UVM - How, why, and is it even necessary?

SystemVerilog

- An extension of the Verilog HDL
- Adds some much-needed features to hardware design
    - Enumerated types
    - Structs and unions (as in C)
    - “Procedural blocks” that help generate correct type of logic
- A much-needed upgrade to modernize Verilog
- A lot of verification features as well

UVM & SystemVerilog

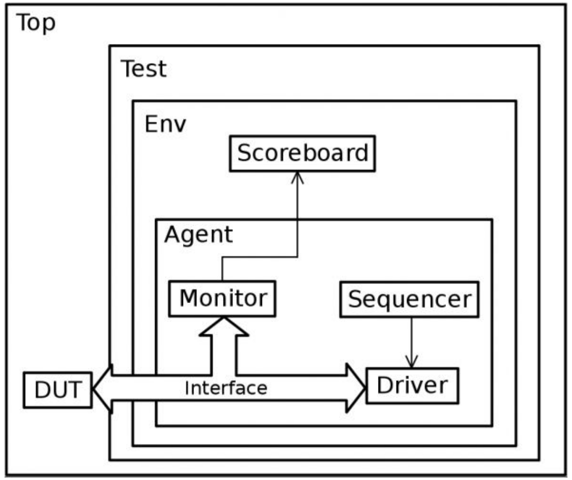
- SystemVerilog adds
  - Functional Coverage
  - Constrained Random Verification
  - Class-based testbench design (but only for verification, not for design)
- HDL + HVL at the same time. Yay or nay?
- UVM is a class library in SystemVerilog

Purpose of UVM

- Standardize testbench designs between vendors
  - Previously: OVM, eRM, OSVVM, VMM
- Make testbenches more streamlined and simpler to extend
- It’s HUGE and quite verbose

Structure of a UVM Testbench

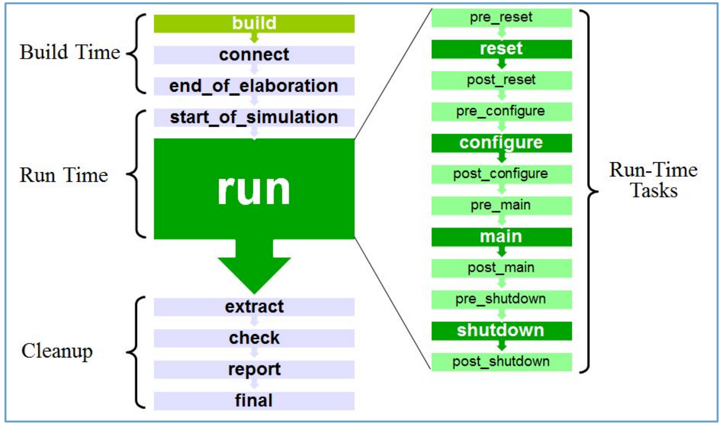
- Each component does only *one* thing
  - Encourages OOP principles
- Agents are an important building block
  - Write once, reuse everywhere
- Using OOP is a really good idea!



Structure of a UVM testbench. By Pedro Araújo / colorlesscube.com

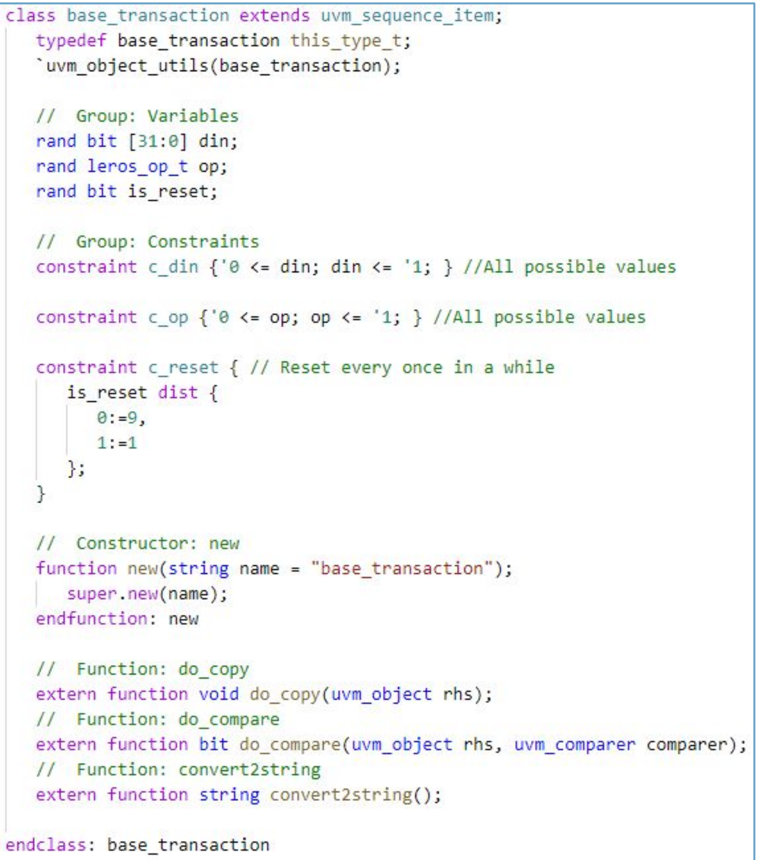
Structure of a UVM Testbench

- UVM uses phases to control execution
- Each phase may "raise an objection"

