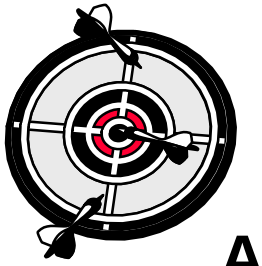
Agenda: Day 3

DAY 3

9	UVM Advanced Sequence/Sequencer
10	UVM Phasing and Objections 
11	UVM Register Abstraction Layer (RAL)
12	Summary 

© 2016 Synopsys, Inc. All Rights Reserved

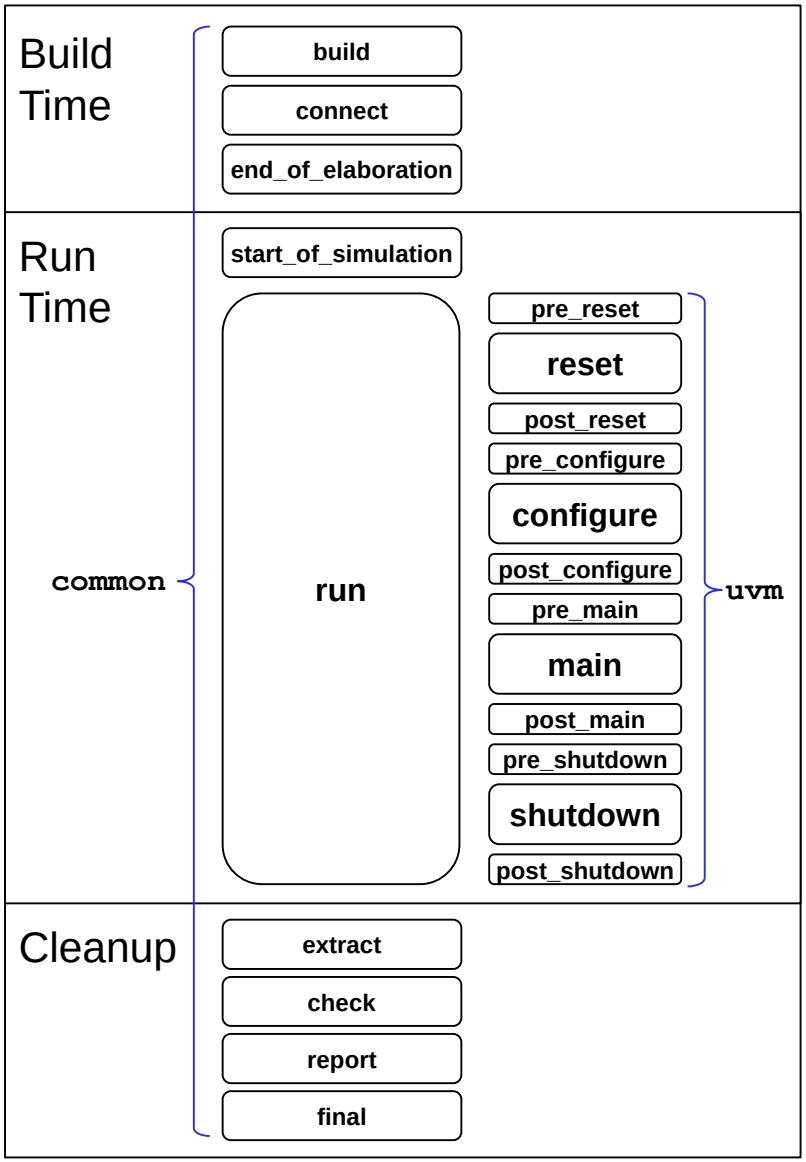
Unit Objectives



After completing this unit, you should be able to:

- **Control phase objections**
- **Control phase timeout**
- **Create user phases**
- **Create phase domains**
- **Implement phase callbacks**

Phasing in UVM 1.0+



- **Synchronized phase execution**
- **Two predefined domains**
  - **`common`** - Simple components (driver/monitor)
    - **`run`**
  - **`uvm`** - Complex components (test/environment/scoreboard)
    - **`reset -> shutdown`**
      - With **`pre_*/post_*`** phases
- **Task phases terminate when objection count reaches zero**
- **Enough flexibility for basic to intermediate user**

