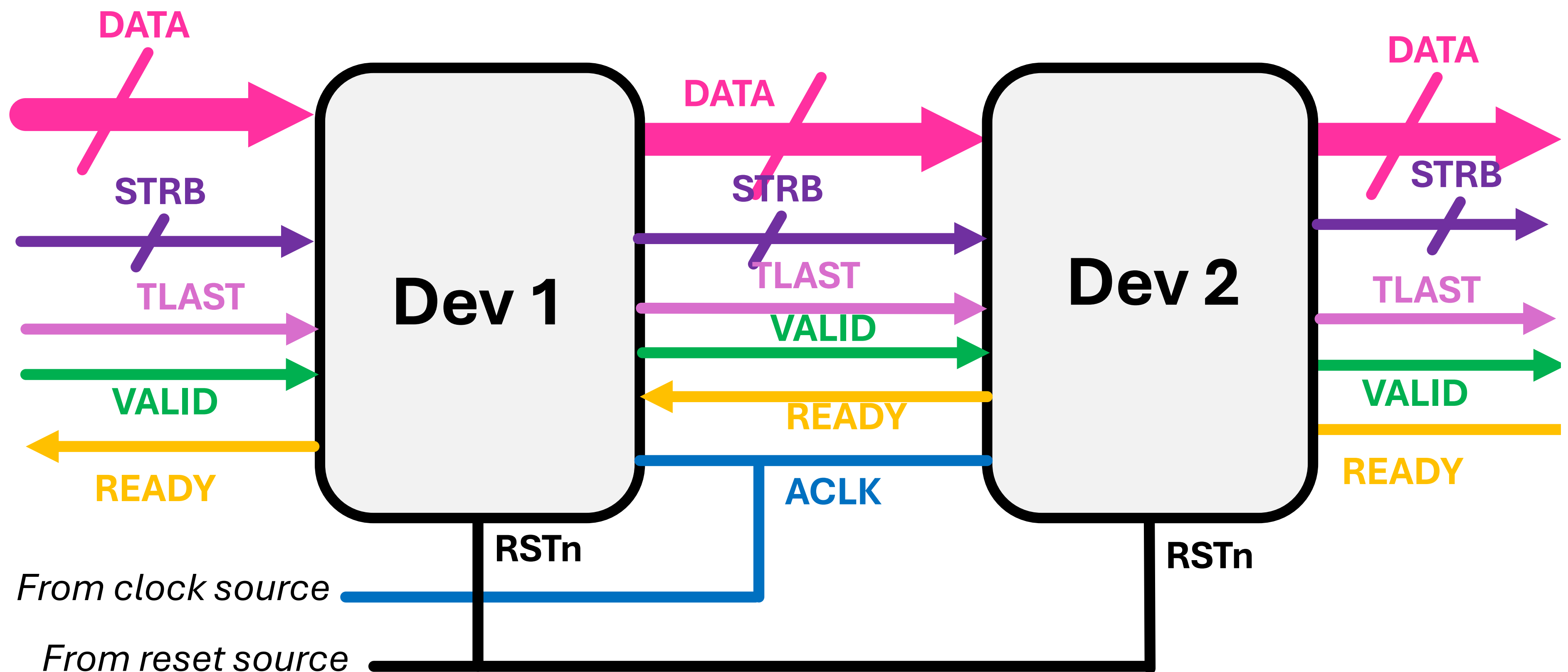
Strobe

- The strobe line will clarify which bytes in data are to be acted upon (default 0b1111 aka all)



TLAST

- For data sent in packets (such as samples of a signal where), a TLAST signal is asserted on the final sample to ensure

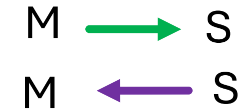


Let’s say you wanted to build a streaming module that…

- Takes in a number, multiplies it by 3 and adds 10000

- An attempt at a streaming module like this would be the following…

Build First Part of Lab 3



s_AXIS_tvalid
s_AXIS_tlast
s_AXIS_tdata[31:0]
×3
+
10000
s_AXIS_tstrb
4
4
s_AXIS_tready
m_AXIS_tvalid
m_AXIS_tlast
m_AXIS_tdata[31:0]
m_AXIS_tstrb
m_AXIS_tready

Where it will get tricky…

- If you have a module that is making calculations based off of more than one sample.
- In previous module, the flops are in place for pipelining and to meet timing
- If you have flops in place to for an:
  - FFT
  - FIR
  - Any sort of stateful calculation…
- Passthrough can get much nastier
- When a LAST signal appears, you'll still need to pump data through the system to clear the buffers.

cocotb

Triggers

Triggers are used to indicate when the cocotb scheduler should resume coroutine execution. To use a trigger, a coroutine should `await` it. This will cause execution of the current coroutine to pause. When the trigger fires, execution of the paused coroutine will resume:

```
async def coro():
    print("Some time before the edge")
    await RisingEdge(clk)
    print("Immediately after the edge")
```

Simulator Triggers

Signals

class **cocotb.triggers.Edge**(*signal*) [source]

Fires on any value change of *signal*.

class **cocotb.triggers.RisingEdge**(*signal*) [source]

Fires on the rising edge of *signal*, on a transition from `0` to `1`.

class **cocotb.triggers.FallingEdge**(*signal*) [source]

Fires on the falling edge of *signal*, on a transition from `1` to `0`.

[source]

class **cocotb.triggers.ReadOnly** [source]

Fires when the current simulation timestep moves to the read-only phase.

