

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	3
SYSTEM OVERVIEW & DESIGN UNDER TEST.....	4
SYSTEM OVERVIEW.....	4
DESIGN UNDER TEST: 10 GIGABIT ETHERNET MEDIA ACCESS CONTROLLER (MAC)	5
DUT BLOCK DIAGRAM & MAJOR DATA PATHS	5
<i>TX PATH : PACKET Transmit PATH interface (PKT_TX) to XGMII TRANSMIT (XGMII_TX)</i>	6
<i>RX PATH : XGMII RECEIVE (GXMII_RXD) to PACKET Receive PATH interface (PKT_RX)</i>	6
<i>Wishbone Path : register interface (WB)</i>	6
VERIFICATION ENVIRONMENT USING UNIVERSAL VERIFICATION METHODOLOGY (UVM)	7
HIGH LEVEL OVERVIEW OF UVM VERIFICATION ENVIRONMENT	7
TESTBENCH TOP : RTL, VIRTUAL INTERFACES, CLOCK GENERATOR	8
VIRTUAL INTERFACE	9
TEST PROGRAM : LAUNCH TESTCASE.....	11
TESTCASE : ENVIRONMENT, VIRTUAL SEQUENCER, VIRTUAL SEQUENCE	12
ENVIRONMENT : AGENT(S), SEQUENCE, COVERAGE, SCOREBOARD	16
AGENT : DRIVER, SEQUENCER, MONITOR, SEQUENCE_ITEM	17
<i>Driver</i>	17
<i>monitor</i>	18
SEQUENCE	19
SEQUENCE_ITEM.....	20
SCOREBOARD	21
COVERAGE.....	22
TESTPLAN – TYING TESTCASE TO COVERAGE	23
PHASE 0 – RESET & BASIC CONNECTIVITY.....	23
PHASE 1 – BASIC TRANSACTIONS	23
PHASE 2 – RANDOM TRANSACTIONS.....	23
SIMULATION ENVIRONMENT.....	24
SIMULATION WAVEFORM AND IMPACT OF SEQUENCES	27
FINAL NOTES, CONCLUSION, & OPINION	29

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 be 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 code 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.

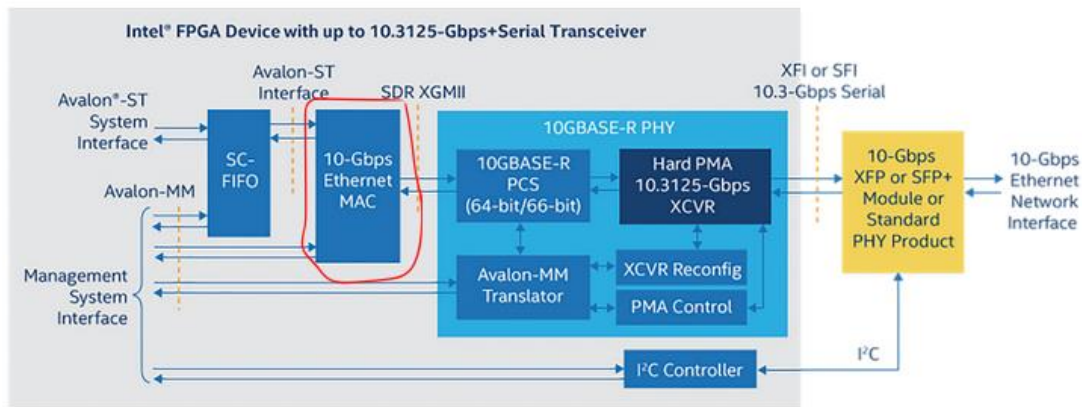
This paper will explore using UVM to verify OpenCores' 10GE MAC is open source in the Verilog language. It comes with a simple testbench to verify its functionality and is also written in the same language of Verilog. While using Verilog for both design and verification seems natural, the Verilog language 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 opinionated 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 built properly for the UVM environment to interact properly with the DUT.

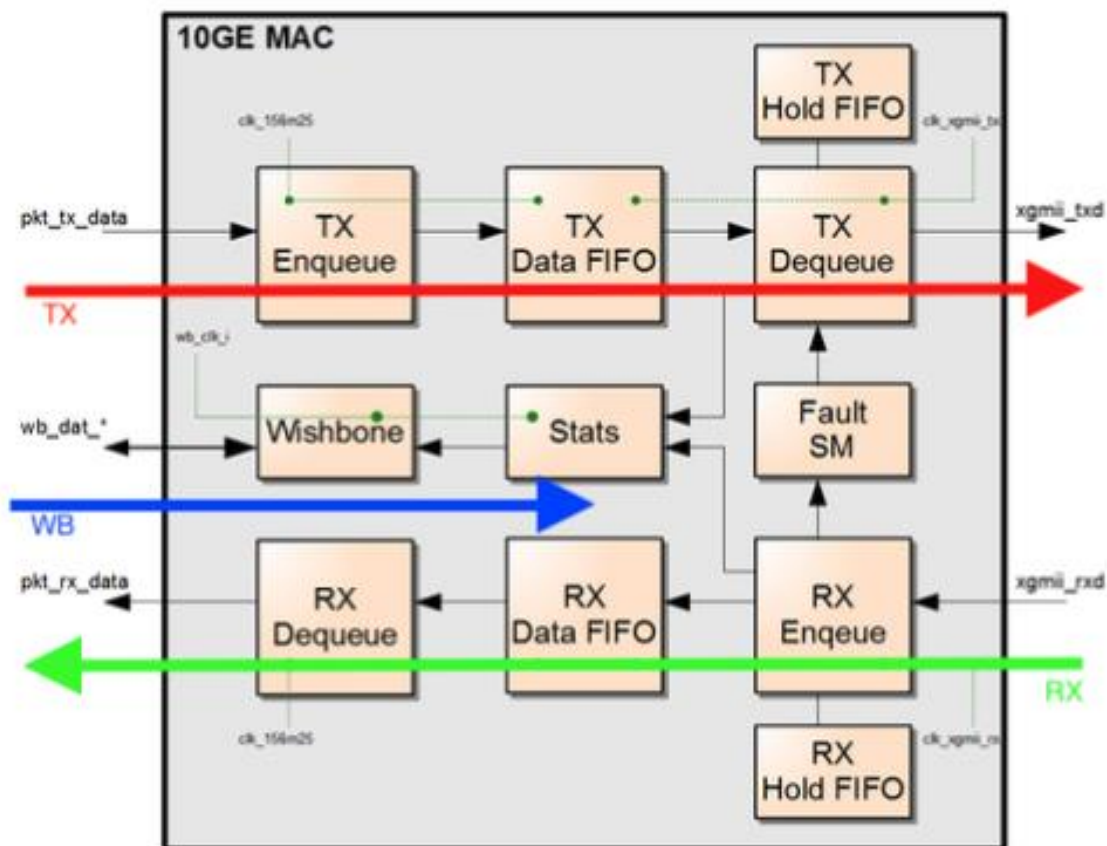


Figure 2: OpenCores 10GE MAC diagram. TX path is `pkt_tx_*` to `xgmii_tx`. RX path is `xgmii_rxd` to `pkt_rx_data`.

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 (XGMII_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_RX 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.

