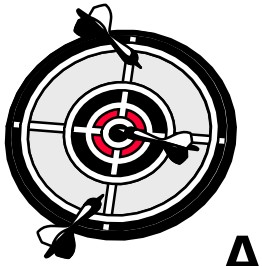
Agenda: Day 1

DAY 1

1. OOP Inheritance Review
2. UVM Structural Overview
3. UVM Transaction
4. UVM Sequence

Unit Objectives

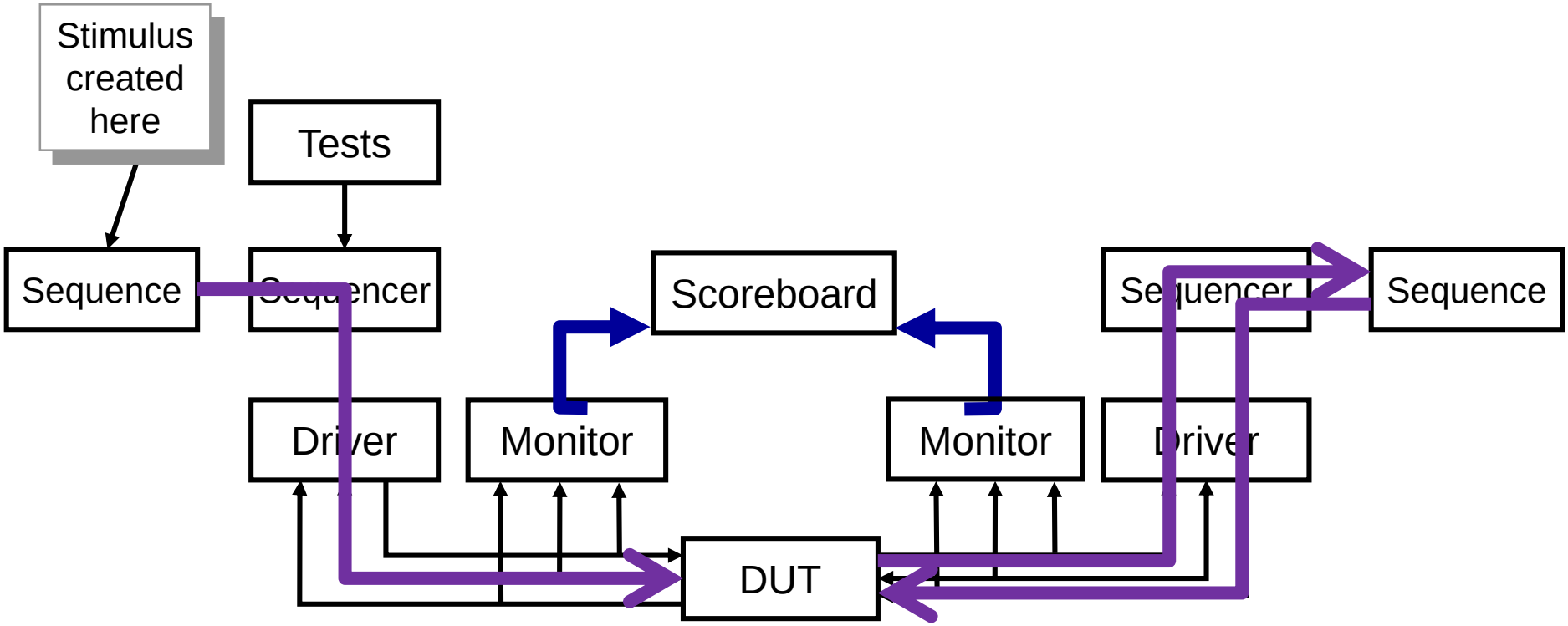


After completing this unit, you should be able to:

- **Build stimulus generator class by inheriting from `uvm_sequence`**
- **Execute sequence explicitly and implicitly**