The read-only phase is entered when the current timestep no longer has any further delta steps. This will be a point where all the signal values are stable as there are no more RTL events scheduled for the timestep. The simulator will not allow scheduling of more events in this timestep. Useful for monitors which need to wait for all processes to execute (both RTL and cocotb) to ensure sampled signal values are final.

SV Time Slot Expanded Out

- Awaiting this will guarantee your results have stabilized

```
await ReadOnly()
```

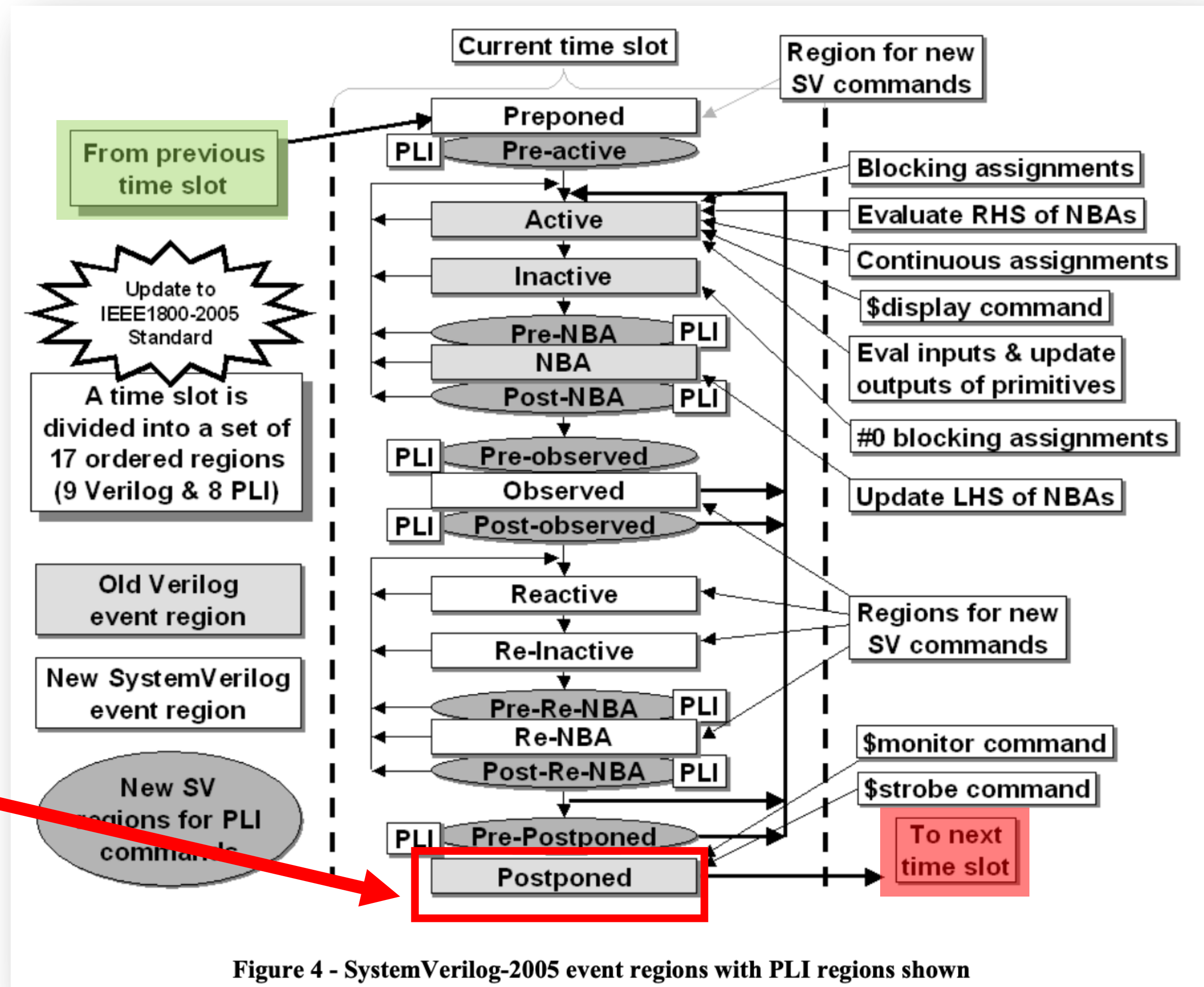


Figure 4 - SystemVerilog-2005 event regions with PLI regions shown

"SystemVerilog Event Regions, Race Avoidance & Guidelines"
Clifford E. Cummings Arturo Salz

If you need to write once in Postponed…

- You can't…

class **cocotb.triggers.ReadWrite** [source]

Fires when the read-write portion of the simulation cycles is reached.

class **cocotb.triggers.NextTimeStep** [source]

Fires when the next time step is started.