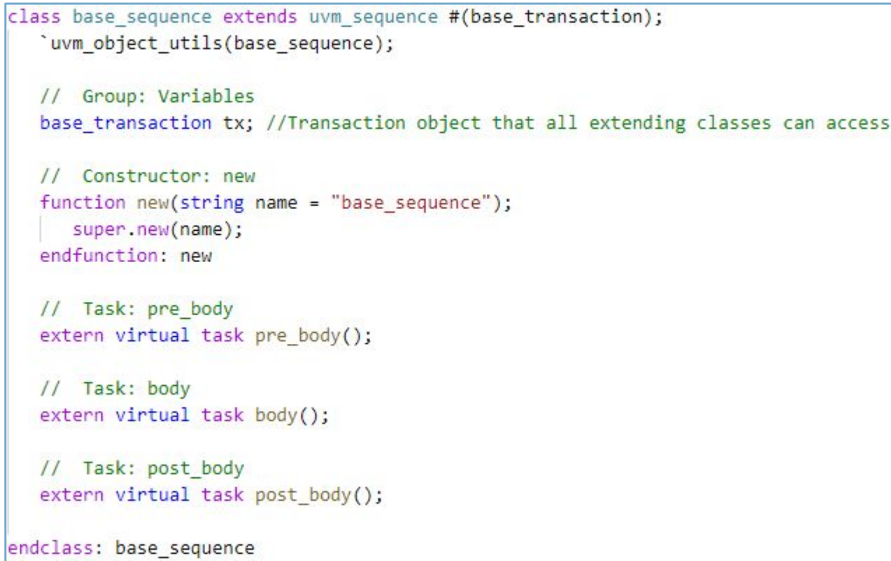


Structure of the UVM Phases

A deeper dive (1) | Transaction and sequence

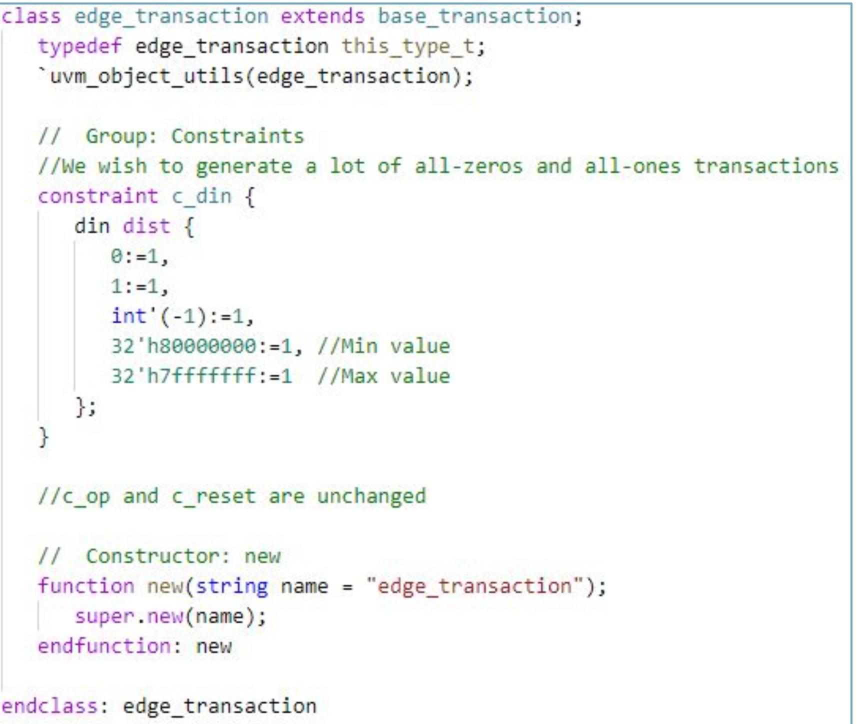


```
class base_transaction extends uvm_sequence_item;
   typedef base_transaction this_type_t;
   `uvm_object_utils(base_transaction);

   //  Group: Variables
   rand bit [31:0] din;
   rand leros_op_t op;
   rand bit is_reset;

   //  Group: Constraints
   constraint c_din {'0 <= din; din <= '1; } //All possible values

   constraint c_op {'0 <= op; op <= '1; } //All possible values

   constraint c_reset { // Reset every once in a while
      is_reset dist {
         0:=9,
         1:=1
      };
   }

   //  Constructor: new
   function new(string name = "base_transaction");
      super.new(name);
   endfunction: new

   //  Function: do_copy
   extern function void do_copy(uvm_object rhs);
   //  Function: do_compare
   extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
   //  Function: convert2string
   extern function string convert2string();

endclass: base_transaction
```