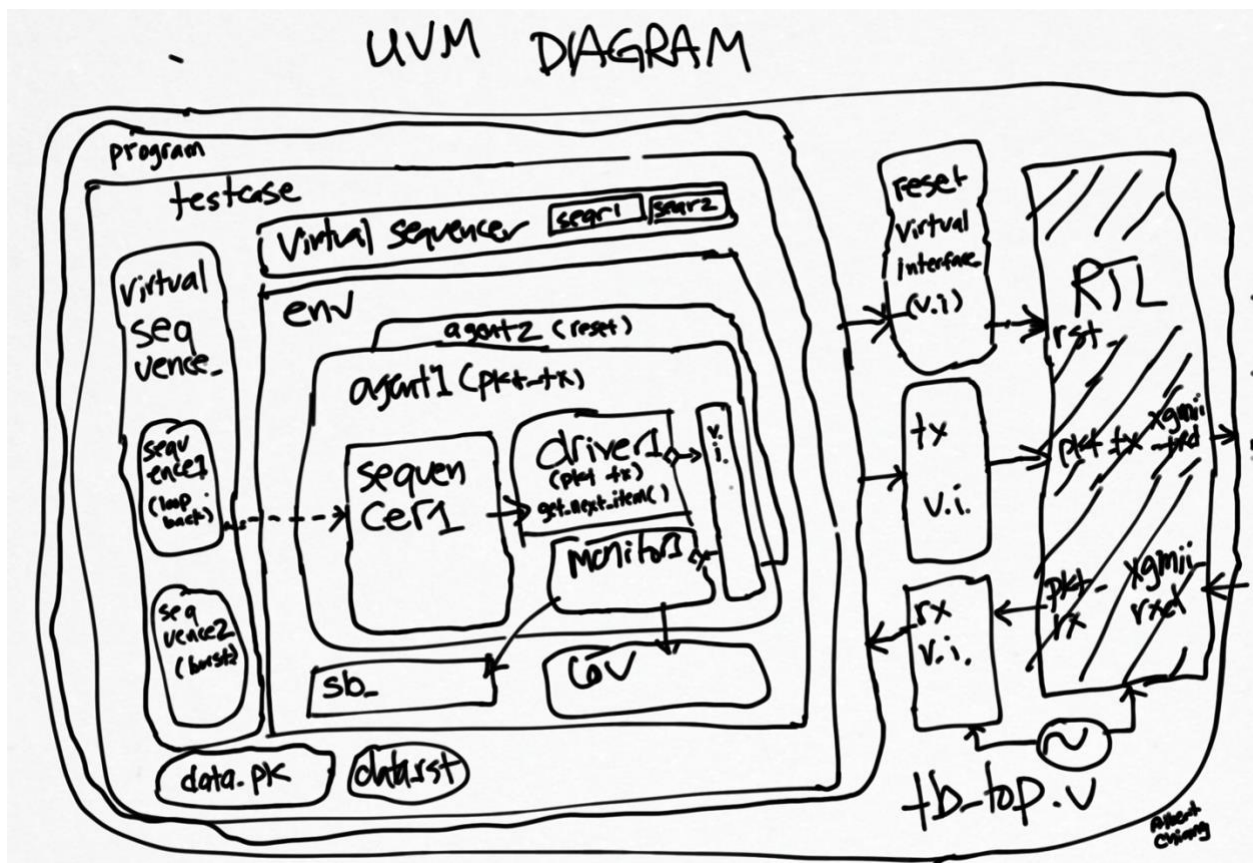


Figure 3. UVM Components Used in This Paper, drawn spaghetti style

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
1
2  module tb_top_xge_mac;
3
4
5  reg clk_156m25;
6
7  // virtual interfaces
8  intf_pkt_tx intf_pkt_tx_0(clk_156m25);
9  intf_rst intf_rst_0(clk_156m25);
10 intf_wb intf_wb_0(clk_wb);
11
12
13 // 10GE MAC DUT
14 xge_mac dut(
15     .pkt_tx_data (intf_pkt_tx_0.pkt_tx_data)
16 );
17
18 // clock generator
19 initial clk_156m25 = 0;
20 always #10 clk_156m25 = ~clk_156m25;
21
22
23 endmodule // tb_top_xge_mac
24
```


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
1  interface intf_pkt_tx( input bit clk); // clk_156m25);
2
3      // input
4      logic [63:0] pkt_tx_data;
5      logic      pkt_tx_eop;
6      logic [2:0] pkt_tx_mod;
7      logic      pkt_tx_sop;
8      logic      pkt_tx_val;
9      logic      reset_156m25_n;
10
11     // output
12     logic pkt_tx_full;
13
14
15     clocking cb_driver @(posedge clk);
16     output pkt_tx_data;
17     output pkt_tx_eop;
18     output pkt_tx_mod;
19     output pkt_tx_sop;
20     output pkt_tx_val;
21     input  pkt_tx_full;
22     endclocking
23
24 endinterface
25
```

Figure 4: Virtual interface for PKT_TX

```
testbench > uvm > ≡ intf_xgmii_tx.sv
1  interface intf_xgmii_tx ( input bit clk);
2
3      logic [63:0] xgmii_tx_txd;
4      logic [7:0] xgmii_tx_txc;
5
6
7      clocking cb_driver @(posedge clk);
8      xgmii_tx_txd;
9      xgmii_tx_txc;
10     endclocking
11
12 endinterface
13
```

Figure 5: Virtual interface for XGMII_TX

```

testbench > uvm > ≡ intf_wb.sv
1  interface intf_wb( input bit wb_clk);
2
3      // input to WB
4      logic [7:0] adr;
5      logic [31:0] dat_i;
6      logic we;
7      logic stb;
8      logic cyc;
9
10     // output from WB
11     logic [31:0] dat_o;
12     logic ack;
13     logic intr;
14
15     clocking cb_driver @(posedge wb_clk);
16     output adr;
17     output dat_i;
18     output we;
19     output stb;
20     output cyc;
21
22
23     input dat_o;
24     input ack;
25     input intr;
26
27     endclocking
28
29 endinterface
30

```

Figure 6: Virtual interface for Wishbone

```

testbench > uvm > ≡ intf_rst.sv
1  interface intf_rst( input bit clk_156m25);
2
3      logic reset_156m25_n;
4      logic wb_rst_i;
5      logic reset_xgmii_rx_n;
6      logic reset_xgmii_tx_n;
7
8      clocking cb_driver @(posedge clk_156m25);
9      output reset_156m25_n;
10     output wb_rst_i;
11     output reset_xgmii_rx_n;
12     output reset_xgmii_tx_n;
13     endclocking
14
15 endinterface
16