- From ReadOnly, these two triggers should be equivalent (I think)

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

cocotb-bus Public

Watch 11

master 4 Branches 6 Tags Go to file t Add file Code

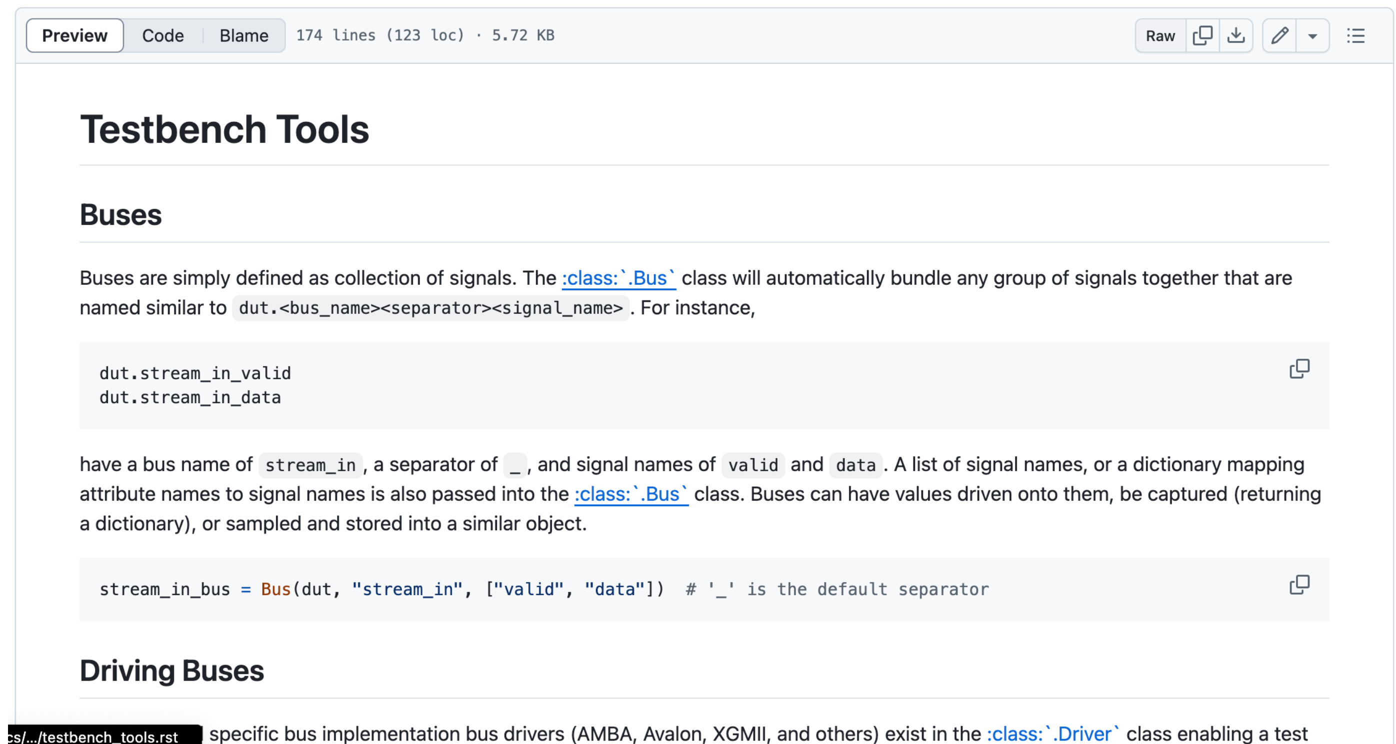
p12tic and ktbarrett Re-add support for event data field that has been rem... b99f1fa · last week 3,134 Commits

Name	Last commit message	Last commit date
.github/workflows	Brush up CI (#75)	2 months ago
bin	Replace xml.etree.cElementTree with ElementTree for Pyt...	4 years ago
docs	Document new scapy dependency; add newsfragment	10 months ago
examples	Ensure that comparisons are run on value returned by sig...	last week
src/cocotb_bus	Re-add support for event data field that has been remove...	last week
tests	Do not implicitly convert signal value to bool	last week
.gitignore	Initial commit of documentation System (#66)	10 months ago
.readthedocs.yml	Initial commit of documentation System (#66)	10 months ago
LICENSE	Create an initial blank repository	3 years ago
Makefile	Fix broken reporting of available SIM settings	4 years ago
README.md	Document new scapy dependency; add newsfragment	10 months ago
noxfile.py	Brush up CI (#75)	2 months ago
pyproject.toml	Initial commit of documentation System (#66)	10 months ago
setup.py	Brush up CI (#75)	2 months ago

README License

cocotb_bus

- Library Built Upon Cocotb that gives three (four) main devices:
  - Bus object



Bus

```
18  class Bus:
19      """Wraps up a collection of signals.
20
21      Assumes we have a set of signals/nets named ``entity.<bus_name><bus_separator><signal>``.
22
23      For example a bus ``stream_in`` with signals ``valid`` and ``data`` is assumed
24      to be named ``dut.stream_in_valid`` and ``dut.stream_in_data`` (with
25      the default separator '_').
26
27      TODO:
28          Support for ``struct``/``record`` ports where signals are member names.
29      """
```

- Let's you read and manipulate different wires in a group:
  - Drive sets all wires together at once
  - Capture gets all measures at once
  - Etc...

Symbols

Find definitions and references for functions and other symbols in this file by clicking a symbol below or in the code.