UVM Transaction Flow - Continued

- **Focused on sequence**



Stimulus Generation Classes

uvm_object

- get_name()
- get_full_name()
- get_type()
- clone()
- copy()
- print()
- sprint()
- compare()
- pack*()
- unpack*()
- record()
- convert2string()

uvm_sequence_item

- set_item_context()
- set_transaction_id()
- get_transaction_id()
- set_sequence_id()
- get_sequence_id()
- set_id_info()
- set_sequencer()
- **get_sequencer()**
- set_parent_sequence()
- **get_parent_sequence()**

uvm_sequence_base

- **set_starting_phase()**
- **get_starting_phase()**
- **set_automatic_phase_objection()**
- **starting_phase**
- **start()**
- pre_start()
- pre_body()
- **body()**
- start_item()
- pre_do()
- mid_do()
- finish_item()
- post_do()
- post_body()
- post_start()
- lock()
- unlock()
- grab()
- ungrab()

UVM 1.2

UVM 1.1

uvm_sequence#(REQ, RSP)

- **REQ req**
- **RSP rsp**
- get_response()

The difference between UVM-1.1 and UVM-1.2 is covered in this unit

Sequence Class

- **The stimulus generator in UVM is called a sequence**
- **User sequence must extended from `uvm_sequence` class and parameterized to the transaction type to be generated**
  - Two handles are available:
    - **`req`** for request to driver and
    - **`rsp`** for response back from driver

uvm_squence #(REQ, RSP)
REQ req **RSP rsp** get_response()

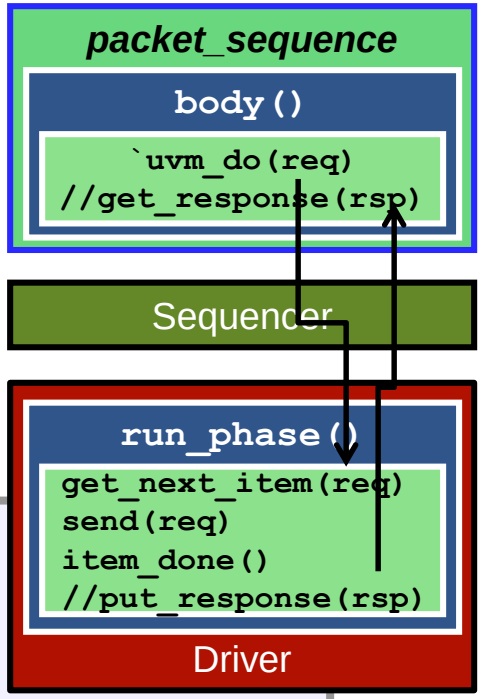
The second parameter defaults to first parameter if not specified

```
class packet_sequence extends uvm_sequence #(packet);
  `uvm_object_utils(packet_sequence)
  function new(string name = "packet_sequence");
    super.new(name);
  endfunction
  // see next slide
```

The string argument in constructor must have a default value, typically the class name

Generate Transactions in Sequence Class

- **Sequence functional code resides in `body()` task**
  - `` `uvm_do `` creates, randomizes and passes transaction to driver through sequencer
  - `` `uvm_do_with `` is the same as `` `uvm_do `` but with additional constraints
  - Optional `get_response()` retrieves response from driver through sequencer
    - See appendix for response implementation




```
  // Continued from previous slide
  virtual task body();
    repeat (10) begin
      `uvm_do(req);
      // or with constraint: `uvm_do_with(req, {da == 3;});
      // optional: get_response(rsp);
    end
  endtask
endclass
```

User Can Manually Create and Send Item

- `uvm_do macro effectively implements the following:

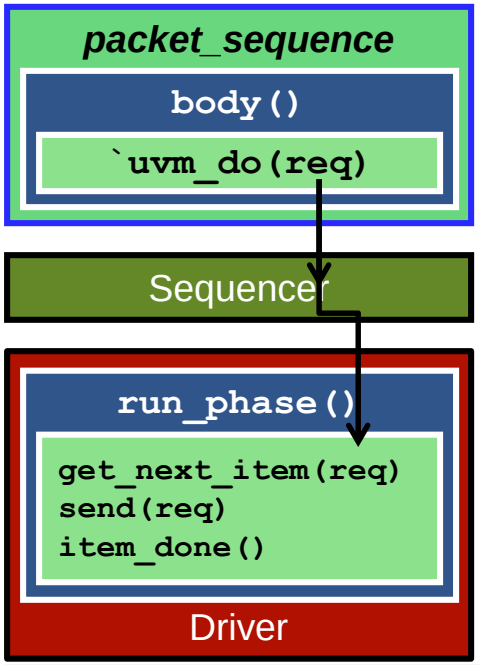
```
// * - Equivalent code, not actual code
`define uvm_do(UVM_SEQ_ITEM) \
  `uvm_create(UVM_SEQ_ITEM) \
  start_item(UVM_SEQ_ITEM); \
  UVM_SEQ_ITEM.randomize(); \
  finish_item(UVM_SEQ_ITEM);
```