```

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_loopback.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 > testcases_base.vv
1  `ifndef _TESTCASE_BASE_
2  `define _TESTCASE_BASE_
3
4  import uvm_pkg::*;
5
6  `include "env.vv"
7
8  class testcase_base extends uvm_test;
9
10
11    `uvm_component_utils(testcase_base)
12
13
14    env env_0;
15
16    function new(string n, uvm_component p);
17    | super.new(n, p);
18    endfunction
19
20
21    virtual function void build_phase(uvm_phase phase) ;
22    | super.build_phase(phase) ;
23    | env_0 = env::type_id::create("env_0", this);
24    | uvm_config_db#(virtual intf_wb)::set(this, "env_0.agent_wb_0.monitor_wb_0", "vi", tb_top_xge_mac.intf
25    | uvm_config_db#(virtual intf_rst)::set(this, "env_0.agent_reset_0.driver_reset_0", "vi", tb_top_xge_m
26    | uvm_config_db#(virtual intf_pkt_tx)::set(this, "env_0.agent_pkt_tx_0.driver_pkt_tx_0", "vi", tb_top_
27    | uvm_config_db#(virtual intf_pkt_tx)::set(this, "env_0.agent_pkt_tx_0.monitor_pkt_tx_0", "vi", tb_top
28    | uvm_config_db#(virtual intf_pkt_rx)::set(this, "env_0.agent_pkt_rx_0.driver_pkt_rx_0", "vi", tb_top_
29    | uvm_config_db#(virtual intf_pkt_rx)::set(this, "env_0.agent_pkt_rx_0.monitor_pkt_rx_0", "vi", tb_top
30    endfunction
31
32
33    virtual function void end_of_elaboration_phase(uvm_phase phase);
34    | super.end_of_elaboration_phase(phase);
35    | uvm_top.print_topology();
36    | factory.print();
37    endfunction
38
39    virtual function void start_of_simulation_phase(input uvm_phase phase);
40    | super.start_of_simulation_phase(phase);
41    endfunction
42
43
44    virtual task run_phase(input uvm_phase phase);
45    | super.run_phase(phase);
46    endtask
47
48    virtual task main_phase(uvm_phase phase) ;
49    | super.main_phase(phase) ;
50    | // phase.raise_objection(this);
51    | // phase.drop_objection(this);
52    endtask
53
54    endclass
55    `endif // _TESTCASE_BASE_
56
```

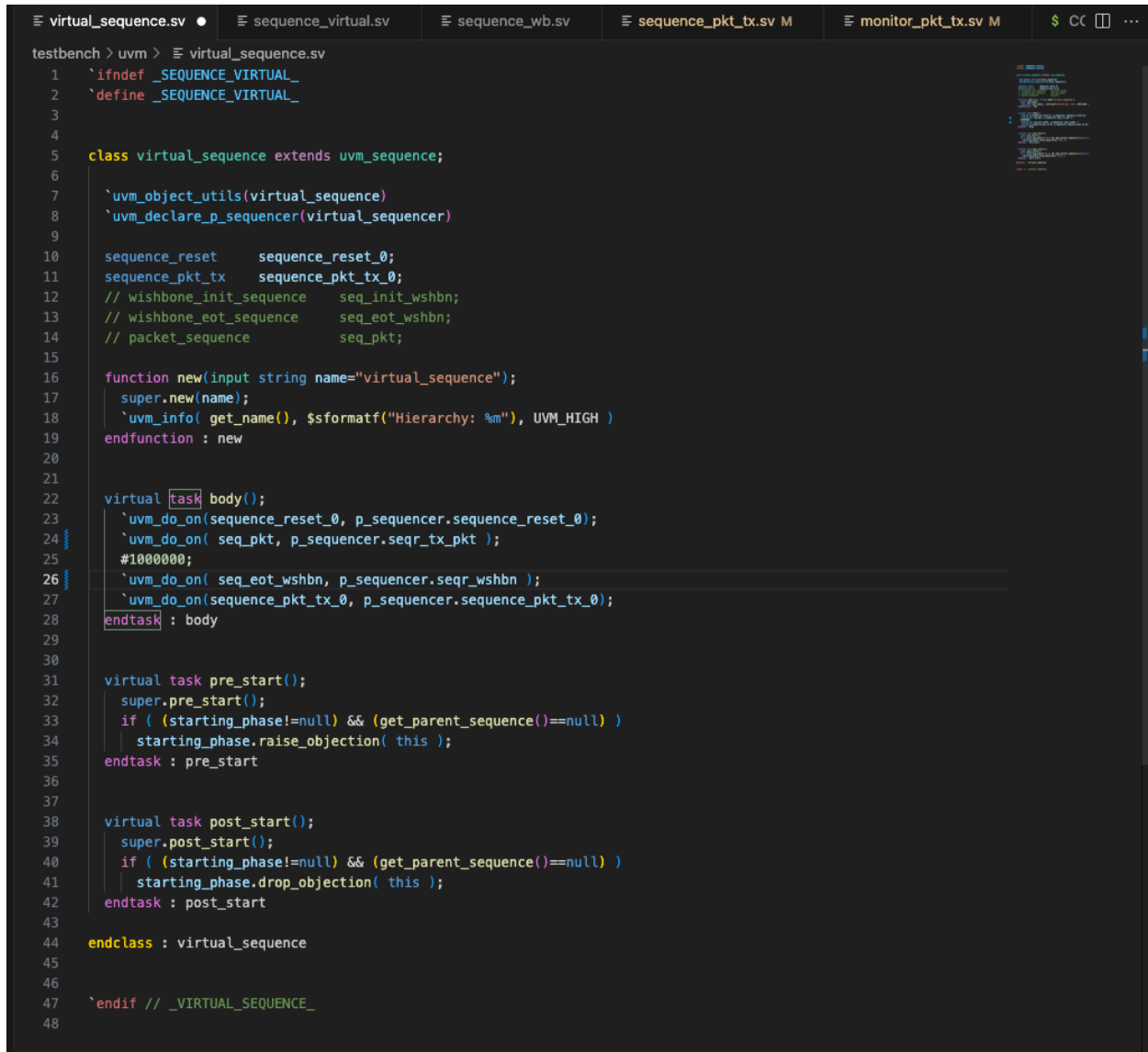
Figure 8: All testcases will derive from a testcase_base that does necessary setup such as virtual interface hookup in build_phase()

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