Filter symbols

- func _build_sig_attr_dict
- class Bus
  - func __init__
  - func _caseInsensGetattr
  - func _add_signal
  - func drive
  - func capture
    - class _Capture
      - func __getattr__
      - func __setattr__
      - func __delattr__
  - func sample

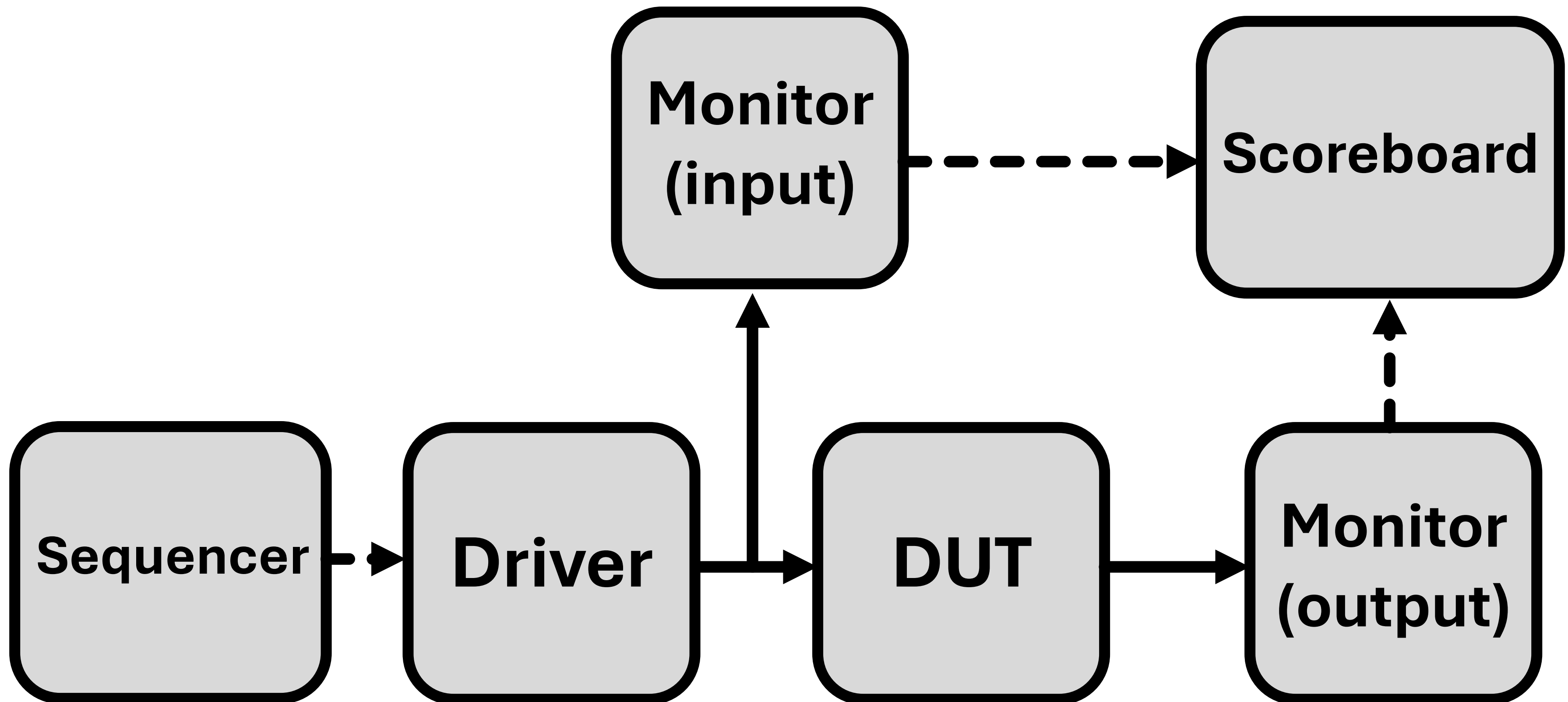
Bus is important because…

- Pretty much every modern digital design uses groups of wires to convey information.
- Packaging them up nicely is super beneficial.

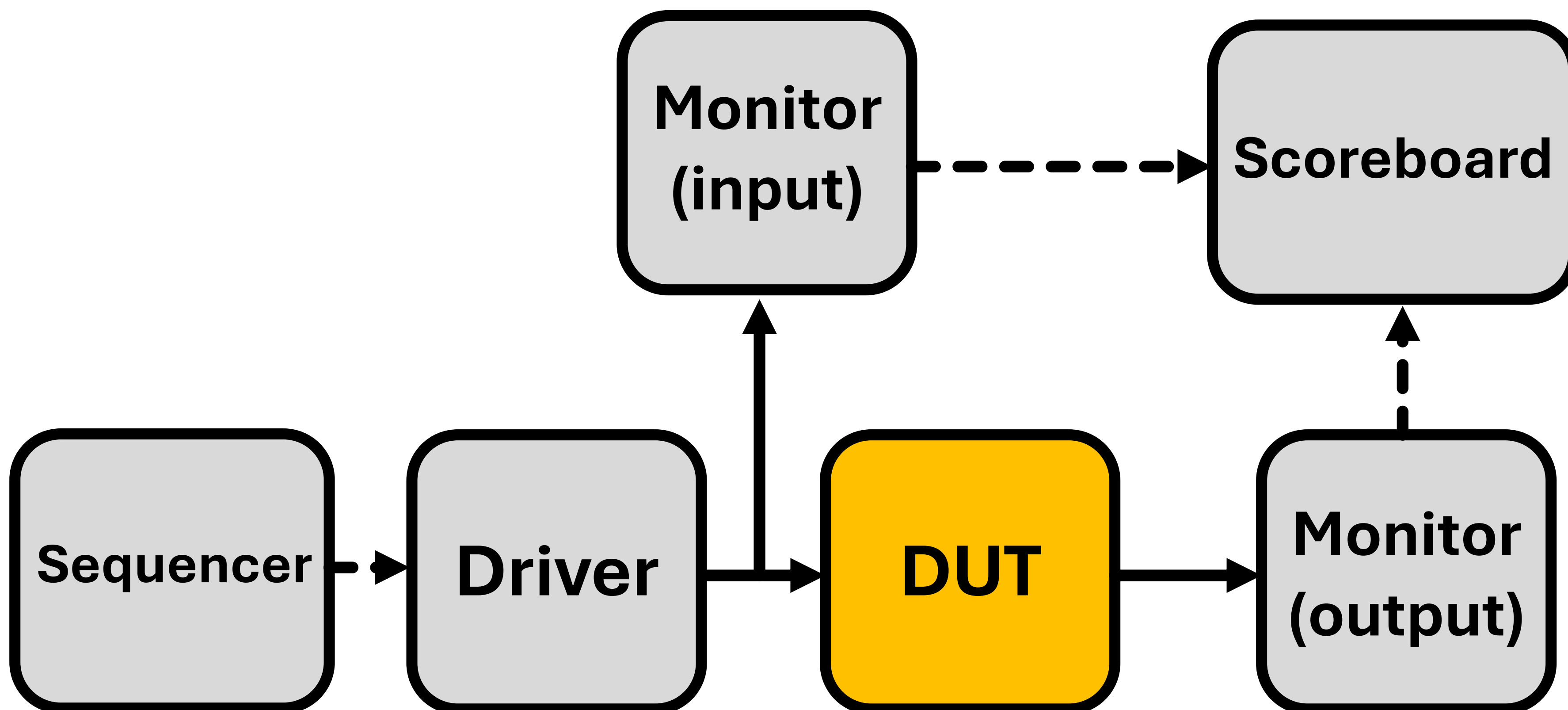
What else does Cocotb_bus bring us?

