Layering Protocols With Chained Sequencers

Brian Hunter,

Cavium

ABSTRACT

A sequencer arbitrates among sequences that operate in parallel. Sequences contain traffic to be driven onto an interface in the form of sequence items. The protocol of an interface--what happens between the traffic--is therefore left to the driver. For complex protocols, this may involve link training initialization, 8b/10b encoding/decoding, flow-control mechanisms, and more.

Both the traffic and the protocols often have a layered hierarchy, in which higher-levels are used to manage acknowledgements, replay, error checking, etc. Having a UVM driver manage these details while juggling traffic is complex to design and difficult to debug.

Design patterns that use a sequence hierarchy to separate the traffic layers are well known. This paper will describe a technique of chaining sequencers together to separate the layers of both the traffic AND the interface protocols to not only simplify design and debug, but also to allow for simulation at higher levels of abstraction without the overhead of low-level protocols.

Table of Contents

Table of Figures

1.Introduction

The success of a constrained random verification environment hinges upon the injection of chaotic —yet meaningful—stimulus into the device under test. The Universal Verification Methodology's (UVM) solution to stimulus generation is sequencers and sequences. These oddly-named classes can drive traffic into a device in a random manner, yet can also be programmed and debugged as individual threads.

Sequencers could have been called arbiters, because that is their primary purpose. These UVM components hold handles to one or many sequences that are said to be running "on" them, and sequencers get to decide which item to send next. Sequencers can be used to make the order of stimulus events chaotic.

The sequences themselves can be thought of as individual threads of procedural tasks. Sequences may operate at various levels of abstraction, but never so low as to perceive individual signals of interfaces. Because sequences are also randomizable classes, sequences are used to cause mayhem, and produce varied stimulus.

Finally, the sequence item contains the data that will be driven onto the interface. At higher levels, this data represents the packetized or transaction level information. At the lowest level, the sequence items received by the driver often map directly to the pins on an interface.

The UVM does not explain how engineers should best use sequences and sequence items to create random transactions and procedures. Failure to optimize this part of a verification environment can lead to several issues. An environment may under-stimulate the design and fail to hit corner cases. Or, it may over-stimulate the design and inject meaningless nonsense. Poorly planned sequences can be difficult to both design and debug.

All random stimulus has some hierarchical aspects. From the highest-level operating systems to the lowest-level pin wiggling, these layers of abstraction are used to help make sense of otherwise chaotic systems. In general, sequence hierarchies can be mapped along these same boundaries.

2.The Hawkins Interface

To best explain sequence hierarchies, consider an example interface protocol named Hawkins[1]. Like many interfaces, such as Ethernet, PCI Express®, or RapidIO®, it has distinct, well-documented protocol layers. While all of these interfaces have unambiguous layering, they are representative of how most stimulus generation might be approached. Unlike the others, though, the Hawkins interface is designed to be simple to understand.

The Hawkins interface is a point-to-point interface connecting two nodes. The Hawkins protocol consists of three distinct layers: Transaction, Link, and Physical Levels.

The transaction level transmits reads, writes, and responses from one agent to the other to be used as memory access transactions. Each read transaction has a TAG that is used to map responses, and a limited number of TAGs are available.

The link level adds cyclical redundancy check (CRC) values to transactions to ensure data correctness. A simple acknowledgement scheme is also used to guarantee transaction reception.

The data at the physical level is sent on a byte-wide interface with a simple valid signal to indicate data. The physical layer clears the valid signal when sending acknowledgements (ACK), non-acknowledgements (NAK), end-of-packets (EOP), and idle symbols. Also, the interface requires that special training symbols be sent every two microseconds.

---

[1] The Hawkins interface is presented in the textbook ***Advanced UVM*** [1], available at http://tinyurl.com/AdvancedUVM or http://tinyurl.com/AdvancedUVM-ebook.

To help readers understand the SystemVerilog code that was developed to support this paper, Chapter 9 provides complete technical details of the Hawkins interface.

3.Chaining Sequencers

3.1.Layering With Encapsulation

As with similar interfaces, the data packets to be injected into a device using the Hawkins interface are encapsulated. From the transaction level down to the physical level, each stage adds more and more data to the packet. Figure 1 shows a complete packet consisting of a transactional command contained within a link packet. The physical layer breaks these into an array of phy_item_c sequence items to be delivered to the driver and adds an end-of-packet (EOP) symbol.

With such a simplistic interface, it may be tempting to just create one class containing all of these elements. But it will soon become evident that maintaining the different levels of abstraction has significant benefits.

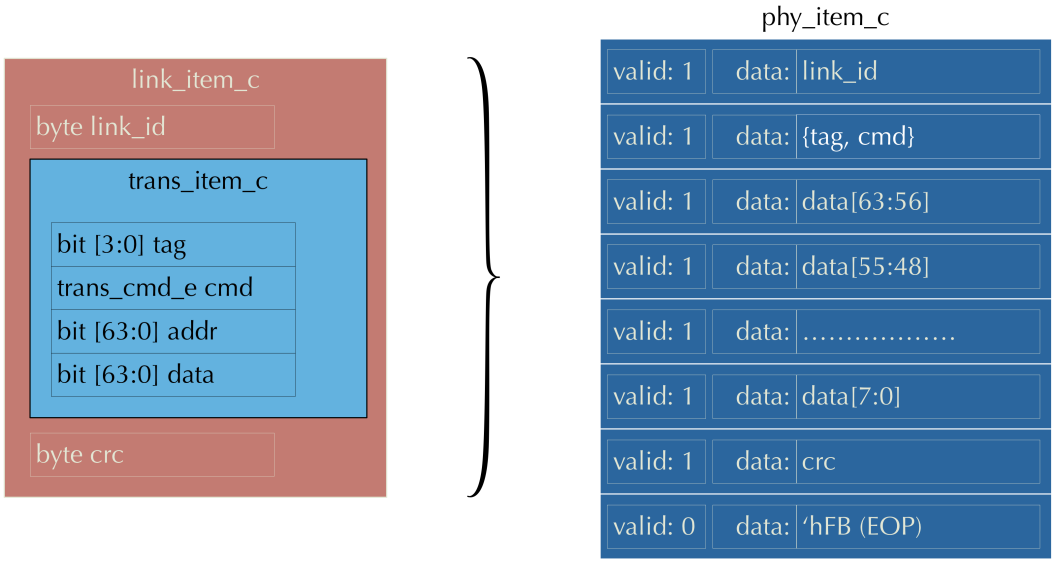

Figure 1: Data Encapsulation Hierarchy

Typically, developers often stop at data encapsulation. But the protocol itself also has a hierarchical nature. Dealing with the link-level's acknowledgements and retries is a separate concern from handling training and idle symbols at the physical level. Developing a robust, debuggable environment that handles all of these scenarios will benefit from a layered approach.