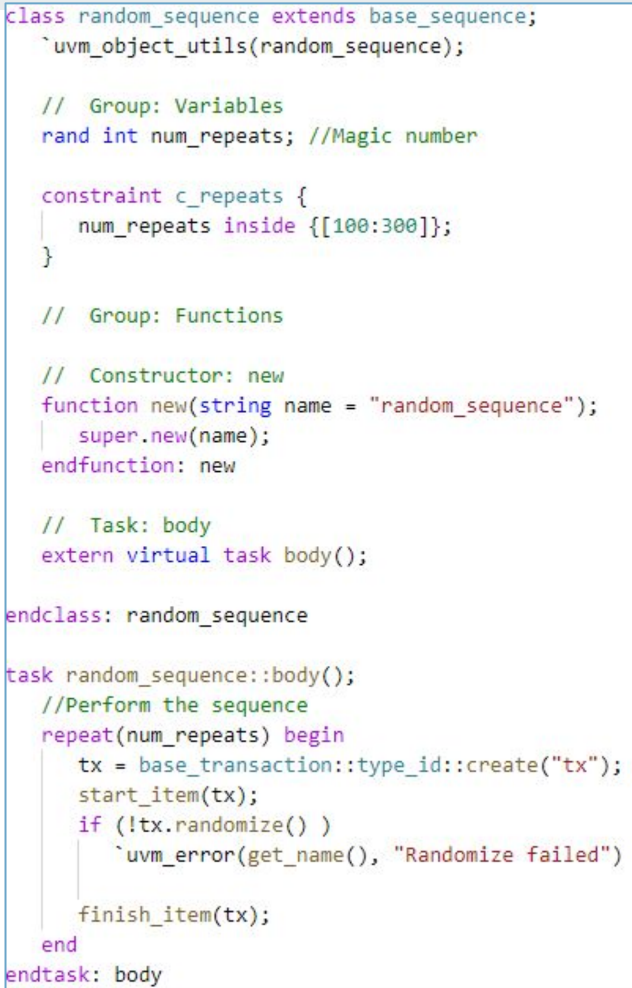


```
class base_sequence extends uvm_sequence #(base_transaction);
   `uvm_object_utils(base_sequence);

   //  Group: Variables
   base_transaction tx; //Transaction object that all extending classes can access

   //  Constructor: new
   function new(string name = "base_sequence");
      super.new(name);
   endfunction: new

   //  Task: pre_body
   extern virtual task pre_body();

   //  Task: body
   extern virtual task body();

   //  Task: post_body
   extern virtual task post_body();

endclass: base_sequence
```

A deeper dive (2) | Inheritance

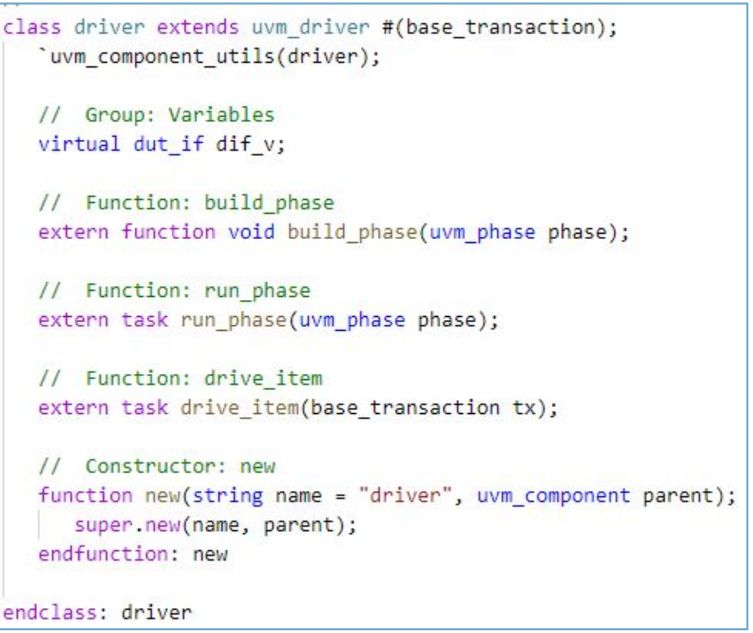


```
class edge_transaction extends base_transaction;
   typedef edge_transaction this_type_t;
   `uvm_object_utils(edge_transaction);

   //  Group: Constraints
   //We wish to generate a lot of all-zeros and all-ones transactions
   constraint c_din {
      din dist {
         0:=1,
         1:=1,
         int'(-1):=1,
         32'h80000000:=1, //Min value
         32'h7fffffff:=1  //Max value
      };
   }

   //c_op and c_reset are unchanged

   //  Constructor: new
   function new(string name = "edge_transaction");
      super.new(name);
   endfunction: new

endclass: edge_transaction
```

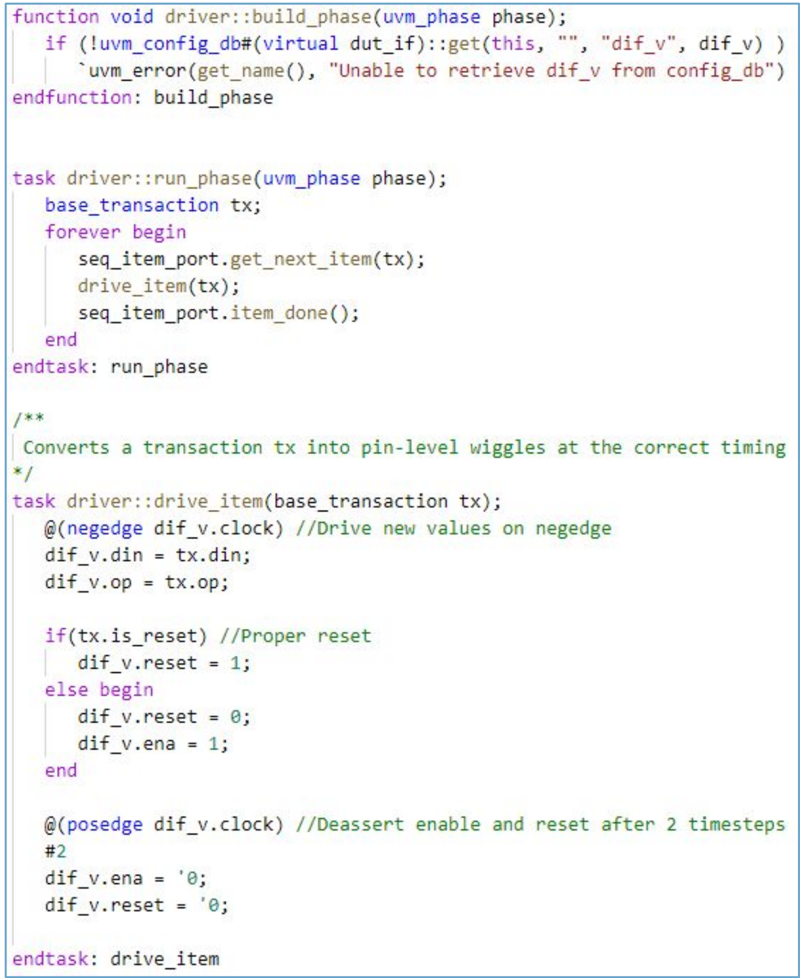


```
class random_sequence extends base_sequence;
   `uvm_object_utils(random_sequence);

   //  Group: Variables
   rand int num_repeats; //Magic number

   constraint c_repeats {
      num_repeats inside {[100:300]};
   }

   //  Group: Functions

   //  Constructor: new
   function new(string name = "random_sequence");
      super.new(name);
   endfunction: new

   //  Task: body
   extern virtual task body();

endclass: random_sequence

task random_sequence::body();
   //Perform the sequence
   repeat(num_repeats) begin
      tx = base_transaction::type_id::create("tx");
      start_item(tx);
      if (!tx.randomize() )
         `uvm_error(get_name(), "Randomize failed")

      finish_item(tx);
   end
endtask: body
```