Common Phases

build	Create and configure testbench components
connect	Establish cross-component connections
end_of_elaboration	Check for correctness of testbench structure
start_of_simulation	Print configuration for components
run	Stimulate the DUT
extract	Extract data from different points of the verification environment
check	Check for any unexpected conditions in the verification environment
report	Report results of the test
final	Tie up loose ends

For more details please see UVM Reference Guide

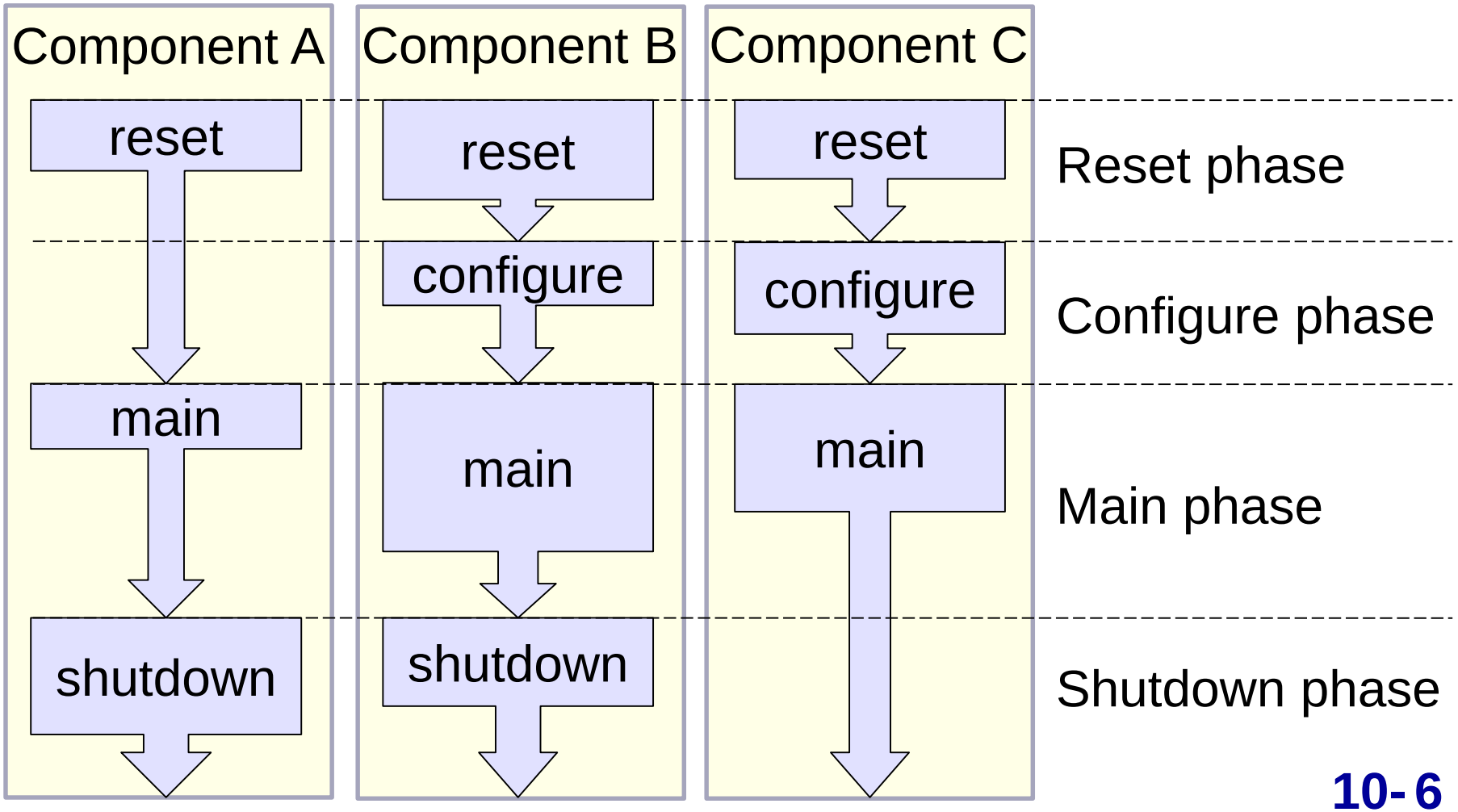
Run-Time Task Phases

pre_reset	Setup/Wait for conditions to reset DUT
reset	Reset DUT/De-assert control signals
post_reset	Wait for DUT to be at a known state
pre_configure	Setup/Wait for conditions to configure DUT
configure	Configure the DUT
post_configure	Wait for DUT to be at a known configured state
pre_main	Setup/Wait for conditions to start testing DUT
main	Test DUT
post_main	Typically a no-op
pre_shutdown	Typically a no-op
shutdown	Wait for data in DUT to be drained
post_shutdown	Perform final checks that consume simulation time