At the highest level of abstraction are the various scenarios for which this interface is responsible. The Hawkins interface facilitates memory accesses, so it may be used for device register configurations, peripheral addressing, or as a network management interface. Regardless of its application, users of this environment will want to interact only at the transaction level and leave the secondary details to the code.

3.2.Keep Your Drivers Simple

Experienced verification engineers will recognize that creating randomizable packets for the Hawkins interface is not the biggest challenge. Managing the protocol in between the packets is the real issue. Drivers send packet data from the sequences, but they also must send the idles and the training and all of the other physical symbols of the Hawkins interface. The real engineering challenge is: at any point in time what should the driver send next?

As stated earlier, sequencers could have been called arbiters. The sole purpose of a sequencer is to decide between multiple sequence items at once. Having the sequencer make the decisions for the driver dramatically simplifies the design of the driver, the monitor, and even the sequence item.

In this pattern, the sequence item, or request, maps closely to the signals of the interface. As shown below, both the phy_item_c and the hawk_intf mirror one another directly.

hawk_phy_item.sv

```
class phy_item_c extends uvm_sequence_item;
   `uvm_object_utils_begin(hawk_pkg::phy_item_c)
      `uvm_field_int(valid, UVM_DEFAULT)
      `uvm_field_int(data, UVM_DEFAULT | UVM_HEX)
   `uvm_object_utils_end

   //---------------------------------------------------------------
   // Group: Fields

   // var: valid
   // Valid Signal
   rand bit valid;

   // var: data
   // Data Signal
   rand byte unsigned data;
```

hawk_intf.sv

```
interface hawk_intf(input logic clk,
                    input logic rst_n);
   import uvm_pkg::*;

   //---------------------------------------------------------------
   // Group: Signals

   // var: valid
   // Valid Wire
   logic valid;

   // var: data
   // Data Signal
   logic [7:0] data;
```

With this implementation, the driver() task in the driver class pulls the next item from the sequencer and sends it. If the sequencer has nothing to send, it will clear the interface (in the case of the Hawkins interface, this happens to be an illegal condition). Shown below is the driver task.

hawk_drv.sv

```
   virtual task driver();
      // ensure that we start on a clock edge
      @(vi.drv_cb);

      forever begin
         // get the next item to send. if none is immediately
         // available, reset the interface, get the next one,
```

```
        // and sync to a clock edge
        seq_item_port.try_next_item(req);
        if(!req) begin
           vi.drv_cb.valid <= 1'b0;
           vi.drv_cb.data <= 'h0;
           seq_item_port.get_next_item(req);
           @(vi.drv_cb);
        end

        // drive this item for one clock
        vi.drv_cb.valid <= req.valid;
        vi.drv_cb.data <= req.data;
        seq_item_port.item_done();
        @(vi.drv_cb);
     end
  endtask : driver
```

Likewise, on every clock cycle the monitor's monitor() task pulls the valid and the data off of the interface, creates a new sequence item, and pushes it out of its analysis port.

hawk_mon.sv

```
  virtual task monitor();
     phy_item_c item;
     forever begin
        // wait until the next clock
        @(vi.mon_cb);
        // create the item from the signal values
        item = phy_item_c::type_id::create("item");
        item.valid = vi.mon_cb.valid;
        item.data = vi.mon_cb.data;
        // announce the item via the analysis port
        phy_item_port.write(item);
     end
  endtask : monitor
```

One of the great benefits of developing thin drivers and monitors is that it makes them much easier to replace when migrating a testbench to an emulation system. To be emulation-friendly, environments must substitute drivers and monitors with synthesizable task and function calls. When protocol handling is not performed by the driver, this process becomes much easier.

Having relegated the driver, monitor, and sequence item classes to simple designs, the interface's protocol details must now be handled by the sequences.

3.3.How to Chain Your Sequencers

Sequences aren't just randomizable data containers. When their body tasks are launched and operate in a forever loop, they become permanent threads in the environment, not unlike a component's run-time phase. In this way, a sequence can be made to perform all of the jobs of a protocol level, receiving data items from the level above and transmitting their own data items to the level below. Sequences that do this are called chaining sequences, because they operate on chained sequencers.

- **Chained Sequencer**

A chained sequencer[2] is a sequencer that sends items to a driver, but can also receive items from another sequencer. A chained sequencer has at least one continuously running sequence called the chaining sequence. Chained sequencers have request items that they send downstream, and different request items that they receive from upstream.

- **Chaining Sequence**

A chaining sequence runs continuously on a chained sequencer and pulls new request items out of its sequencer's upstream item port and uses these to create downstream request items, which are sent to the next sequencer in the chain (or to the driver).

The Hawkins protocol maps neatly to a chained sequencer approach. As shown in Figure 2, three separate sequencers are connected together before sending items into the Hawkins interface driver. One or more chaining sequences are running on each sequencer. Each chained sequencer sends "requests" downstream (towards the driver), and receives "traffic" upstream (away from the driver) and up the protocol stack. Shown in Figure 2 are the primary responsibilities of each of the chaining sequences.

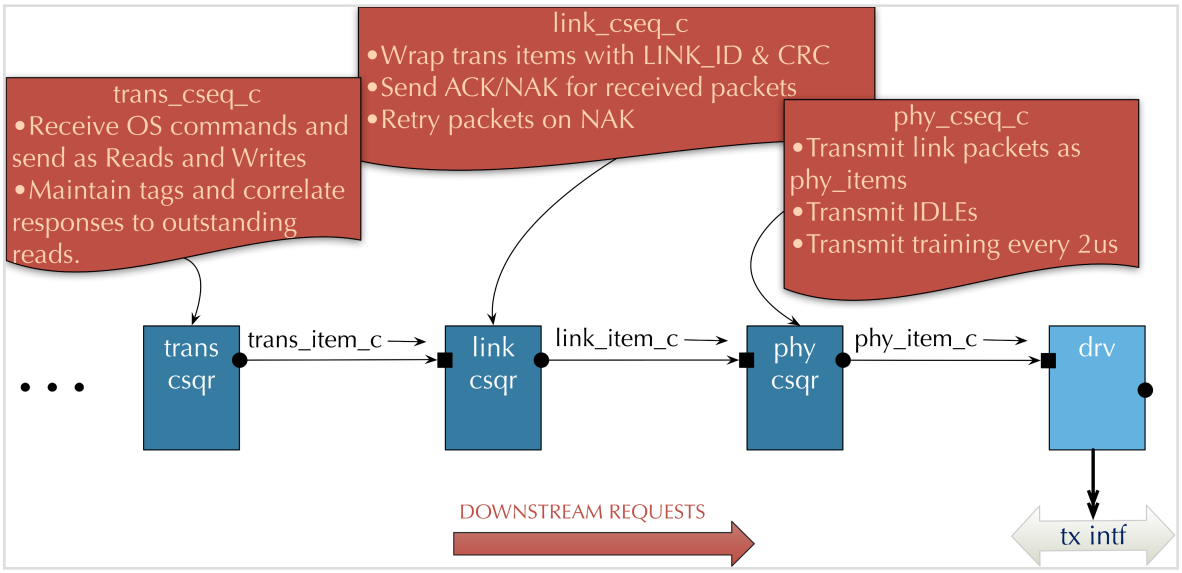

Figure 2: Chained Sequencers for the Hawkins Interface Showing Downstream Direction

There are numerous advantages to this unorthodox approach:

First, there is no one giant class that needs to maintain the complete protocol hierarchy. Each chaining sequence can manage its own level of the protocol. Isolating separate parts of the protocol tends to simplify the design and debug of each. In the Hawkins interface example, the link level chaining sequence is entirely responsible for calculating CRCs and managing its own retry buffer, while caring nothing for training symbols or reads and writes.

