Verification Plan for

OpenCores' 10 Gigabit Ethernet Media Access Controller Core

using

Universal Verification Methodology (UVM)

Albert Chiang

Updated : June 19, 202

Table of Content

INTRODUCTION

Universal Verification Methodology (UVM) is a framework written in the SystemVerilog language, created to speed up verification of RTL designs. UVM is an opinionated approach to how a verification environment should built and enhanced. Which means if something needs to be done, there is usually one way to do it. And you implicitly agreed on “the way” if you adopt UVM. An analogy to this in the software world is : Java is a general purpose programming language (purported the #1 programming language used by enterprise software such as ERP). To verify Java programs, a framework called Spring was invented to help accelerate verification of Java programs. Spring Boot adds an opinionated approach to Spring. Without UVM, spaghetti coded ruled the RTL verification world. And spaghetti code made the task harder for the next engineer to learn, debug, fix, and enhance.

In today's hyper connected life style, Ethernet is the networking backbone of providing a seamless experience, connecting users to services. Ethernet is a highly robust and resilient network technology that was invented in the 1970s, and has survived the test of time to connect people at a global scale. One key component of an Ethernet network is the Media Access Control (MAC). MAC provides the fundamental technology governing how devices communicate over local area networks (LANs). At its core, Ethernet MAC is responsible for facilitating the transmission of data packets between devices within a network, ensuring efficient and reliable communication.

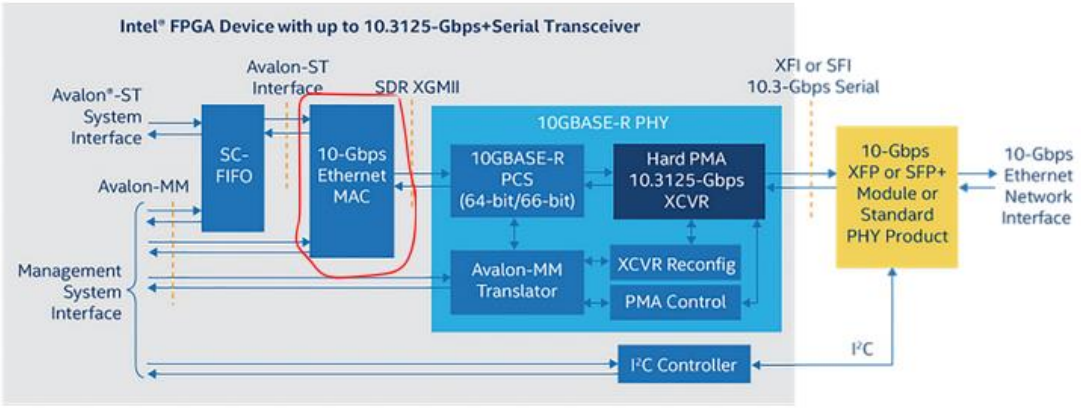
This paper will explore using UVM to verify OpenCores’ 10GE MAC is open source in the Verilog language. It comes with a simple testbench to verify its functionality and is also written in the same language of Verilog. While using Verilog for both design and verification seems natural, the Verilog language does not natively come with important verification elements such as randomization, coverage, assertions, advanced data structures, and object-oriented programming. The SystemVerilog language was created to augment Verilog to support these concepts. UVM, based on SystemVerilog, provides an opinioned framework of how SystemVerilog should be used to build a verification environment quickly. In this paper, we will examine the functionality of a MAC, create a simulation environment to verify its functionality using UVM, and share the results.

SYSTEM OVERVIEW & DESIGN UNDER TEST

SYSTEM OVERVIEW

Ethernet is an established, tried and true networking technology that has connected the world since its inception in the 1980s. Specifically, Ethernet is standard that defines how devices (clients, servers) can talk to each other on a shared network called Local Area Network (LAN).

A Media Access Controller (MAC) is a key component of an Ethernet network environment. In the block diagram downloaded from Intel shown below, the MAC receives command from the “Avalon” system interface, then outputs via 10 Gigabit Media Independent Interface (XGMII) interface to the next block – the PHY block. In this paper, the input interface to the 10Gigabit Ethernet MAC will be called “PKT_TX_”, such as “PKT_TX_DATA”



https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/10g-base-r-pcs.html

Figure 1 : The role of the 10GE MAC in an Ethernet device. Picture from Intel.

DESIGN UNDER TEST: 10 GIGABIT ETHERNET MEDIA ACCESS CONTROLLER (MAC)

The 10 Gigabit Ethernet (10GE) MAC core that we will be using the Design Under Test (DUT) in our verification is a soft core from OpenCores. OpenCores provide the Register Transfer Logic (RTL) source code for the 10GE MAC, written in the Verilog language. I will use the word "DUT" and "10GE MAC" interchangeably, depending on context.

DUT BLOCK DIAGRAM & MAJOR DATA PATHS

In order to verify the 10GE MAC DUT, the major interfaces need to be identified and understood so that the UVM environment, specifically virtual interfaces, drivers, and monitors, needs be bult properly for the UVM environment to interact properly with the DUT.

