Run-Time Configuration of a Verification Environment

A Novel Use of the OVM/UVM Analysis Pattern
Authors

Paul Marriott
paul.marriott@verilab.com
Authors

Mark Ronan
mronan@diablo-technologies.com
Typical OVM Environment

Common analysis port usage:
Data transactions  
e.g. monitor to scoreboard
Example PHY Circuit

Process/temperature/voltage sensitive delay
Zero, one or more delays elements can subscribe to a group. Driver sends updates to the delay values at run time via analysis port `write()` calls.
virtual class delay_api_abstract extends ovm_pkg::ovm_subscriber#(time signed);
class unidir_delay_api extends global_type_Pkg :: delay_api_abstract;

virtual function void set_delay (time the_delay);
   super.set_delay (the_delay);
   delay_val = the_delay;
endfunction

virtual function void offset_delay (time signed the_offset);
   super.offset_delay (the_offset);
   delay_val = new_delay;
endfunction

virtual function void write (time signed t);
   this.offset_delay (t);
endfunction

undir_delay_api class IsA ovm_subscriber parameterized for time signed
Implementation of the write() method calls the offset_delay() method of the concrete class
This applies an update to the delay value (either positive or negative)
Checks are made to ensure the time delay never goes negative
A write() to the analysis port of any group updates all the connected subscribers’ delay values
A method-based API is also available to implement delays that do not vary at run-time
A call to the set_delay() method applies values from the configuration object
class delay_timing_driver extends ovm_component;
    root_cfg my_cfg;
    ovm_analysis_port #(time signed) delay_group_control_aps [delay_group_t];

function void connect_group_bl0();
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ00__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ01__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ02__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ03__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ04__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ05__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ06__inst.delay_element_in.api.analysis_export);
    this.delay_group_control_aps[group_bl0].connect (`PHY.DQ07__inst.delay_element_in.api.analysis_export);
endfunction

function void connect_group_bl1();
    ...
endfunction

function void connect_all_groups();
    connect_group_bl0();
    connect_group_bl1();
    ...
endfunction

function void connect();
    super.connect();
    connect_all_groups();
endfunction : connect

Associated array of analysis ports
Connection to analysis export in the interface
Connection function for each group
OVM connect() phase to make the connections

delay_timing_driver

delay_control_agent
class delay_group_sequencer extends ovm_component;

function void connect();

  my_cfg = my_parent.parent_cfg;

case (my_cfg.delay_timing_config.padgroup_timing_config[my_group].waveform)
  triangle: this.delay_function = ac_triangle_wave::type_id::create("delay_lut_triangle",this);
  sine: this.delay_function = sine_wave::type_id::create("delay_lut_sine",this);
  square: this.delay_function = square_wave::type_id::create("delay_lut_square",this);
  ramp: this.delay_function = ramp_wave::type_id::create("delay_lut_ramp",this);
  impulse: this.delay_function = impulse_wave::type_id::create("delay_lut_impulse",this);
  noise: this.delay_function = noise_wave::type_id::create("delay_lut_noise",this);
endcase
endfunction

function void send_update(time signed the_time);

  my_parent.delay_group_control_aps [my_group].write(the_time);
endfunction

ev_trig task run();

  time signed the_delay_offset;
  if (this.enabled) begin
    #this.delay_function.wait_to_start
    while (this.enabled) begin
      if (this.running) begin
        the_delay_offset = this.delay_function.get_next_value();
        send_update(the_delay_offset);
      end
      #this.delay_function.update_interval
    end //while enabled
  end //if enabled
endtask

Each group has a sequencer
Waveshape is set from the configuration
Sequencer calls the write() method of the driver's analysis port for that group
Get the next delay value from the delay_function class's lookup table – allows easy creation of any waveshape
Performance considerations
396 delay elements in 49 groups updated once every 1000ns
Typical simulation time = 2ms
792,000 value updates require 98,000 calls to write()

If using traditional OVM approach:
With 49 groups of elements
Would require 98,000 calls to set_config_int() and 792,000 calls to get_config_int()

set_config_int() and get_config_int() use string lookups and are expensive in compute time

Example simulation
Shows several groups
Triangle-wave timing variation
Conclusions

• Methodology can be applied to any arbitrary type that needs to be communicated
  • Type *time* used in this particular project
  • More complex transaction type could be further decoded on reception

• Significant performance advantage for this project
  • Alternative would be a large number of calls to set/get_config_int
  • This is expensive due to the string lookups required
  • Set-once delay values were just applied via the delay control class’s API
    • No difference in performance to usual set/get_config_int
    • (though a special configuration object used in this project)

• This methodology can also be used with UVM environments