Secondly, multiple chained sequences may operate in parallel on the same chained sequencer. By using varying priority levels, this further isolates the different tasks between the levels of the protocol. For example, the physical level chaining sequence may be split into three separate

[2] Chained sequencers and chaining sequences are elsewhere referred to as layered agents and sequences [2] or translation sequences [3]. A chained sequencer approach differs from other methods by using multiple TLM interfaces for bidirectional traffic flow, and by providing reusable base classes to promote a uniform implementation.

sequences: one that sends IDLE symbols to the driver at a low-level of priority; a second that receives link-level items, creates physical level items from them, and transmits them to the driver; and a third that sends four consecutive training symbols to the driver every two microseconds.

A third advantage is that one can break the chain at any point and send its request items either directly into a driver, or skip the device under test altogether and send them via TLM ports directly to the corresponding chained sequencer. This allows engineers to simulate the design at different levels of abstraction. If it is desirable to operate at a transaction-level of the hierarchy only, simulations can be sped up considerably by skipping the link and physical levels entirely.

3.4.Upstream Traffic

The requests in the chaining sequence described in Figure 2 all move towards the driver (the downstream direction) which delivers them to the TX interface. The verification environment must also be able to respond to data monitored on the RX interface. As shown in Figure 3, the monitor attached to the RX interface is designed to emit these items out of its phy_item_port.

Chained sequencers receive traffic items from the downstream sequencer. These traffic items represent the data being **received** by this agent. The chained sequencer of a given protocol layer processes those traffic items relevant to its level, and forwards the remaining traffic items upstream.

Typical sequencer-sequence interaction involves both requests and “responses” being transported bidirectionally over the same seq_item_pull_port. Using this method, drivers must correlate each response with an associated request before pushing it back upstream. Traffic items differ from response items in that there is no direct correlation. As will be shown later, response items may still be used where appropriate. The primary upstream transmission, however, are traffic items.

The following definitions may help keep these ideas clear:

- **Requests.** These are sequence items that go from upstream to downstream (towards the driver).
- **Traffic.** These are sequence items that are monitored and have been received by the agent. Traffic items flow upstream.
- **Responses.** Responses are sequence items that flow upstream and correlate directly to request items. Responses are usually only used to exit the chain upstream.

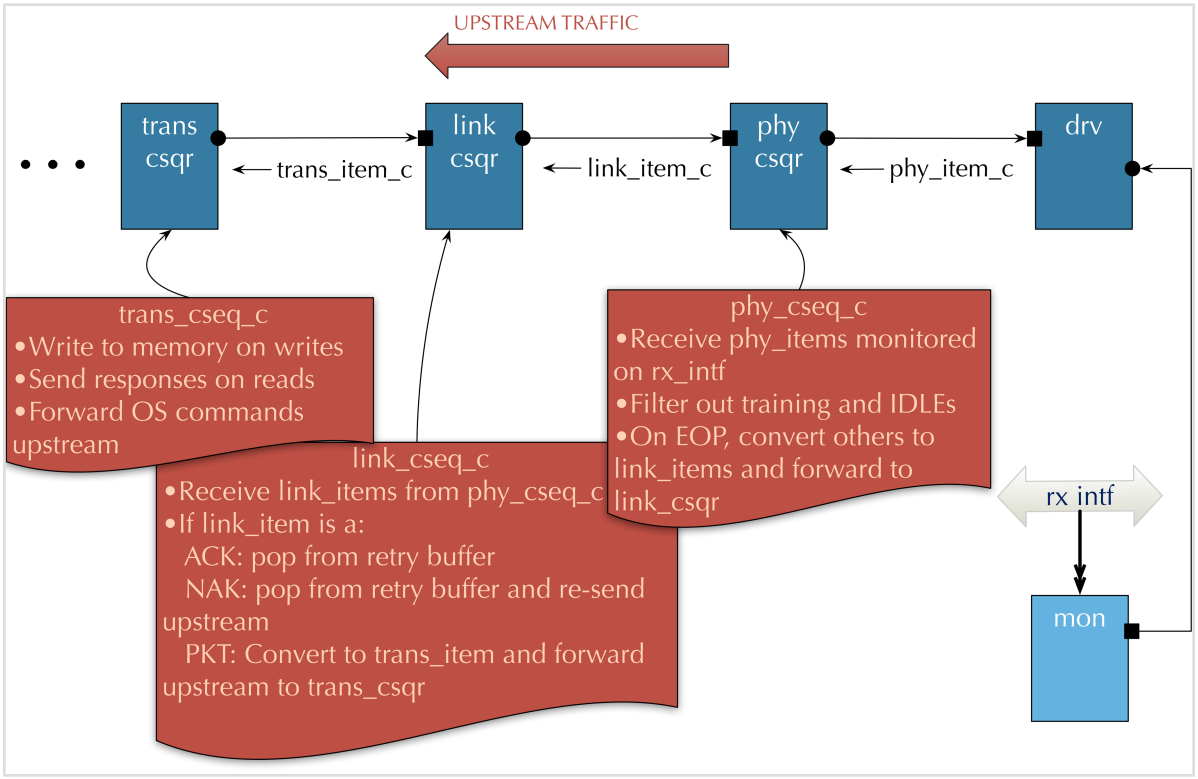

Figure 3: Chained Sequencers on the Hawkins Interface Showing Upstream Direction

4.Implementation

This section will present the details of how chained sequencers and chaining sequences are built.

4.1.Chained Sequencer

The chained sequencer is a parameterized class deriving from the uvm_sequencer class. As shown in Figure 4, the uvm_sequencer normally has only one TLM interface—seq_item_export—which is used by drivers to pull the next sequence item to be driven. UVM Sequencers are parameterized with both request and response types.

Figure 4: Standard uvm_sequencer

Chained sequencers contain additional TLM interfaces and are parameterized by both upstream and downstream request types, as well as upstream and downstream traffic types. Figure 5 shows the additional ports that are added.

