Universal Verification Method Testbench

Design and creation of a re-usable UVM testbench

TOM DIEDEREN

Revision History

Date	Version	Comments
May 2023	0.5	Structured document and added rudimentary descriptions
June 2023	1.0	Initial Release

LIST OF FIGURES

ACRONYMS

Acronym	Unabbreviated	Description
BCD	Binary Coded Decimal	Number representation format which uses 4 bits per base 10 digit.
DUT	Device Under Test	The design being verified with the testbench.
FPGA	Field Programmable Gate Array	Circuit containing reconfigurable logic which allows for redesigning/updating hardware after point of sale ("in the field").
HDL	Hardware Description Layer	Code in this layer represents hardware (synthesizable)
HVL	Hardware Verification Layer	Code in this layer is used for verification (not synthesizable)
IC	Integrated Circuit	An electronic circuit on a single piece of Si (also informally called "chip")
OOP	Object Oriented Programming	Programming paradigm.
PCB	Printed Circuit Board	Printed Interconnect for electrical components.
RTL	Register Transfer Layer	An abstraction layer of digital or mixed signal design.
Si	Silicon	Semiconductor. Element 14 on the periodic table.
TCL	Tool Command Language	Programming Language
UVM	Universal Verification Method	A standardized framework of SW classes enabling more reuse and standardized design verification.

SUMMARY

In order to get familiar with the UVM, I created a basic testbench that can be re-used for and improved upon during more elaborate projects. To fully focus on the UVM and testbench design, I decided to pick a simple DUT: a binary to BCD converter from a demo FPGA project, [1].

This document describes the testbench architecture, System Verilog code, and shows the final verification output from the simulator. The final chapter lists some lessons learned as well as suggested future improvements. The code can be found on my [Github](#) as well.

1 SPECIFICATION & REQUIREMENTS

The testbench must have the following classes:

- a test
- an environment
- a sequence item
- a sequence
- an agent containing a:
    - monitor
    - driver
    - scoreboard
- Configuration objects for the environment and agent

2 ARCHITECTURE

The test architecture is similar to the block level testbench architecture as described in the verification cookbook from Mentor/Siemens [2]. Because the DUT is simpler than the one used in the book, the overall architecture, most notably the BFM's and configuration objects, can be simplified. The high-level architecture I designed after reading the first couple of chapter of the book is shown in Figure 1. A full page view can be found in Appendix I.

It contains two top layers: one hardware one, hdl top, and a verification one, hvl top. The hdl top layer contains the RTL (Verilog) for the DUT and the interface. The hvl layer starts the UVM test which is the first component created for the testbench. The hierarchy of the paragraphs describing the hvl match the hierarchy of the actual testbench.
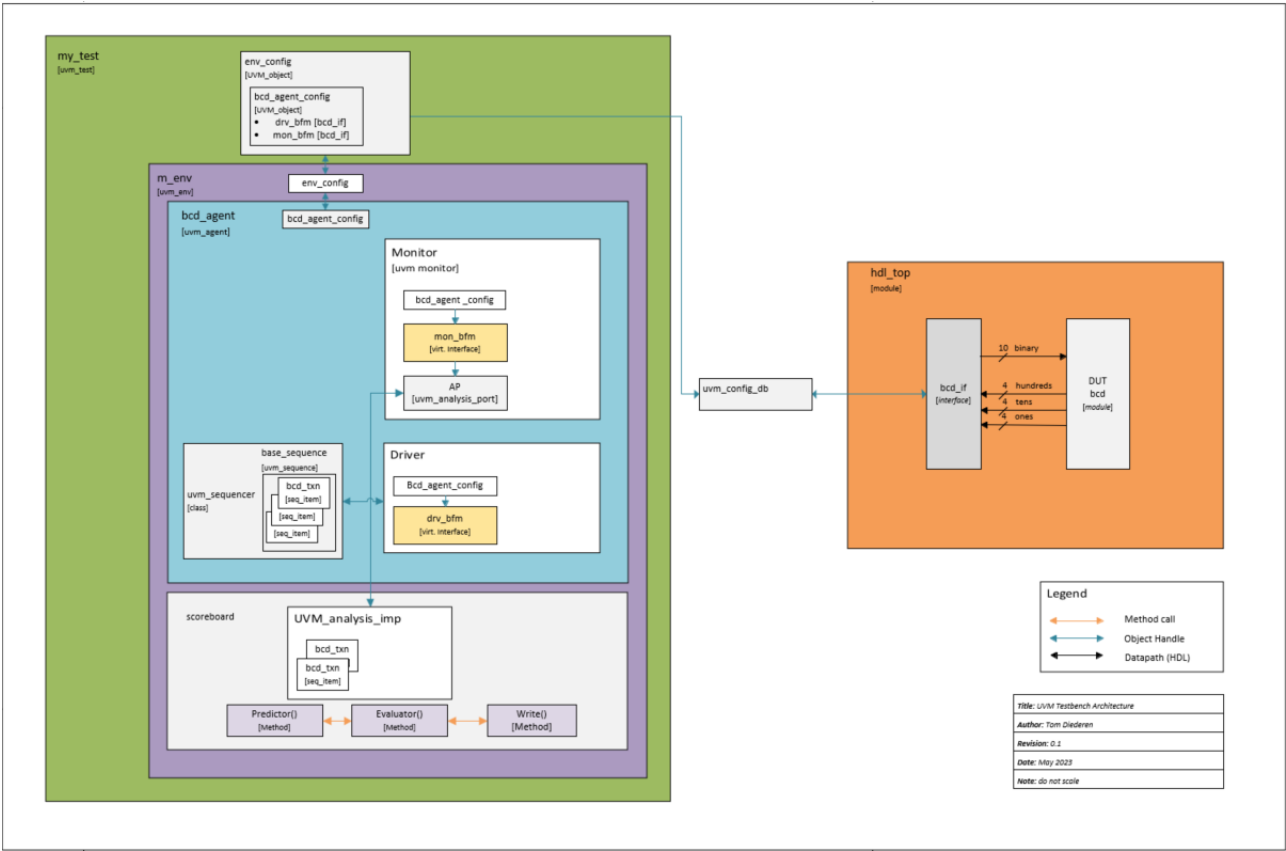
Figure 1 Testbench Architecture