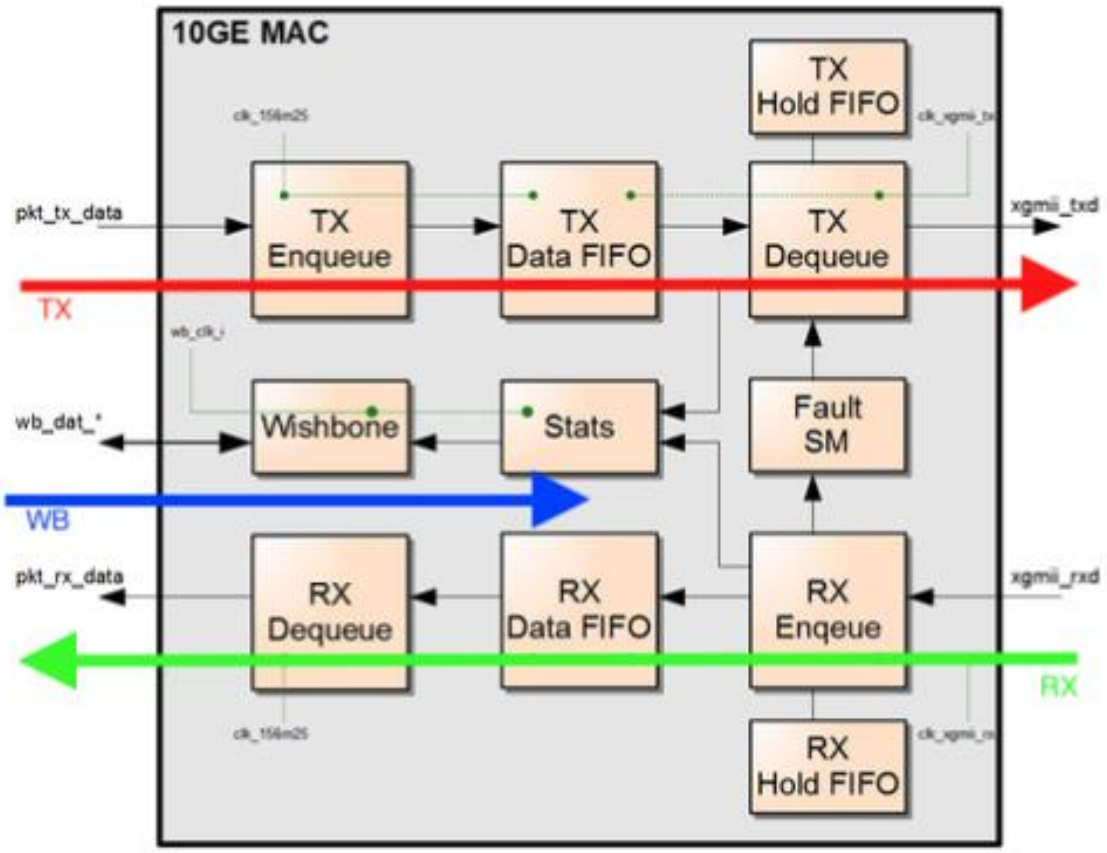


Figure 2: OpenCores 10GE MAC diagram. TX path is pkt_tx_* to xgmii_tx. RX path is xgmii_rxd to pkt_rx_data.

TX PATH : PACKET TRANSMIT PATH INTERFACE (PKT_TX) TO XGMII TRANSMIT (XGMII_TX)

For the device to transmit Ethernet packets out, the device receives data from the PKT_TX interface, shown on the left of Figure 2. Once the packet is received, it is processed by the 10GE MAC and sent back out to the XGMII_TXD port. The PKT_TX interface uses the “standard” SOP, VAL, DATA, EOP protocol. The XGMII_TXD uses TXC for control and TXD for data. More details below.

RX PATH : XGMII RECEIVE (GXMII_RXD) TO PACKET RECEIVE PATH INTERFACE (PKT_RX)

For the device to receive Ethernet packets in, the device receives data from the XGMII_RXC & XGMII_RXD ports.. Once the packet is received, it is processed by the 10GE MAC and sent into the device via PKT_RX interface. The PKT_TRX interface uses the “standard” SOP, VAL, DATA, EOP protocol. The XGMII_RXD uses RXC for control and RXD for data. More details below.

WISHBONE PATH : REGISTER INTERFACE (WB)

The 10GE MAC DUT contains eight 32-bit registers, used for configuration, counting, and interrupt control. In this paper, the DUT can perform basic functions without using the WB interface, and hence I will not be actively using this interface.

VERIFICATION ENVIRONMENT USING UNIVERSAL VERIFICATION METHODOLOGY (UVM)

HIGH LEVEL OVERVIEW OF UVM VERIFICATION ENVIRONMENT

UVM is implemented in SystemVerilog, an Object-Oriented Programming (OOP) language. One beauty of OOP is the concept of encapsulation – components can be constructed systematically, then assembled together to build a system. In UVM, the components are pre-defined templates. This diagram attempts to show the major components and their role in the UVM verification environment. The stripped-out component called "RTL" is a non UVM component.