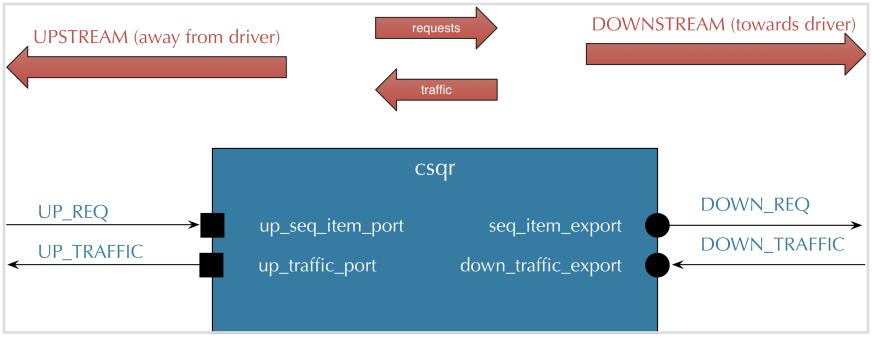

Figure 5: Chained Sequencers

The base uvm_sequencer class is parameterized with the DOWN_REQ sequence item type, and as before is expected to be connected to a seq_item_port such as that of a driver. When connected to another chained sequencer, it connects to the other sequencer's up_seq_item_port.

Traffic moving upstream arrives via the uvm_analysis_export named down_traffic_export. The prefix 'down' indicates that it comes from the downstream and uses the parameter type DOWN_TRAFFIC. Chaining sequences operate on these items as they arrive. Those that are useful to the protocol layer are processed, and the rest are converted to the UP_TRAFFIC type and sent out the chained sequencer's up_traffic_port.

Figure 6 shows the connectivity of the link-level chained sequencer (link_csqr). As shown in Figure 2, transaction-level request items flow from the transaction sequencer to the link-level sequencer. The link-level sequence converts these to link-level request items and forwards them downstream to the physical-level sequencer.

Meanwhile, link-level traffic flows from the physical-level up to the link-level. These are then converted to transaction-level items and continue upstream to the transaction-level.

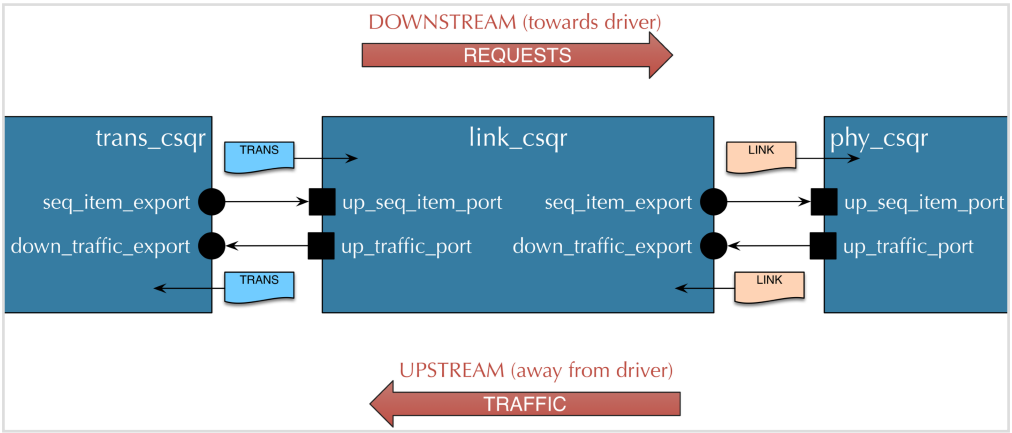

Figure 6: Link-Level Chained Sequencer Connections

The implementation details of a chained sequencer are shown in Figure 7. The chained sequencer instantiates two uvm_tlm_analysis_fifos to receive upstream requests and downstream traffic. As upstream requests flow into a chained sequencer, its up_fetcher() task pulls them from the upstream sequencer and writes them into the FIFO, to be retrieved by the chaining sequence. Likewise, monitored traffic from downstream are written directly into a different FIFO. These implementations are hidden from the user by functions and tasks provided by the chaining sequence base class.

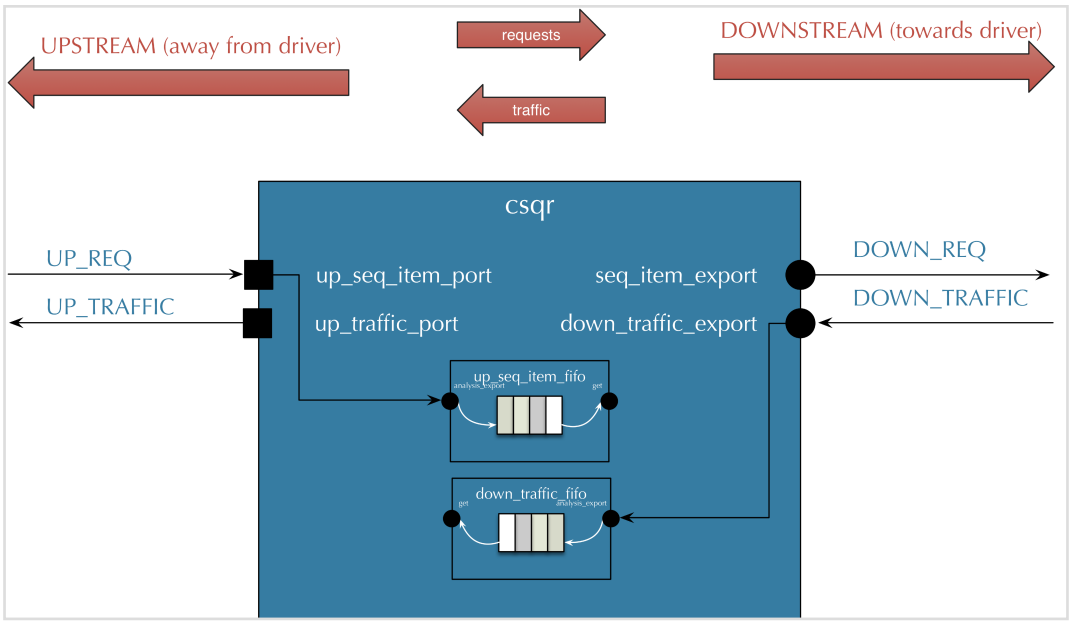

Figure 7: Chained Sequencer Implementation

4.2.Chaining Sequence

A parameterized base class is provided that defines the most common operations of a chaining sequence. Users extend from the class and provide the implementation details as needed. As written, the sequence forks two processes off when its body task is run: handle_up_items() and handle_down_traffic(). These methods are often overwritten, but they provide a working template for how chaining sequences may be coded.

The FIFO implementations within the chained sequencer are obscured from the user by providing the following methods in the chaining sequence base class.

cmn_cseq.sv

```
  virtual task get_next_up_item(ref UP_REQ _item);
  virtual function bit try_get_next_up_item(ref UP_REQ _item);
  virtual function void put_up_traffic(UP_TRAFFIC _up_traffic);
  virtual function void put_up_response(UP_TRAFFIC _up_traffic);
  virtual function bit try_get_down_traffic(
      ref DOWN_TRAFFIC _traffic
   );
  virtual task get_down_traffic(ref DOWN_TRAFFIC _down_traffic);
```

5.Breaking the Chain

An important advantage of layered protocols is the ability to simulate at higher levels of abstraction by skipping lower-levels. Chained sequencers facilitate this model by offering the configuration setting, chain_break.

