Reactive agents and backpressure

This page answers: *the design decides when each beat appears on its output. How does a testbench answer each one, and how do you choose how long each beat waits?*

Your Lab 04 sink flipped a coin in every cycle to set `out_ready`, which was 1 with probability `ready_pct` percent whether or not a beat was on offer. That is backpressure, but `ready_pct` was the only thing about it you could set. In Lab 05, Part 1, the covergroup `cg_wait` takes each packet of the graded test and records the longest that any of its beats waited for `ready`. Its bins are `none` for 0 cycles, `short_wait` for 1 to 3, `long_wait` for 4 to 20 and `stalled` for more than 20, and with Lab 04's sinks it reports:

```
          0  __vlAnonCG_cg_pkt.cp_len.oversize
        257  __vlAnonCG_cg_wait.cp_wait.none
          2  __vlAnonCG_cg_wait.cp_wait.short_wait
         18  __vlAnonCG_cg_wait.cp_wait.long_wait
         57  __vlAnonCG_cg_wait.cp_wait.stalled
```

Only two of the 334 packets had a longest wait of 1 to 3 cycles. Every other packet either went through without waiting at all or had at least one beat that waited 4 cycles or more. (The first line counts packets of 9 beats or more, and it is 0 because no Lab 04 test sends one.) In Part 3 you'll finish a sink driver that answers each beat separately, with an item from a sequence. This page shows how an agent like that is put together, using a small design of our own in `guide-examples/reactive-agents-and-backpressure/`, and measures how long the beats wait with each kind of sink.

Who starts a transfer

In the agents of Labs 01 and 04 that take items from a sequence, the testbench decides when each transfer happens. Lab 04's source driver, for example, calls `get_next_item()`, waits until its sequence hands over a packet, and then puts the packet on the input. Nothing happens on the input until the driver has a packet, so the driver can wait as long as it likes.

A **reactive** agent works the other way round. The design decides when something happens, and the testbench answers. A sink is the simplest example. The router decides when a beat appears on an output, and the sink decides how long that beat waits before it's taken. Other examples are a memory model that answers a processor's read requests, and a bus slave. A reactive agent has the usual pieces (an item, a sequence, a sequencer, a driver and a monitor), but each of its items describes an answer to something the design does, rather than a transfer for the driver to start. Its driver also has to drive its side of the interface in every cycle, whether or not it has an item.

Our example's design, `beat_src`, numbers its beats 0, 1, 2 and so on. After each handshake it leaves its output idle for 1 to 4 cycles, a number it takes from an LFSR, and then puts the next beat on the output, where the beat stays until the sink takes it. The sink agent has two modes. In `coin` mode its driver does what Lab 04's did. In `beat` mode the driver answers each beat with one `snk_item`, and the item says how many cycles the beat waits after it first appears:

```
class snk_item extends uvm_sequence_item;
  rand bit [4:0] delay;
  int unsigned   max_delay = 12;
  constraint c_delay { int'(delay) <= max_delay; }
  ...
```

A delay of 0 means `ready` is already 1 when the beat appears.

A sequence that never ends

The sink's sequence makes one item per beat, for as long as the test runs:

```
task body();
  forever begin
    it = snk_item::type_id::create("it");
    start_item(it);
    it.max_delay = cfg.max_delay;
    ok = it.randomize();
    if (ok == 0) `uvm_fatal("RAND", "snk_item randomize() failed")
    finish_item(it);
  end
endtask
```

The agent starts this sequence itself, from its own `run_phase`, on its own sequencer at time 0, and the sequence never returns. Lab 05's `out_agent` starts its `out_ready_seq` the same way. The sequence also raises no objection. The test's objection decides when the test ends, and a sequence that never finishes would keep the test running forever if it raised one.

Each call to `finish_item()` blocks until the driver has taken the item and called `item_done()`, so the sequence is never more than one item ahead of the beats. The item the driver holds is the answer to the beat on offer, or, if no beat is on offer yet, to the next one.