- User can manually execute the methods:

```
  virtual task body();
//  Instead of `uvm_do_with(req, {da == 3;})
//  User can call these mechanisms individually
    `uvm_create(req);  // constructs sequence item and sets association with sequence
    start_item(req);   // wait for parent sequencer to get request from driver
    req.randomize() with {da == 3};
    finish_item(req);  // use parent sequencer to pass item to driver and wait for done
  endtask
```

`uvm_do Macro Interaction Detailed

- **Pseudo code illustration:**




```
virtual task body();
  repeat (item_count) begin
    `uvm_do(req);
```

```
`define uvm_do(req)
  `uvm_create(req);
  start_item(req);
  req.randomize();
  finish_item(req);
```

```
task start_item(req);
  sqr.wait_for_grant();
  sqr.begin_tr(req);
endtask
```

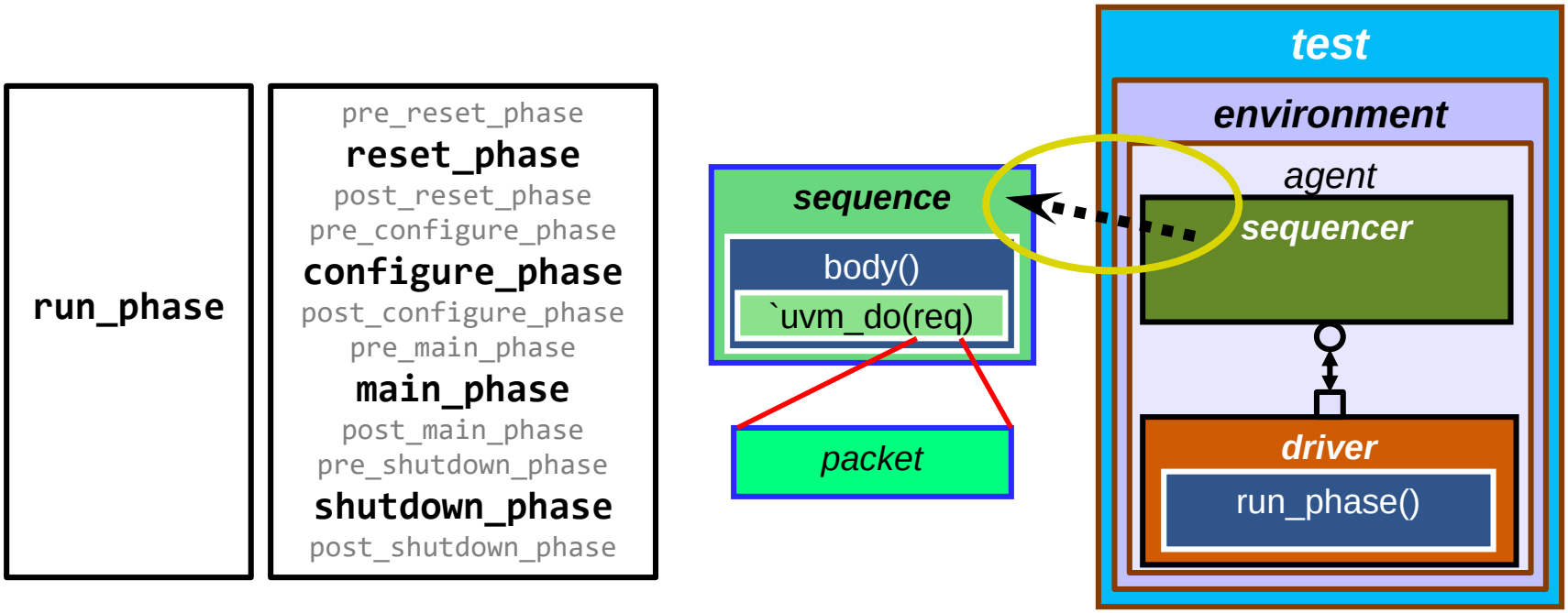
```
class driver …
 virtual task run_phase(…);
  forever begin
   seq_item_port.get_next_item(req);
   send(req);
   seq_item_port.item_done();
  end
 endtask
endclass
```

```
task finish_item(req);
  sqr.send_request(req);
  sqr.wait_for_item_done();
  sqr.end_tr(req);
endtask
```