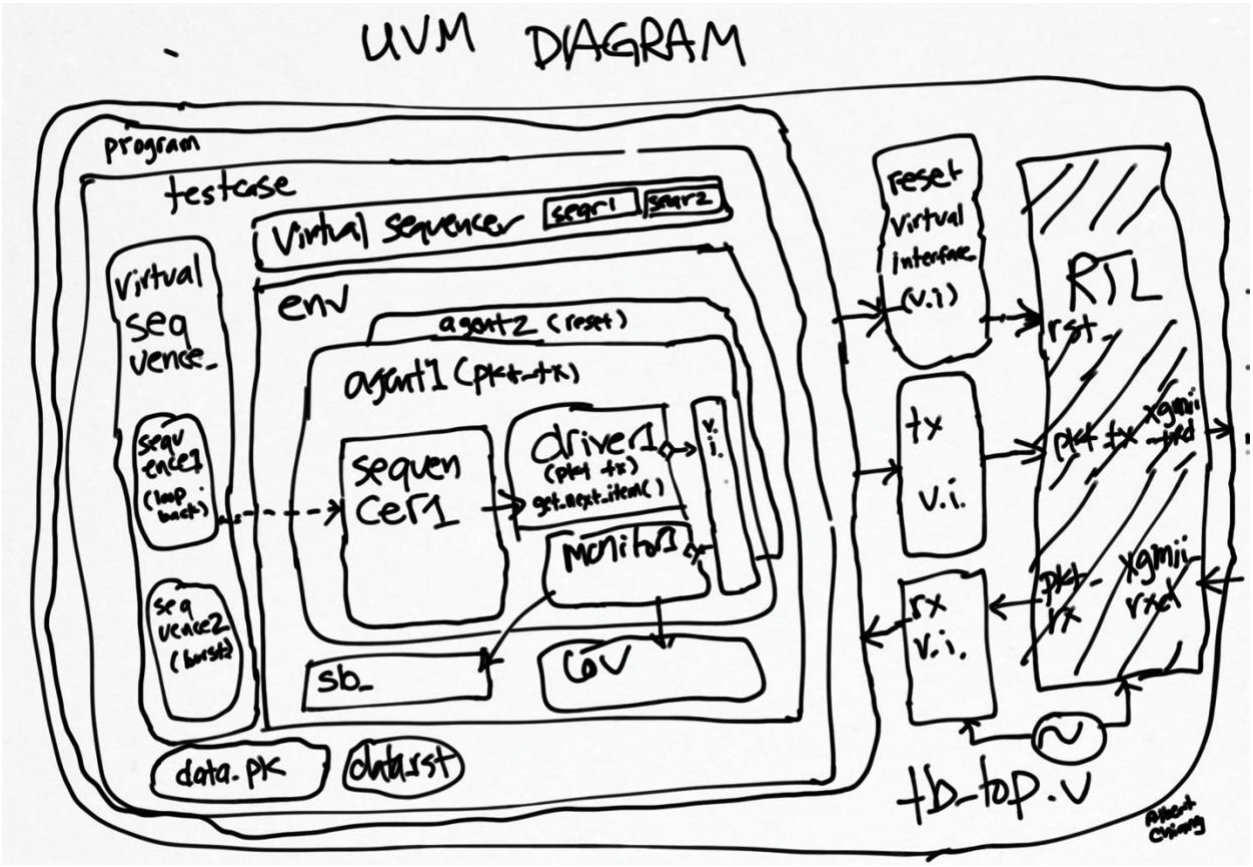


Figure 3. UVM Components Used in This Paper, drawn spaghetti style

TESTBENCH TOP : RTL, VIRTUAL INTERFACES, CLOCK GENERATOR

This is a simplified version of the very top of the UVM verification environment – a Verilog module. Key components instantiated are 1) the 10GE MAC called “xge_mac” 2) virtual interfaces, such as “intf_pkt_tx” 3) clock generator(s).

tb_top_xge_mac.v

testbench > uvm > tb_top_xge_mac.v

```
module tb_top_xge_mac;


reg clk_156m25;

// virtual interfaces
intf_pkt_tx intf_pkt_tx_0(clk_156m25);
intf_rst intf_rst_0(clk_156m25);
intf_wb intf_wb_0(clk_wb);


// 10GE MAC DUT
xge_mac dut(
    .pkt_tx_data (intf_pkt_tx_0.pkt_tx_data)
);

// clock generator
initial clk_156m25 = 0;
always #10 clk_156m25 = ~clk_156m25;


endmodule // tb_top_xge_mac
```

VIRTUAL INTERFACE

“Virtual interface”, a SystemVerilog concept, is a data type that connects the 10GE MAC DUT to the UVM verification environment.

testbench > uvm > intf_pkt_tx.sv

```
interface intf_pkt_tx( input bit clk); // clk_156m25);

   // input
   logic [63:0] pkt_tx_data;
   logic        pkt_tx_eop;
   logic [2:0]  pkt_tx_mod;
   logic        pkt_tx_sop;
   logic        pkt_tx_val;
   logic        reset_156m25_n;

   // output
   logic pkt_tx_full;


   clocking cb_driver @(posedge clk);
      output pkt_tx_data;
      output pkt_tx_eop;
      output pkt_tx_mod;
      output pkt_tx_sop;
      output pkt_tx_val;
      input pkt_tx_full;
   endclocking

endinterface
```

Figure 4: Virtual interface for PKT_TX

testbench > uvm > intf_xgmii_tx.sv

```
interface intf_xgmii_tx ( input bit clk);

   logic [63:0] xgmii_tx_txd;
   logic [7:0] xgmii_tx_txc;


   clocking cb_driver @(posedge clk);
      xgmii_tx_txd;
      xgmii_tx_txc;
   endclocking

endinterface
```

Figure 5: Virtual interface for XGMII_TX

testbench > uvm > intf_wb.sv

```
interface intf_wb( input bit wb_clk);

   // input to WB
   logic [7:0] adr;
   logic [31:0] dat_i;
   logic we;
   logic stb;
   logic cyc;

   // output from WB
   logic [31:0] dat_o;
   logic ack;
   logic intr;

   clocking cb_driver @(posedge wb_clk);
      output adr;
      output dat_i;
      output we;
      output stb;
      output cyc;


      input dat_o;
      input ack;
      input intr;

   endclocking

endinterface
```

Figure 6: Virtual interface for Wishbone

testbench > uvm > intf_rst.sv

```
interface intf_rst( input bit clk_156m25);

   logic reset_156m25_n;
   logic wb_rst_i;
   logic reset_xgmii_rx_n;
   logic reset_xgmii_tx_n;

   clocking cb_driver @(posedge clk_156m25);
      output reset_156m25_n;
      output wb_rst_i;
      output reset_xgmii_rx_n;
      output reset_xgmii_tx_n;
   endclocking

endinterface
```

Figure 7: Virtual interface for Reset

TEST PROGRAM : LAUNCH TESTCASE

The main UVM verification execution starts here – hence called program. Right off the bat, only one method is called : run_test(). Ideally, it should look like this:

```
testbench > uvm > tb_prog.sv
1  program tb_prog;
2     import uvm_pkg::*;
3     initial begin
4        run_test();
5     end
6  endprogram
7
```

But because this is where everything starts for UVM, what is done here will have repercussions later. Some key items to consider that I have implemented :

1) run_test() is called without passing a testcase argument - instead, I pass it in at runtime (+UVM_TESTNAME), which makes running other testcases easier because running a different test does NOT require editing and compiling this program file.
2) the testcase source code is NOT included here - instead all testcase(s) are compiled in, which makes running other testcases easier because running a different test does NOT require editing and compiling this program file. Instead testcase selection is now completely a runtime option (+UVM_TESTNAME).
3) the use of Verilog $display, even in the program world, aids in debug and sanity checking.

```
tb_top_xge_mac.v ●   tb_prog.sv ●   testcase_loopback.sv
testbench > uvm > tb_prog.sv
1  program tb_prog;
2     import uvm_pkg::*;
3     // `include "testcase_loo[back].sv"
4     initial begin
5        $display("Hello from tb_prog");
6        run_test(); // uvm_top.run_test();
7        // run_test("testcase_loopback"); // +UVM_TESTNAME=testcase_loopback
8     end
9  endprogram
10
```

TESTCASE : ENVIRONMENT, VIRTUAL SEQUENCER, VIRTUAL SEQUENCE

A UVM testcase is the first official top level class, and it encapsulates the components needed for the verification environment. It contains the 1) verification environment (think of it as a test fixture) 2) virtual sequencers (a library of sequencers) and virtual sequences (a library of sequences, coordinates timing amongst sequences). But testcase is NOT where the actual test is written – it is located in sequences.