A deeper dive (3) | Driver

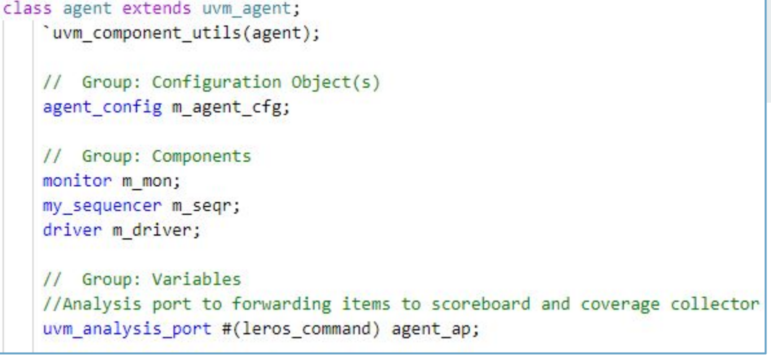


```
class driver extends uvm_driver #(base_transaction);
   `uvm_component_utils(driver);

   //  Group: Variables
   virtual dut_if dif_v;

   //  Function: build_phase
   extern function void build_phase(uvm_phase phase);

   //  Function: run_phase
   extern task run_phase(uvm_phase phase);

   //  Function: drive_item
   extern task drive_item(base_transaction tx);

   //  Constructor: new
   function new(string name = "driver", uvm_component parent);
      super.new(name, parent);
   endfunction: new

endclass: driver
```

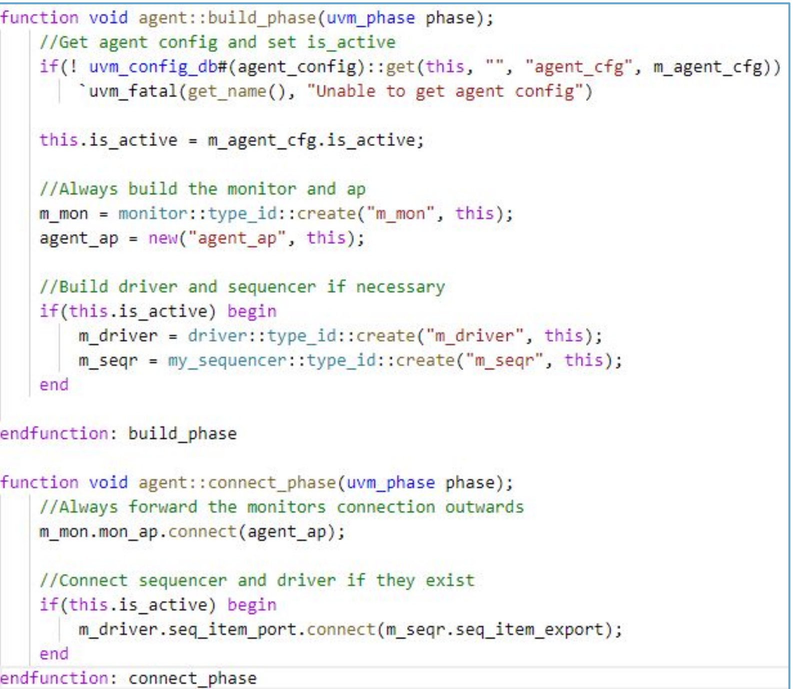


```
function void driver::build_phase(uvm_phase phase);
   if (!uvm_config_db#(virtual dut_if)::get(this, "", "dif_v", dif_v) )
      `uvm_error(get_name(), "Unable to retrieve dif_v from config_db")
endfunction: build_phase


task driver::run_phase(uvm_phase phase);
   base_transaction tx;
   forever begin
      seq_item_port.get_next_item(tx);
      drive_item(tx);
      seq_item_port.item_done();
   end
endtask: run_phase

/**
 Converts a transaction tx into pin-level wiggles at the correct timing
*/
task driver::drive_item(base_transaction tx);
   @(negedge dif_v.clock) //Drive new values on negedge
   dif_v.din = tx.din;
   dif_v.op = tx.op;

   if(tx.is_reset) //Proper reset
      dif_v.reset = 1;
   else begin
      dif_v.reset = 0;
      dif_v.ena = 1;
   end

   @(posedge dif_v.clock) //Deassert enable and reset after 2 timesteps
   #2
   dif_v.ena = '0;
   dif_v.reset = '0;

endtask: drive_item
```