Caution: Pseudo code, not actual code

Sequence Execution: Starting a Sequence

- **Sequences are executed in task phases**



- **Sequence executes when its `start()` method is called**

`start()` Method in Sequence Class

- The `start()` method calls the `body()` method in turn in the order shown below

```
start() {
  pre_start()   (task)  // UVM-1.1
  pre_body()    (task)  // UVM-1.0
  body()        (task)  // Your stimulus generation code
  post_body()   (task)  // UVM-1.0
  post_start()  (task)  // UVM-1.1
}
```

- The purpose of the pre/post hook/callback methods is explained in the upcoming slides

Sequence Execution Methodologies

- **There are two ways to execute a sequence**
  - Explicitly
    - **Test writer** must create the sequence object then call the **`start()`** method

```
packet_sequence seq;
seq = packet_sequence::type_id::create("seq", this);
seq.start(env.agt.sqr);
```

  - Implicitly
    - Test writer just populate the **`uvm_config_db`** with the intended sequence execution
      - Can be done in test code or via run-time switch
    - The **sequencer** will pick it up from the **`uvm_config_db`**, create the sequence object and call the **`start()`** method automatically

```
uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
                "default_sequence", packet_sequence::get_type());
```

Explicit Sequence Execution

```
class test_base extends uvm_test;
  // Other code not shown
  virtual task main_phase(uvm_phase phase);
    packet_sequence seq;
    phase.raise_objection(this);
    seq = packet_sequence::type_id::create("seq", this);
    seq.randomize(); // optional
    seq.start(env.agt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Construct sequence object

Raise and drop phase objection

Execute sequence

- **Only do this in test!**
  - Simple to implement but hard to control and reuse
- **Must implement the phase method**
- **Must raise and drop phase objection in phase method**
- **Must call sequence's `start()` method and specify a chosen sequencer**

Implicit Sequence Execution (1/2)

```
class test_base extends uvm_test;  // Other code not shown
  virtual function void build_phase(uvm_phase phase);  ...;
    uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
                        "default_sequence", packet_sequence::get_type());
  endfunction
endclass
```

Populate sequence in **uvm_config_db**

- **Populate `uvm_config_db` with sequence to be executed by the chosen sequencer in phase specified**
  - Set configuration in `build_phase`
  - Can be done in environment class or test class
  - Can be overridden in higher layer structure or derived class
    - e.g. configuration in test overrides environment's configuration
- **Recommended Methodology**
  - Requires more code in sequence class (coming up)

Implicit Sequence Execution (2/2)

- **Automatically happens in the configured phase:**

Sequencer

Retrieves "default_sequence" from uvm_config_db

↓

Constructs **seq** object

↓

Sets sequence's **`starting_phase`**

↓

Randomizes sequence

↓

Calls sequence's **start()** method

In implicit sequence execution, the required sequence execution steps are done for you by the sequencer automatically

`starting_phase` is equivalent to the **`phase`** handle in the component class's phase method (use is explained in next few slides)

```
// From explicit sequence execution
virtual task main_phase(uvm_phase phase);
  phase.raise_objection(this);
  ...;
  phase.drop_objection(this);
endtask
```

Implicit Sequence Execution and Objection

- **In implicit sequence execution, no phase objections are raised or dropped in test**

Cannot raise or drop phase objection for sequence execution in **`build_phase`**