testcases > testcase_base.sv

```
`ifndef _TESTCASE_BASE_
`define _TESTCASE_BASE_

import uvm_pkg::*;

`include "env.sv"

class testcase_base extends uvm_test;


   `uvm_component_utils(testcase_base)


   env env_0;

   function new(string n, uvm_component p);
      super.new(n, p);
   endfunction


  virtual function void build_phase(uvm_phase phase) ;
     super.build_phase(phase) ;
     env_0  = env::type_id::create("env_0", this);
     uvm_config_db#(virtual intf_wb)::set(this, "env_0.agent_wb_0.monitor_wb_0", "vi", tb_top_xge_mac.int
     uvm_config_db#(virtual intf_wb)::set(this, "env_0.agent_wb_0.driver_wb_0", "vi", tb_top_xge_mac.intf
     uvm_config_db#(virtual intf_rst)::set(this, "env_0.agent_reset_0.driver_reset_0", "vi", tb_top_xge_m
     uvm_config_db#(virtual intf_pkt_tx)::set(this, "env_0.agent_pkt_tx_0.driver_pkt_tx_0", "vi", tb_top_
     uvm_config_db#(virtual intf_pkt_tx)::set(this, "env_0.agent_pkt_tx_0.monitor_pkt_tx_0", "vi", tb_top
     uvm_config_db#(virtual intf_pkt_rx)::set(this, "env_0.agent_pkt_rx_0.driver_pkt_rx_0", "vi", tb_top_
     uvm_config_db#(virtual intf_pkt_rx)::set(this, "env_0.agent_pkt_rx_0.monitor_pkt_rx_0", "vi", tb_top
  endfunction

   virtual function void end_of_elaboration_phase(uvm_phase phase);
      super.end_of_elaboration_phase(phase);
      uvm_top.print_topology();
      factory.print();
   endfunction

virtual function void start_of_simulation_phase(input uvm_phase phase);
     super.start_of_simulation_phase(phase);
  endfunction


  virtual task run_phase(input uvm_phase phase);
     super.run_phase(phase);
  endtask

  virtual task main_phase(uvm_phase phase) ;
     super.main_phase(phase) ;
     // phase.raise_objection(this);
     // phase.drop_objection(this);
  endtask

endclass
`endif //  _TESTCASE_BASE_
```

Ln 1, Col 1 Spaces: 3 UTF-8 LF System Verilog

Figure 8: All testcases will derive from a testcase_base that does necessary setup such as virtual interface hookup in build_phase()

All testcases will inherit from testcase_base, such as test testcase_loopback:

testcases > testcase_loopback_reset.sv

```
import uvm_pkg::*;

`include "env.sv"
`include "sequence_pkt_tx.sv"
`include "virtual_sequencer.sv"
`include "virtual_sequence.sv"

class testcase_loopback_reset extends testcase_base;


  `uvm_component_utils(testcase_loopback_reset)


  // env env_0; // in testcase_base
  sequence_reset sequence_reset_0;
  sequence_pkt_tx sequence_pkt_tx_0;
  virtual_sequencer  virtual_sequencer_0;

  function new(string n, uvm_component p);
    super.new(n, p);
  endfunction


 virtual function void build_phase(uvm_phase phase) ;
   super.build_phase(phase) ;
     virtual_sequencer_0  = virtual_sequencer::type_id::create("virtual_sequencer_0", this);

 endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    virtual_sequencer_0.sequencer_pkt_tx_0 = env_0.agent_pkt_tx_0.sequencer_pkt_tx_0;
    virtual_sequencer_0.sequencer_reset_0 = env_0.agent_reset_0.sequencer_reset_0;
   uvm_config_db #(uvm_object_wrapper)::set(this, "virtual_sequencer_0.reset_phase", "default_sequence"
  endfunction // connect_phase


  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    uvm_top.print_topology();
    factory.print();
  endfunction

virtual function void start_of_simulation_phase(input uvm_phase phase);
    super.start_of_simulation_phase(phase);
 endfunction


 virtual task run_phase(input uvm_phase phase);
    super.run_phase(phase);
    sequence_pkt_tx_0 = sequence_pkt_tx::type_id::create("sequence_pkt_tx_0", this);
    sequence_reset_0 = sequence_reset::type_id::create("sequence_reset_0", this);
 endtask

 virtual task main_phase(uvm_phase phase) ;

   uvm_objection objection;

   super.main_phase(phase) ;

   phase.raise_objection(this);
   sequence_pkt_tx_0.start(virtual_sequencer_0.sequencer_pkt_tx_0);
   sequence_reset_0.start(virtual_sequencer_0.sequencer_reset_0);
   phase.drop_objection(this);

   // from Benjamin's book
   objection = phase.get_objection();
   objection.set_drain_time(this, 90us);  // works!
                                          // 100us  ->Time: 100617600 ps
                                          // 90us -> Time: 90617600 ps



 endtask

endclass
```

Figure 9: A testcase that encapsulates the environment, sequences

Although the testcase is the top most representative of a test, the sequence(s) that the testcase uses is the heart of the stimulus.

virtual_sequence.sv ● | sequence_virtual.sv | sequence_wb.sv | sequence_pkt_tx.sv M | monitor_pkt_tx.sv M

testbench > uvm > virtual_sequence.sv

```
`ifndef _SEQUENCE_VIRTUAL_
`define _SEQUENCE_VIRTUAL_


class virtual_sequence extends uvm_sequence;

  `uvm_object_utils(virtual_sequence)
  `uvm_declare_p_sequencer(virtual_sequencer)

  sequence_reset     sequence_reset_0;
  sequence_pkt_tx    sequence_pkt_tx_0;
  // wishbone_init_sequence    seq_init_wshbn;
  // wishbone_eot_sequence     seq_eot_wshbn;
  // packet_sequence           seq_pkt;

  function new(input string name="virtual_sequence");
    super.new(name);
    `uvm_info( get_name(), $sformatf("Hierarchy: %m"), UVM_HIGH )
  endfunction : new


  virtual task body();
    `uvm_do_on(sequence_reset_0, p_sequencer.sequence_reset_0);
    `uvm_do_on( seq_pkt, p_sequencer.seqr_tx_pkt );
    #1000000;
    `uvm_do_on( seq_eot_wshbn, p_sequencer.seqr_wshbn );
    `uvm_do_on(sequence_pkt_tx_0, p_sequencer.sequence_pkt_tx_0);
  endtask : body


  virtual task pre_start();
    super.pre_start();
    if ( (starting_phase!=null) && (get_parent_sequence()==null) )
      starting_phase.raise_objection( this );
  endtask : pre_start


  virtual task post_start();
    super.post_start();
    if ( (starting_phase!=null) && (get_parent_sequence()==null) )
      starting_phase.drop_objection( this );
  endtask : post_start

endclass : virtual_sequence


`endif // _VIRTUAL_SEQUENCE_
```