The DUT features the following inputs and outputs:

Table 1 I/O Table of the DUT

Signal Name	Acronym	I/O	Width (bits)	Summary
Binary	N/A	I	10 [9:0]	Binary input number
Hundreds	N/A	O	4 [3:0]	4-bit output representing base 10 hundreds
Tens	N/A	O	4 [3:0]	4-bit output representing base 10 tens
Ones	N/A	O	4 [3:0]	4-bit output representing base 10 ones

3 RTL

The hardware layer is written in Verilog and the verification layer is written in System Verilog.

3.1 HDL

3.1.1 DUT

The DUT is a simple binary to BCD converter based on the "double dabble" algorithm. It appeared in a FPGA demo project, [1], but didn't have an interface, so I created one.

```verilog
18   module bcd (
19   //   input [7:0] binary,
20        input  [9:0] binary,
21        output logic [3:0] hundreds,
22        output logic [3:0] tens,
23        output logic [3:0] ones
24        //output logic output_ready
25   );
26
27        integer i;
28
29        always @(binary) begin
30
31            // new input received, set ready to 0
32            //output_ready = 0;
33            // set 100's, 10's, and 1's to zero
34            hundreds = 4'd0;
35            tens = 4'd0;
36            ones = 4'd0;
37
38   //       for (i=7; i>=0; i=i-1) begin
39            for (i=9; i>=0; i=i-1) begin
40                // add 3 to columns >= 5
41                if (hundreds >= 5)
42                    hundreds = hundreds + 3;
43                if (tens >= 5)
44                    tens = tens + 3;
45                if (ones >= 5)
46                    ones = ones + 3;
47
48                // shift left one
49                hundreds = hundreds << 1;
50                hundreds[0] = tens[3];
51                tens = tens << 1;
52                tens[0] = ones[3];
53                ones = ones << 1;
54                ones[0] = binary[i];
55            end
56
57            // Output ready / stable for read out
58            //output_ready = 1;
59        end
60   endmodule
61
```

Figure 2 RTL for the DUT: "double dabble" binary to bcd converter [1]

3.1.2   Interface

A pointer to the interface will later be used by the UVM classes (virtual interface). It is designed to match the ports of the DUT.

```
1    interface bcd_if ();
2        logic [9:0] if_binary;
3        logic [3:0] if_hundreds;
4        logic [3:0] if_tens;
5        logic [3:0] if_ones;
6        //logic if_ready;
7
8        modport mp_drv(
9            output if_binary
10       );
11
12       modport mp_mon(
13           input if_binary, if_hundreds, if_tens, if_ones
14       );
15
16   endinterface: bcd_if
17
```