```
testcases > E testcase_loopback_reset.sv
1  import uvm_pkg::*;
2
3  `include "env.sv"
4  `include "sequence_pkt_tx.sv"
5  `include "virtual_sequencer.sv"
6  `include "virtual_sequence.sv"
7
8  class testcase_loopback_reset extends testcase_base;
9
10
11     `uvm_component_utils(testcase_loopback_reset)
12
13
14     // env env_0; // in testcase_base
15     sequence_reset sequence_reset_0;
16     sequence_pkt_tx sequence_pkt_tx_0;
17     virtual_sequencer virtual_sequencer_0;
18
19     function new(string n, uvm_component p);
20         super.new(n, p);
21     endfunction
22
23
24     virtual function void build_phase(uvm_phase phase);
25         super.build_phase(phase);
26         virtual_sequencer_0 = virtual_sequencer::type_id::create("virtual_sequencer_0", this);
27     endfunction
28
29     virtual function void connect_phase(uvm_phase phase);
30         super.connect_phase(phase);
31         virtual_sequencer_0.sequencer_pkt_tx_0 = env_0.agent_pkt_tx_0.sequencer_pkt_tx_0;
32         virtual_sequencer_0.sequencer_reset_0 = env_0.agent_reset_0.sequencer_reset_0;
33         uvm_config_db #(uvm_object_wrapper)::set(this, "virtual_sequencer_0.reset_phase", "default_sequence"
34     endfunction // connect_phase
35
36
37     virtual function void end_of_elaboration_phase(uvm_phase phase);
38         super.end_of_elaboration_phase(phase);
39         uvm_top.print_topology();
40         factory.print();
41     endfunction
42
43
44     virtual function void start_of_simulation_phase(input uvm_phase phase);
45         super.start_of_simulation_phase(phase);
46     endfunction
47
48
49     virtual task run_phase(input uvm_phase phase);
50         super.run_phase(phase);
51         sequence_pkt_tx_0 = sequence_pkt_tx::type_id::create("sequence_pkt_tx_0", this);
52         sequence_reset_0 = sequence_reset::type_id::create("sequence_reset_0", this);
53     endtask
54
55     virtual task main_phase(uvm_phase phase);
56
57         uvm_object obj;
58
59         super.main_phase(phase);
60
61         phase.raise_objection(this);
62         sequence_pkt_tx_0.start(virtual_sequencer_0.sequencer_pkt_tx_0);
63         sequence_reset_0.start(virtual_sequencer_0.sequencer_reset_0);
64         phase.drop_objection(this);
65
66         // from Benjamin's book
67         obj = phase.get_objection();
68         obj.set_drain_time(this, 90us); // works!
69         // 100us -> Time: 100617600 ps
70         // 90us -> Time: 90617600 ps
71
72
73
74     endtask
75
76 endclass
77
```

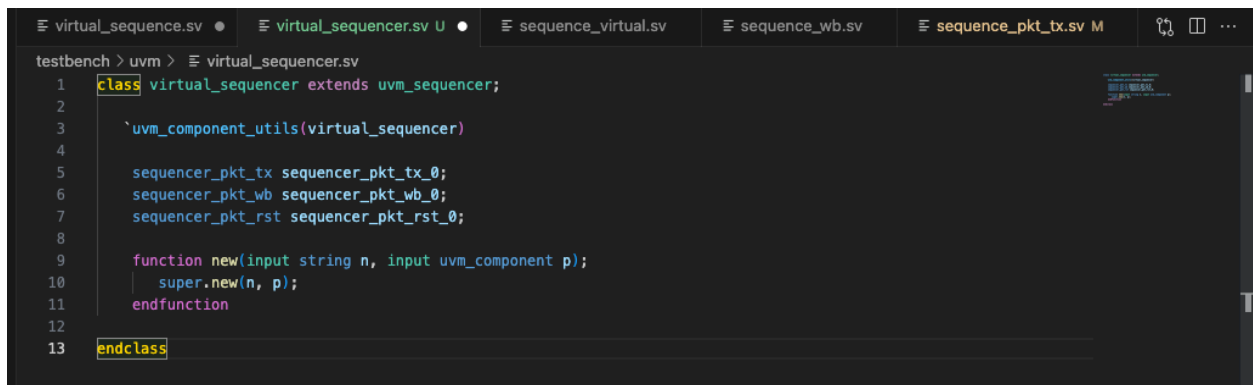
Figure 9: A testcase that encapsulates the environment, sequences

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



```
testbench > uvm > virtual_sequence.sv
1  `ifndef _SEQUENCE_VIRTUAL_
2  `define _SEQUENCE_VIRTUAL_
3
4
5  class virtual_sequence extends uvm_sequence;
6
7      `uvm_object_utils(virtual_sequence)
8      `uvm_declare_p_sequencer(virtual_sequencer)
9
10     sequence_reset      sequence_reset_0;
11     sequence_pkt_tx      sequence_pkt_tx_0;
12     // wishbone_init_sequence      seq_init_wshbn;
13     // wishbone_eot_sequence      seq_eot_wshbn;
14     // packet_sequence      seq_pkt;
15
16     function new(input string name="virtual_sequence");
17         super.new(name);
18         `uvm_info( get_name(), $sformatf("Hierarchy: %m"), UVM_HIGH )
19     endfunction : new
20
21
22     virtual task body();
23         `uvm_do_on(sequence_reset_0, p_sequencer.sequence_reset_0);
24         `uvm_do_on( seq_pkt, p_sequencer.seqr_tx_pkt );
25         #1000000;
26         `uvm_do_on( seq_eot_wshbn, p_sequencer.seqr_wshbn );
27         `uvm_do_on(sequence_pkt_tx_0, p_sequencer.sequence_pkt_tx_0);
28     endtask : body
29
30
31     virtual task pre_start();
32         super.pre_start();
33         if ( (starting_phase!=null) && (get_parent_sequence()==null) )
34             starting_phase.raise_objection( this );
35     endtask : pre_start
36
37
38     virtual task post_start();
39         super.post_start();
40         if ( (starting_phase!=null) && (get_parent_sequence()==null) )
41             starting_phase.drop_objection( this );
42     endtask : post_start
43
44 endclass : virtual_sequence
45
46
47 `endif // _VIRTUAL_SEQUENCE_
48
```

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



The image shows a screenshot of a Verilog code editor with a dark theme. At the top, there is a tab bar with five tabs: 'virtual_sequence.sv', 'virtual_sequencer.sv' (which is active and highlighted with a green dot), 'sequence_virtual.sv', 'sequence_wb.sv', and 'sequence_pkt_tx.sv'. Below the tabs, the editor displays the code for 'virtual_sequencer.sv'. The code starts with a package declaration 'package uvm;' followed by an import statement 'import uvm_sequencer.*;'. A comment '// virtual sequencer' is present. The code then defines a class 'virtual_sequencer' that extends 'uvm_sequencer'. Inside the class, there is a local task 'uvm_component_utils(virtual_sequencer)' which instantiates three objects: 'sequencer_pkt_tx sequencer_pkt_tx_0;', 'sequencer_pkt_wb sequencer_pkt_wb_0;', and 'sequencer_pkt_rst sequencer_pkt_rst_0;'. A 'function new' is defined to call 'super.new(n, p);'. The class ends with 'endclass'. Line numbers 1 through 13 are visible on the left side of the code.

