# Monitors, analysis ports and scoreboards

This page answers: *how does what a monitor sees on the pins get to the scoreboard and the coverage, and what does the scoreboard do with it?*

Lab 01′s pyuvm testbench for the ALU already used an analysis port. Its monitor wrote each vector to the port, and the scoreboard checked each vector in its `write()` method. Lab 02′s router testbench was built without pyuvm, and its classes passed packets through queues. Your monitor turned the handshakes on a stream into whole packets and put each one in a queue, and your scoreboard took them out in two loops of its own, one for each monitor's queue. Lab 03 added a tracker that read the input's and output 0′s pins itself, in a loop of its own.

In Lab 04 the router testbench is rewritten in pyuvm, and it uses **analysis ports**, as Lab 01′s did. Each of its monitors writes every beat, or every APB transfer, to its analysis port as soon as it sees it, and the scoreboard has no coroutine at all. It's a component with three write methods, `write_in()`, `write_out()` and `write_apb()`, and every write to a monitor's analysis port ends in a call to one of them.

The example for this page tests SimProc's ALU with a complete agent, made of a sequencer, a monitor and the driver from the example for Sequences, sequencers and drivers. Its source is in `guide-examples/monitors-analysis-ports-and-scoreboards/`. The environment connects three subscribers to the monitor's one analysis port: a scoreboard, a coverage collector and a history that keeps every item it's given. The ALU is the one from Lab 01, and its bug switch works here too. You set it when you run the simulator, so `make run BUG=A01` runs the same test, on the same build, with the ALU whose `SUB` computes `B - A`.

## The monitor

A monitor reads the pins and reports what it saw. It doesn't drive anything, and it doesn't know what the driver sent or what the answer should be, so the same monitor can feed the scoreboard, the coverage collector, the history and any subscriber you add later, without a change to the monitor. The example's monitor reads the pins 5 ns into every 10 ns period, as Lab 01′s did, while the driver holds them steady:

```
def next_item(self):
    return AluItem.create("seen")          # a new item every time

async def run_phase(self):
    vif, d = self.vif, self.vif.dut
    await Timer(vif.PERIOD_NS // 2, "ns")
    while True:
        if vif.valid:
            t = self.next_item()
            t.op = AluOp(int(d.ALUop.value))
            t.A = int(d.A.value)
            t.B = int(d.B.value)
            t.ALUout = int(d.ALUout.value)
            t.N = int(d.N.value)
            t.Z = int(d.Z.value)
            self.n_seen += 1
            self.ap.write(t)
        await Timer(vif.PERIOD_NS, "ns")
```

(`next_item()` is a separate method only so that a test can swap in a monitor that gets its items some other way, as `MonAliasTest` does in "One object per write", below.)

Lab 04's monitors do the same job for the router. The monitor on each stream writes one `StreamBeat` for every handshake, stamped with the number of the cycle it happened in, and the APB monitor writes one `ApbItem` for every transfer that completes. All of them read the pins at the falling edge in the middle of each cycle, as the course's monitors have done since Lab 02.

## Analysis ports

The monitor makes its port in `build_phase`, with `self.ap = uvm_analysis_port("ap", self)`. The port's `write(t)` is an ordinary method, not a coroutine, so it can't wait. It calls the `write()` of every subscriber connected to the port, one after another in the order they were connected, before it returns. A port with no subscribers is allowed, and `write()` then does nothing. Nothing in the monitor depends on who, if anyone, is connected to its port. The environment makes the connections, in its `connect_phase`:

```
def connect_phase(self):
    self.agt.mon.ap.connect(self.sb.analysis_export)
    self.agt.mon.ap.connect(self.cov.analysis_export)
    self.agt.mon.ap.connect(self.hist.analysis_export)
```

All three subscribers derive from `uvm_subscriber`. Its constructor makes the subscriber's `analysis_export`, an export that passes every item written to it on to the subscriber's `write(t)`, so `write(t)` is the only method you have to write. The port has to be connected to that export, not to the subscriber itself. `MonConnectTest` swaps in an environment whose `connect_phase` calls `self.agt.mon.ap.connect(self.sb)` instead, and the run stopped there with this error:

```
 BROKEN: UVMTLMConnectionError   1.83 s   log: build/runs/MonConnectTest-s1/run.log
     MonConnectTest: UVMTLMConnectionError: uvm_test_top.env.sb must be a subclass of
uvm_export_base
```