Figure 3 Interface to RTL

3.1.3   HDL Top Layer

The hardware description language top layer instantiates the interface and DUT, connects them, and adds the interface to the UVM config database

```
430  module hdl_top;
431      import uvm_pkg::*;
432
433      //Instantiate pin interface to DUT
434      bcd_if BCD_if();
435
436      //Connect DUT to bcd_if0
437      bcd dut0(
438          .binary(BCD_if.if_binary),
439          .hundreds(BCD_if.if_hundreds),
440          .tens(BCD_if.if_tens),
441          .ones(BCD_if.if_ones)
442          //.output_ready(BCD_if.if_ready)
443      );
444
445      //Add virtual interfaces to uvm config db
446      initial begin
447          uvm_config_db #(virtual bcd_if)::set(null, "uvm_test_top", "BCD_if", BCD_if);
448      end
449
450  endmodule: hdl_top
451
```

3.2   HVL

The hardware verification language top layer starts the UVM test, which is at the top of the testbench architecture's hierarchy.

```
452  //HW Verification Language Top Layer. Starts test.
453  module hvl_top;
454      import uvm_pkg::*;
455
456      initial begin
457          run_test("my_test");
458      end
459  endmodule: hvl_top
```

3.2.1   The test

The test is the highest level of the testbench. During the UVM build phase, the test creates the environment. It does so based on an environment configuration object.

```systemverilog
379  class my_test extends uvm_test;
380      //Register with UVM Factory
381      `uvm_component_utils(my_test)
382
383      //Environment class
384      bcd_env m_env;
385
386      //Config Objects
387      env_config m_env_cfg;
388      bcd_agent_config m_bcd_cfg;
389
390      //Constructor
391      function new(string name="my_test", uvm_component parent=null);
392          super.new(name, parent);
393      endfunction
394
395      //Configure bcd agent
396      function void configure_bcd_agent(bcd_agent_config cfg);
397          cfg.active = UVM_ACTIVE;
398          cfg.has_functional_coverage = 0;
399          //cfg.has_scoreboard = 1;
400      endfunction: configure_bcd_agent
401
402      //Build Phase
403      function void build_phase(uvm_phase phase);
404          m_env_cfg = env_config::type_id::create("m_env_cfg"); //Create env config object
405          m_bcd_cfg = bcd_agent_config::type_id::create("m_bcd_cfg"); //Create bcd agent config object
406          configure_bcd_agent(m_bcd_cfg);
407
408          // Get monitor and driver bfm handles
409          if(!uvm_config_db #(virtual bcd_if)::get(this, "", "BCD_if", m_bcd_cfg.drv_bfm) ) `uvm_fatal(get_type_name(), "BCD_if not found in UVM_
410          if(!uvm_config_db #(virtual bcd_if)::get(this, "", "BCD_if", m_bcd_cfg.mon_bfm) ) `uvm_fatal(get_type_name(), "BCD_if not found in UVM_
411
412          m_env_cfg.m_bcd_agent_cfg = m_bcd_cfg; //set agent config member of env config
413          uvm_config_db #(env_config)::set(this, "*", "env_config", m_env_cfg); //Add env_config to uvm_config_db
414
415          //Create environment
416          m_env = bcd_env::type_id::create("m_env", this);
417      endfunction: build_phase
418
419  //   //Run Phase
420      task run_phase(uvm_phase phase);
421          base_sequence b_seq = base_sequence::type_id::create("b_seq");
422          //`uvm_info("", "Test run phase started", UVM_LOW)
423          b_seq.start(m_env.m_bcd_agent.m_sequencer);
424      endtask
425  endclass: my_test
426
```

Figure 4 The UVM test

3.2.1.1   Configuration

Configuration objects contain options to determine what happens in the build phase. This testbench has one for the environment and the agent. The agent's config object is embedded in the environment config object.

Additionally, the configuration objects hold handles to the interface, i.e. virtual interfaces, that will be used by the driver and monitor. Obtaining these through the UVM config data base facilitates flexibility. They are referred to as "BFM" in the code below.

```
8    //Agent Configuration.
9    //  Driver and Monitor BFMs.
10   //  Active vs Passive
11   //  has_functional coverage switch
12   class bcd_agent_config extends uvm_object;
13       `uvm_object_utils(bcd_agent_config)
14
15       //BFM Virtual Interfaces
16       virtual bcd_if mon_bfm;
17       virtual bcd_if drv_bfm;
18
19       //Agent configuration options
20       uvm_active_passive_enum active = UVM_ACTIVE;
21       bit has_functional_coverage = 0;
22
23       //Constructor
24       function new(string name = "bcd_agent_config");
25           super.new(name);
26       endfunction
27
28   endclass: bcd_agent_config
29
```