Figure 10: A virtual sequence is a library and coordinate of sequences

virtual_sequence.sv ● | virtual_sequencer.sv U ● | sequence_virtual.sv | sequence_wb.sv | sequence_pkt_tx.sv M

testbench > uvm > virtual_sequencer.sv

```
class virtual_sequencer extends uvm_sequencer;

   `uvm_component_utils(virtual_sequencer)

    sequencer_pkt_tx sequencer_pkt_tx_0;
    sequencer_pkt_wb sequencer_pkt_wb_0;
    sequencer_pkt_rst sequencer_pkt_rst_0;

    function new(input string n, input uvm_component p);
       super.new(n, p);
    endfunction

endclass
```

Figure 11: A virtual sequencer is a library of sequencers that already reside in agents

ENVIRONMENT : AGENT(S), SEQUENCE, COVERAGE, SCOREBOARD

The environment is considered a “test fixture” that a test person will use to test the RTL. perform tests will be interfacing withing to test the DUT. Sequence, coverage, and scoreboard are criterial, highly customized, design specific components of the environment and will be discussed separately below.

AGENT : DRIVER, SEQUENCER, MONITOR, SEQUENCE_ITEM

DRIVER

testbench > uvm > driver_reset.sv

```
`ifndef _DRIVER_RESET_
`define _DRIVER_RESET_

class driver_reset extends uvm_driver #(data_reset);

   `uvm_component_utils(driver_reset)


   virtual intf_rst vi;

   function new(input string name="Driver for reset", input uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual function void  build_phase(input uvm_phase phase);
      super.build_phase(phase);
      uvm_config_db#(virtual intf_rst)::get(this, "", "vi", vi);
      if (vi == null) begin
         `uvm_fatal("Driver for reset ", "VI for DRIVER not in CONFIG_DB");
      end
   endfunction

   virtual task run_phase(input uvm_phase phase);

      `uvm_info("DRIVER CLASS", "HIERARCHY: %m", UVM_HIGH);

      forever begin

      seq_item_port.get_next_item(req);

      `uvm_info("XAC RESET DRIVER run_phase received this packet ", req.sprint(), UVM_HIGH);

      vi.wb_rst_i <= req.reset_;
      vi.reset_xgmii_rx_n <= req.reset_;
      vi.reset_xgmii_tx_n <= req.reset_;
      vi.reset_156m25_n <= req.reset_;

      repeat(req.cycles) @(posedge vi.clk_156m25);
      vi.wb_rst_i <= ~req.reset_;
      vi.reset_xgmii_rx_n <= ~req.reset_;
      vi.reset_xgmii_tx_n <= ~req.reset_;
      vi.reset_156m25_n <= ~req.reset_; /


      seq_item_port.item_done();
      end // forever
   endtask

endclass

`endif // _DRIVER_RESET_

```

MONITOR

Monitor watch activities on the RTL, capture the data, then packages them into packet data.

virtual_sequencer.sv U | sequence_virtual.sv | sequence_wb.sv | sequence_pkt_tx.sv M | monitor_pkt_tx.sv M

testbench > uvm > monitor_pkt_tx.sv

```
`ifndef _MONITOR_PACKET_TX_MONITOR_
`define _MONITOR_PACKET_TX_MONITOR_


class monitor_pkt_tx extends uvm_monitor;

  virtual intf_pkt_tx     vi;
  int unsigned                  m_num_captured;
  uvm_analysis_port #(data_pkt_tx)   ap_tx_mon;

  `uvm_component_utils( monitor_pkt_tx )

  function new(input string name="monitor_pkt_tx", input uvm_component parent);
    super.new(name, parent);
  endfunction : new


  virtual function void build_phase(input uvm_phase phase);
    super.build_phase(phase);
    m_num_captured = 0;
    ap_tx_mon = new ( "ap_tx_mon", this );
    uvm_config_db#(virtual intf_pkt_tx)::get(this, "", "vi", vi);
    if ( vi==null )
      `uvm_fatal(get_name(), "Virtual Interface for monitor not set!");
  endfunction : build_phase


  virtual task run_phase(input uvm_phase phase);
    data_pkt_tx      tmp_rcv_pkt;
    bit [7:0]   q_rx_data[$];
    int         idx;

    `uvm_info( get_name(), $sformatf("HIERARCHY: %m"), UVM_HIGH);


    forever begin


      @(vi.clk) // @(vi.mon_cb)
      if ( vi.pkt_rx_val ) begin
        // if ( vi.pkt_rx_sop && !vi.pkt_rx_eop && pkt_in_progress==0 ) begin
        tmp_rcv_pkt = data_pkt_rx::type_id::create("tmp_rcv_pkt", this);

        if ( vi.pkt_rx_sop && !vi.pkt_rx_eop  ) begin
           $display("XAC PKT_RX_MON sop : %b", vi.pkt_rx_sop);
           $display("XAC PKT_RX_MON sop : %b", vi.pkt_rx_sop);
           $display("XAC PKT_RX_MON data : %h", vi.pkt_rx_data);
           tmp_rcv_pkt.data = vi.pkt_rx_data;
           `uvm_info("XAC Hey from PKT_RX MON ","SEE THIS", UVM_HIGH);
           // `uvm_info("XAC PKT RX MON run_phase received this packet ", req.sprint(), UVM_HIGH);

        end // vi.pkt_rx_sop && !vi.pkt_rx_eop  ) begin
      end // if vi.pkt_rx_val
    end // forever begin
    ap_rx_mon.write(tmp_rcv_pkt);
  endtask : run_phase


  function void report_phase( uvm_phase phase );
    `uvm_info( get_name( ), $sformatf( "REPORT: Captured %0d packets", m_num_captured ), UVM_LOW )
  endfunction : report_phase

endclass : monitor_pkt_tx

`endif


```

SEQUENCE

Sequence is the heart and soul of stimulus generation. Here I show three methods 1) using uvm_do for quick stimulus 2) uvm_do_with for constrained random and 3) start_item(req)/finish_item(req) to fine tune stimulus.