For more details please see UVM Reference Guide

Task Phase Synchronization

- **Run-Time task phases for all components will move on to the next phase only when all objections for that phase are dropped**



Phase Objection (1/6)

- **Do not raise/drop objection in component's run_phase**
  - Impacts simulation with excessive objection count

```
class driver extends ...;
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      phase.raise_objection(this);
      send(req);
      seq_item_port.item_done();
      phase.drop_objection(this);
    end
  endtask
endclass
```

Trace with
+UVM_OBJECTION_TRACE
run-time switch

```
class monitor extends ...;
  virtual task run_phase(uvm_phase phase);
    forever begin
      @(negedge sigs.monClk.frame_n[port_id]);
      phase.raise_objection(this);
      get_packet(tr);
      phase.drop_objection(this);
      ...
    end
  endtask
endclass
```

Phase Objection (2/6)

- **Objections should be raised/dropped in sequence**

```
class packet_sequence ...;  // other code not shown
  function packet_sequence::new(string name = "packet_sequence");
    super.new(name);
    set_automatic_phase_objection(1);          // UVM-1.2 Only!
  endfunction
  virtual task pre_start();
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.raise_objection(this);  // UVM-1.1 ONLY!
  endtask
  virtual task post_start();
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.drop_objection(this);   // UVM-1.1 ONLY!
  endtask
```

Objection set for main phase

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

Phase Objection (3/6)

- **Objections in sequence may not be enough**
  - There may be latency within the DUT
- **Known latency can be taken care of with drain time**
  - **`set_drain_time()`** method extends the phase for the specified amount of simulation time after objection count reaches 0

```
class test_drain extends test_base; // other code not shown

  virtual task main_phase(uvm_phase phase);
    uvm_objection objection;
    super.main_phase(phase);
    objection = phase.get_objection();
    objection.set_drain_time(this, 1us);
  endtask
endclass
```

Change objection drain time in component's phase method

Get objection handle

Set drain time

Phase Objection (4/6)

- **A common way of handling unknown latency of DUT is to wait for the scoreboard's expect queue to be empty**

```
class scoreboard#(type T=packet) extends scoreboard_base; // other code left off
  T expected_queue[$];  // queue of expected transactions
  virtual task wait_for_empty();
    wait (expected_queue.size() == 0);
  endtask
endclas
```

```
class test_shutdown extends test_base;              // other code left off
  virtual task shutdown_phase(uvm_phase phase);     // phase specific
    uvm_component comps[$}; scoreboard_base sb;
    phase.raise_objection(this, "Wait for scoreboard to empty");
    uvm_root::get().find_all("*", comps);
    foreach (comps[i]) begin
      if ($cast(sb, comps[i]) begin
        sb.wait_for_expected_q_empty();
      end
    end
    phase.drop_objection(this, "Scoreboard queues emptied");
  endtask
endclass
```

Wait for scoreboard queue to empty

Drop objection when scoreboard queues are emptied

Phase Objection (5/6)

- **Test can extend phase via phase callback**
  - **`phase_ready_to_end()`**
    - Called when all objections are dropped for a phase
  - Requires the test writer to create and manage a *BUSY* flag

```
class my_test extends test_base;                          // other code left off
  virtual function void phase_ready_to_end(uvm_phase phase); // all phases
    // BUSY flag must be set and clear by test
    if (BUSY) begin
      phase.raise_objection(this, "In middle of protocol");
      fork
        begin
          wait(BUSY == 0);
          phase.drop_objection(this, "Completed protocol");
        end
      join_none
    end
  endfunction
endclass
```

Raise objection when monitor is busy

Drop objection when monitor is idle

Phase Objection (6/6)

