Sequences, sequencers and drivers

This page answers: *how does a transaction get from the code that makes it to the design's pins?*

In Labs 02 and 03 the generator made packets and put them into a mailbox, and the driver took them out one at a time and drove them. In Lab 04 a **sequence** makes the items, a **sequencer** passes them on, and a **driver** puts each one on the pins. The table in Lab 04's Part 5 lists what changed. The sequence calls `start_item` and `finish_item` where the generator called the mailbox `put`, and the driver calls `get_next_item` and `item_done` where it called the mailbox `get`. Everything else the driver does is the same as before, including the idle gaps it leaves before beats and its drives through the clocking block.

This page follows one item along that path. The example for it drives SimProc's ALU from Lab 01 and prints a line, with the time, at every step. Its files are in `guide-examples/sequences-sequencers-and-drivers/`.

The pieces

The item is an object with `rand` fields, and its class extends `uvm_sequence_item`. Lab 04's item is `router_pkt`, and the example's is `alu_item`, which holds an operation and two operands.

A sequence is an object too, not a component. Its class extends `uvm_sequence #(alu_item)`, and its `body()` task makes the items and sends them. A test makes one during its run phase, when it needs it, sets its knobs and calls `seq.start(sequencer)`. `start()` returns when `body()` has finished.

The sequencer is a component in the agent. It sits between any number of sequences and one driver, and it hands the driver one item at a time. You hardly ever write a sequencer class yourself. Lab 04 has two sequencers, in the input agent and the APB agent, and it declares each of them with a `typedef`:

```
typedef uvm_sequencer #(router_pkt) src_sequencer;
```

The driver is a component too. Lab 04's `src_driver` extends `uvm_driver #(router_pkt)`, and that base class gives it a port called `seq_item_port`, which the agent connects to the sequencer in `connect_phase`:

```
drv.seq_item_port.connect(sqr.seq_item_export);
```

The port is on the driver because the driver asks for each item when it's ready for one, and the sequencer hands items over only when asked. That lets the driver set the pace. It takes the next item only when the pins can take it, and the sequence waits until then.

One item through the handshake

Here are the loops in the example's sequence and driver:

```
// alu_rand_seq::body()
req = alu_item::type_id::create("req");
start_item(req);                       // wait until the driver asks for an item
if (!req.randomize()) `uvm_fatal("RAND", "randomize() failed")
finish_item(req);                      // hand it over; wait until the driver is done

// alu_driver::run_phase()
forever begin
  seq_item_port.get_next_item(req);    // wait for an item
  @(vif.drv_cb);                       // drive from a clock edge
  vif.drv_cb.valid <= 1'b1;
  vif.drv_cb.ALUop <= req.op;
```

```
  vif.drv_cb.A     <= req.A;
  vif.drv_cb.B     <= req.B;
  @(vif.drv_cb);                      // the vector holds for one cycle
  vif.drv_cb.valid <= 1'b0;
  seq_item_port.item_done();          // let finish_item() return
end
```

(The files in the example also print a line at each step, and they tag each item with a short label for the sequence that sent it, which is `a` here.) `sq_timeline_test` runs this sequence for three items. The clock has a 10 ns period, and its first rising edge is at 5 ns. The first two items of the run print these lines:

```
0    ns  [drv]    get_next_item: waiting for an item
0    ns  [seq a]  start_item(req): waiting for the sequencer
0    ns  [seq a]  start_item returned; randomize()
0    ns  [drv]    get_next_item returned a: OR A=b0 B=b2
15   ns  [drv]    item_done, ALUout was b2
15   ns  [drv]    get_next_item: waiting for an item
15   ns  [seq a]  finish_item returned: a: OR A=b0 B=b2 is done
15   ns  [seq a]  start_item(req): waiting for the sequencer
15   ns  [seq a]  start_item returned; randomize()
15   ns  [drv]    get_next_item returned a: SHL A=c9 B=02
35   ns  [drv]    item_done, ALUout was 24
```

`start_item()` returns once the sequencer has picked this sequence and the driver is waiting for an item. At time 0 the driver was already waiting, so `start_item()` returned at once. The sequence then randomizes the item and calls `finish_item()`, which hands the item over and blocks. The driver's `get_next_item()` returns with the item in the same time step. The driver waits for the clock edge at 5 ns and drives the vector through its clocking block, and the vector reaches the pins 1 ns later, because the block's output skew is 1 ns. The driver holds the vector through the edge at 15 ns and then calls `item_done()`, and that is when `finish_item()` returns. So the sequence's loop runs at the driver's pace, which here is one item every 20 ns. This timing diagram shows the same two items:

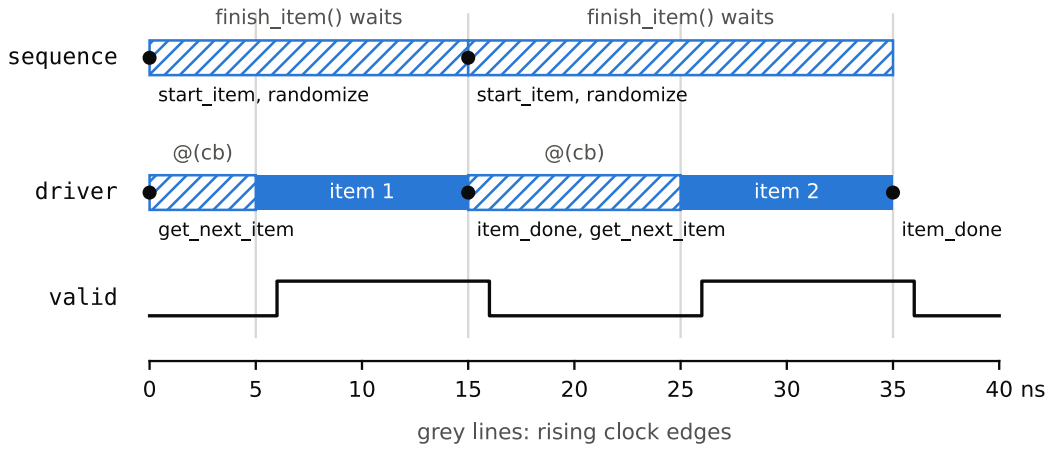


Figure 1: The first two items of sq_timeline_test: the sequence waits in finish_item from 0 to 15 ns while the driver waits for the edge at 5 ns and holds the vector until 15 ns.

The handshake for the first two items of `sq_timeline_test`. The sequence spends most of its time blocked in `finish_item()`, and the driver spends most of its time waiting on clock edges.

On the ALU's pins, recorded with `make waves`, the run looks like this:



Figure 2: sq_timeline_test on the ALU in Surfer, 0 to 55 ns: three vectors, each with valid high for one cycle.

The three vectors on the ALU's pins. Each one goes on 1 ns after a rising edge and holds for one cycle, and `ALUout` shows the ALU's answer (`b2`, `24`, `ff`). Between two vectors `valid` is low for one cycle. The driver drives `valid <= 0` right after the edge that ends a vector, before it asks for the next item, and it drives the next vector only after the edge that follows.

Because the sequence randomizes the item after `start_item()` returns, the item gets its values when the driver is ready for it rather than when the sequence made it, and the constraints can use anything that the sequence has set by then. Lab 04's `router_rand_seq` copies `route_en` into each packet before it randomizes the packet. `route_en` holds the enable bits of the routing table that the test wrote for the current phase, and the packet's `c_dst` constraint uses those bits to keep the packet on an enabled route.

Two sequences on one sequencer

A sequencer can serve several sequences at once, so one test can run background traffic and a directed burst on the same agent at the same time. `sq_two_test` starts two sequences of two items each, `a` and `b`, in a `fork`. These are the lines it prints, with some of the driver's left out:

```
0    ns  [seq a]  start_item(req): waiting for the sequencer
0    ns  [seq b]  start_item(req): waiting for the sequencer
0    ns  [seq a]  start_item returned; randomize()
0    ns  [drv]    get_next_item returned a: SUB A=05 B=80
15   ns  [seq a]  finish_item returned: a: SUB A=05 B=80 is done
15   ns  [seq a]  start_item(req): waiting for the sequencer
15   ns  [seq b]  start_item returned; randomize()
15   ns  [drv]    get_next_item returned b: OR A=cc B=ea
35   ns  [seq b]  finish_item returned: b: OR A=cc B=ea is done
35   ns  [seq b]  start_item(req): waiting for the sequencer
35   ns  [seq a]  start_item returned; randomize()
35   ns  [drv]    get_next_item returned a: SHR A=79 B=bd
```

The items alternate between the two sequences. The sequencer's default arbitration grants requests in the order they arrived, and each sequence asks again only after its last item is done. So whenever an item finishes, the other sequence's request has been waiting longer than the new one from the sequence that just finished, and the other sequence goes next. A sequence that you start on a sequencer that's already busy therefore takes turns with the one running there, item by item. Each of Lab 05's sinks has a sequence that its agent starts and that runs for the whole test, which is why Lab 05's virtual sequence steers a sink's backpressure by changing its `out_cfg` rather than by starting another sequence on the sink's sequencer.