testbench > uvm > sequence_pkt_tx.sv

```
`ifndef _SEQUENCNE_PKT_TX_
`define _SEQUENCNE_PKT_TX_
`include "data_pkt.sv"

class sequence_pkt_tx extends uvm_sequence#(data_pkt);



   `uvm_object_utils(sequence_pkt_tx)

   virtual task body();

      reg [7:0] taddr;
      reg [31:0] tdata;



      // method #1 just randomize using uvm_do
      repeat(10) `uvm_do(req);


      // method #2 randomize using uvm_do_with


      // `uvm_do_with(req, {src_addr != 48'h1234; src_data != 32'h5678;})
      `uvm_do_with(req, {data == 64'h5A5ABABA;data2.size() > 10; data2.size() < 50; });
      `uvm_do_with(req, {data == 64'hDEADCAFE;data2.size() > 200; data2.size () < 500; ifg == 1; });
      `uvm_do_with(req, {data == 64'hBEEFFACE;data2.size() > 40; data2.size() < 45; ifg == 30; });
      `uvm_do_with(req, {data == 64'hBAD00100; data2.size() == 3; ifg == 3; });
      `uvm_do_with(req, {data == 64'hA1B00B1A;data2.size() < 70; ifg == 18; });



      // method #3  using start_item and finish_item

      taddr = 8'hAB;
      tdata = 32'h00112233;
      $display("XAC Hey from Sequence aka Generator start_item ","producing addr=%h data=%h", taddr, tdat
      req = data_pkt::type_id::create("req");
      start_item(req);
      req.adr = taddr;
      req.dat_i = tdata;
      req.we = 1;
      req.cyc = 1;
      req.stb = 1;
      finish_item(req);

      finish_item(req);


   endtask
endclass
`endif //  _SEQUENCNE_PKT_TX_
```

Figure 12: A UVM Sequence is the heart of stimulus generation.

SEQUENCE_ITEM

For each major interface (RX, TX, RST, WB, XGMIITX, XGMIIRX), a sequence_item is created to convey control and/or data. For the TX and RX interfaces, here are both the packet data and meta data such is inter-frame gap.

Figure 13: The

testcase_base.sv | testcase_loopback_reset.sv | scoreboard_pkt_tx.sv | sequence_pkt_tx.sv

testbench > uvm > data_pkt.sv

```
`ifndef _DATA_PKT_
`define _DATA_PKT_

class data_pkt extends uvm_sequence_item;

   rand bit [63:0]  data;
   rand bit [63:0]  data2 [];

   rand bit val;
   rand bit sop;
   rand bit eop;
   rand bit [2:0] mod;

   rand int ifg;
   int len; // not used for pkt gen; used to pass len info to coverage

   `uvm_object_utils_begin(data_pkt)
      `uvm_field_int (data, UVM_ALL_ON)
      `uvm_field_array_int (data2, UVM_ALL_ON)
      `uvm_field_int (mod, UVM_ALL_ON)
      `uvm_field_int (val, UVM_ALL_ON)
      `uvm_field_int (sop, UVM_ALL_ON)
      `uvm_field_int (eop, UVM_ALL_ON)
      `uvm_field_int (ifg, UVM_ALL_ON)
   `uvm_object_utils_end


   constraint cons_ifg {
      ifg < 5000;
      ifg > 0;
   }
endclass

`endif //  _DATA_PKT_TX_

```

core element of traffic data or commands

SCOREBOARD

A scoreboard is the UVM component that decides if a testcase passes or fails. In this project, we simply collect data being transmitted out on the TX interface, then collect the packet returning back on the RX interface. A TX MONITOR collects what was sent. The RX MONIOTR collects what was retuned. The UVM_SCOREBOARD has a built in COMPARATOR to compare IN-ORDER transmission and reception of packets.

testcase_loopback_reset.sv ● | scoreboard_pkt_tx.sv | sequence_pkt_tx.sv ● | data_pkt.sv | scoreboard_pkt.sv ×

testbench > uvm > scoreboard_pkt.sv

```
class scoreboard_pkt extends uvm_scoreboard;

    `uvm_component_utils(scoreboard_pkt)

    typedef uvm_in_order_class_comparator#(data_pkt) packet_comparator;
    packet_comparator comparator;
    `uvm_analysis_imp_decl(ap_from_rx)

    uvm_analysis_export#(data_pkt) from_agent_out; // out from RTL
    uvm_analysis_export#(data_pkt) from_agent_in; // in to RTL

    function new(input string name = "sb", input uvm_component parent);
      super.new(name, parent);
    endfunction


    virtual function void build_phase(input uvm_phase phase);
       super.build_phase(phase);
       comparator  = packet_comparator::type_id::create("comparator", this);
       from_agent_out =new("from_agent_out", this);
       from_agent_in = new("from_agent_in", this);
    endfunction

    virtual function void connect_phase(input uvm_phase phase);
       super.connect_phase(phase);
       this.from_agent_out.connect(comparator.after_export);
       this.from_agent_in.connect(comparator.before_export);
    endfunction

    virtual function void report_phase(input uvm_phase phase);
       super.report_phase(phase);
       // `uvm_info("Scoreboard Report", $sformat("Matches : %0d. Mismatches : %0d", comparator.m_matches
       $display("Scoreboard Report : Matches : %0d. Mismatches : %0d", comparator.m_matches, comparator.m

    endfunction
endclass
```

Figure 14: Scoreboard stores outputs from the RTL (passed to it from the monitor) and compares the data to expected data.

COVERAGE

There are 3 major types of coverage:

Functional coverage : SystemVerilog natively supports functional coverage via the “covergroup” construct. It is idea to track states, crosses, and transitions of transactions.

Assertion coverage : SystemVerilog natively supports assertion coverage. Assertion coverage is ideal for directly looking at RTL signals check the “timing” aspects of the protocol.

Code coverage : Code coverage is supported internally by the simulator such as VCS.

testbench > uvm > coverage_pkt_tx.sv

```
class coverage_pkt_tx  extends uvm_subscriber#(data_pkt);

   `uvm_component_utils(coverage_pkt_tx )

   data_pkt data_pkt_to_cov;

   covergroup cg_pkt_tx;
      ifg : coverpoint data_pkt_to_cov.ifg {
               bins bin_low_ifg =  {[1:9]};
               bins bin_high_ifg =  {[10:999]};
            }
      len : coverpoint data_pkt_to_cov.len {
               bins bin_low_len =  {[1:99]};
               bins bin_high_ifg =  {[100:3999]};
            }

   endgroup

   function new(input string name = "coverage_pkt_tx ", input uvm_component parent);
      super.new(name, parent);
      cg_pkt_tx = new();
   endfunction

   function void build_phase(input uvm_phase phase);
      data_pkt_to_cov = new();
   endfunction // build_phase(input uvm_phase phase);

   // function void write(input data_pkt pkt_from_agent_out);
   function void write(input data_pkt t);
      // this.data_pkt_to_cov = pkt_from_agent_out;
      this.data_pkt_to_cov = t;
      cg_pkt_tx.sample();
   endfunction // write(input data_pkt pkt_from_agent_out);
endclass // coverage_pkt_tx  extends uvm_subscriber#(data_pkt);
```

Figure 15: Functional coverage is implemented as a subscriber to a monitor that captures what is seen on an interface. The native SystemVerilog "covergroup" is used to collect the coverage data.