A driver that can't wait

The driver's loop has two halves, one for each half of the clock cycle:

```
forever begin
  @(vif.cb);                                  // a rising edge
  vif.cb.ready <= rdy;                        // ready for the cycle that starts here
  @(negedge vif.clk);                         // the middle of the cycle
  if (vif.valid && rdy) begin                 // a handshake at the next edge
    seq_item_port.item_done();
    it = null;
  end else if (...)
    ...                                       // one more cycle of this beat's wait
  if (it == null)
    seq_item_port.try_next_item(it);          // returns at once, with or without an item
  rdy = ...;                                  // ready in the next cycle
end
```

It drives `ready` just after the rising edge, through the clocking block, and it looks at `valid` at the falling edge, when `valid` and `ready` have settled. At that point it knows whether a beat moves at the next rising edge. If one does, that beat has had its answer, so the driver calls `item_done()`.

The driver asks for items with `try_next_item()`, not with `get_next_item()` as Lab 04's source driver does. `get_next_item()` blocks until the sequence hands over an item. `try_next_item()` returns straight away, and leaves the handle `null` if the sequence has nothing ready. The driver then drives `ready` to 0 for the next cycle, and asks again in the middle of that cycle. Lab 05's `out_ready_driver` has the same shape, and Part 3 asks you why it uses `try_next_item()`.

On Verilator 5.050, a drive issued in the same time step as a clock edge, but after the sequencer has handed over an item, lands a cycle late. The labs found this, and the comment at the top of `tb/router_agents_pkg.sv` in Labs 04 to 07 records it. This is the second reason the driver's loop is split between the edge and the middle of the cycle; the first is to look at `valid` once it has settled. When the driver asks for items in the middle of the cycle and drives only from an edge, the handover and the drive happen in different time steps.

The rest of the loop counts the cycles the beat has been on offer, starting from the first cycle it appears, and sets `rdy` once the item's delay has gone by. You'll write that countdown for the router in Part 3, so it's left out above.

Measured waits for each sink

We ran 2,000 beats through each sink and counted, for each beat, the cycles it waited between appearing and being taken. The table sorts the waits into the same ranges as `cg_wait`'s bins, but it counts every beat, where `cg_wait` counts each packet once, by its longest wait.

Sink	Waited 0	1 to 3	4 to 20	More than 20	Mean wait
coin flip, 5%	99	237	1014	650	18.27
coin flip, 50%	1017	854	129	0	0.98
coin flip, 95%	1907	93	0	0	0.05
per beat, delays 0 to 12 from `randomize()`	116	390	1494	0	6.81
per beat, delays 0 to 12 from `$urandom_range`	169	466	1365	0	6.01