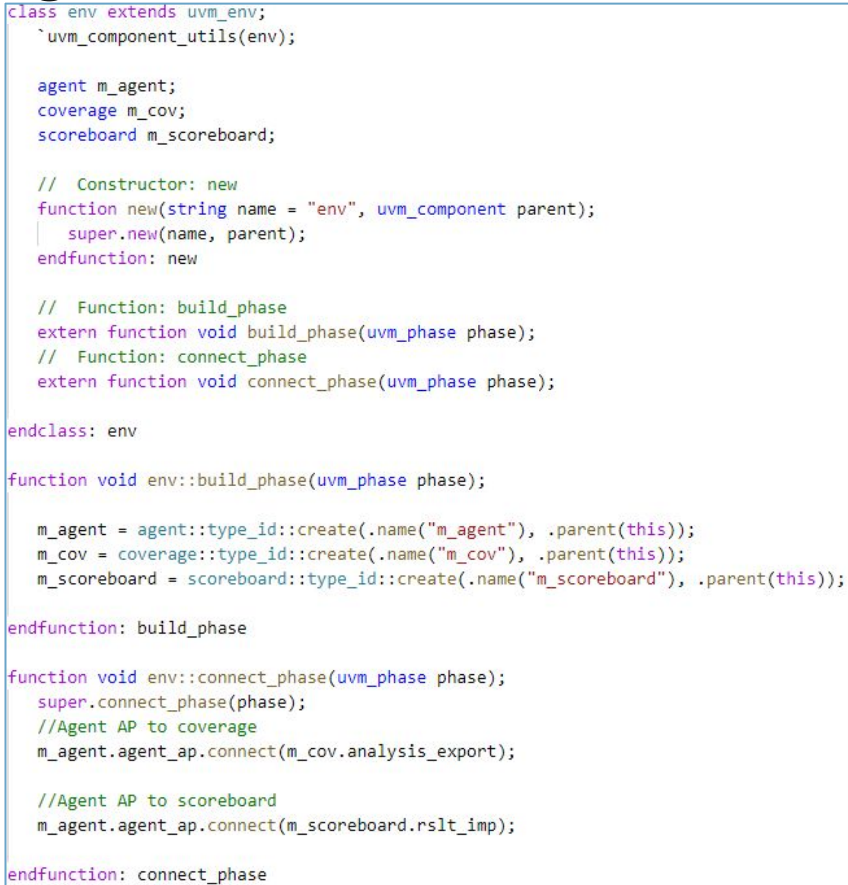
Env and Agent



```
class agent extends uvm_agent;
    `uvm_component_utils(agent);

    //  Group: Configuration Object(s)
    agent_config m_agent_cfg;

    //  Group: Components
    monitor m_mon;
    my_sequencer m_seqr;
    driver m_driver;

    //  Group: Variables
    //Analysis port to forwarding items to scoreboard and coverage collector
    uvm_analysis_port #(leros_command) agent_ap;
```

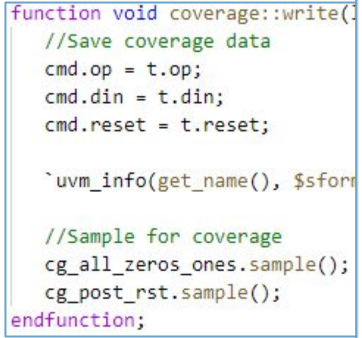


```
function void agent::build_phase(uvm_phase phase);
    //Get agent config and set is_active
    if(! uvm_config_db#(agent_config)::get(this, "", "agent_cfg", m_agent_cfg))
        `uvm_fatal(get_name(), "Unable to get agent config")

    this.is_active = m_agent_cfg.is_active;

    //Always build the monitor and ap
    m_mon = monitor::type_id::create("m_mon", this);
    agent_ap = new("agent_ap", this);

    //Build driver and sequencer if necessary
    if(this.is_active) begin
        m_driver = driver::type_id::create("m_driver", this);
        m_seqr = my_sequencer::type_id::create("m_seqr", this);
    end

endfunction: build_phase

function void agent::connect_phase(uvm_phase phase);
    //Always forward the monitors connection outwards
    m_mon.mon_ap.connect(agent_ap);

    //Connect sequencer and driver if they exist
    if(this.is_active) begin
        m_driver.seq_item_port.connect(m_seqr.seq_item_export);
    end
endfunction: connect_phase
```