Figure 8 shows what the environment looks like when the physical-layer is removed. In this scenario, the physical-level chained sequencer, the drivers, the monitors, and even the DUT are bypassed. Instead, the link-level sequencers talk directly to one another.

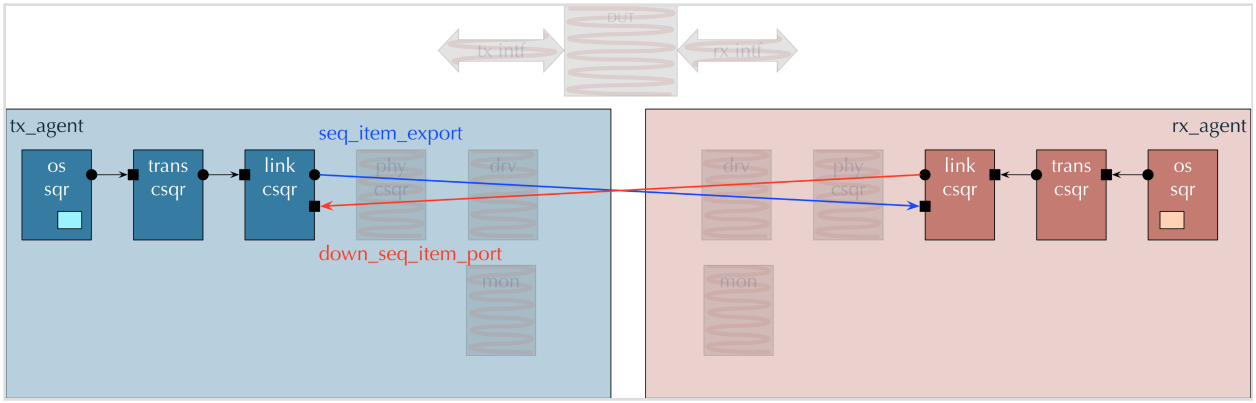

Figure 8: Breaking Chained Sequencers at the Link-Layer

The link_csqr of the tx_agent is connected directly to the rx_agent's link_csqr. Because the rx_agent normally expects upstream traffic from its physical-level over an analysis export, another TLM interface must be added to the chained sequencer when it operates in a broken chain mode. This interface is of type uvm_seq_item_pull_port and is called down_seq_item_port. This additional interface is shown below in Figure 9. Because this port feeds into the same downstream traffic FIFO as before, the link-level chaining sequence need not be re-programmed nor even be aware that the chain is broken. Only the TLM connections need modification.

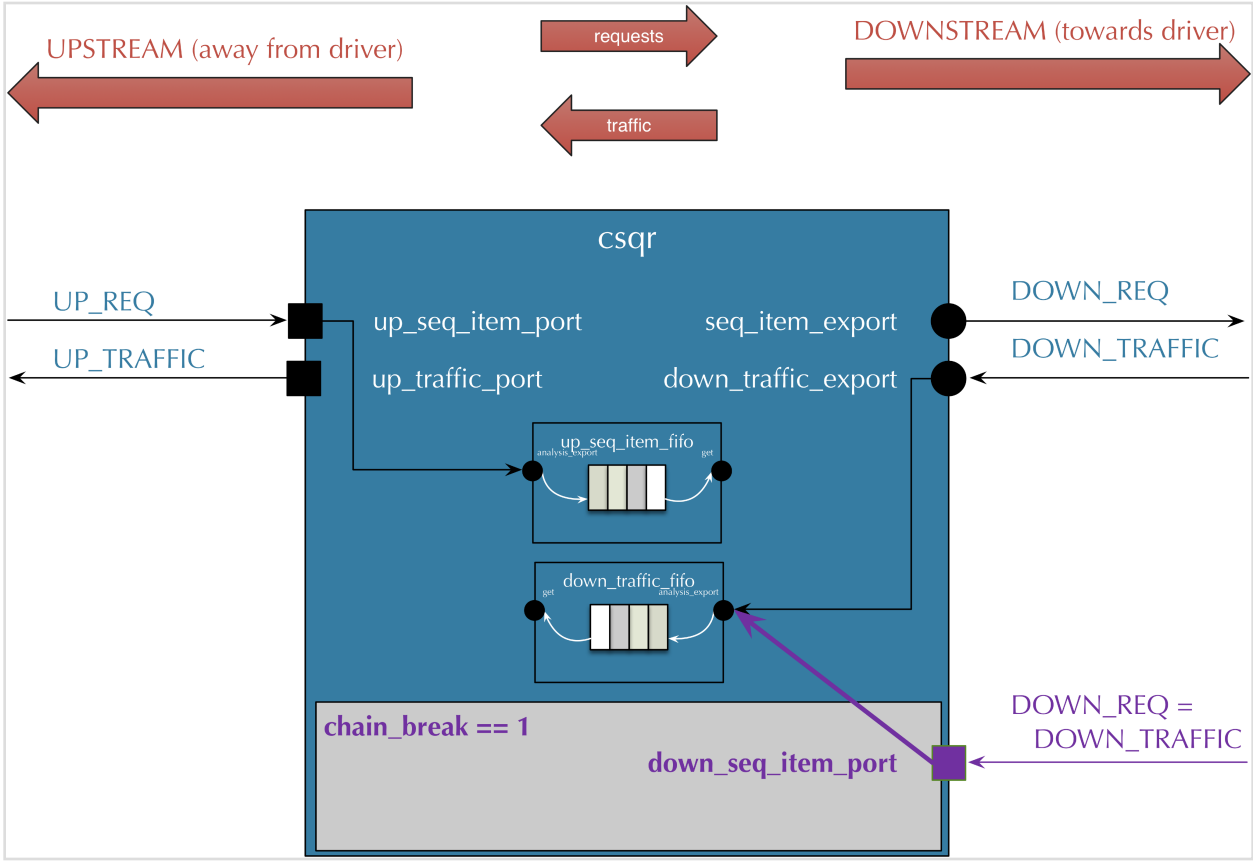

Figure 9: Chained Sequencer When chain_break is Set

With this implementation, the link-level chaining sequence is isolated from the downstream arrangement. This has profound implications. As stimulus protocols grow in complexity, there is a greater need to test the verification environment itself—independently of the device under test.

Moreover, layers may be tested without the lower-layers in very fast simulations. Because these complex sequences need no modifications, this additional benefit is essentially free.