- **Objections can be tracked via phase callback**

```
virtual function void phase_started(uvm_phase phase);
  if (uvm_report_enabled(UVM_DEBUG, UVM_INFO, "OBJECTIONS")) begin
    if (phase.get_objection() == null) return;
    fork begin
      // Needed to give sequences opportunity to raise objection
      phase.wait_for_state(UVM_PHASE_EXECUTING);
      fork
        uvm_objection objection = phase.get_objection();
        forever begin
          objection.display_objections();
          #1us; // Objection sample period
        end
        begin
          objection.wait_for_total_count();
        end
      join_any
      disable fork;
    end join_none
  end
endfunction
```

Message control

Print objectors

Wait for objection count to be 0
Terminate threads when reached

UVM Timeout

- **Run-Time switch:**
  - **`+UVM_TIMEOUT`**
  - Maximum absolute time before fatal is called
  - Defaults to 9,200 sec
    - Causes fatal message if reached
- **Embed in test:**

```
uvm_root::get().set_timeout(.timeout(1ms))
```

  - Overridden by **`+UVM_TIMEOUT`** if called in **`new()`**
    - Recommended
  - Overrides **`+UVM_TIMEOUT`** if called in phase method
    - Not recommended

Advanced Features

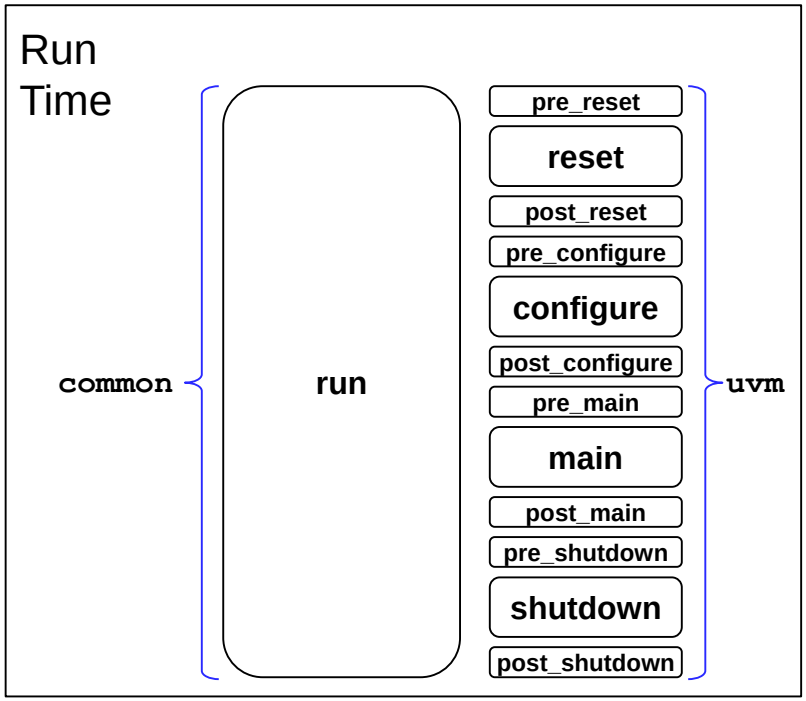
- **Phase Domains**
  - UVM has two default phase domains: common and uvm
  - Phases within same domain are synchronized
  - Inter-domain phases can be synchronized
    - User can set full, partial or no synchronization
- **User Defined Phases**
  - Can be mixed with predefined phases
- **Phase Jumping**
  - Forwards and Backwards

Only recommended for (Environment) Implementers not (Component) Developers

Phase Domains (1/2)

- **Task phases are organized as domains**
  - There are two task domains: **common** and **uvm**
    - Executing concurrently as two synchronized threads

```
virtual task main_phase(uvm_phase phase);
  $display("Phase in %s domain", phase.get_domain_name());
```



Phase Domains (2/2)

- **User can create customized domains**

```
class top_env extends uvm_env;
  sub_env env0, env1;
  uvm_domain domain0=new("domain0"), domain1=new("domain1");

  virtual function void connect_phase(uvm_phase phase);
    env0.set_domain(domain0);
    env1.set_domain(domain1);

    domain1.sync(.target(this.get_domain());
//  domain0.sync(.target(this.get_domain()),
//               .phase(uvm_main_phase::get()));
//  domain0.sync(.target(this.get_domain()),
//               .phase(uvm_post_main_phase::get()),
//               .with_phase(uvm_shutdown_phase::get()));
    ...
  endfunction
endclass
```