```
class test_base extends uvm_test;  // Other code not shown
  virtual function void build_phase(uvm_phase phase);  ...;
    uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
                      "default_sequence", packet_sequence::get_type());
  endfunction
endclass
```

- **The sequence must manage objection!**
  - Implementations in UVM-1.1 and UVM-1.2 are different!
  - Shown in the next few slides

UVM-1.1 Sequence Phase Objection

- **Where to raise and drop phase objection in sequence?**
  - A **`uvm_phase`** handle called **`starting_phase`** is built-in to the **`uvm_sequence`** base class
  - Use this handle to raise phase objection in **`pre_start()`**
  - Then, drop phase objection in **`post_start()`**

```
class packet_sequence extends uvm_sequence #(packet);
  virtual task pre_start();
    if ((get_parent_sequence() == null) && (starting_phase != null))
      starting_phase.raise_objection(this, "Starting");
  endtask

  virtual task post_start();
    if ((get_parent_sequence() == null) && (starting_phase != null))
      starting_phase.drop_objection(this, "Ending"));
  endtask
  virtual task body(); // Same as before
endclass
```

Optional debug message

UVM-1.1 ONLY!
Will not compile
in UVM-1.2

UVM-1.2 Sequence Phase Objection

- **Where to raise and drop phase objection in sequence?**
  - **`starting_phase`** handle access has been deprecated

Compile error in UVM-1.2

```
task packet_sequence::pre_start();
  if ((get_parent_sequence() == null) && (starting_phase != null))
    starting_phase.raise_objection(this, "Starting");
endtask
```

  - Replaced by **`get_starting_phase()`** method
    - Still too much of a headache because you still need to implement **`pre_start()`** and **`post_start()`** methods
  - A better way is to automate this with a new UVM-1.2 method called **`set_automatic_phase_objection()`**
    - Argument of "1" turns it on, "0" turns it off

```
function packet_sequence::new(string name = "packet_sequence");
  super.new(name);
  set_automatic_phase_objection(1); // UVM-1.2 Only!
endfunction
```

Code That Can Work in UVM-1.1 and UVM-1.2

Use UVM VERSION to be compile-able under either version

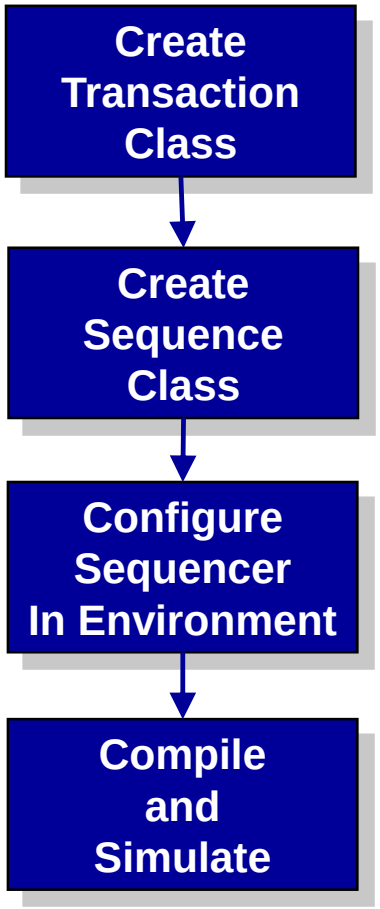
```
function packet_sequence::new(string name = "packet_sequence");
  super.new(name);
  `ifdef UVM_POST_VERSION_1_1
   set_automatic_phase_objection(1);
  `endif
endfunction

`ifdef UVM_VERSION_1_1
task packet_sequence::pre_start();
  if ((get_parent_sequence() == null) && (starting_phase != null))
    starting_phase.raise_objection(this);
endtask

task packet_sequence::post_start();
  if ((get_parent_sequence() == null) && (starting_phase != null))
    starting_phase.drop_objection(this);
endtask
`endif
```

uvm_pkg provides compile-time switches to let you figure out which version of UVM you are compiling for

Lab 2 Introduction

30 min

Implement transaction and sequence classes



Sequence with randc Transaction Property

- `uvm_do macro does not support randc behavior