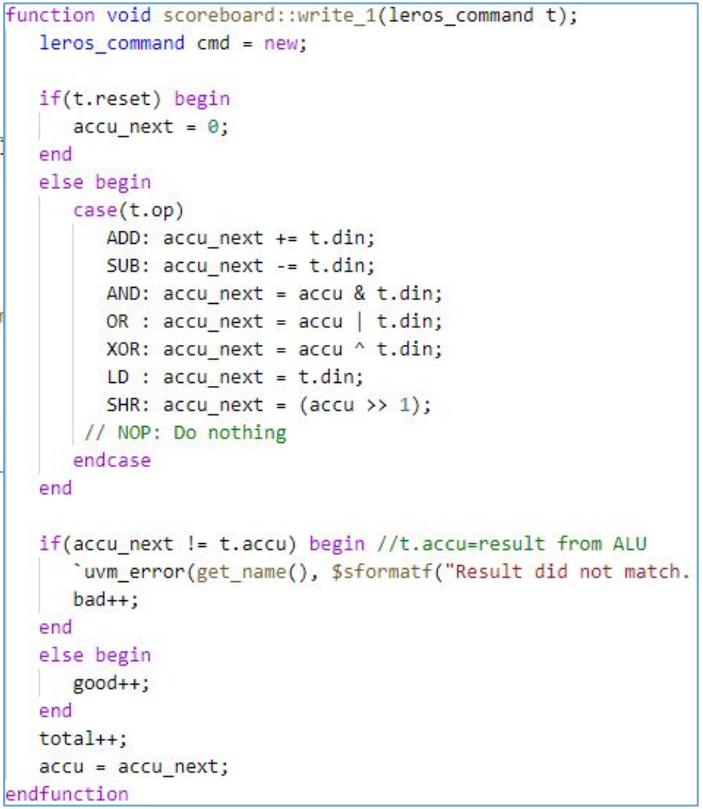


```
class env extends uvm_env;
    `uvm_component_utils(env);

    agent m_agent;
    coverage m_cov;
    scoreboard m_scoreboard;

    //  Constructor: new
    function new(string name = "env", uvm_component parent);
        super.new(name, parent);
    endfunction: new

    //  Function: build_phase
    extern function void build_phase(uvm_phase phase);
    //  Function: connect_phase
    extern function void connect_phase(uvm_phase phase);

endclass: env

function void env::build_phase(uvm_phase phase);

    m_agent = agent::type_id::create(.name("m_agent"), .parent(this));
    m_cov = coverage::type_id::create(.name("m_cov"), .parent(this));
    m_scoreboard = scoreboard::type_id::create(.name("m_scoreboard"), .parent(this));

endfunction: build_phase

function void env::connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //Agent AP to coverage
    m_agent.agent_ap.connect(m_cov.analysis_export);

    //Agent AP to scoreboard
    m_agent.agent_ap.connect(m_scoreboard.rslt_imp);

endfunction: connect_phase
```

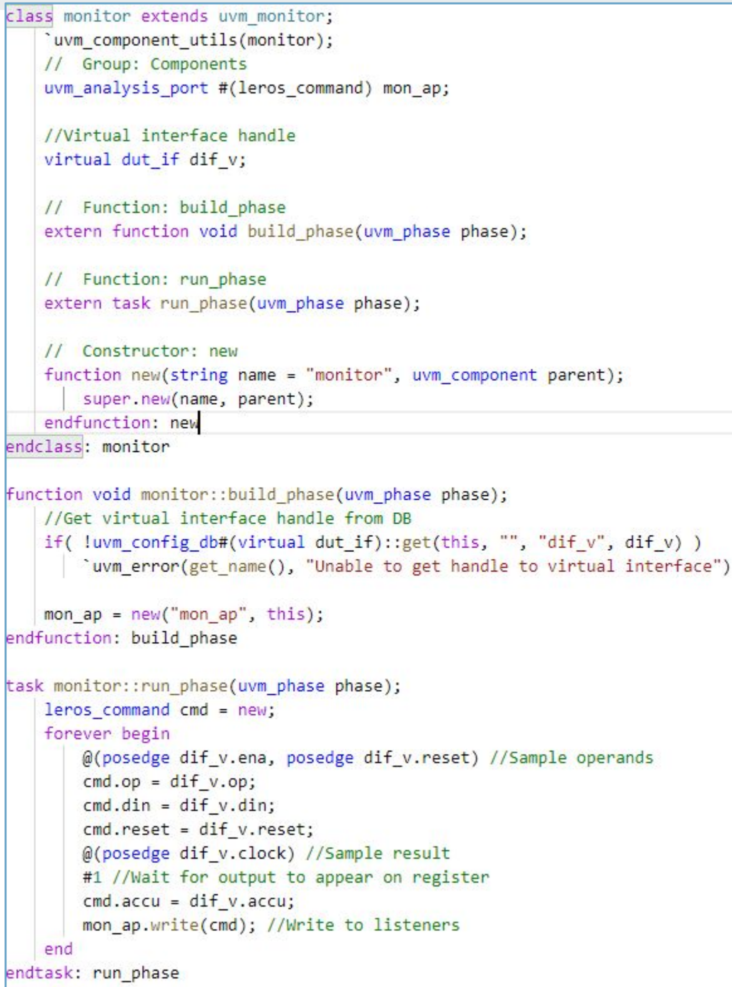
A deeper dive (5) | Analysis



```
function void coverage::write(
    //Save coverage data
    cmd.op = t.op;
    cmd.din = t.din;
    cmd.reset = t.reset;

    `uvm_info(get_name(), $sfor

    //Sample for coverage
    cg_all_zeros_ones.sample();
    cg_post_rst.sample();
endfunction;
```

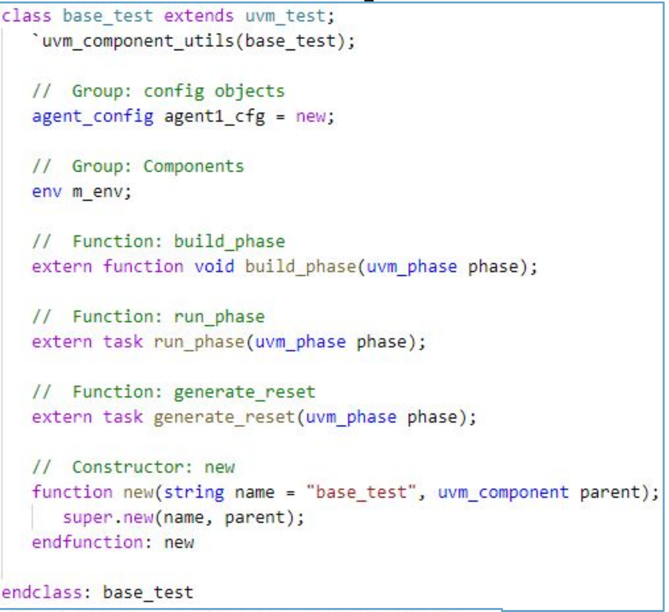


```
function void scoreboard::write_1(leros_command t);
   leros_command cmd = new;

   if(t.reset) begin
      accu_next = 0;
   end
   else begin
      case(t.op)
         ADD: accu_next += t.din;
         SUB: accu_next -= t.din;
         AND: accu_next = accu & t.din;
         OR : accu_next = accu | t.din;
         XOR: accu_next = accu ^ t.din;
         LD : accu_next = t.din;
         SHR: accu_next = (accu >> 1);
       // NOP: Do nothing
      endcase
   end

   if(accu_next != t.accu) begin //t.accu=result from ALU
      `uvm_error(get_name(), $sformatf("Result did not match.
      bad++;
   end
   else begin
      good++;
   end
   total++;
   accu = accu_next;
endfunction
```