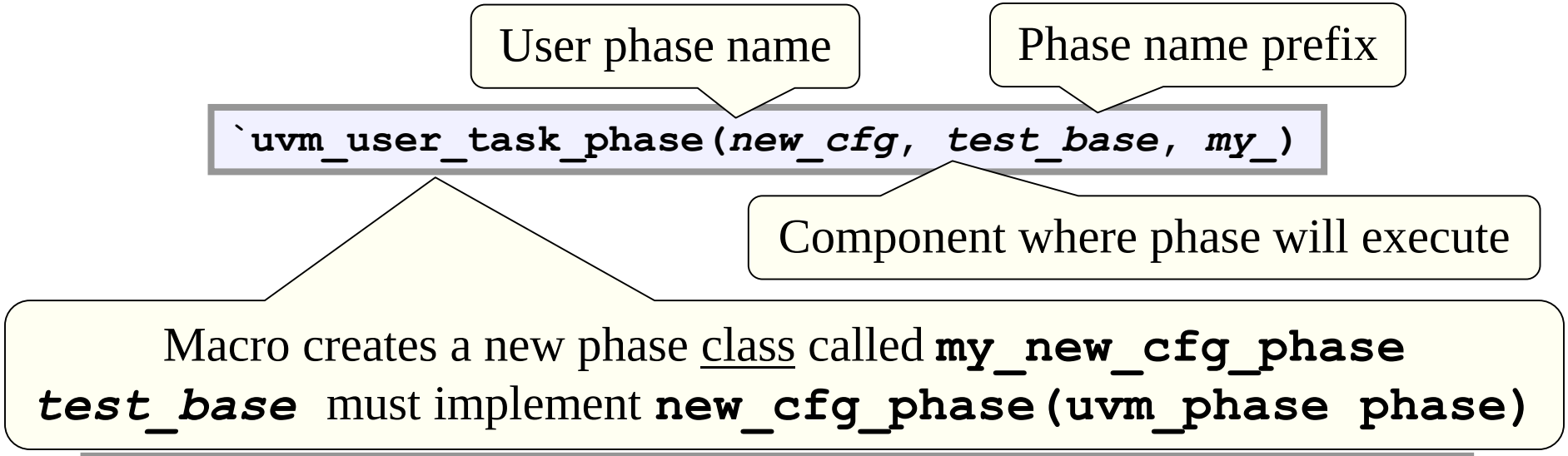
Created two user domains

env0's phases are independent of all other domains
env1's phases are synchronized with top_env

env0's specified phase is sync'ed with top_env

User Defined Phase

- **User can create customized phases**

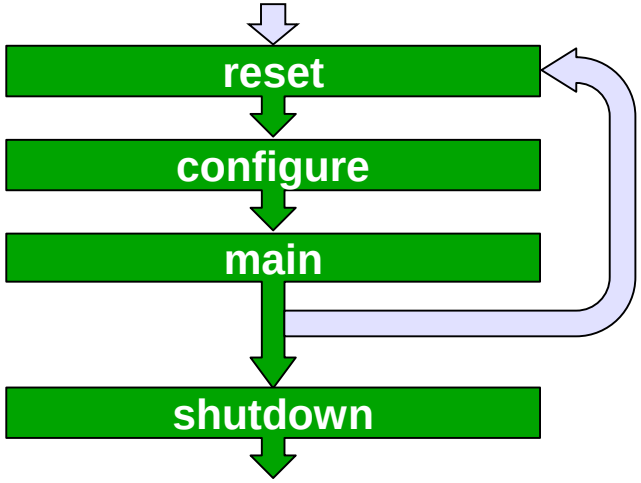



```
class test_base extends uvm_test; // utils left off
  virtual function void build_phase(uvm_phase phase);
    ... // all environment are constructed here
    begin
      uvm_domain env_domain = env.get_domain();
      env_domain.add(my_new_cfg_phase::get(),
                     .after_phase(uvm_configure_phase::get()));
    end
  endfunction
endclass
```