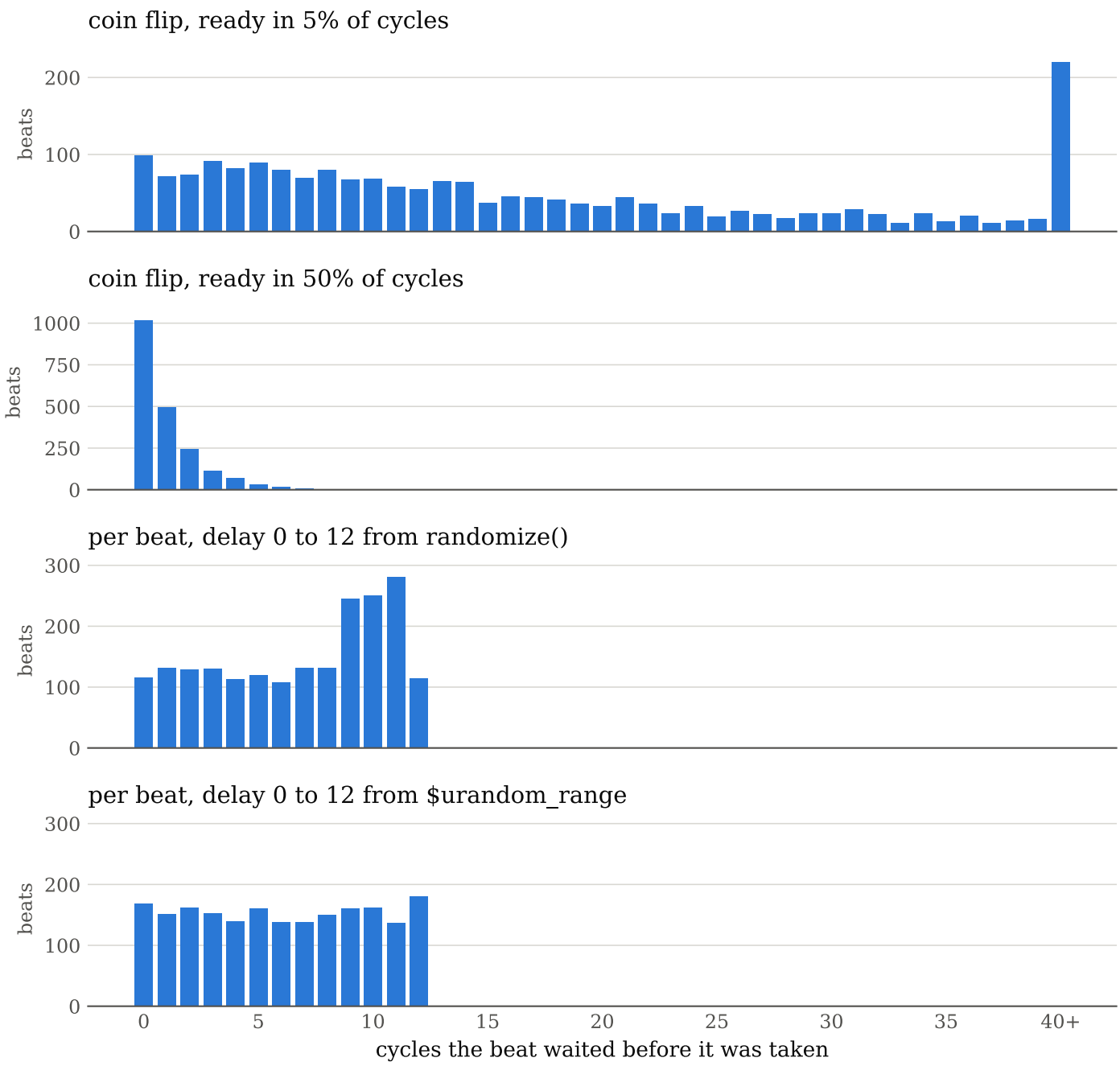


Figure 1: How long each of 2,000 beats waited, for four of the sinks in the table

A coin-flip sink gives each beat the same chance of being taken in every cycle, so its waits fall off geometrically. If p is `ready_pct` as a fraction, a beat waits w cycles with probability $p\,(1-p)^w$, and the mean wait is $(1-p)/p$, which is 19 cycles at $p = 5\%$ and 1 cycle at 50%, against our measured 18.27 and 0.98. Whatever the setting, a wait of 0 cycles is the most likely, and each wait is $1-p$ times as likely as a wait one cycle shorter, so `ready_pct` sets only how quickly the waits fall off. At 5%, most beats waited 4 cycles or more, and only 237 of the 2,000 waited 1 to 3. At 50%, no beat waited more than 11 cycles. At 95%, 1,907 of the 2,000 beats were taken in the cycle they appeared.

The test you inherit from Lab 04 runs each sink at 100% or 5% in its four phases, and stalls output 0 completely while `close_holes()` runs. At 100% a beat never waits. At 5% a packet of more than one beat nearly always has

a beat that waits 4 cycles or more, so a packet whose longest wait is 1 to 3 cycles is rare, and that test produced only two.