- A set of nice Components that can form the basis of reusable testing infrastructure.

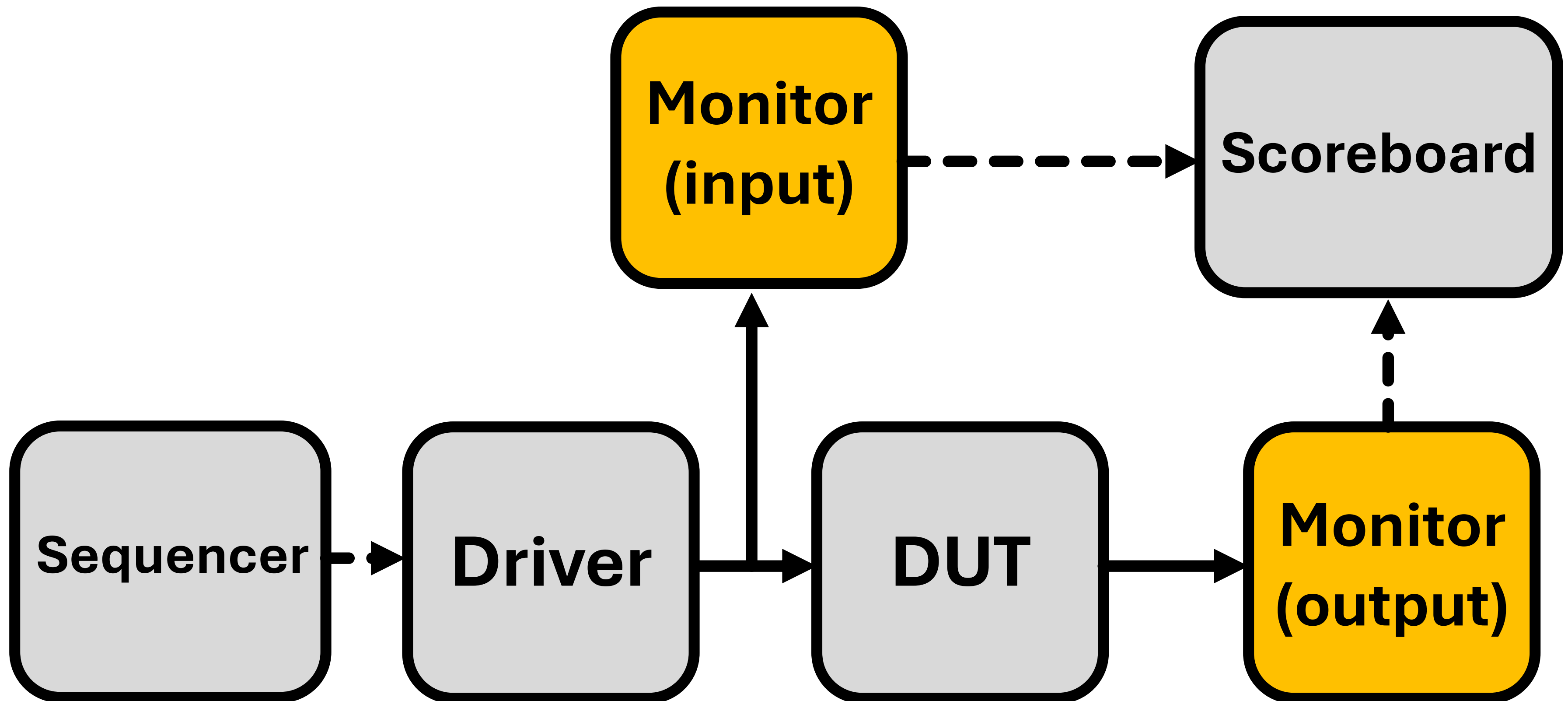
Standard Testing Framework



DUT



Monitors



Monitor/Bus Monitor

```
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 overridde
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        """
```