```
class transaction extends uvm_sequence_item;
  randc bit[3:0] value;
endclass
```

```
class t_sequence extends uvm_sequence#(transaction);
  virtual task body();
    repeat(10) begin
      `uvm_do(req);
    end
  endtask
endclass
```

Does not work for randc!

uvm_do always creates a new object

randc requires randomization on the same object

- Implement discrete code :

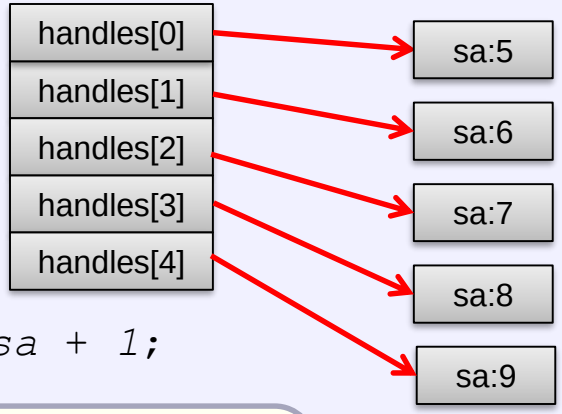
```
virtual task body(); transaction rand_obj;
  rand_obj = transaction::type_id::create("rand_obj");
  repeat(10) begin
    `uvm_create(req);
    start_item(req);
    rand_obj.randomize();
    req.copy(rand_obj);
    finish_item(req);
  end
endtask
```

Randomize same object

Pass a copy of the object to driver

Creating a Sequence of Related Items

```
class packet_scenario extends sequence_base;
  // macro and constructor not shown
  int item_count = 10;
  rand packet handles[];
  constraint array_constraint {
    foreach(handles[i]) {
      (i > 0) -> handles[i].sa == handles[i-1].sa + 1;
  } }

  function void pre_randomize();
    super.pre_randomize();
    handles = new[item_count];
    foreach(handles[i]) begin
      `uvm_create(handles[i]);    // create transaction objects
    end
  endfunction
  virtual task body();
    foreach (handles[i]) begin
      `uvm_send(handles[i]);    // only executes start_item() and finish_item()
    end
  endtask
endclass
```



Use array of rand sequence items to simplify defining relationship via constraint

Nested Sequences

- **Sequences can be nested**

```
class noise_sequence extends sequence_base; // other code
  virtual task body();
```

```
class burst_sequence extends sequence_base; // other code
  virtual task body();
```

```
class congested_sequence extends sequence_base; // other code
  virtual task body();
```

```
class nested_sequence extends sequence_base;
  // utils macro and constructor not shown
  noise_sequence      noise;
  burst_sequence      burst;
  congested_sequence  congestion;
  virtual task body();
    `uvm_do(noise);
    `uvm_do(burst);
    `uvm_do(congestion);
  endtask
endclass
```

`uvm_do macro also applies to sequences

Implicit Sequence Execution Overrides

- **Default test (*test_base*) executes default sequence specified by embedded environment**

```
class router_env extends uvm_env;  // other code not shown
  virtual function void build_phase(…); super.build_phase(…);
    uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
                    "default_sequence", packet_sequence::get_type());
```

```
class test_base extends uvm_test;
  router_env env;
  // other code not shown
endclass
```

sequence set by environment

- **Test writer can override or turn this off in tests**

```
class test_nested extends test_base;  // other code not shown
 virtual function void build_phase(…); super.build_phase(…);
  uvm_config_db#(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
     "default_sequence", <usr_seq>);   // tells sqr to run usr_seq in main phase
 endfunction
endclass
```

Setting <*usr_seq*> to **`null`** turns off implicit sequence execution

Otherwise, overrides with specified <*usr_seq*>

Implicit Sequence Execution at Phases

- **Sequences be targeted for a chosen phase**
  - Typically done at the testcase level