A per-beat sink makes each beat wait for the delay in its item, so the spread of the waits is the spread of the delays that the sequence produced. With delays of 0 to 12, between 1,365 and 1,494 beats waited 4 to 20 cycles, and none waited more than 20. Lab 05′s `out_cfg.mode` offers both kinds of sink. `SNK_PCT` is Lab 04′s coin flip, and in `SNK_FAST`, `SNK_SLOW` and `SNK_RANDOM` the driver answers each beat with an item.

Check the distribution you got

The delays from `randomize()` weren't uniform, as the fourth row of the table and the third chart of the figure show. The values 9, 10 and 11 came up about twice as often as the others (245, 251 and 281 times out of 2,000 on seed 1, and 234, 235 and 260 on seed 2, where a uniform choice would give about 154 each). With `$urandom_range(12)` in place of `randomize()`, every value from 0 to 12 came up between 137 and 180 times. The standard says that each value the constraint `delay <= 12` allows is equally likely, but Verilator 5.050′s solver didn't pick them equally often. As in Lab 03, you can't rely on a constraint for how often each value comes out, so count what the stimulus actually produced. Lab 05′s `out_ready_item` uses `randcase` in `pre_randomize()` to pick the range a delay comes from (0, 1 to 3, or 4 to 12), and leaves only the value inside that range to the solver. `cg_wait` is where you check the result.

The sink's configuration object

The sink's driver reads its mode and `ready_pct` from a configuration object, `snk_cfg`, in every cycle, and its sequence reads `max_delay` for every item. So a test, or a virtual sequence, can change the backpressure in the middle of a run just by writing a field of that object, without starting anything on the sink's sequencer. In Lab 05, Part 4, `backpressure()` changes both sinks' modes this way, every 30 to 250 cycles while the packets go in. Virtual sequences explains how a sequence gets hold of the object.

Check yourself

1. A coin-flip sink has `ready_pct` set to 20. What fraction of the beats are taken in the cycle they appear, and what is the mean wait?

   Answer: A1, in Answers at the end.

2. Why does the sink's sequence raise no objection?

   Answer: A2, in Answers at the end.

3. An item with a delay of 0 reaches the driver while the design's output is idle. What does the driver drive, and how long does the beat wait when it appears?

   Answer: A3, in Answers at the end.

Further reading

- UVM Reactive Sequence: the responder pattern for a slave agent, with a sequence that keeps its own memory model and answers the design's reads and writes from it. The test starts that sequence in a background thread, with no objection.
- Using get_next_item(): the driver-sequencer handshake, which call blocks and what releases it, and `try_next_item()`.
- UVM Driver. Correction: its "simplified" loop calls `item_done()` after one edge and never clears the request, which is the mistake the same page warns against further down.
- Using get() and put(). Its log comes from a different version of the code: the driver's path in the log has an environment the test never creates.
- Handshake Properties, in the formal section, on why a sink that is always ready hides bugs in the sender.

Next: Virtual sequences.

Answers

A1. Check yourself

20% of beats wait 0 cycles, because the coin comes up 1 in the beat's first cycle with probability 0.2. The mean wait is $(1 - p)/p = 0.8/0.2 = 4$ cycles.

A2. Check yourself

It never finishes. It makes one item per beat for as long as the test runs, so if it raised an objection, the run phase could never end. The test's own objection decides when the test is over. Our example's test holds it until the monitor has counted 2,000 beats, and Lab 05's tests hold theirs while the source sends its packets and the sinks drain them.

A3. Check yourself

From the next edge on it drives `ready` = 1, because a delay of 0 leaves nothing to wait for. When the beat appears, `ready` is already 1, so the handshake happens in the beat's first cycle and the beat waits 0 cycles. A coin-flip sink can promise that for a particular beat only when its `ready_pct` is 100.