```
testbench > uvm > virtual_sequencer.sv
1  class virtual_sequencer extends uvm_sequencer;
2
3      `uvm_component_utils(virtual_sequencer)
4
5      sequencer_pkt_tx sequencer_pkt_tx_0;
6      sequencer_pkt_wb sequencer_pkt_wb_0;
7      sequencer_pkt_rst sequencer_pkt_rst_0;
8
9      function new(input string n, input uvm_component p);
10         super.new(n, p);
11     endfunction
12
13 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
1  `ifndef _DRIVER_RESET_
2  `define _DRIVER_RESET_
3
4  class driver_reset extends uvm_driver #(data_reset);
5
6      `uvm_component_utils(driver_reset)
7
8
9      virtual intf_rst vi;
10
11      function new(input string name="Driver for reset", input uvm_component parent);
12          super.new(name, parent);
13      endfunction
14
15      virtual function void build_phase(input uvm_phase phase);
16          super.build_phase(phase);
17          uvm_config_db#(virtual intf_rst)::get(this, "", "vi", vi);
18          if (vi == null) begin
19              `uvm_fatal("Driver for reset ", "VI for DRIVER not in CONFIG_DB");
20          end
21      endfunction
22
23      virtual task run_phase(input uvm_phase phase);
24
25          `uvm_info("DRIVER CLASS", "HIERARCHY: %m", UVM_HIGH);
26
27          forever begin
28
29              seq_item_port.get_next_item(req);
30
31              `uvm_info("XAC RESET DRIVER run_phase received this packet ", req.sprint(), UVM_HIGH);
32
33              vi.wb_rst_i <= req.reset_;
34              vi.reset_xgmii_rx_n <= req.reset_;
35              vi.reset_xgmii_tx_n <= req.reset_;
36              vi.reset_156m25_n <= req.reset_;
37
38              repeat(req.cycles) @(posedge vi.clk_156m25);
39              vi.wb_rst_i <= ~req.reset_;
40              vi.reset_xgmii_rx_n <= ~req.reset_;
41              vi.reset_xgmii_tx_n <= ~req.reset_;
42              vi.reset_156m25_n <= ~req.reset_; /
43
44              seq_item_port.item_done();
45          end // forever
46      endtask
47
48
49  endclass
50
51  `endif // _DRIVER_RESET_
52
```

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
1 `ifndef _MONITOR_PACKET_TX_MONITOR_
2 `define _MONITOR_PACKET_TX_MONITOR_
3
4
5 class monitor_pkt_tx extends uvm_monitor;
6
7     virtual intf_pkt_tx    vi;
8     int unsigned          m_num_captured;
9     uvm_analysis_port #(data_pkt_tx) ap_tx_mon;
10
11     `uvm_component_utils( monitor_pkt_tx )
12
13     function new(input string name="monitor_pkt_tx", input uvm_component parent);
14         super.new(name, parent);
15     endfunction : new
16
17
18     virtual function void build_phase(input uvm_phase phase);
19         super.build_phase(phase);
20         m_num_captured = 0;
21         ap_tx_mon = new ( "ap_tx_mon", this );
22         uvm_config_db#(virtual intf_pkt_tx)::get(this, "", "vi", vi);
23         if ( vi==null )
24             `uvm_fatal(get_name(), "Virtual Interface for monitor not set!");
25     endfunction : build_phase
26
27
28     virtual task run_phase(input uvm_phase phase);
29         data_pkt_tx    tmp_rcv_pkt;
30         bit [7:0]      q_rx_data[$];
31         int            idx;
32
33         `uvm_info( get_name(), $sformatf("HIERARCHY: %m"), UVM_HIGH);
34
35
36         forever begin
37
38
39             @(vi.clk) // @(vi.mon_cb)
40             if ( vi.pkt_rx_val ) begin
41                 // if ( vi.pkt_rx_sop && !vi.pkt_rx_eop && pkt_in_progress==0 ) begin
42                 tmp_rcv_pkt = data_pkt_rx::type_id::create("tmp_rcv_pkt", this);
43
44                 if ( vi.pkt_rx_sop && !vi.pkt_rx_eop ) begin
45                     $display("XAC PKT_RX_MON sop : %b", vi.pkt_rx_sop);
46                     $display("XAC PKT_RX_MON sop : %b", vi.pkt_rx_sop);
47                     $display("XAC PKT_RX_MON data : %h", vi.pkt_rx_data);
48                     tmp_rcv_pkt.data = vi.pkt_rx_data;
49                     `uvm_info("XAC Hey from PKT_RX MON ", "SEE THIS", UVM_HIGH);
50                     // `uvm_info("XAC PKT RX MON run_phase received this packet ", req.sprint(), UVM_HIGH);
51
52                     end // vi.pkt_rx_sop && !vi.pkt_rx_eop ) begin
53                     end // if vi.pkt_rx_val
54                 end // forever begin
55                 ap_rx_mon.write(tmp_rcv_pkt);
56             endtask : run_phase
57
58
59             function void report_phase( uvm_phase phase );
60                 `uvm_info( get_name( ), $sformatf( "REPORT: Captured %0d packets", m_num_captured ), UVM_LOW )
61             endfunction : report_phase
62
63     endclass : monitor_pkt_tx
64
65 `endif
66
67
```

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.

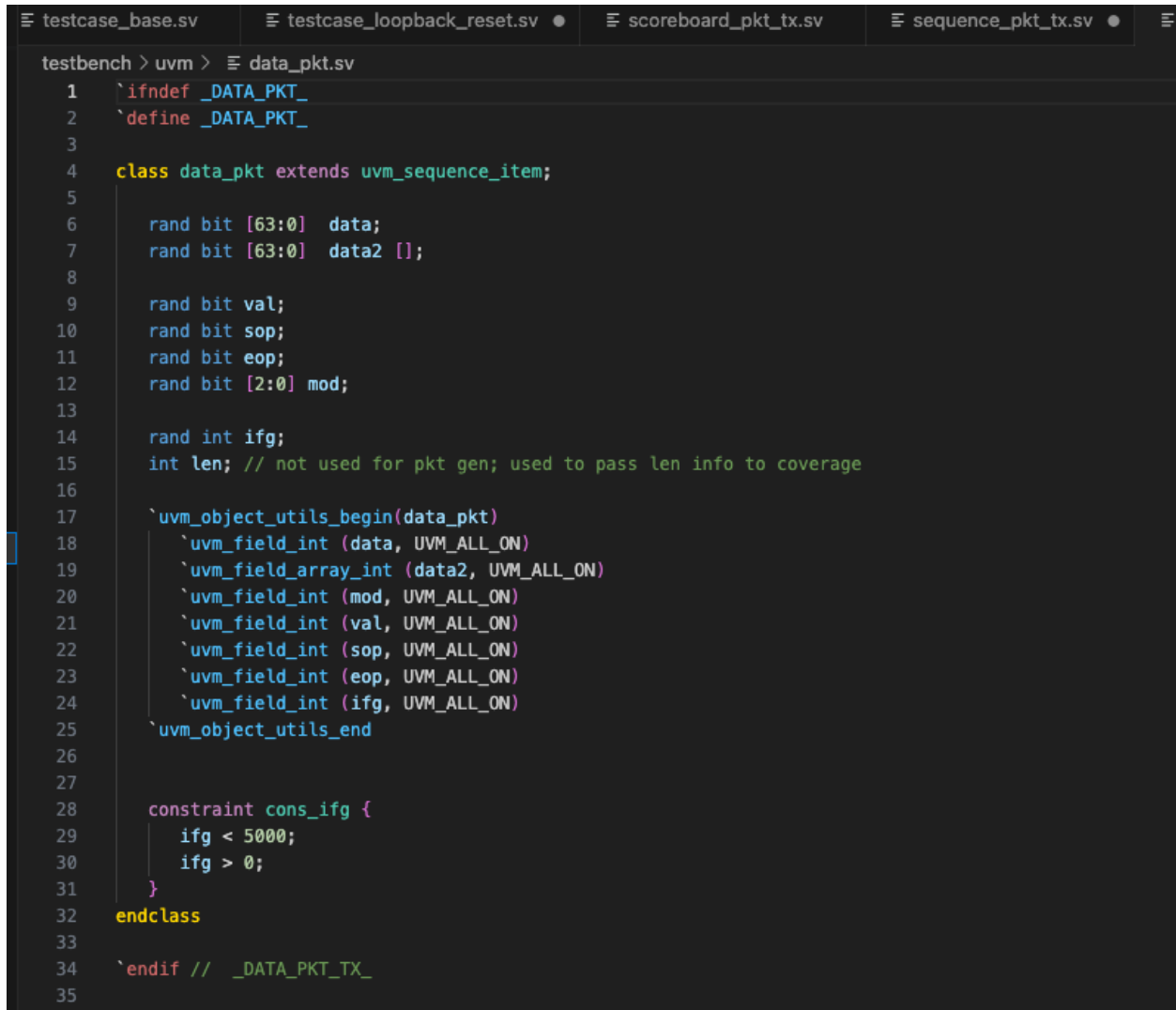
```
testbench > uvm > sequence_pkt_tx.sv
1  `ifndef _SEQUENCNE_PKT_TX_
2  `define _SEQUENCNE_PKT_TX_
3  `include "data_pkt.sv"
4
5  class sequence_pkt_tx extends uvm_sequence#(data_pkt);
6
7
8
9  `uvm_object_utils(sequence_pkt_tx)
10
11  virtual task body();
12
13      reg [7:0] taddr;
14      reg [31:0] tdata;
15
16
17
18      // method #1 just randomize using uvm_do
19      repeat(10) `uvm_do(req);
20
21
22      // method #2 randomize using uvm_do_with
23
24
25      // `uvm_do_with(req, {src_addr != 48'h1234; src_data != 32'h5678;})
26      `uvm_do_with(req, {data == 64'h5A5ABABA; data2.size() > 10; data2.size() < 50; });
27      `uvm_do_with(req, {data == 64'hDEADCAFE; data2.size() > 200; data2.size() < 500; ifg == 1; });
28      `uvm_do_with(req, {data == 64'hBEEFFACE; data2.size() > 40; data2.size() < 45; ifg == 30; });
29      `uvm_do_with(req, {data == 64'hBAD00100; data2.size() == 3; ifg == 3; });
30      `uvm_do_with(req, {data == 64'hA1B00B1A; data2.size() < 70; ifg == 18; });
31
32
33
34      // method #3 using start_item and finish_item
35
36      taddr = 8'hAB;
37      tdata = 32'h00112233;
38      $display("XAC Hey from Sequence aka Generator start_item ", "producing addr=%h data=%h", taddr, tdata);
39      req = data_pkt::type_id::create("req");
40      start_item(req);
41      req.adr = taddr;
42      req.dat_i = tdata;
43      req.we = 1;
44      req.cyc = 1;
45      req.stb = 1;
46      finish_item(req);
47
48      finish_item(req);
49
50
51
52  endtask
53 endclass
54 `endif // _SEQUENCNE_PKT_TX_
55
```