```
class simple_seq_RST extends sequence_base;
```

```
class simple_seq_CFG extends sequence_base;
```

```
class simple_seq_MAIN extends sequence_base;
```

```
class test_multi_phase extends test_base;
  typedef uvm_config_db #(uvm_object_wrapper) seq_phase;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seq_phase::set(this, "env.agt.sqr.reset_phase",
                   "default_sequence", simple_seq_RST::get_type());
    seq_phase::set(this, "env.agt.sqr.configure_phase",
                   "default_sequence", simple_seq_CFG::get_type());
    seq_phase::set(this, "env.agt.sqr.main_phase",
                  "default_sequence", simple_seq_MAIN::get_type());
  endfunction
endclass
```

Unit Objectives Review

Having completed this unit, you should be able to:

- **Build stimulus generator class by inheriting from `uvm_sequence`**
- **Execute sequence explicitly and implicitly**

Appendix

Sequence Macros

Sequence Priority/Weight

Sequencer-Driver Response Port

Out-Of-Order Sequencer-Driver Port

Sequence Macros

Sequence Macros

- **Please read UVM class reference document for details**

```
`uvm_create(SEQ_OR_ITEM)
`uvm_do(SEQ_OR_ITEM)
`uvm_do_pri(SEQ_OR_ITEM, PRIORITY)
`uvm_do_with(SEQ_OR_ITEM, CONSTRAINTS)
`uvm_do_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS)
`uvm_create_on(SEQ_OR_ITEM, SQR)
`uvm_do_on(SEQ_OR_ITEM, SQR)
`uvm_do_on_pri(SEQ_OR_ITEM, SQR, PRIORITY)
`uvm_do_on_with(SEQ_OR_ITEM, SQR, CONSTRAINTS)
`uvm_do_on_pri_with(SEQ_OR_ITEM, SQR, PRIORITY, CONSTRAINTS)
`uvm_send(SEQ_OR_ITEM)
`uvm_send_pri(SEQ_OR_ITEM, PRIORITY)
`uvm_rand_send(SEQ_OR_ITEM)
`uvm_rand_send_pri(SEQ_OR_ITEM, PRIORITY)
`uvm_rand_send_with(SEQ_OR_ITEM, CONSTRAINTS)
`uvm_rand_send_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS)
`uvm_create_seq(UVM_SEQ, SQR_CONS_IF)
`uvm_do_seq(UVM_SEQ, SQR_CONS_IF)
`uvm_do_seq_with(UVM_SEQ, SQR_CONS_IF, CONSTRAINTS)
```

Sequence Priority/Weight

Sequence Priority/Weight

- **Sequences can be assigned priority/weight**

```
class nested_sequence extends sequence_base;
  // utils macro and other code not shown
  virtual task body();
    uvm_sequencer_base sqr = get_sequencer();
    sqr.set_arbitration(SEQ_ARB_STRICT_FIFO)
    fork
      `uvm_do_pri(noise, 1000);
      `uvm_do_pri(burst, 50);
      `uvm_do(congestion);
    join
  endtask
endclass
```

Must set arbitration
(Defaults to **SEQ_ARB_FIFO**)

Defaults to priority/weight of 100
Higher value has higher priority

SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
SEQ_ARB_RANDOM	Requests are granted randomly
SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted randomly
SEQ_ARB_USER	Arbitration is delegated to the user-defined function, user_priority_arbitration()

Sequencer-Driver Response Port

Sequencer-Driver Response Port

- **Driver can send response back to sequence**

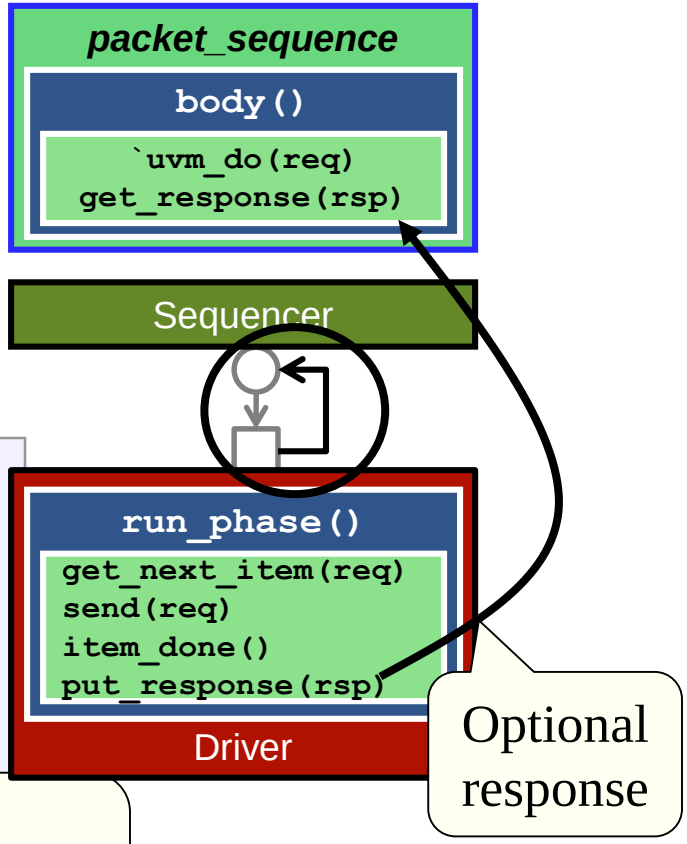
```
task body();
  repeat (item_count) begin
    `uvm_do(req);
    get_response(rsp);
    // process response
  end
endtask
```

Retrieve response