Add phase into domain

Phase Jump: Backward

- **Phases can be rolled back**
- **Mid-simulation reset can be accomplished by jumping back to reset phase**
- **Environment must be designed to support jumping**
  - All active components must return control signals to de-asserted state
  - All residual properties must be deleted
  - All residual processes must be killed




```
class test_jump2reset extends test_base;
  // code not shown
  virtual task main_phase(…);
    // detect some condition
    if (phase.get_run_count() < 5)
      phase.jump(uvm_reset_phase::get());
  endtask
```

Phase Jump: Forward

- **Can be used for a test to skip a behavior**
  - Example: skip rest of main if coverage is met
- **Environment must be designed to support jumping**
  - May need to wait for components to be in idle state before calling jump

```
class driver extends uvm_driver #(...);
  covergroup d_cov ... endgroup
  virtual task run_phase(...);
    forever begin
      seq_item_port.get_next_item(req);
      process(req);
      seq_item_port.item_done();
      d_cov.sample(req);
      if ($get_coverage() == 100.0)
        phase.jump(uvm_post_main_phase::get());
    end
  endtask
endclass
```

Phase Jumping Cleanup

- **`phase_ended()`**
  - Called before each phase terminates

```
virtual function void phase_ended(uvm_phase phase);
  uvm_phase jump_phase = phase.get_jump_target();
  if (jump_phase != null) begin
    if (jump_phase.is_before(phase)) begin
      ...
    end else begin
      if (jump_phase.is_after(phase)) begin
        ...
      end
    end
  end
endfunction: phase_ended
```

Get Phase Execution Count

- **`get_run_count()`**
  - Returns the number of times this phase has executed

```
virtual function void phase_started(uvm_phase phase);
  if (phase.is(uvm_reset_phase::get())) begin
    int count = phase.get_run_count();
    if (count > 1) begin
      ...
    end    ...
  end
endfunction : phase_started
```

Unit Objectives Review

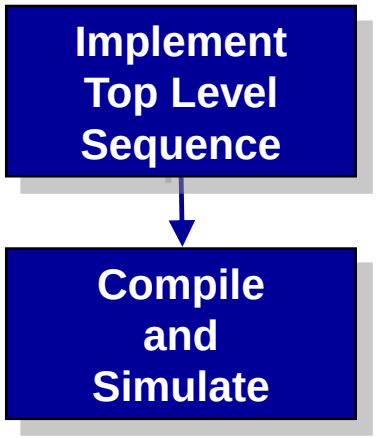
Having completed this unit, you should be able to:

- **Control phase objections**
- **Control phase timeout**
- **Create user phases**
- **Create phase domains**
- **Implement phase callbacks**

Lab 5 Introduction

30 min

Implement top level sequence



Appendix

`uvm_phase` Class Key Members

Component Phasing Guidelines

Jump Code Example

`uvm_phase` Class Key Members

uvm_phase Class Key Methods

- **uvm_phase class key methods are as follows:**

```
class uvm_phase extends uvm_object;
  function bit is(uvm_phase phase);
  function bit is_before(uvm_phase phase);
  function bit is_after(uvm_phase phase);
  function uvm_objection get_objection();
  virtual function void raise_objection(uvm_object obj, string description="", int count=1);
  virtual function void drop_objection(uvm_object obj, string description="", int count=1);
  function uvm_phase find(uvm_phase phase, bit stay_in_scope=1);
  function uvm_phase find_by_name(string name, bit stay_in_scope=1);
  function uvm_phase get_schedule(bit hier=0);
  function string get_schedule_name(bit hier=0);
  function uvm_domain get_domain();
  function string get_domain_name();
  function void add(uvm_phase phase, with_phase=null, after_phase=null, before_phase=null);
  function void sync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);
  function void unsync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);
  function void jump(uvm_phase phase);
  static function void jump_all(uvm_phase phase);
  function uvm_phase get_jump_target();
  function int get_run_count();
  function int unsigned get_ready_to_end_count();

  function uvm_phase_state get_state();                                  // see next slide for possible states
  task wait_for_state(uvm_phase_state state, uvm_wait_op op=UVM_EQ);     // possible op: UVM_EQ, UVM_NE, UVM_LT,
                                                                         // UVM_LTE, UVM_GT, UVM_GTE
endclass
```

`uvm_phase` Class States

- **The `get_state()` and `wait_for_state()` methods operates on the following possible states of a phase**