Here are the report lines from one run of `MonRandTest`, which sends 200 vectors. The history kept 200 items and the scoreboard checked 200, the same number that the monitor wrote:

```
   2000.00ns INFO      ..-and-scoreboards/mon_env.py(184) [uvm_test_top]: [MON] the monitor
wrote 200 items
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(106) [uvm_test_top.env.sb]: [SB] 200 vectors
checked: 0 wrong
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(126) [uvm_test_top.env.cov]: [COV] alu: 20
of 20 bins hit
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(141) [uvm_test_top.env.hist]: [HIST] kept
200 handles to 200 different objects
```

### A scoreboard with several inputs

A subscriber has one export, and every item written to it arrives in the same method, `write()`. A scoreboard that listens to several monitors needs a separate method for each input, so that it knows which monitor an item came from. SystemVerilog UVM declares a new imp class for each input with the macro `` `uvm_analysis_imp_decl ``, and each of those classes calls a different method of the scoreboard. pyuvm 4.0.1 has no such macro. Instead, Lab 04 gives its scoreboard a small subscriber for each input, a `Fwd`, which passes every item written to it on to one method of the scoreboard:

```
class Fwd(uvm_subscriber):
    def __init__(self, name, parent, fn):
        super().__init__(name, parent)
        self.fn = fn

    def write(self, t):
        self.fn(t)
```

`RouterScoreboard.build_phase` makes three of them as its children:

```
self.in_imp = Fwd("in_imp", self, self.write_in)
self.out_imp = Fwd("out_imp", self, self.write_out)
self.apb_imp = Fwd("apb_imp", self, self.write_apb)
```

and the environment connects each monitor to the right one. The figure shows which monitors write to each.

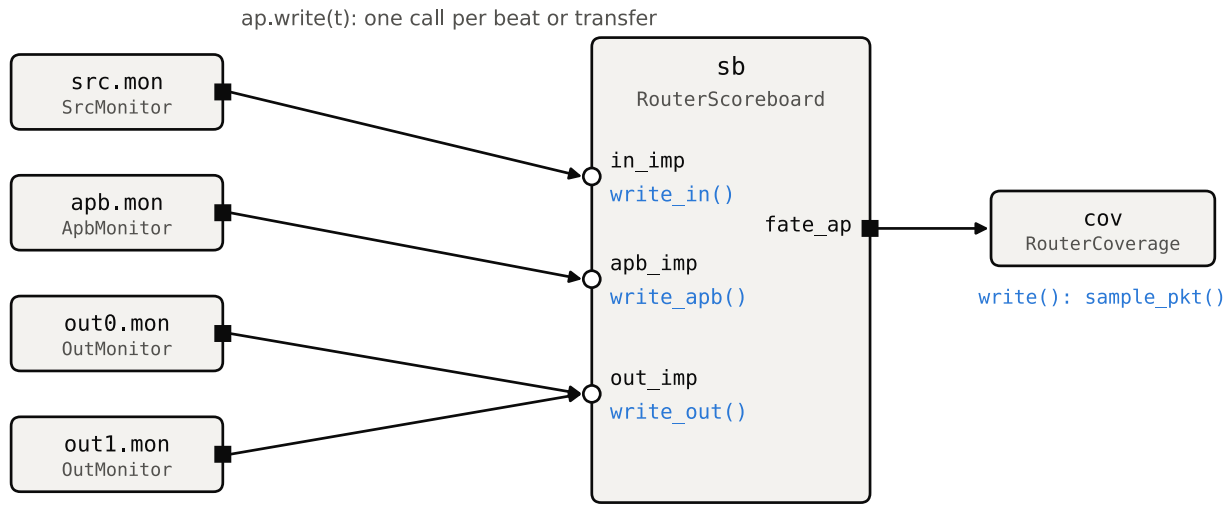



Figure 1: Lab 04's analysis connections: four monitors write to three Fwd inputs on the scoreboard, and the scoreboard's own analysis port feeds the coverage.