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

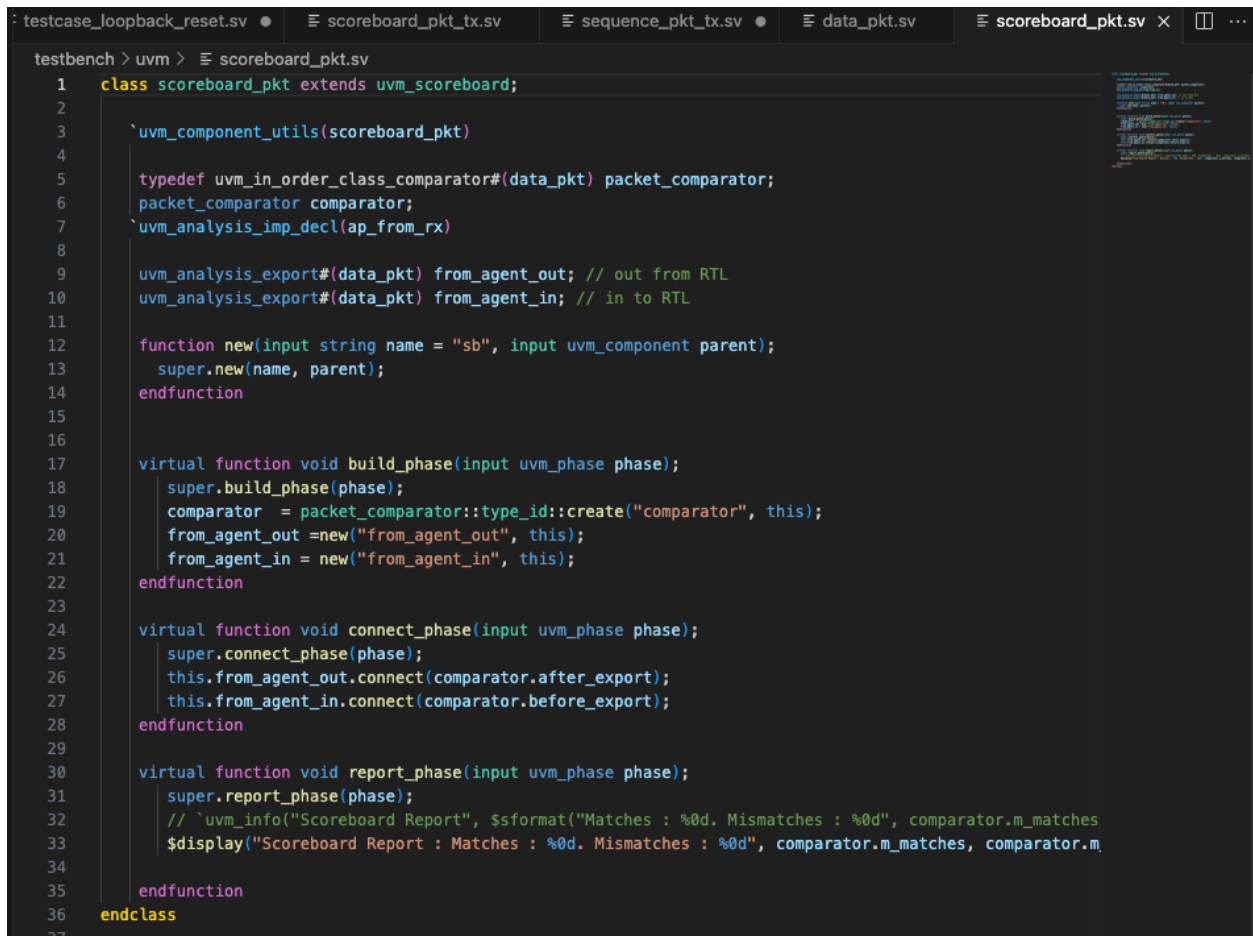


```
testbench > uvm > data_pkt.sv
1  `ifndef _DATA_PKT_
2  `define _DATA_PKT_
3
4  class data_pkt extends uvm_sequence_item;
5
6      rand bit [63:0] data;
7      rand bit [63:0] data2 [];
8
9      rand bit val;
10     rand bit sop;
11     rand bit eop;
12     rand bit [2:0] mod;
13
14     rand int ifg;
15     int len; // not used for pkt gen; used to pass len info to coverage
16
17     `uvm_object_utils_begin(data_pkt)
18         `uvm_field_int (data, UVM_ALL_ON)
19         `uvm_field_array_int (data2, UVM_ALL_ON)
20         `uvm_field_int (mod, UVM_ALL_ON)
21         `uvm_field_int (val, UVM_ALL_ON)
22         `uvm_field_int (sop, UVM_ALL_ON)
23         `uvm_field_int (eop, UVM_ALL_ON)
24         `uvm_field_int (ifg, UVM_ALL_ON)
25     `uvm_object_utils_end
26
27
28     constraint cons_ifg {
29         ifg < 5000;
30         ifg > 0;
31     }
32 endclass
33
34 `endif // _DATA_PKT_TX_
35
```

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.