6.Protocol Management with Priorities

Consider the requirement that four data training cycles must be injected every two microseconds. The Hawkins specification is clear that this is the highest priority. Without a layered protocol environment, a naïve algorithm to push these cycles might appear as follows:

```
if($time - 2000 == last_training_time) begin
   inject_training();
   last_training_time = $time;
end else if(data_cycles_to_send) begin
   ...
end
```

This algorithm would need to include all possible physical-level types before it was complete—acknowledgements, IDLE cycles, and end-of-packet (EOP) cycles. The method above is a form of arbitration that must be developed, debugged, and maintained. When stimulus is generated with protocol layering, however, the arbitration is done by the sequencer and a simple priority scheme is used to determine what happens next.

The following sequence injects training cycles:

hawk_phy_train_seq.sv

```
virtual task body();
   phy_item_c train_item;

   forever begin
      #(2us);
      repeat(4) begin
         `uvm_do_pri_with(trn_item, TRAIN_PRI, {
           valid == 0;
           data == TRAIN;
         })
      end
   end
endtask : body
```

This sequence transmits IDLE cycles:

hawk_phy_idle_seq.sv

```
virtual task body();
   byte unsigned idle_cnt = 0;
   phy_item_c idle_item;

   forever begin
      `uvm_do_pri_with(idle_item, IDLE_PRI, {
         valid == 0;
         data == idle_cnt;
      })
      idle_cnt = (idle_cnt == 'hFF)? 0 : idle_cnt + 1;
   end
endtask : body
```

Both of these sequences, as well as the task that handles the sequence items from upstream, run in parallel on the PHY chained sequencer. The arbitration is encoded by these priorities, which were developed according to the requirements:

hawk_types.sv:

```
// enum: priority_e
// Priorities for sending traffic
typedef enum int {
   IDLE_PRI    = 100,
   PKT_PRI     = 200,
   REPLAY_PRI  = 400,
   ACK_NAK_PRI = 500,
   TRAIN_PRI   = 1000
} priority_e;
```

These sequences are elementary to develop because they are written in isolation. Following a methodology that enables the separation of concerns design principle is highly advantageous. Debugging, adding, removing, and modifying other sequences that operate in isolation becomes a simple task.

7.Conclusions

As stimulus generation grows more complicated, protocol layering is well known to have numerous advantages to flattened techniques:

- **Design modularity:** Each layer operates in isolation of the other layers and is therefore easier to develop, debug, and maintain. Modularity also allows layers to be swapped out with alternative implementations so long as they conform to the same sequence item interface. This technique can be used, for example, to inject errors or to provide faster implementations to speed simulations.
- **Protocol Modularity.** The protocol specific to stimulus generation may also be independent of the data. Sometimes, managing these rules can be more complicated than generating the data. Chaining sequences provide additional isolation when protocols are layered.
- **Breaking the chain:** Only the TLM interfaces that connect chained sequencers together must be re-wired to run simulations at higher-levels of abstractions. Changing abstraction layers is a snap because sequence algorithms do not need modification.
- **Thin drivers and monitors:** Chaining enforces the creation of “thin” drivers and monitors. By simplifying these components, they can easily be swapped out with synthesizable models for emulation, or for various other reasons.
- **Sequencer arbitration:** Giving the UVM sequencer the job of arbitration—as it was intended—saves one the effort of developing a complex arbitration scheme.

Layering sequences is not an entirely new concept. Previous methods, though, have expected users to manage with the classes available in the standard UVM library and have left the challenge of forwarding upstream traffic to the developer. Chained sequencers provide a uniform methodology in which layered stimulus generation can easily be developed.

8.Code

8.1.The Hawkins Environment

A complete test environment containing two Hawkins agents was created. In this environment, each agent performs memory reads and writes on the other, and tests are provided to show how knobs can be manipulated to increase NAK percentages, to inject bad CRCs, or to operate at higher levels of abstraction.

The environment conforms to the coding guidelines presented in Advanced UVM [1]. The environment also contains numerous examples from and makes use of all of the techniques described in that textbook. The environment code is available for download at:

https://github.com/advanced-uvm/hawkins

The complete hierarchy of the hawkins environment is shown in Figure 10. Each agent contains the chained sequencers as shown from Figures 2 and 3. The agents also contain an operating system-level sequencer, os_sqr, that represents how a user might interact with chained sequencers. The sequences which operate on the os_sqr are:

- **os_mem_seq_c:** The purpose of this sequence is to manage the agent's memory. This sequence receives the traffic from the transaction-layer chaining sequence in the form of OS-level sequence items. These items represent simple reads, writes, and responses. The os_mem_seq_c class writes data to the sequencer's instantiated memory and provides responses to reads.
- **os_main_seq_c:** This is the main test-level sequence. It creates a random list of fifty 64-bit addresses that will be tested in the the opposite agent's memory space. It then primes these locations by performing fifty random writes, and then performs 100 random reads to these addresses. Each response is checked for correct data by comparing to the data that was previously written.

A key feature of the os_main_seq_c class is that it uses the familiar request/response mechanism of the uvm_seq_item_pull_port. The transaction-level chaining sequence sends responses back via its up_seq_item_port rather than sending them out of the up_traffic_port. It does this by calling put_up_response() rather than put_up_traffic(). This permits users of agents that use chained sequencers to be isolated from this design choice.

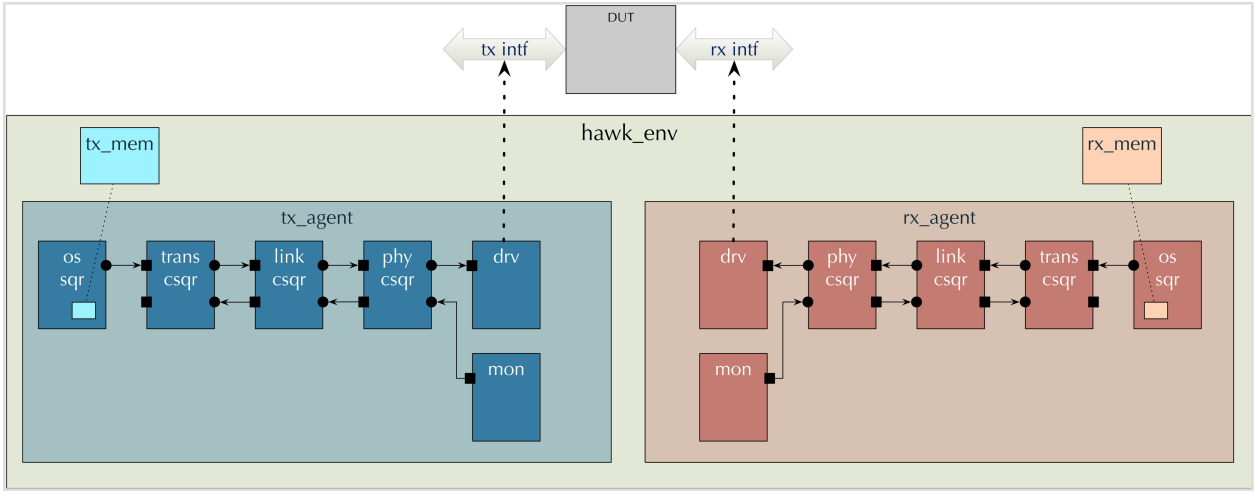

Figure 10: Hawkins Environment

8.2.Chained Sequencer Class

The following code is also available within the hawkins environment repository, in the file:

https://github.com/advanced-uvm/hawkins/blob/master/verif/vkits/cmn/cmn_csqr.sv.

