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 cocotb `Queue`, and the driver took them out one at a time and drove them. In Lab 04 a **sequence** makes the packets, a **sequencer** hands them to the driver, and the **driver** puts each one on the pins. Where the generator called the queue's `put()`, the sequence calls `start_item()` and `finish_item()`, and where the driver called the queue's `get()`, it now calls `get_next_item()` and `item_done()`. The idle cycles that the driver leaves before beats, and the way it drives the pins, are the same as in Lab 03. The table in Lab 04′s Part 5 goes through the two testbenches piece by piece.

This page follows one item from a sequence, through the sequencer, to the pins. Its example drives SimProc's ALU as Lab 01′s pyuvm testbench did, holding each vector on the pins for one 10 ns period, and at every step it prints a line that starts with the simulation time. The example's files are in `guide-examples/sequences-sequencers-and-drivers/`.

The pieces

An item is an object of a class derived from `uvm_sequence_item`. Lab 04′s item class is `RouterPkt`. The example's is `AluItem`, which holds an operation and two operands, and also holds the ALU's answer after the driver has put the vector on the pins. Both classes are decorated with `@vsc.randobj` as well, which makes them pyvsc classes, so a sequence can call `randomize()` on them.

A sequence is an object too, not a component. Its class derives from `uvm_sequence`, and its `body()` coroutine makes the items and sends them. A test makes a sequence when it needs one, sets its knobs, and runs it with `await seq.start(sequencer)`, which returns when `body()` has finished. A pyuvm sequence has no logger of its own. The example's sequences print their lines with `print()`, and the register sequences in the course's reference testbench, `labs/ref`, report through a component's logger.

The sequencer is a component inside the agent. Any number of sequences can send items to it, and it hands them to its one driver one at a time. You don't write a sequencer class in this course. In Lab 04 the input agent and the APB agent each make one of pyuvm's own, with `uvm_sequencer("sqr", self)`.

The driver is a component too. Lab 04′s `SrcDriver` derives from `uvm_driver`, and that base class gives it a port called `seq_item_port`, which the agent connects to the sequencer in `connect_phase`:

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

The port is on the driver because the driver sets the pace. It asks for the next item only when it's ready to put one on the pins, and the sequence waits until it does.

One item through the handshake

Here are the loops in the example's sequence and driver, without the lines that print each step:

```
# AluRandSeq.body()
for _ in range(self.n):
    req = AluItem.create("req")
    await self.start_item(req)          # wait until the driver takes it
    req.randomize()
    await self.finish_item(req)         # hand it over; wait until the driver is done

# AluDriver.run_phase()
while True:
```

```
    req = await self.seq_item_port.get_next_item()   # wait for an item
    d.ALUop.value = int(req.op)
    d.A.value = int(req.A)
    d.B.value = int(req.B)
    vif.valid = True
    await Timer(vif.PERIOD_NS, "ns")    # the vector holds for one 10 ns period
    req.ALUout = int(d.ALUout.value)
    vif.valid = False
    self.seq_item_port.item_done()      # let finish_item() return
```

The example's files also tag each item with the name of the sequence that sent it, and that name is the `a` in the lines below. `SqTimelineTest` runs this sequence for three items, and these are the lines it prints up to the driver's `item_done()` for the second item:

```
0    ns  [drv]     get_next_item: waiting for an item
0    ns  [seq a]   start_item(req): waiting for the driver
0    ns  [seq a]   start_item returned; randomize()
0    ns  [drv]     get_next_item returned a: SHR A=bd B=14
10   ns  [drv]     item_done, ALUout was bd
10   ns  [drv]     get_next_item: waiting for an item
10   ns  [seq a]   finish_item returned: a: SHR A=bd B=14 is done
10   ns  [seq a]   start_item(req): waiting for the driver
10   ns  [seq a]   start_item returned; randomize()
10   ns  [drv]     get_next_item returned a: SHL A=ce B=eb
20   ns  [drv]     item_done, ALUout was 70
```

`start_item()` puts the item in the sequencer's queue and returns once the driver's `get_next_item()` has taken it. At time 0 the driver was already waiting, so `start_item()` returned at once. The sequence then randomized the item and called `finish_item()`, which hands the item over and waits. `get_next_item()` returned after that, in the same time step, because pyuvm's `get_next_item()` waits for `finish_item()` before it gives the driver the item. The driver held the vector on the pins for one period and called `item_done()` at 10 ns, which let `finish_item()` return in the same time step. So the sequence's loop runs at the driver's pace, which here is one item every 10 ns:

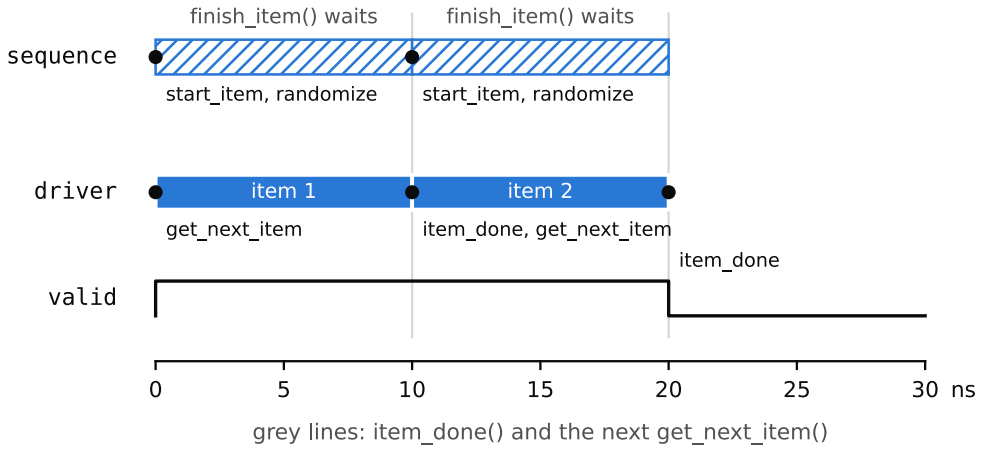


Figure 1: The first two items of SqTimelineTest: the sequence waits in finish_item from 0 to 10 ns and from 10 to 20 ns while the driver holds each vector for one period.

The handshake for the first two items of `SqTimelineTest`. The sequence is blocked in `finish_item()` for the whole of each period, while the driver holds the vector on the pins. `valid` is the Python flag that the driver sets while a vector is on the pins, as in Lab 01. It stays high from 0 to 20 ns, because the driver puts the second vector on in the same time step as it calls `item_done()` for the first.

The sequence randomizes the item after `start_item()` returns, so the item gets its values when the driver is ready for it rather than when the sequence made it. The constraints can use anything that the sequence has put in the item by then. Lab 04′s `RouterRandSeq`, for example, 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.

The ALU has no clock, so the example's driver drives the vector as soon as it has the item. The router has a clock, and Lab 04′s drivers follow the timing rule that the course set in Lab 02, Part 2. Every driver changes the router's inputs right after a rising edge, and every monitor reads the pins at the falling edge in the middle of the cycle. So when `get_next_item()` returns, `SrcDriver` looks at the clock before it drives the packet's first beat. If the clock is low, it waits for the next rising edge, and if the clock is high, it's already just after one. `ApbDriver` always waits for the next rising edge, and starts the transfer's setup cycle there. The sequence waits in `finish_item()` all the while.

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. `SqTwoTest` starts two sequences of two items each, `a` and `b`, as two cocotb tasks, and waits for both. These are the lines it prints up to 20 ns, without the driver's `get_next_item: waiting` and `item_done` lines:

```
0    ns  [seq a]   start_item(req): waiting for the driver
0    ns  [seq b]   start_item(req): waiting for the driver
0    ns  [seq a]   start_item returned; randomize()
0    ns  [drv]     get_next_item returned a: ADD A=6d B=23
10   ns  [seq a]   finish_item returned: a: ADD A=6d B=23 is done
10   ns  [seq a]   start_item(req): waiting for the driver
10   ns  [seq b]   start_item returned; randomize()
10   ns  [drv]     get_next_item returned b: ADD A=85 B=c8
20   ns  [seq b]   finish_item returned: b: ADD A=85 B=c8 is done
20   ns  [seq b]   start_item(req): waiting for the driver
20   ns  [seq a]   start_item returned; randomize()
20   ns  [drv]     get_next_item returned a: SUB A=22 B=c4
```

The items alternate between the two sequences. pyuvm's sequencer hands the driver items in the order that their `start_item()` calls arrived, whichever sequence made them, 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. That order is the only one that pyuvm 4.0.1′s sequencer has. SystemVerilog UVM's sequencer has arbitration modes, priorities and locks, and pyuvm's source has none of them. Lab 05 starts a sequence that runs for the whole test on each output agent, so another sequence started on one of those sequencers would take turns with it.

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. `start()` stores the sequencer in the sequence's `sequencer` attribute, so a child sequence is started with `await child.start(self.sequencer)`. (SystemVerilog UVM calls that attribute `m_sequencer`, and has a

`p_sequencer` as well; pyuvm has neither.) `SqNestedTest` runs a sequence that starts two others, first one that sends two random vectors and then one that sends two directed `SUB`s. The run prints these lines from the point where the random sequence finishes, at 20 ns, to the driver's `item_done()` for the second `SUB`:

```
20   ns  [seq r]   finish_item returned: r: SHL A=ff B=ec is done
20   ns  [drv]     get_next_item returned sub: SUB A=5a B=5a
30   ns  [drv]     item_done, ALUout was 00
30   ns  [drv]     get_next_item: waiting for an item
30   ns  [seq sub] sent sub: SUB A=5a B=5a
30   ns  [drv]     get_next_item returned sub: SUB A=5a B=21
40   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. Lab 04′s `close_holes()` builds its fill packets by hand too, with `RouterPkt.make()`, and sends each one with `RouterOneSeq`, a sequence that only calls `start_item()` and `finish_item()`.

Lab 04′s base test writes a register by starting a one-item sequence on the APB agent's sequencer, and it reads a register the same way:

```
async def write_reg(self, addr, data):
    await ApbRwSeq("wr", True, addr, data).start(self.env.apb.sqr)

async def read_reg(self, addr):
    s = ApbRwSeq("rd", False, addr)
    await s.start(self.env.apb.sqr)
    return s.t.rdata
```

`read_reg()` takes the value it returns from 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 `finish_item()` has. `SqReadTest` does the same on the ALU, whose driver fills in `ALUout`:

```
[READ] SUB(c8, 3c) = 8c
[READ] NAND(c3, 1d) = fe
[READ] SHR(f8, 03) = 1f
```

get_next_item() and item_done()

A SystemVerilog UVM driver can call `try_next_item()`, which returns at once, with no item, when none is waiting. pyuvm 4.0.1′s `seq_item_port` has no `try_next_item()`. The only call it has that gets an item is `get_next_item()`, and that call waits until there is one. So a driver that must act in every cycle, whether or not an item has arrived, needs a sequence that always has the next item ready. The sink driver that you finish in Lab 05, Part 3 is one of these. Lab 05′s base test starts a `SinkSeq` on each sink, and that sequence never ends.

A driver must also call `item_done()` before it asks for the next item. `SqDoubleGetTest` has a driver that calls `get_next_item()` twice without calling `item_done()` in between, and pyuvm stopped the test at the second call:

```
 BROKEN: UVMSequenceError   1.88 s   log: build/runs/SqDoubleGetTest-s1/run.log
   SqDoubleGetTest: UVMSequenceError: You must call item_done() before calling get_next_item
again
```

The graders call this run BROKEN rather than FAIL, because the exception came from the testbench, and a run that ends that way tells you about the testbench and nothing about the design.

The time on the first line is how long the simulator ran. We built the simulator twice and ran `SqDoubleGetTest` five times. The first run after each build took us 3.54 s and 3.56 s, and each of the other three runs took 1.88 s.

Check yourself

1. In `SqTimelineTest`, `finish_item()` for the first item returns at 10 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()` stalls output 0 and fills it with packets for DST 0 whose lengths come from a `FillPlan` and add up to exactly 8. Why does it build them with `RouterPkt.make()` and send them with `RouterOneSeq`, instead of starting `RouterRandSeq`?

   Answer: A3, in Answers at the end.

Further reading

pyuvm's sequences are in its source, `s14_15_python_sequences.py` at 4.0.1. Its comments sketch both sides of the handshake: a driver's loop that calls `get_next_item()` and `item_done()`, and a sequence's `body()` that calls `start_item()` and `finish_item()`. ChipVerify's walkthrough builds the same pieces in SystemVerilog UVM, for a register block that it calls `reg_block`: UVM Sequence, UVM Sequencer and UVM Driver.

- Using get_next_item(), for the handshake in more detail. It also covers `try_next_item()`, which pyuvm 4.0.1 doesn't have.
- How to execute sequences via start(), for the hooks around `body()`. pyuvm's `start()` takes only the sequencer and a flag for whether to call `pre_body()` and `post_body()`, with no parent sequence or priority.
- UVM Sequence Arbitration describes arbitration modes that pyuvm 4.0.1′s sequencer doesn't have; it hands out items in the order they were asked for.

Next: Monitors, analysis ports and scoreboards.

Answers

A1. Check yourself

The driver's `item_done()`. The driver got the item at 0 ns, held it on the pins for one 10 ns period, and then called `item_done()`, which let `finish_item()` return.

A2. Check yourself

At 10 ns. Its `start_item()` can't return until the driver takes it, and the driver asks for another item only after it has held the first one for its period and called `item_done()`.

A3. Check yourself

`RouterRandSeq` makes each packet and calls `randomize()` on it, so the solver would choose the DST and the length, and the packet's `post_randomize()` would put idle cycles before about one beat in four. `RouterRandSeq` also makes each packet with `RouterPkt.create()`, through the factory, so a test's override, such as the one that swaps in `NoroutePkt` for `RouterInjectTest`, would reach the fill packets too. `make()` sets the DST and the length by hand, puts the beats back to back and uses the constructor, which no override reaches, and `RouterOneSeq` sends the packet it's given without changing it. (Lab 03, Part 4 says that `randomize_with()` with an inline `length == n` would work too, as line 3 of its experiments shows, and the packets would then have the idle cycles.)