TESTPLAN – TYING TESTCASE TO COVERAGE

The testplan how to systematically verify the DUT. And more importantly, how to track that the verification was done. A proper testplan is typically created by the verification engineer (as opposed to the design engineer) – so as to keep a check and balance between the creator and the verifier. The testplan is a critical piece of document and hence once written, is approved by the team.

PHASE 0 – RESET & BASIC CONNECTIVITY

Testcase Number	Testcase Name	Description	Coverage (CODE, FC, ASSERTION)
TC0.0	Testcase_reset	Reset DUT pins, read quiescent values on PKT_RX, PKT_TX, XGMII_TX, XGMII_RX, WB interfaces	ASSERTION: a_rst_xgmii_rx, xgmii_tx, packet_intf, wb
TC0.1	Testcase_wb_rst	After reset, read configuration registers via WB	FC : compare to hardwired expected values, or use RAL

PHASE 1 – BASIC TRANSACTIONS

Testcase Number	Testcase Name	Description	Coverage
TC1.0	Testcase_loopback	Drive packets from PKT_TX, read back on PKT_RX, compare using scoreboard	FC and ASSERTION : check for valid PKT_TX and PKT_RX transfers
TC1.1	Testcase_large_burst	Drive multiple long packets on PKT_TX to mimic streaming music	FC : packet_tx length greater than 1000
TC1.2	Testcase_many_small	Drive multiple short packets on PKT_TX to mimic client interface to services	FC : packet_tx length shorter than 10
TC1.3	Testcase_long_gap	Leave long gaps between packets	ASSERTION : count number of long gaps
TC0.4	Testcase_under		

PHASE 2 – RANDOM TRANSACTIONS

Testcase Number	Testcase Name	Description	Coverage
TC2.0	Testcase_random_pkt_tx	Drive random length and gap from PKT_TX	FC : cross length and gap to target all corners
TC2.1	Testcase_random_xgmii_rx	Drive random D & C from XGMII_RX	FC : XGMII_RXD & RXC
TC2.2	Testcase_random_reset	Random reset to test or document DUT resilience	ASSERTION : random assert reset to
TC2.3	Testcase_hit_codecoverage	By now, functionality is verified, but code coverage might not be 100%	CODE : achieve 100% or create exclustions
TC2.4	Testcase_packet_proto	Break packet_tx and/or packet_rx protocol	ASSERTION : packet_tx/rx checker

SIMULATION ENVIRONMENT

Navigating (in the UCSC-X network) to my top level directory:

```
jless — sf10274@VLSI04:~/AlbertLand/0UVMgetinfo/lab-project-10gEthernetMAC/sim/uvm — ssh -X sf10274@...
[sf10274@VLSI04 lab-project-10gEthernetMAC]$ pwd
/home/sf10274/AlbertLand/0UVMgetinfo/lab-project-10gEthernetMAC
[sf10274@VLSI04 lab-project-10gEthernetMAC]$ ls
doc  rtl  scripts  sim  testbench  testcases
[sf10274@VLSI04 lab-project-10gEthernetMAC]$ cd sim/uvm/
[sf10274@VLSI04 uvm]$ pwd
/home/sf10274/AlbertLand/0UVMgetinfo/lab-project-10gEthernetMAC/sim/uvm
[sf10274@VLSI04 uvm]$ ls
CLEAN        novas.conf      README   runuvm   simv.daidir   vcs.log
COMP         novas_dump.log  RUN      SETUP    simv.log      verdi_config_file
csrc         novas.fsdb      RUN.old  SIM      ucli.key      verdiLog
ENV_WORKED   novas.rc        runsim   simv     vc_hdrs.h
[sf10274@VLSI04 uvm]$
```

Compile : Note that all testcases are compiled in at the same time, so that testcase selection becomes a runtime option (no need to recompile to run a different testcase).

```
sim > uvm > $ COMP
vcs \
-full64 \
+vcs+lic+wait \
-sverilog  \
-l vcs.log \
-override_timescale=1ps/1ps \
-ntb_opts uvm-1.1 \
-debug_access+all \
-kdb \
../../rtl/verilog/*.v \
+incdir+../../rtl/include \
+incdir+../../testcases \
+incdir+../../testbench/uvm \
../../testbench/uvm/intf_rst.sv \
../../testbench/uvm/intf_wb.sv \
../../testbench/uvm/intf_pkt_tx.sv \
../../testbench/uvm/tb_top_xge_mac.v \
../../testbench/uvm/tb_prog.sv  \
../../testcases/testcase_base.sv \
../../testcases/testcase_wb.sv \
../../testcases/testcase_reset.sv  \
../../testcases/testcase_loopback.sv
```

Simulate & picking a test via “+UVM_TESTNAME”

```
sim > uvm > SIM
simv \
-l simv.log \
+UVM_VERBOSITY=HIGH \
+UVM_TESTNAME=testcase_reset

# +UVM_TESTNAME=testcase_loopback
```

Writing a new test (rough flow):

To quickly create a new test, follow these steps. Basically

1) create a new sequence by copying the existing “sequence_template.sv” as a starting point
2) create a new testcase by copying the existing “testcase_template.sv” as a starting point
3) in the new testcase, include the new sequence, then declare, and finally instantiate
4) edit COMP script to compile new testcase
5) edit SIM script to simulate with new testcase using +UVM_TESTNAE=testcase_new

```
jless — sf10274@VLSI04:~/AlbertLand/0UVMgetinfo/lab-project-10gEthernetMAC/sim/uvm — ssh -X sf10274@...
[sf10274@VLSI04 lab-project-10gEthernetMAC]$ pwd
/home/sf10274/AlbertLand/0UVMgetinfo/lab-project-10gEthernetMAC
[sf10274@VLSI04 lab-project-10gEthernetMAC]$ cd testbench/uvm/
[sf10274@VLSI04 uvm]$ cp sequence_template.sv sequence_new.sv
[sf10274@VLSI04 uvm]$ cd ../../testcases/
[sf10274@VLSI04 testcases]$ cp testcase_template.sv testcase_new.sv
[sf10274@VLSI04 testcases]$ cd ../sim/uvm
[sf10274@VLSI04 uvm]$ vi COMP
[sf10274@VLSI04 uvm]$ vi SIM
[sf10274@VLSI04 uvm]$
```

SIMULATION WAVEFORM AND IMPACT OF SEQUENCES

Sequences is the heart of stimulus generation. Here I created a random length packet, but constrain the first data to be a recognizable string such as “DEADCAFE” (the DRIVER for PKT_TX read the “data” field instead of the “data2” array.