The image shows a code editor window with a dark theme. At the top, there are several tabs: 'testcase_loopback_reset.sv', 'scoreboard_pkt_tx.sv', 'sequence_pkt_tx.sv', 'data_pkt.sv', and 'scoreboard_pkt.sv'. The 'scoreboard_pkt.sv' tab is active. The code is a SystemVerilog class definition for 'scoreboard_pkt' which extends 'uvm_scoreboard'. It includes a 'uvm_component_utils' block, a typedef for 'packet_comparator', and several virtual functions: 'new', 'build_phase', 'connect_phase', and 'report_phase'. The 'report_phase' function includes a commented-out line for 'uvm_info' and a 'display' statement for the scoreboard report. The code is numbered from 1 to 37 on the left margin.

```
testbench > uvm > scoreboard_pkt.sv
1  class scoreboard_pkt extends uvm_scoreboard;
2
3      `uvm_component_utils(scoreboard_pkt)
4
5      typedef uvm_in_order_class_comparator#(data_pkt) packet_comparator;
6      packet_comparator comparator;
7      `uvm_analysis_imp_decl(ap_from_rx)
8
9      uvm_analysis_export#(data_pkt) from_agent_out; // out from RTL
10     uvm_analysis_export#(data_pkt) from_agent_in; // in to RTL
11
12     function new(input string name = "sb", input uvm_component parent);
13         super.new(name, parent);
14     endfunction
15
16
17     virtual function void build_phase(input uvm_phase phase);
18         super.build_phase(phase);
19         comparator = packet_comparator::type_id::create("comparator", this);
20         from_agent_out = new("from_agent_out", this);
21         from_agent_in = new("from_agent_in", this);
22     endfunction
23
24     virtual function void connect_phase(input uvm_phase phase);
25         super.connect_phase(phase);
26         this.from_agent_out.connect(comparator.after_export);
27         this.from_agent_in.connect(comparator.before_export);
28     endfunction
29
30     virtual function void report_phase(input uvm_phase phase);
31         super.report_phase(phase);
32         // `uvm_info("Scoreboard Report", $sformat("Matches : %0d. Mismatches : %0d", comparator.m_matches
33         $display("Scoreboard Report : Matches : %0d. Mismatches : %0d", comparator.m_matches, comparator.m
34
35     endfunction
36 endclass
37
```

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 ideal 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
1  class coverage_pkt_tx extends uvm_subscriber#(data_pkt);
2
3      `uvm_component_utils(coverage_pkt_tx )
4
5      data_pkt data_pkt_to_cov;
6
7      covergroup cg_pkt_tx;
8          ifg : coverpoint data_pkt_to_cov.ifg {
9              bins bin_low_ifg = {[1:9]};
10             bins bin_high_ifg = {[10:999]};
11         }
12         len : coverpoint data_pkt_to_cov.len {
13             bins bin_low_len = {[1:99]};
14             bins bin_high_ifg = {[100:3999]};
15         }
16
17     endgroup
18
19     function new(input string name = "coverage_pkt_tx ", input uvm_component parent);
20         super.new(name, parent);
21         cg_pkt_tx = new();
22     endfunction
23
24     function void build_phase(input uvm_phase phase);
25         data_pkt_to_cov = new();
26     endfunction // build_phase(input uvm_phase phase);
27
28     // function void write(input data_pkt pkt_from_agent_out);
29     function void write(input data_pkt t);
30         // this.data_pkt_to_cov = pkt_from_agent_out;
31         this.data_pkt_to_cov = t;
32         cg_pkt_tx.sample();
33     endfunction // write(input data_pkt pkt_from_agent_out);
34 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 exclusions
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     verilog  
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
1  vcs \
2  -full64 \
3  +vcs+lic+wait \
4  -sverilog \
5  -l vcs.log \
6  -override_timescale=1ps/1ps \
7  -ntb_opts uvm-1.1 \
8  -debug_access+all \
9  -kdb \
10 ../../rtl/verilog/*.v \
11 +incdir+../../rtl/include \
12 +incdir+../../testcases \
13 +incdir+../../testbench/uvm \
14 ../../testbench/uvm/intf_rst.sv \
15 ../../testbench/uvm/intf_wb.sv \
16 ../../testbench/uvm/intf_pkt_tx.sv \
17 ../../testbench/uvm/tb_top_xge_mac.v \
18 ../../testbench/uvm/tb_prog.sv \
19 ../../testcases/testcase_base.sv \
20 ../../testcases/testcase_wb.sv \
21 ../../testcases/testcase_reset.sv \
22 ../../testcases/testcase_loopback.sv
23
```

Simulate & picking a test via "+UVM_TESTNAME"

```
sim > uvm > ≡ SIM
1  simv \
2  -l simv.log \
3  +UVM_VERBOSITY=HIGH \
4  +UVM_TESTNAME=testcase_reset
5
6  # +UVM_TESTNAME=testcase_loopback
7
```

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]$
```

A TYPICAL SIMULATION

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).