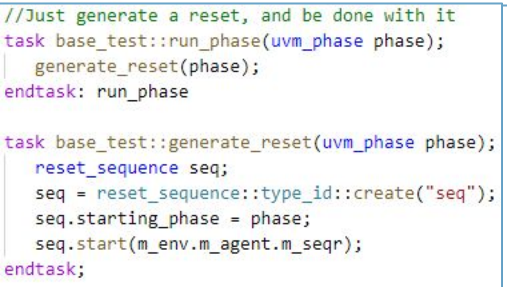


```
class monitor extends uvm_monitor;
    `uvm_component_utils(monitor);
    //  Group: Components
    uvm_analysis_port #(leros_command) mon_ap;

    //Virtual interface handle
    virtual dut_if dif_v;

    //  Function: build_phase
    extern function void build_phase(uvm_phase phase);

    //  Function: run_phase
    extern task run_phase(uvm_phase phase);

    //  Constructor: new
    function new(string name = "monitor", uvm_component parent);
        super.new(name, parent);
    endfunction: new
endclass: monitor

function void monitor::build_phase(uvm_phase phase);
    //Get virtual interface handle from DB
    if( !uvm_config_db#(virtual dut_if)::get(this, "", "dif_v", dif_v) )
        `uvm_error(get_name(), "Unable to get handle to virtual interface")

    mon_ap = new("mon_ap", this);
endfunction: build_phase

task monitor::run_phase(uvm_phase phase);
    leros_command cmd = new;
    forever begin
        @(posedge dif_v.ena, posedge dif_v.reset) //Sample operands
        cmd.op = dif_v.op;
        cmd.din = dif_v.din;
        cmd.reset = dif_v.reset;
        @(posedge dif_v.clock) //Sample result
        #1 //Wait for output to appear on register
        cmd.accu = dif_v.accu;
        mon_ap.write(cmd); //Write to listeners
    end
endtask: run_phase
```

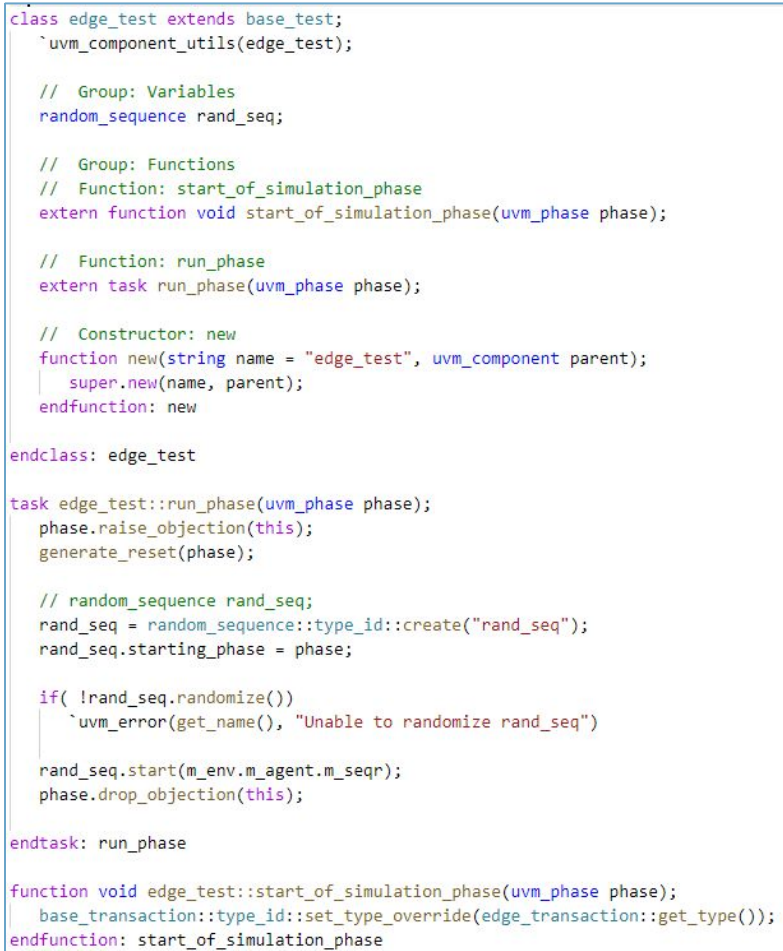
A deeper dive (6) | The tests



```
class base_test extends uvm_test;
   `uvm_component_utils(base_test);

   //  Group: config objects
   agent_config agent1_cfg = new;

   //  Group: Components
   env m_env;

   //  Function: build_phase
   extern function void build_phase(uvm_phase phase);

   //  Function: run_phase
   extern task run_phase(uvm_phase phase);

   //  Function: generate_reset
   extern task generate_reset(uvm_phase phase);

   //  Constructor: new
   function new(string name = "base_test", uvm_component parent);
      super.new(name, parent);
   endfunction: new

endclass: base_test
```