*The connections that Lab 04's `RouterEnv.connect_phase` makes once you've done Part 2, which adds output 1's agent. Before Part 2, only output 0's monitor is connected to `out_imp`. Both output monitors write to the same `out_imp`, and each `StreamBeat` carries its port number, so `write_out()` can tell which output a beat came from. The scoreboard decides what happens to each packet and writes a record of it to its own analysis port, `fate_ap`, which the coverage subscribes to.*

## The scoreboard

A scoreboard predicts what the design should have done and compares that with what the monitors saw. The example's scoreboard compares each item inside `write()`, as soon as the monitor writes it. Like Lab 01's scoreboard, it gets the expected answer from `predict()`, a model of the ALU written from SimProc's instruction set, and compares that answer with the item:

```
def write(self, t):
    out, n, z = predict(t)
```

```
    bad = t.ALUout != out
    if self.check_flags:
        bad = bad or t.N != n or t.Z != z
    self.n_checked += 1
    if bad:
        self.n_bad += 1
        self.logger.error(f"[SB] {t}, expected ALUout={out:02x} N={n} Z={z}")
```

With bug A01 on, `MonRandTest` logged 30 errors, and this is the first:

```
    75.00ns ERROR    ..s-and-scoreboards/mon_env.py(97) [uvm_test_top.env.sb]: [SB] SUB A=9a
B=db -> ALUout=41 N=0 Z=0, expected ALUout=bf N=1 Z=0
```

In pyuvm, logging an error doesn't fail the test by itself. The test derives from the course's `BaseTest`, which gives every component's logger a handler that counts the errors logged through it. Here the handler counted 30, so `BaseTest` failed the test in `final_phase`, and the verdict was `caught: 30 errors`. In `check_phase` the scoreboard also makes sure that it checked at least one vector, because a scoreboard that was never called would report no errors either.

Checking inside `write()` works for the ALU because the ALU is combinational. The monitor reads its inputs and its result in the same period, and both arrive in the same item. In the router, a packet goes in on one stream and comes out on another stream two or more cycles later, and the scoreboard can't wait inside a write method for it to come out. So Lab 04's scoreboard splits the work between its write methods. When a packet's header comes in, `write_in()` decides what the router must do with the packet: forward it to output 0 or 1, or drop it as NOROUTE or FULL. It decides from its copy of the routing table, which it keeps from the APB writes it sees, and from the level of the output that the packet's route names. `level()` works out that level from the beats written into the output and taken from it. If the router should forward the packet, `write_in()` queues each beat that it expects to come out of that output. `write_out()` takes the next expected beat off the queue for the output that the beat came from, and compares the two. `write_apb()` keeps the scoreboard's copy of the routing table up to date, and checks every read of `DROP_CNT` against the drops that the scoreboard has counted since `DROP_CNT` was last written. At the end of the test, `check_phase` reports any beats still left in the queues, because those are beats that the router never delivered. (As shipped, `write_in()` queues only output 0's beats, and `level()` takes output 1's level as 0. Part 2 of Lab 04 has you change the first, and Part 3 the second.)

All the monitors read the pins at the same falling edge, and cocotb runs them in no fixed order. So in the cycle that a header comes in, an output's monitor may already have reported a beat taken from that output in the same cycle, and `write_out()` will have counted it. The specification's level counts only the handshakes in earlier cycles, so `level()` adds that beat back.

A scoreboard catches only the bugs that show up in what it compares. With `check_flags` off, which is the default, the example's scoreboard compares only `ALUout`. Against Lab 01's three bugs, `make check-all` gave:

```
  caught  tier 1  A01_sub_operands_swapped        30 errors                    2.5s
  caught  tier 1  A02_nand_is_nor                 25 errors                    2.5s
 MISSED  tier 1  A03_zero_flag_low_nibble        0 errors                     2.5s
```

A03 makes the zero flag look only at the low four bits of `ALUout`. The monitor put every wrong `Z` into an item, but no subscriber compared `Z` with anything. The next page, The factory and ConfigDB, turns `check_flags` on from a test, without editing the scoreboard.