Sequences that start sequences

A sequence's `body()` can start other sequences on the same sequencer, so you can build a larger sequence out of smaller ones. The call is `child.start(m_sequencer, this)`, where `m_sequencer` is the sequencer that the parent is running on. `sq_nested_test` runs a sequence that starts two others: first one that sends two random vectors, then one that sends two directed SUBs. From the point where the random sequence finishes, the run prints these lines (one of the driver's lines is left out):

```
35   ns  [seq r]  finish_item returned: r: SHR A=db B=14 is done
35   ns  [drv]    get_next_item returned sub: SUB A=5a B=5a
55   ns  [drv]    item_done, ALUout was 00
55   ns  [seq sub] sent sub: SUB A=5a B=5a
55   ns  [drv]    get_next_item returned sub: SUB A=5a B=21
75   ns  [drv]    item_done, ALUout was 39
```

The directed sequence never calls `randomize()`. It sets `op`, `A` and `B` by hand between `start_item()` and `finish_item()`, which is the simplest way to send exactly the item you mean. On this simulator, `randomize() with { ... }` fails when its inline constraint rules out the entry that a `dist` picked, and setting the fields by hand is also how the course avoids those failures (see Constrained random on Verilator).

A test starts sequences too. Lab 04′s base test writes a register by starting a one-item sequence on the APB agent's sequencer:

```
task write_reg(bit [7:0] addr, bit [31:0] data);
  apb_rw_seq s = apb_rw_seq::type_id::create("wr");
  s.write = 1;
  s.addr  = addr;
  s.wdata = data;
  s.start(env.apb.sqr);
endtask
```

`read_reg()` does the same with `write = 0`, and then reads the result out of the sequence's item. That works because the APB driver fills in the item's `rdata` before it calls `item_done()`, and `start()` doesn't return until the driver has called it.

Driving from a clock edge

Lab 04′s drivers wait for a clock edge before their first drive of an item, as the example's driver does with `@(vif.drv_cb)` straight after `get_next_item()`. They do it because of a timing problem in this simulator that turned up while the course's reference testbench was being written (`labs/ref/README.md`, "A clocking-block drive after an NBA-region wait"). `try_next_item()` doesn't return until the nonblocking assignments of its time step have been made. If the driver then issues a drive through the clocking block in that same time step, and the time step is a clock edge, the drive takes effect at the next edge instead of this one. The same thing happens to Lab 04′s drivers, which use `get_next_item()`, when the sequencer hands one of them an item in the time step of an edge and the driver drives straight away. Waiting for an edge after getting the item puts every drive at the start of a time step. Lab 01′s driver calls `try_next_item()` instead of `get_next_item()`, and it too asks for the item before it waits for the edge.

Drive through the clocking block too, as every driver on this page does. On Verilator 5.050, a module-level task that does `x <= 1; @(posedge clk); x <= 0;` loses some of its assignments when it's called in a `repeat` loop. Class drivers kept every assignment in every shape that the course tried. Use the clocking block anyway, because it gave

the right result in every shape, and it doesn't depend on the order in which Verilator commits two nonblocking assignments in one time step (`labs/common/README.md`, "Drive through a clocking block").

Check yourself

1. In `sq_timeline_test`, `finish_item()` for the first item returns at 15 ns, not at 0 ns. What is it waiting for?

   Answer: A1, in Answers at the end.

2. At what time is the second item randomized, and what has to happen first?

   Answer: A2, in Answers at the end.

3. Lab 04's `close_holes()` needs its fill packets to arrive exactly as it built them, with set lengths. Should it use `router_rand_seq` with `randomize() with { len == n; }`?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's walkthrough builds these pieces for a small register file: Defining the Transaction Class, UVM Sequence, UVM Sequencer and UVM Driver. Two of those pages have mistakes in them. The driver page's "simplified" `get_next_item()` loop never clears the request signals, and it calls `item_done()` straight after one clock wait, which is the mistake that the page's own list of common mistakes warns about ("Driving signals with only one clock wait"). The transaction page's exercise solution adds an `idle` field that can never be 1 while the original constraint stays in.

- Using get_next_item(), for the handshake in more detail, with `try_next_item()` and responses.
- How to execute sequences via start(), for the arguments of `start()` and the hooks around `body()`. Its example test starts a sequence without raising an objection, which works there only because the sequence uses no simulated time.
- UVM Sequence Arbitration, for the other ways a sequencer can choose between sequences.
- Executing Sequences with uvm_do Macros shows what the macros expand into. This course writes out `start_item` and `finish_item` instead of using the macros. The expansions on that page are from an older version of UVM than the course's library.

Next: Monitors, analysis ports and scoreboards.

Answers

A1. Check yourself

The driver's `item_done()`. The driver got the item at 0 ns, drove it after the edge at 5 ns, held it through the edge at 15 ns and then called `item_done()`, which let `finish_item()` return.

A2. Check yourself

At 15 ns. Its `start_item()` can't return until the driver asks for another item, and the driver asks only after it has driven the first item, held it through the edge at 15 ns and called `item_done()`.

A3. Check yourself

No. `router_pkt` has a `dist` on `kind`, and on this simulator an inline constraint that rules out the entry that the `dist` picked makes the call fail. `len == n` rules out two of the three kinds. So `close_holes()` makes its fill packets by hand with `router_pkt::make()`, and it sends each one with Lab 04's `router_one_seq`, a sequence that only calls `start_item()` and `finish_item()`.