Phase State	Description
UVM_PHASE_DORMANT	Domain inactive
UVM_PHASE_SCHEDULED	Phase scheduled waiting for preceding phases to complete
UVM_PHASE_SYNCING	All preceding phases completed
UVM_PHASE_STARTED	Phase ready, phase_started() executes
UVM_PHASE_EXECUTING	Phase method executes
UVM_PHASE_READY_TO_END	No objections, phase_ready_to_end() executes
UVM_PHASE_ENDED	Phase completed, phase_ended() executes
UVM_PHASE_CLEANUP	Phase related threads killed
UVM_PHASE_DONE	Done, execute succeeding phase

Component Phasing Guidelines

Driver Guideline

- **Emulates launch and capture registers**
- **Build time phases**
  - build_phase() // retrieve configuration
- **Run time phases**
  - start_of_simulation_phase() // print configuration
  - run_phase() // get and process item
- **Callbacks**
  - phase_ended() // for jump back

Monitor Guideline

- **Emulates capture register only**
- **Build time phases**
  - build_phase() // retrieve configuration
- **Run time phases**
  - start_of_simulation_phase() // print configuration
  - run_phase() // report observed transaction via analysis port
- **Callbacks**
  - phase_ended() // for jump back

Agent Guideline

- **A container class for interface-base components**
  - No behavioral code
- **Build time phases**
  - build_phase() // construct sub-components
    // retrieve and set sub-component configuration
  - connect_phase() // connect sub-components
- **Run time phases**
  - start_of_simulation_phase() // report configuration

Scoreboard Guideline

- **Tracks correctness of operation**
- **Build time phases**
  - build_phase() // construct sub-components if needed
    // retrieve configuration
  - connect_phase() // connect analysis ports if needed
- **Run time phases**
  - start_of_simulation_phase() // print configuration
- **Callbacks**
  - phase_ended() // for jump back

Environment Phase Guideline

- **A container class for all DUT verification components**
  - No behavioral code
- **Build time phases**
  - build_phase() // construct sub-components
    // retrieve and set sub-component configuration
  - connect_phase() // connect sub-components
- **Run time phases**
  - start_of_simulation_phase() // display configuration

Test Phase Guideline

- **No restriction/limitation of phases**
- **Build time phases**
  - build_phase() // construct and configure environment
  - connect_phase() // make changes to environment connections
- **Run time phases**
  - start_of_simulation_phase() // display configuration
  - run_phase() // set dynamically changing configurations
  - reset_phase() // execute reset sequence
  - configure_phase() // execute configure sequence
  - main_phase() // execute stimulus sequence
  - shutdown_phase() // additional end of test condition
- **Cleanup phases and callbacks**
  - As needed by test

Jump Code Example

Driver Code for Jump

```
class driver extends uvm_driver #(packet);   // other code left off
  process m_proc[$];

  function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
      `uvm_info("JUMP", {"to", jump_phase.get_name()}, UVM_MEDIUM);
      foreach (m_proc[i]) begin
        m_proc[i].kill();
      end
      m_proc.delete();
      if (req != null) begin
        seq_item_port.item_done();
        req = null;
      end
    end
  endfunction
  // continue on next page
```

Driver Code for Jump

```
  // continue from previous page
  virtual task run_phase(uvm_phase phase);
    forever begin
      fork begin
        process p = process::self();
        m_proc.push_front(p);
        drive_transaction();
        m_proc.pop_front();
      end join
    end
  endtask
  virtual task drive_transaction();
    seq_item_port.get_next_item(req);
    send(req);
    seq_item_port.item_done();
    req = null;
  endtask
endclass
```

Monitor Code for Jump

```
class monitor extends uvm_monitor; // other code left off
  process m_proc[$];
  function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
      foreach (m_proc[i]) begin
        m_proc[i].kill();
      end
      m_proc.delete();
    end
  endfunction
  virtual task run_phase(uvm_phase phase);
    forever begin
      fork begin
        process p = process::self();
        m_proc.push_front(p);
        get_packet();
        m_proc.pop_front();
      end join
    end
  endtask
endclass
```

Scoreboard Jump Code

```
class scoreboard#(type T=packet) extends uvm_scoreboard; // other code left off
  T expected_queue[$];
  function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
      expected_queue.delete();
    end
  endfunction
  virtual function void write_before(T tr);
    expected_queue.push_back(tr);
  endfunction
endclass
```