Because the GitHub repository will be maintained after this document's finalization, readers should consider fetching the chained sequencer class from there.

cmn_csqr.sv

```
class csqr_c#(type UP_REQ=uvm_sequence_item, UP_TRAFFIC=UP_REQ,
              DOWN_REQ=uvm_sequence_item, DOWN_TRAFFIC=DOWN_REQ)
              extends uvm_sequencer#(DOWN_REQ);
   `uvm_component_utils_begin(csqr_c)
      `uvm_field_int(chain_break, UVM_DEFAULT)
      `uvm_field_enum(uvm_sequencer_arb_mode, sqr_arb_mode,
                      UVM_DEFAULT)
   `uvm_component_utils_end

   //---------------------------------------------------------------
   // Group: Configuration Fields

   // var: sqr_arb_mode
   // The sequencer arbitration mode
   uvm_sequencer_arb_mode sqr_arb_mode = UVM_SEQ_ARB_STRICT_FIFO;

   // var: chain_break
   // Set this configuration variable if this chained sequencer is the
   // end of the chain but does NOT send to a driver.
   //
   // When set, the down_seq_item_port is created and takes the place
   // of another chained sequencer's driver. Downstream requests are
   // then automatically pulled from the opposite side's chained
   // sequencer.
   bit chain_break = 0;

   //---------------------------------------------------------------
   // Group: TLM Ports

   // var: up_seq_item_port
   // Gets the next sequence from the upstream sequencer
   uvm_seq_item_pull_port#(UP_REQ) up_seq_item_port;

   // var: up_traffic_port
   // Drives traffic back upstream
   uvm_analysis_port#(UP_TRAFFIC) up_traffic_port;

   // var: down_traffic_export
   // Receives traffic from the downstream sequencer
   uvm_analysis_export #(DOWN_TRAFFIC) down_traffic_export;

   // var: down_seq_item_port
   // Pulls downstream items from another chained sequencer just as a
   // driver would.
   // only created when chain_break == 1
```

```
   uvm_seq_item_pull_port#(DOWN_REQ, DOWN_TRAFFIC) down_seq_item_port;

   //------------------------------------------------------------------
   // Group: Fields
   // var: up_seq_item_fifo
   // Receives upstream sequence items
   uvm_tlm_analysis_fifo#(UP_REQ) up_seq_item_fifo;

   // var: down_traffic_fifo
   // Receives the traffic from the downstream sequencer
   uvm_tlm_analysis_fifo#(DOWN_TRAFFIC) down_traffic_fifo;

   //------------------------------------------------------------------
   // Group: Methods
   function new(string name="csqr",
                uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   ////////////////////////////////////////////
   // func: build_phase
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      up_traffic_port = new("up_traffic_port", this);
      down_traffic_export = new("down_traffic_export", this);
      down_traffic_fifo = new("down_traffic_fifo", this);
      up_seq_item_port = new("up_seq_item_port", this);
      up_seq_item_fifo = new("up_seq_item_fifo", this);
      set_arbitration(sqr_arb_mode);
      if(chain_break)
         down_seq_item_port = new("down_seq_item_port", this);
   endfunction : build_phase

   ////////////////////////////////////////////
   // func: connect_phase
   // Connect downstream traffic fifo to export
   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      down_traffic_export.connect(down_traffic_fifo.analysis_export);
   endfunction : connect_phase

   ////////////////////////////////////////////
   // func: run_phase
   virtual task run_phase(uvm_phase phase);
      fork
         up_fetcher();
         if(chain_break)
            downstream_driver();
      join
   endtask : run_phase

   ////////////////////////////////////////////
   // func: up_fetcher
```

```
   // continuously get the next item from upstream and write it
   // into the up_seq_item_fifo. Then tell the upstream sequencer that
   // the item is done.
   virtual task up_fetcher();
      UP_REQ item;

      forever begin
         up_seq_item_port.get_next_item(item);
         up_seq_item_fifo.analysis_export.write(item);
         up_seq_item_port.item_done();
      end
   endtask : up_fetcher

   ////////////////////////////////////////////
   // func: downstream_driver
   // Pulls the requests out of the down_seq_item_port as a
   // driver would. Converts these to downstream traffic and pushes
   // it into the downstream fifo
   virtual task downstream_driver();
      DOWN_REQ down_req;
      DOWN_TRAFFIC down_traffic;

      forever begin
         down_seq_item_port.get_next_item(down_req);
         down_traffic = convert_down_req(down_req);
         down_traffic_fifo.analysis_export.write(down_traffic);
         down_traffic_user_task(down_traffic);
         down_seq_item_port.item_done();
      end
   endtask : downstream_driver

   ////////////////////////////////////////////
   // func: convert_down_req
   // Convert a downstream request to downstream traffic
   // By default, these are the same types and a cast will work.
   // Override if necessary.
   virtual function DOWN_TRAFFIC convert_down_req(
      ref DOWN_REQ _down_req
   );
      $cast(convert_down_req, _down_req);
   endfunction : convert_down_req

   ////////////////////////////////////////////
   // func: down_traffic_user_task
   // Allow the user to handle what happens to downstream traffic
   // before its item_done is called. This task is only ever called
   // when chain_break is set.
   virtual task down_traffic_user_task(
      ref DOWN_TRAFFIC _down_traffic
   );
   endtask : down_traffic_user_task
endclass : csqr_c
```

8.3.Chaining Sequence Base Class

The following code is also available within the hawkins environment repository, in the file:

https://github.com/advanced-uvm/hawkins/blob/master/verif/vkits/cmn/cmn_cseq.sv.

Because the GitHub repository will be maintained after this document's finalization, readers should consider fetching the chaining sequencer class from there.