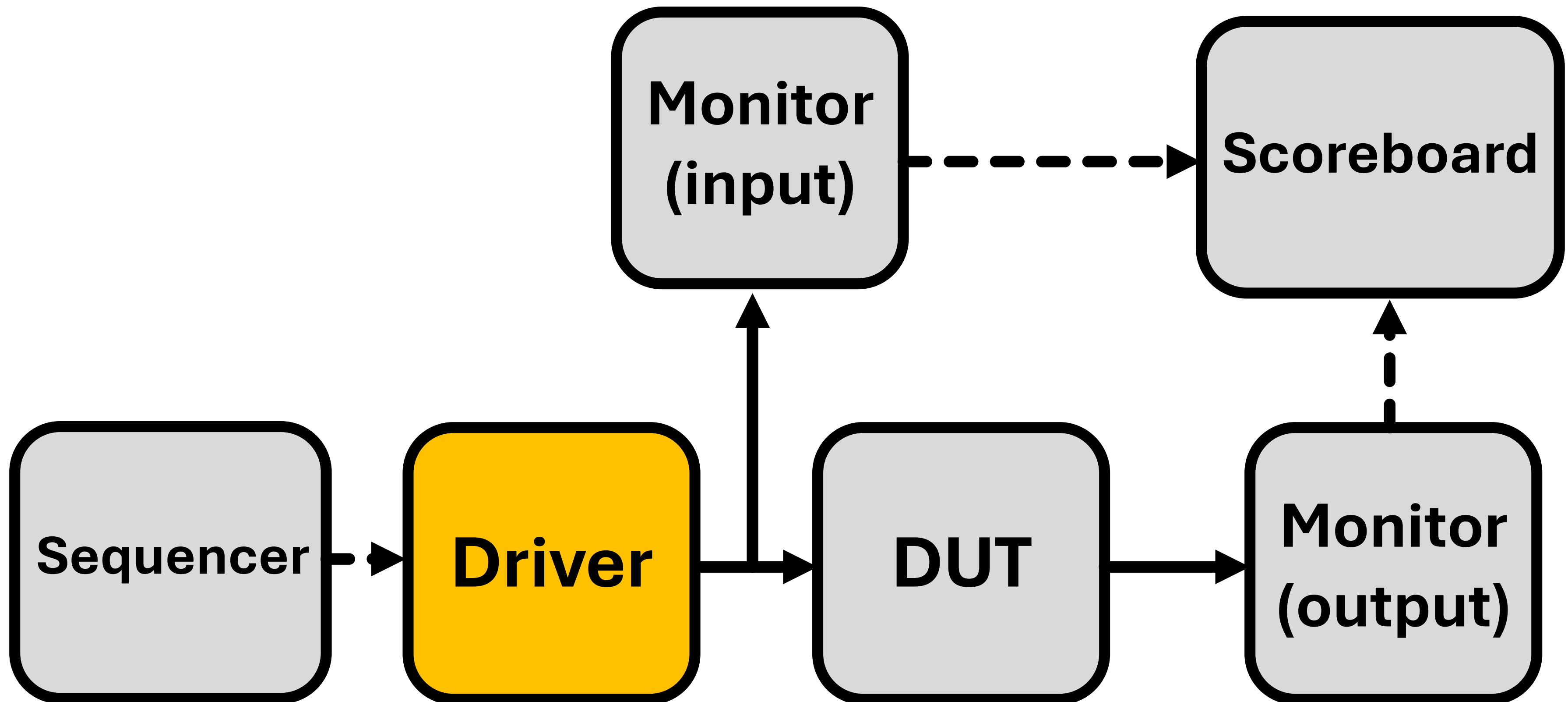
Monitors

- Monitors should listen to signals on a bus and legitimate transactions are observed then log 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.