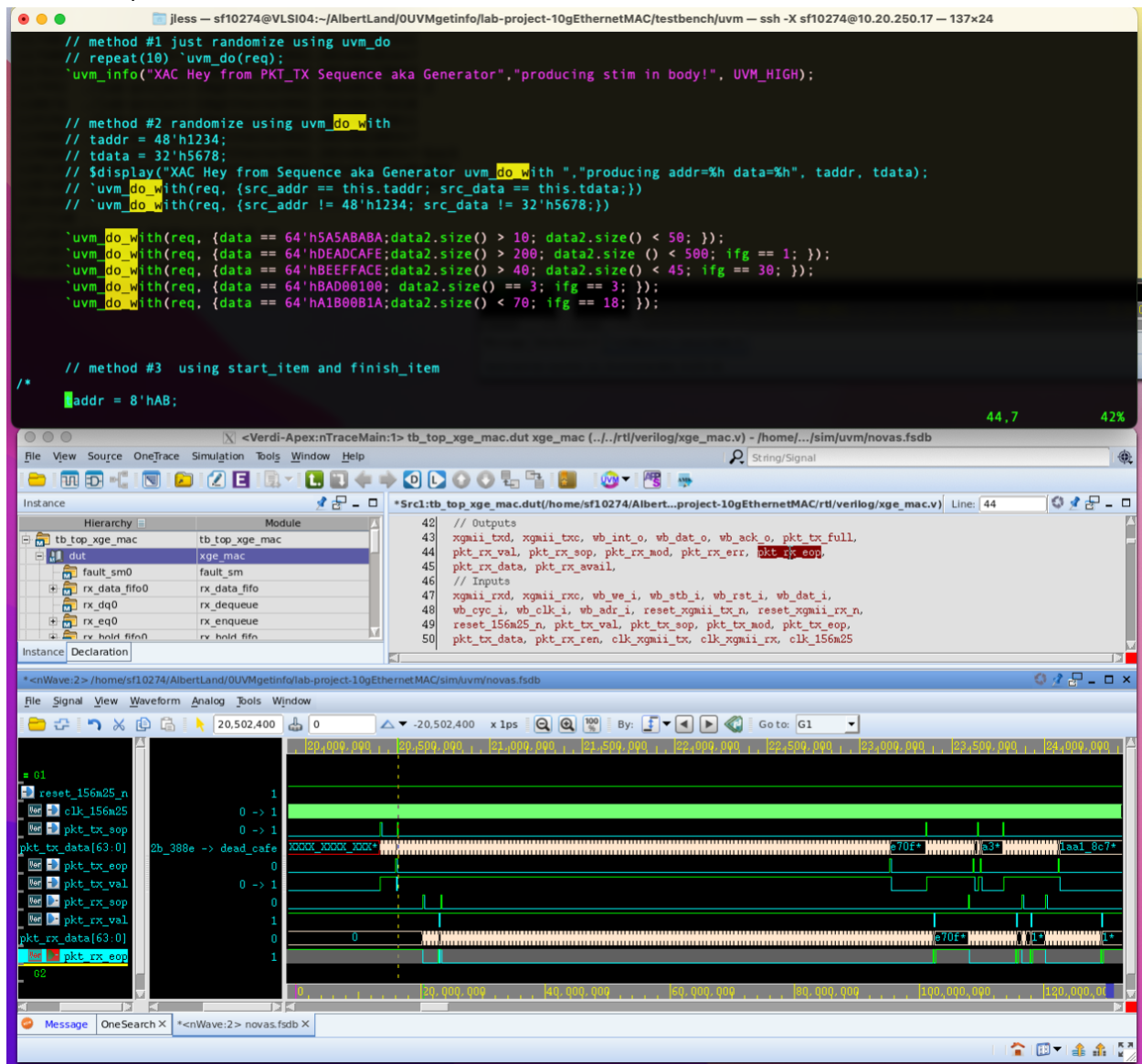


Figure 16: The 2nd packet starts with "0xDEADCAFE", as seen in both the SEQUENCE constraint and WAVEFORM

UVM TOPOLOGY

```

(C) 1996 - 2023 by Synopsys, Inc.
*Verdi* : Create FSDb file 'novas.fsd'
*Verdi* : Begin traversing the scopes, layer (0).
*Verdi* : End of traversing.
XAC hello from tb_prog
UVM_INFO @ 0: reporter [RNTST] Running test testcase_loopback_reset...
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
-----
Name                                     Type                                     Size  Value
-----
uvm_test_top                             testcase_loopback_reset                  -    @460
env_0                                     env                                       -    @468
agent_pkt_rx_0                           agent_pkt_rx                             -    @639
analysis_port                             uvm_analysis_port                       -    @807
monitor_pkt_rx_0                         monitor_pkt_rx                           -    @799
ap                                         uvm_analysis_port                       -    @816
sequencer_pkt_rx_0                       uvm_sequencer                           -    @676
rsp_export                               uvm_analysis_export                     -    @684
seq_item_export                           uvm_seq_item_pull_imp                   -    @790
arbitration_queue                        array                                    0      -
lock_queue                               array                                    0      -
num_last_reqs                             integral                                32    'd1
num_last_rsps                             integral                                32    'd1
agent_pkt_tx_0                           agent_pkt_tx                             -    @631
analysis_port                             uvm_analysis_port                       -    @984
driver_pkt_tx_0                           driver_pkt_tx                             -    @958
rsp_port                                  uvm_analysis_port                       -    @975
seq_item_port                             uvm_seq_item_pull_port                 -    @966
monitor_pkt_tx_0                         monitor_pkt_tx                           -    @993
ap                                         uvm_analysis_port                       -    @1002
sequencer_pkt_tx_0                       uvm_sequencer                           -    @835
rsp_export                               uvm_analysis_export                     -    @843
seq_item_export                           uvm_seq_item_pull_imp                   -    @949
arbitration_queue                        array                                    0      -
lock_queue                               array                                    0      -
num_last_reqs                             integral                                32    'd1
num_last_rsps                             integral                                32    'd1
agent_reset_0                             agent_reset                             -    @615
driver_reset_0                           driver_reset                             -    @1144
rsp_port                                  uvm_analysis_port                       -    @1161
seq_item_port                             uvm_seq_item_pull_port                 -    @1152
sequencer_reset_0                       uvm_sequencer                           -    @1021
rsp_export                               uvm_analysis_export                     -    @1029
seq_item_export                           uvm_seq_item_pull_imp                   -    @1135
arbitration_queue                        array                                    0      -
lock_queue                               array                                    0      -
num_last_reqs                             integral                                32    'd1
num_last_rsps                             integral                                32    'd1
agent_wb_0                               agent_wb                                 -    @623
driver_wb_0                             driver_wb                                 -    @1303
rsp_port                                  uvm_analysis_port                       -    @1320
seq_item_port                             uvm_seq_item_pull_port                 -    @1311
monitor_wb_0                             monitor_wb                               -    @1329
ap                                         uvm_analysis_port                       -    @1338
sequencer_wb_0                           uvm_sequencer                           -    @1180
rsp_export                               uvm_analysis_export                     -    @1188
seq_item_export                           uvm_seq_item_pull_imp                   -    @1294
arbitration_queue                        array                                    0      -
lock_queue                               array                                    0      -
num_last_reqs                             integral                                32    'd1
num_last_rsps                             integral                                32    'd1
coverage_pkt_tx_0                       coverage_pkt_tx                           -    @655
analysis_imp                             uvm_analysis_imp                       -    @663
scoreboard_pkt_0                       scoreboard_pkt                           -    @647
comparator                               uvm_in_order_class_comparator #(T)      -    @1357
after                                     uvm_tlm_analysis_fifo #(T)             -    @1445
analysis_export                           uvm_analysis_imp                       -    @1489
get_ap                                    uvm_analysis_port                       -    @1480
get_peek_export                           uvm_get_peek_imp                       -    @1462
put_ap                                    uvm_analysis_port                       -    @1471
put_export                                uvm_put_imp                             -    @1453
after_export                              uvm_analysis_export                     -    @1374
before                                    uvm_tlm_analysis_fifo #(T)             -    @1392
analysis_export                           uvm_analysis_imp                       -    @1436
get_ap                                    uvm_analysis_port                       -    @1427
get_peek_export                           uvm_get_peek_imp                       -    @1409
put_ap                                    uvm_analysis_port                       -    @1418
put_export                                uvm_put_imp                             -    @1400
before_export                              uvm_analysis_export                     -    @1365
pair_ap                                    uvm_analysis_port                       -    @1383
from_agent_in                             uvm_analysis_export                     -    @1507
from_agent_out                             uvm_analysis_export                     -    @1498
virtual_sequencer_0                       virtual_sequencer                       -    @492
rsp_export                               uvm_analysis_export                     -    @500
seq_item_export                           uvm_seq_item_pull_imp                   -    @606
arbitration_queue                        array                                    0      -
lock_queue                               array                                    0      -
num_last_reqs                             integral                                32    'd1
num_last_rsps                             integral                                32    'd1
-----

```

Figure 17: A UVM function to show the components in my UVM environment

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 after a packet transfer (say to reads number of packets transferred) – 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, 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 – so used lots of \$display / uvm_info instead
- 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/>)?