cmn_cseq.sv

```
class cseq_c#(type DOWN_REQ=uvm_sequence_item,
              DOWN_TRAFFIC=DOWN_REQ,
              UP_REQ=DOWN_REQ,
              UP_TRAFFIC=DOWN_REQ,
              CSQR=cmn_pkg::csqr_c)
              extends uvm_sequence#(DOWN_REQ);

   `uvm_object_utils_begin(cmn_pkg::cseq_c)
   `uvm_object_utils_end
   `uvm_declare_p_sequencer(CSQR)

   //------------------------------------------------------------
   // Group: Methods
   function new(string name="cseq_seq");
      super.new(name);
   endfunction : new

   ////////////////////////////////////////////
   // func: body
   virtual task body();
      fork
         handle_up_items();
         handle_down_traffic();
      join
   endtask : body

   ////////////////////////////////////////////
   // func: handle_up_items
   // Get items from the upstream chained sequencer and send them
   // as downstream items
   virtual task handle_up_items();
      UP_REQ up_req_item;
      DOWN_REQ down_req_item;

      forever begin
         get_next_up_item(up_req_item);
         `uvm_create(down_req_item)
         make_down_req(down_req_item, up_req_item);
         `uvm_send(down_req_item)
      end
   endtask : handle_up_items

   ////////////////////////////////////////////
   // func: handle_down_traffic
   // Get traffic from downstream, create upstream traffic, and
```

```
   // push it up
   virtual task handle_down_traffic();
      DOWN_TRAFFIC down_traffic;
      UP_TRAFFIC up_traffic;

      forever begin
         get_down_traffic(down_traffic);
         up_traffic = create_up_traffic(down_traffic);
         if(up_traffic)
            put_up_traffic(up_traffic);
      end
   endtask : handle_down_traffic

   ////////////////////////////////////////////
   // func: get_next_up_item
   // Get the next upstream item request
   virtual task get_next_up_item(ref UP_REQ _item);
      p_sequencer.up_seq_item_fifo.get(_item);
   endtask : get_next_up_item

   ////////////////////////////////////////////
   // func: try_get_next_up_item
   // Returns a 1 if an upstream item was available, otherwise
   // returns 0
   virtual function bit try_get_next_up_item(ref UP_REQ _item);
      try_get_next_up_item =
         p_sequencer.up_seq_item_fifo.try_get(_item);
   endfunction : try_get_next_up_item

   ////////////////////////////////////////////
   // func: put_up_traffic
   // Send traffic upstream.
   virtual function void put_up_traffic(UP_TRAFFIC _up_traffic);
      p_sequencer.up_traffic_port.write(_up_traffic);
   endfunction : put_up_traffic

   ////////////////////////////////////////////
   // func: put_up_response
   // Send a response upstream using the sequence item port
   virtual function void put_up_response(UP_TRAFFIC _up_traffic);
      p_sequencer.up_seq_item_port.put_response(_up_traffic);
   endfunction : put_up_response

   ////////////////////////////////////////////
   // func: try_get_down_traffic
   // Returns a 1 if any downstream traffic item was available,
   // otherwise returns 0. Fills in the _traffic
   virtual function bit try_get_down_traffic(
      ref DOWN_TRAFFIC _traffic
   );
      return p_sequencer.down_traffic_fifo.try_get(_traffic);
   endfunction : try_get_down_traffic

   ////////////////////////////////////////////
```

```
   // func: get_down_traffic
   // Return the next available piece of traffic from downstream
   virtual task get_down_traffic(ref DOWN_TRAFFIC _down_traffic);
      p_sequencer.down_traffic_fifo.get(_down_traffic);
   endtask : get_down_traffic

   ////////////////////////////////////////////
   // func: make_down_req
   // make a downstream request item from an upstream
   // request item
   virtual function DOWN_REQ make_down_req(
      ref DOWN_REQ _down_req_item,
          UP_REQ _up_req_item
   );
   endfunction : make_down_req

   ////////////////////////////////////////////
   // func: create_up_traffic
   // Create an upstream traffic item from the downstream traffic
   virtual function UP_TRAFFIC create_up_traffic(
      ref DOWN_TRAFFIC _down_traffic);
   endfunction : create_up_traffic
endclass : cseq_c
```

9.Hawkins Specification

To help readers understand the code in the Hawkins environment, the following is a complete specification of the Hawkins interface.

9.1.Transaction Level

Each command is either a read, write, or response command. The command opcodes are as follows:

```
RD  = 4'h1
WR  = 4'h2
RSP = 4'h4
```

9.1.1.Read Commands

A read command is composed of nine consecutive bytes:

```
| TAG[3:0],RD | ADDR[63:56] | ADDR[55:48] | ... | ADDR[7:0] |
```

The TAG is a four bit value. Only sixteen reads may be outstanding at any time. The RD opcode is 4'b0001.

9.1.2.Write Commands

A write command is composed of seventeen consecutive bytes:

```
| 4'b0,WR | ADDR[63:56] | ADDR[55:48] | ... | ADDR[7:0] |
|         | DATA[63:56] | DATA[55:48] | ... | DATA[7:0] |
```

Writes have no TAG value. The WR opcode is 4'b0010.

9.1.1.Response Commands

The response command is composed of nine consecutive bytes

```
| TAG[3:0],RSP | DATA[63:56] | DATA[55:48] | ... | DATA[7:0] |
```

The TAG must match an outstanding read request. The RSP opcode is 4'b0100.

9.2.Link Level

9.2.1.Packets

Each command received from the transaction level is wrapped as follows:

```
| LINK_ID | <transaction-command> | CRC |
```

The link ID is the link identifier value of the node that is transmitting.

The CRC is the sum of all data in the packet, truncated to the least significant eight bits. The CRC exists only to show the hierarchical layering of the protocol.

9.2.2.Acknowledgements

The Link level sends an ACK for every packet received with correct CRC. The ACK is a single beat of value 'hFC while the valid signal is low.

The Link level sends a NAK for every packet received with incorrect CRC. The NAK is a single beat of value 'hFE when the valid signal is low.

9.2.3.Retries

The link level contains a retry buffer. Each packet sent out of the link level is stored in FIFO order. When an ACK is received, the oldest packet is popped from the retry buffer. When a NAK is received, the oldest packet is popped from the retry buffer and re-tried as soon as possible.

9.3.Physical Level

The Hawkins interface is composed of a TX and an RX interface. Each is eight bits wide with its own valid signal. Both operate on the same clock and reset. While valid is low, a physical level transmitter transmits Physical Level Symbols, corresponding to the following list:

```
IDLE - data inside ['h00:'hF0]
EOP  - data == 'hFB
ACK  - data == 'hFC
NAK  - data == 'hFE
Training - data == 'hFF
```

All other values are reserved.

9.3.1.Packet Data

When valid is high, the byte data on the interface is in unsigned bytes. The valid signal may go low in the middle of a packet, with the packet continuing on the next cycle where valid is high. When the packet is completed, the physical level will send an EOP symbol.

9.3.2.Data Training

Every 2µs, four consecutive training cycles must appear on the bus. The data training may interrupt a packet if necessary.

9.3.3.Data Idle

When valid is low and the transmitter is sending neither ACK, NAK, nor data training symbols, the byte data must be an incrementing number from where previously left off, counting from 8'h0 to 8'hF0 and repeating.

10.References

1. Hunter, Brian. (2015). Advanced UVM (1st ed.). CreateSpace Independent Publishing Platform. Available at http://tinyurl.com/AdvancedUVM or http://tinyurl.com/AdvancedUVM-ebook.

2. Aynsley, John. "Easier UVM Guidelines." Retrieved from https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/layering.

3. Bergeron, Janick, Fabian Delguste, Steve Knoeck, Steve McMaster, Aron Pratt, Amit Sharma. (2013) "Beyond UVM: Creating Truly Reusable Protocol Layering." Paper presented at DVCon 2013, San Jose, CA.