```
class driver extends uvm_driver #(packet);
 task run_phase(uvm_phase phase);
  forever begin
   seq_item_port.get_next_item(req);
   send(req);
   rsp = packet::type_id::create("rsp", this);
   rsp.set_id_info(req);
   // set rsp response
   seq_item_port.item_done();
   seq_item_port.put_response(rsp);
  end
 endtask
endclass
```

Copy response id (**required!**)

Set response



Out-Of-Order Sequencer-Driver Port

Sequence with Out-Of-Order Response (1/2)

```
class packet_sequence extends packet_sequence_base;
  packet pkt_req[int];
  packet rand_obj = packet::type_id::create("rand_obj");
  task body();
    repeat(item_count) begin
      wait(pkt_req.size() < 5);
      `uvm_create(req);
      start_item(req);
      if (!rand_obj.randomize()) ...;
      req.copy(rand_obj);
      finish_item(req);
      pkt_req[req.get_transaction_id()] = req;
      fork
        packet in_driver = req;
        process_rsp(in_driver);
      join_none // see next slide
    end
    wait(pkt_in_drv.size() == 0);
  endtask
endclass
```

throttle traffic

Stores what the driver is operating on

Wait for sepecific transaction response

Block until array is empty

Sequence with Out-Of-Order Response (2/2)

```
  virtual task process_rsp(packet in_driver);
    packet from_driver;
    get_response(from_driver, in_driver.get_transaction_id());
    // process response
    pkt_in_drv.delete(from_driver.get_transaction_id());
  endtask
endclass
```

retrieve response based on transaction_id

Remove response from in-operation array

Out-Of-Order Driver (1/2)

- **Need a queue to store transactions**
- **Implement thread to get transactions from sequence (next slide)**
- **Implement thread to execute transaction in the desired order (next slide)**

```
class driver extends uvm_driver#(packet); // other code not shown
  packet pkt_q[$];

  virtual task run_phase(uvm_phase phase);
    fork
      get_item();
      execute_item();
    join
  endtask
  // see next slide for method implementation
endclass
```

Call thread to get transaction

Call thread to execute transaction

Out-Of-Order Driver (2/2)

```
virtual task get_item()
  forever begin
    seq_item_port.get_next_item(req);
    accept_tr(req);                                   // transaction accepted
    pkt_q.push_back(req);
    seq_item_port.item_done();
  end
endtask

virtual task execute_item();
  forever begin
    int index;
    wait (pkt_q.size() != 0);
    index = $urandom_range(pkt_q.size()-1);
    begin_tr(pkt_q[index]);                           // transaction recording starts
    // process transaction code left off
    seq_item_port.put_response(pkt_q[index]);         // indicate tranaction completed
    end_tr(in_use);                                   // transaction recording ends
    pkt_q.delete(index);
  end
endtask
```

Get transaction

Store transaction in queue

Wait for transaction to process

Indicate which transaction has completed processing

- **Requires +define+UVM_DISABLE_AUTO_ITEM_RECORDING**