Driver

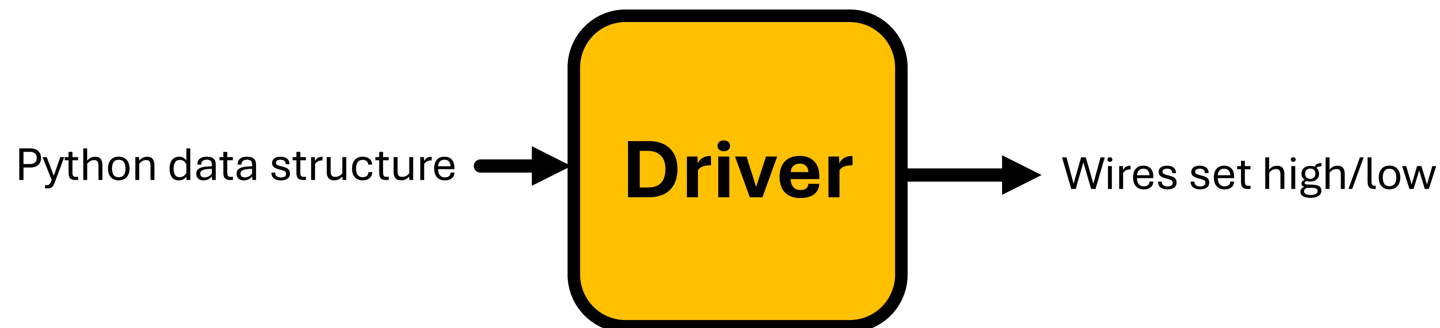


Driver

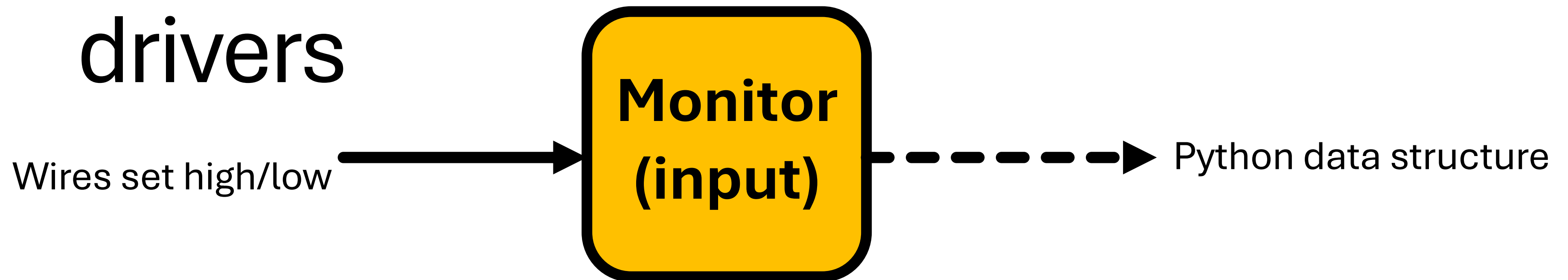
```
205
206     class BusDriver(Driver):
207         """Wrapper around common functionality for buses which have:
208
209             * a list of :attr:`_signals` (class attribute)
210             * a list of :attr:`_optional_signals` (class attribute)
211             * a clock
212             * a name
213             * an entity
214
215         Args:
216             entity: A handle to the simulator entity.
217             name: Name of this bus. ``None`` for a nameless bus, e.g.
218                 bus-signals in an interface or a ``modport``.
219                 (untested on ``struct``/``record``, but could work here as well).
220             clock: A handle to the clock associated with this bus.
221             **kwargs: Keyword arguments forwarded to :class:`cocotb.Bus`,
222                 see docs for that class for more information.
223
224         """
225
226         _optional_signals = []
227
228         def __init__(self, entity: SimHandleBase, name: Optional[str], clock: SimHandleBase, **kwar
229             index = kwargs.get("array_idx", None)
230
231             self.log = logging.getLogger("cocotb.%s.%s" % (entity._name, name))
232             Driver.__init__(self)
233             self.entity = entity
234             self.clock = clock
235             self.bus = Bus(
236                 self.entity, name, self._signals, optional_signals=self._optional_signals,
237                 **kwargs
238             )
```

Driver

- Instead of listening to a bus, it takes in high-level commands about things to put on the bus and takes care of the appropriate signaling.
-

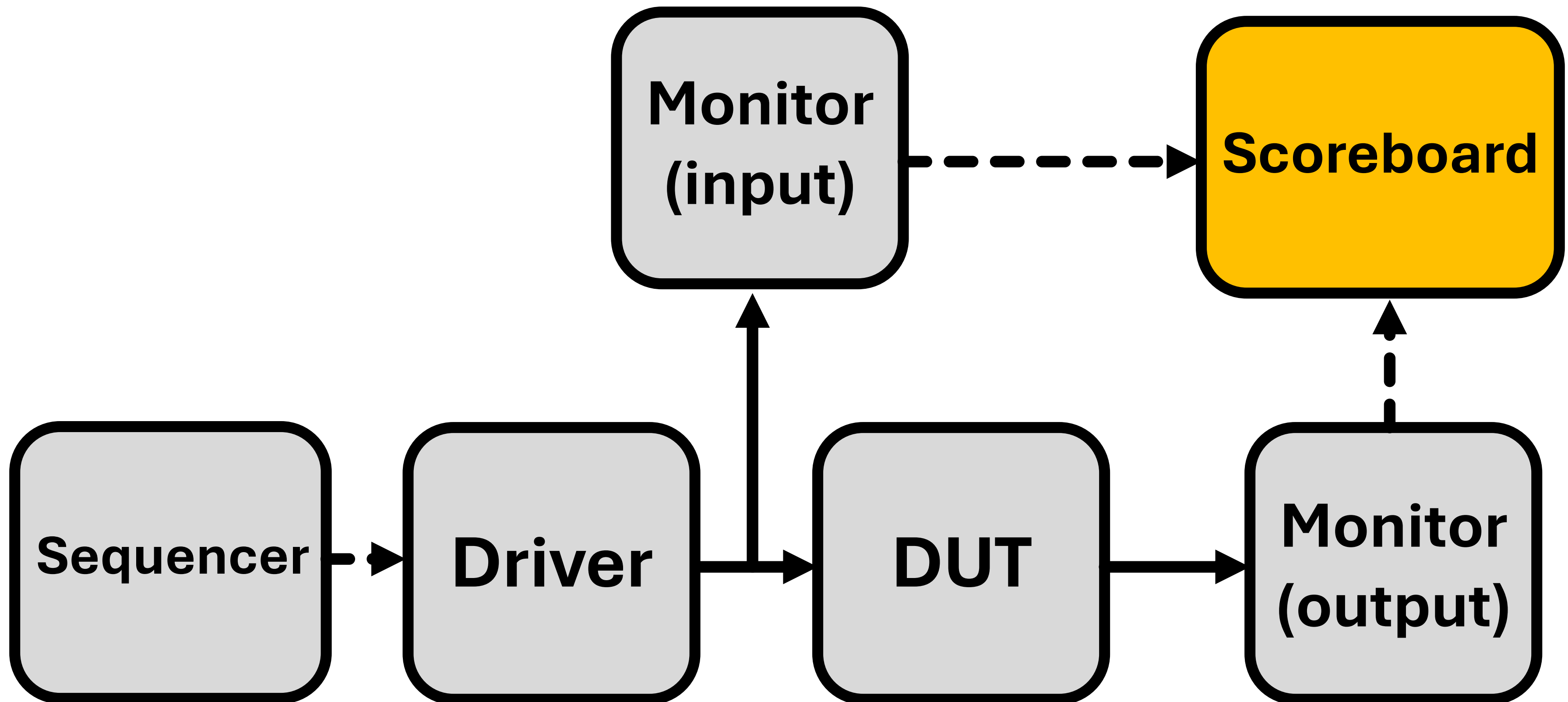


Monitors kinda the opposite of drivers



- While having both might seem kinda stupid/redundant, its strength comes from the compartmentalization of roles.
- It is much easier to just design a thing that turns bus