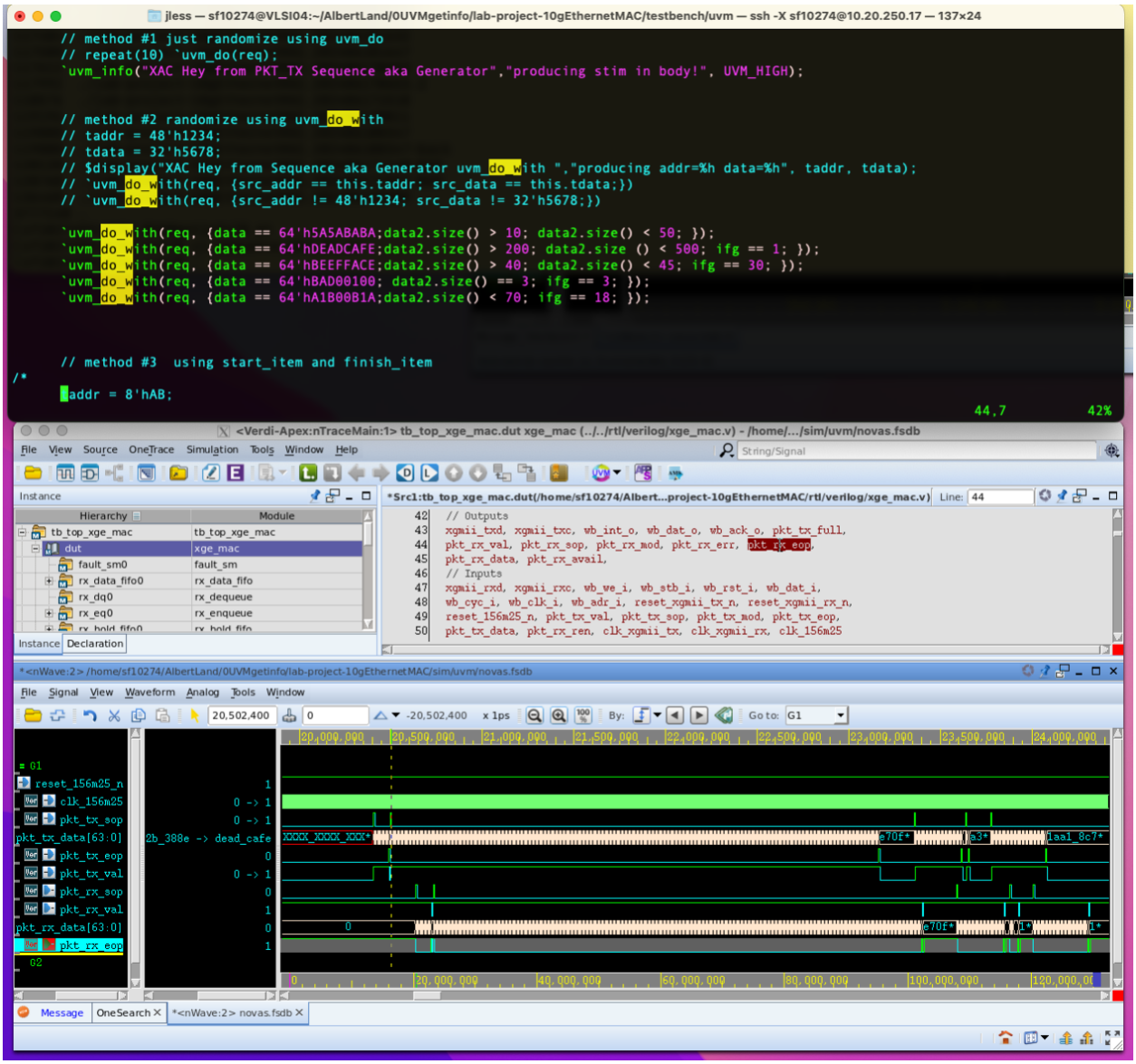


Figure 16: The 2nd packet starts with "0xDEADCAFE", as seen in both the SEQUENCE contraint and WAVEFORM

FINAL NOTES, CONCLUSION, & OPINION

In this paper, we examine how UVM can be used to exhaustively verify the functionality of a 10GE MAC. UVM provides an opinionated framework for new comers to spool up quickly, as well as old timers to know where to look exactly to augment an existing UVM verification environment. My opinion is that UVM is ideal for network and bus centric design – generate lots of interesting and random packets to stress test the network. But for traditional SOC tasks, where there is an orderly access of I/O (program the I/O, enable it, poll or wait for interrupt) or orderly access of memory, UVM is an overkill. From this project, in order to access the DUT registers via the WB interface, constrained fixed packets are created to "mimic" orderly WB bus traffic. And reading back the register to compare should not be via the scoreboard, but rather some sort of direct compare. Despite these drawbacks, the predictability that UVM brings to an otherwise potentially messy verification environment is worth the overhead.

Things that I might improve upon:

1) register testing: figure a better way out to perform simple WB read and write – not sure if RAL is an overkill? I currently have a sequence that performs the READ, but rely on the MON_WB to tell me what was picked, which makes self-checking more tedious (as opposed to just a CPU WRITE, READ, and compare)
2) Constraint debug: I set safety constraints in the SEQUENCE_ITEM, but sometimes they conflict with a testcase sequence constraint and I don't know discover this until much later on
3) Revision control: I basically make a copy of the entire directory and found myself using it a few times, but would like to use something more modern like Git. The UCSC-X seems to block the webs interface to Github.
4) Waveform viewing: Verdi was SLOW
5) Quick start: I think several attempts were made to allow new users to quick start a UVM testbench, but haven't seen much traction. Perhaps follow the example of Spring Inializr (https://start.spring.io/)?