## One object per write

`ap.write(t)` passes the same handle to every subscriber, so they all see one object. That's cheap, and it's safe as long as no code changes the object after the write. The example's monitor makes a new `AluItem` for every vector. `MonAliasTest` swaps in a monitor that makes one item and fills it in again for every vector. (The swap is a factory override, which the next page explains.) The history subscriber keeps every handle it's given, and at the end of that test it prints this:

```
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(141) [uvm_test_top.env.hist]: [HIST] kept
200 handles to 1 different objects
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST] vector
0: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST] vector
1: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST] vector
2: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST] vector
199: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
```

All 200 handles point at one object, which holds the last vector. The scoreboard in the same run still reported `200 vectors checked: 0 wrong`, because it finishes with each item inside `write()`, before the monitor fills the object in with the next vector. A subscriber that keeps the handle for later, or queues it to compare when the answer arrives, finds a later vector's values in the object by the time it looks. So a monitor should make a new object for every write. A subscriber that keeps what it's given should copy the values it needs, as Lab 04′s scoreboard does when it queues `(data, last)` for each expected beat. It can keep the handle instead only when the monitor makes a new object for every write, and then it must never change the object, because every other subscriber has the same one.

## Coverage as a subscriber

A coverage collector is a simple subscriber, because its `write()` only samples the item. The example's collector passes each item to `sample(t)`, a function decorated with a coverpoint for the operation, a coverpoint for whether `A` equals `B`, and a cross of the two, all from cocotb-coverage. In `report_phase` the collector writes `coverage.xml` into the run's folder, where `make cov` finds it. Functional coverage you can read explains the coverpoints.

What a collector subscribes to depends on what it covers. The example covers what went into the ALU, so it subscribes to the monitor. Lab 04′s `RouterCoverage` covers each packet's length, its DST, its fate, and the level of its route's output in the cycle that its header came in. Only the scoreboard works out the fate and the level, so `RouterCoverage` subscribes to the scoreboard's `fate_ap`.

## Check yourself

1. An analysis port's `write()` is an ordinary method. What does that rule out for a scoreboard's `write()`, and how does Lab 04′s scoreboard live with it?

   *Answer: A1, in Answers at the end.*

2. In `MonAliasTest` the scoreboard passed and the history was wrong. Would a scoreboard that kept each input item on a list and compared it when an output arrived have passed?

   *Answer: A2, in Answers at the end.*

3. Two output monitors write `StreamBeat`s to the same `out_imp`. How does `write_out()` tell output 0′s beats from output 1′s?

*Answer: A3, in Answers at the end.*

## Further reading

pyuvm 4.0.1 defines its analysis ports in `s12_uvm_tlm_interfaces.py` and `uvm_subscriber` in `s13_predefined_component_classes.py`. ChipVerify's pages are about SystemVerilog UVM:

- UVM Monitor builds a monitor for a register file. Correction: it says a monitor that didn't get its virtual interface can keep running after a `uvm_error`. Such a monitor can't do anything useful, and in pyuvm a `ConfigDB().get()` that finds nothing and has no default raises `UVMConfigItemNotFound`, which ends the test, as the next page shows.
- UVM TLM Analysis Port shows a broadcast to three subscribers. It warns that every receiver gets the same handle, which is what `MonAliasTest` shows. Its log comes from UVM 1.1d.
- Using _decl macro in TLM builds a two-input scoreboard with a macro that pyuvm doesn't have; Lab 04′s `Fwd` does the same job. Correction: it says the macro can't go inside a class because SystemVerilog doesn't allow a class inside another class. SystemVerilog does allow nested classes.
- In UVM Scoreboard, the only check at the end of the test prints a count of mismatches, so a scoreboard that was never called passes it. The example's scoreboard and Lab 04′s check for that in `check_phase`.

Next: The factory and ConfigDB.

## Answers

### A1. Check yourself

The scoreboard's `write()` can't wait for anything, so it can't `await` a clock edge or a queue. The router's answer comes out cycles after the packet goes in, so `write_in()` predicts and queues the expected beats, and `write_out()` compares each beat when an output monitor reports it. The scoreboard doesn't need a coroutine of its own for any of that.

### A2. Check yourself

No. Every handle on its list would point at the one object, which by the time an output arrives holds a later vector, so the scoreboard would compare the wrong inputs with each output. The example's scoreboard passed only because it finished with each item inside `write()`.

### A3. Check yourself

Each output monitor stamps every beat with its port number, which it takes from its `OutCfg`, and `write_out()` uses `b.port` to pick which queue of expected beats to compare with.