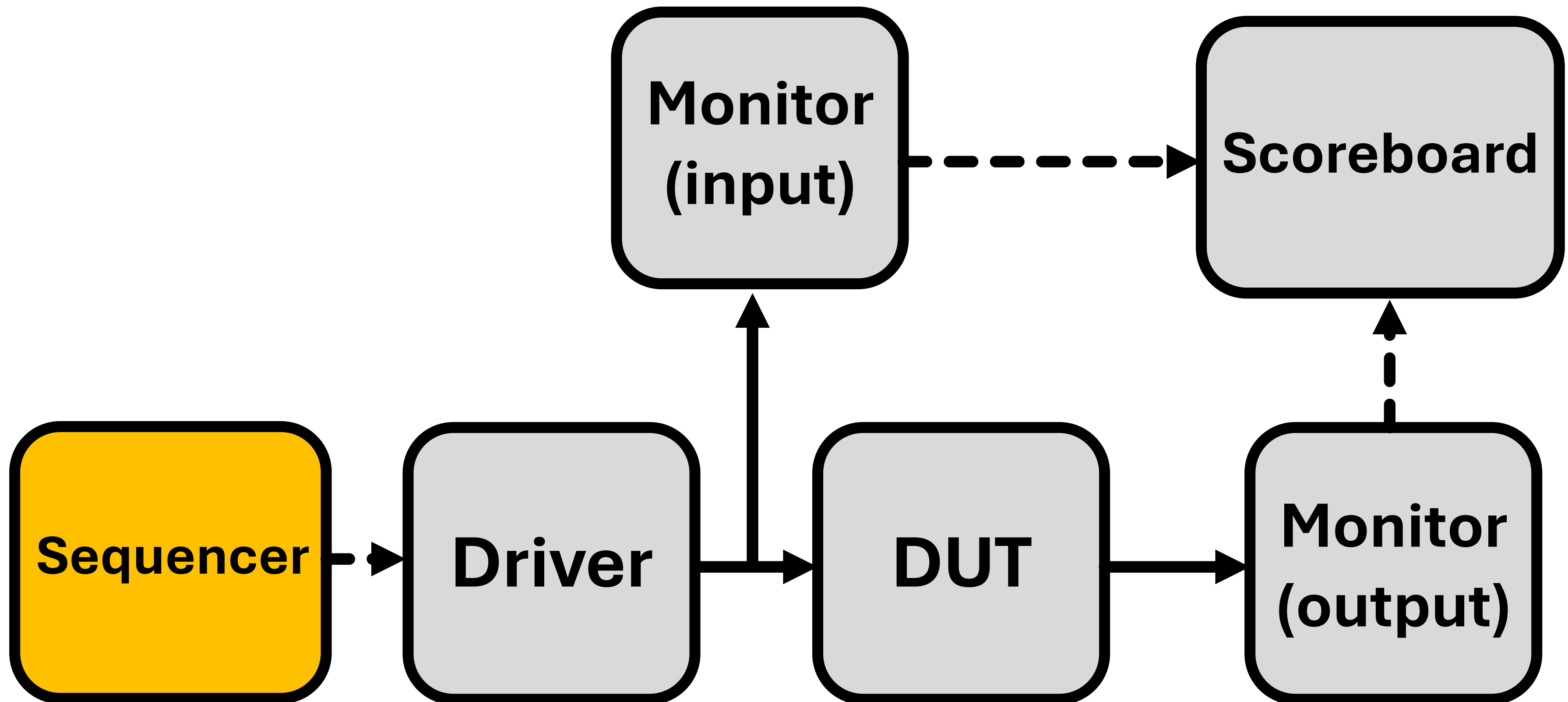
Scoreboard



ScoreBoard

```python
class Scoreboard:
    """Generic scoreboarding class.

    We can add interfaces by providing a monitor and an expected output queue.

    The expected output can either be a function which provides a transaction
    or a simple list containing the expected output.

    TODO:
        Statistics for end-of-test summary etc.

    Args:
        dut (SimHandle): Handle to the DUT.
        reorder_depth (int, optional): Consider up to `reorder_depth` elements
            of the expected result list as passing matches.
            Default is 0, meaning only the first element in the expected result list
            is considered for a passing match.
        fail_immediately (bool, optional): Raise :exc:`AssertionError`
            immediately when something is wrong instead of just
            recording an error. Default is ``True``.
    """

    def __init__(self, dut, reorder_depth=0, fail_immediately=True):  # FIXME: reorder_depth
        self.dut = dut
        self.log = logging.getLogger("cocotb.scoreboard.%s" % self.dut._name)
        self.errors = 0
        self.expected = {}
        self._imm = fail_immediately

    @property
    def result(self):
        """Determine the test result, do we have any pending data remaining?

        Raises:
```

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.