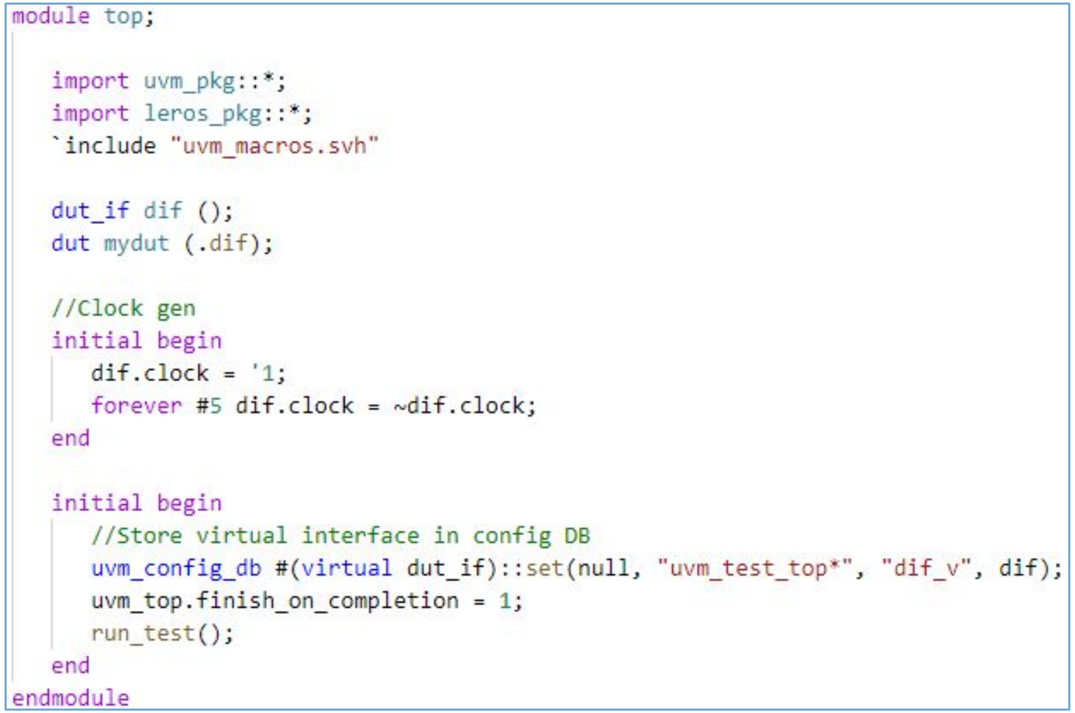


```
//Just generate a reset, and be done with it
task base_test::run_phase(uvm_phase phase);
   generate_reset(phase);
endtask: run_phase

task base_test::generate_reset(uvm_phase phase);
   reset_sequence seq;
   seq = reset_sequence::type_id::create("seq");
   seq.starting_phase = phase;
   seq.start(m_env.m_agent.m_seqr);
endtask;
```



```
class edge_test extends base_test;
   `uvm_component_utils(edge_test);

   //  Group: Variables
   random_sequence rand_seq;

   //  Group: Functions
   //  Function: start_of_simulation_phase
   extern function void start_of_simulation_phase(uvm_phase phase);

   //  Function: run_phase
   extern task run_phase(uvm_phase phase);

   //  Constructor: new
   function new(string name = "edge_test", uvm_component parent);
      super.new(name, parent);
   endfunction: new

endclass: edge_test

task edge_test::run_phase(uvm_phase phase);
   phase.raise_objection(this);
   generate_reset(phase);

   // random_sequence rand_seq;
   rand_seq = random_sequence::type_id::create("rand_seq");
   rand_seq.starting_phase = phase;

   if( !rand_seq.randomize())
      `uvm_error(get_name(), "Unable to randomize rand_seq")

   rand_seq.start(m_env.m_agent.m_seqr);
   phase.drop_objection(this);

endtask: run_phase

function void edge_test::start_of_simulation_phase(uvm_phase phase);
   base_transaction::type_id::set_type_override(edge_transaction::get_type());
endfunction: start_of_simulation_phase
```

A deeper dive (7) | The top



```
module top;

   import uvm_pkg::*;
   import leros_pkg::*;
   `include "uvm_macros.svh"

   dut_if dif ();
   dut mydut (.dif);

   //Clock gen
   initial begin
      dif.clock = '1;
      forever #5 dif.clock = ~dif.clock;
   end

   initial begin
      //Store virtual interface in config DB
      uvm_config_db #(virtual dut_if)::set(null, "uvm_test_top*", "dif_v", dif);
      uvm_top.finish_on_completion = 1;
      run_test();
   end
endmodule
```

So, UVM?

- I need 7 slides to explain how and why a very simple testbench works
- Just about 1000 LOC for this example
    - With comments and whitespace, probably 7-800 without
- Martin’s Scala tester does almost the same
    - Fits in 74 lines, with whitespace

So, UVM?

- I think it has potential
  - I like OOP
  - UVM is **not** the correct choice for simple designs like an ALU
  - For complex designs it makes a lot of sense
- Has an advantage over Chisel in supporting functional coverage
- Chisel testers are simpler to write + more intuitive
  - Does Chisel scale well enough for large-scale projects?
- Chisel also has OOP hardware design, which is a major bonus
  - Seriously, why didn’t they make this a part of SV modules as well?