Figure 5 Configuration class used by the UVM Agent

```
30   //Environment Configuration
31   //  Driver and Monitor BFMs.
32   //  has_functional_coverage, has_scoreboard
33   //  bcd_agent_config
34   class env_config extends uvm_object;
35       `uvm_object_utils(env_config);
36
37       //BFM Virtual Interfaces
38       virtual bcd_if bcd_mon_bfm;
39       virtual bcd_if bcd_driv_bfm;
40
41       //Environment configuration options
42       bit has_functional_coverage = 0;
43       bit has_scoreboard = 1;
44
45       //Configurations for the sub_component(s)
46       bcd_agent_config m_bcd_agent_cfg;
47
48
49       function new(string name="env_config");
50           super.new(name);
51       endfunction
52
53   endclass
54
```

Figure 6 Configuration class used by UVM Environment

The test's build phase does the following:

- Create configuration objects for the environment and agent
- Configure the agent (no config of the environment in this design)
- Get the virtual interfaces for the driver and monitor from the UVM config database
- Set's the agent config attribute of the environment config object
- Store the environment config, also holding the agent config, in the UVM config database
- Create the environment

After the test's build phase has created the environment, a sequence is created and started in the run phase of the UVM. Since the UVM build phase works top down, next up is the environment's build phase.

3.2.1.2   The environment

The environment contains the agent and scoreboard. It also holds the config object described in 3.2.1.1.

During the build phase, it retrieves this config object from the UVM config database and unpacks the agent config object from it which is then stored back in the UVM config database. The agent will refer to it during its build phase later. The environment then creates the agent and, if the environment config's object has it selected, the scoreboard.

The connect phase checks the config object for scoreboard presence and if detected connects the agent's analysis port, ap, to the scoreboards analysis imp (implementation, i.e. the write() method). This is standard UVM practice for communication between the two.

```
337    class bcd_env extends uvm_env;
338        //Register with UVM Factory
339        `uvm_component_utils(bcd_env)
340
341        //Sub component handles
342        bcd_agent m_bcd_agent;
343        scoreboard m_scoreboard;
344
345        //Config objects
346        env_config m_cfg;
347
348        //Constructor
349        function new(string name="bcd_env", uvm_component parent=null);
350            super.new(name, parent);
351        endfunction
352
353        //Build Phase
354        function void build_phase(uvm_phase phase);
355            if(!uvm_config_db #(env_config)::get(this, "", "env_config", m_cfg))`uvm_fatal("CONFIG_LOAD", "Cannot get() configuration env_config fr
356            uvm_config_db #(bcd_agent_config)::set(this, "m_bcd_agent*", "bcd_agent_config", m_cfg.m_bcd_agent_cfg);
357
358            m_bcd_agent = bcd_agent::type_id::create("m_bcd_agent", this); //Create agent
359            if(m_cfg.has_scoreboard) begin
360                m_scoreboard = scoreboard::type_id::create("m_scoreboard", this); //Create scoreboard
361            end
362        endfunction: build_phase
363
364        function void connect_phase(uvm_phase phase);
365            if(m_cfg.has_scoreboard) begin
366                m_bcd_agent.ap.connect(m_scoreboard.sb_ap_imp);
367            end
368        endfunction: connect_phase
369
370    endclass: bcd_env
```

Figure 7 The UVM Environment

3.2.1.2.1   The agent

The agent's main components are the monitor, driver, and sequence. It also contains a configuration object, which is referenced during the build phase, and an analysis port which is connected to the monitor to obtain data that the monitor read.

During the build phase, the agent creates the monitor and gets the configuration object from the UVM config database. If the agent is in active mode, it will create a driver and sequencer.

The connect phase connects the agent's analysis port to the monitor's ap and sets the monitors virtual interface attribute (called mon_bfm). If the config indicates active mode, the sequencer and driver are

connected in the standard UVM manner, using a sequence item port / export, and the driver's virtual interface is set through the config object.

```
216    class bcd_agent extends uvm_agent;
217        //Register with UVM Factory
218        `uvm_component_utils(bcd_agent)
219
220        //Config Object
221        bcd_agent_config m_cfg;
222
223        //Component Members
224        bcd_monitor m_monitor;
225        bcd_driver m_driver;
226        uvm_analysis_port #(bcd_txn) ap;
227        uvm_sequencer #(bcd_txn) m_sequencer;
228
229        //Constructor
230        function new(string name="bcd_agent", uvm_component parent=null);
231            super.new(name, parent);
232        endfunction
233
234        //Build Phase
235        function void build_phase(uvm_phase phase);
236            m_monitor = bcd_monitor::type_id::create("m_monitor", this); //Always present
237
238            if (m_cfg == null)
239                if(!uvm_config_db #(bcd_agent_config)::get(this, "", "bcd_agent_config", m_cfg) ) `uvm_fatal(get_type_name(), "bcd_agent_config not
240            //Create driver and sequencer if agent is in active state (set in agent config object)
241            if(m_cfg.active == UVM_ACTIVE) begin
242                m_driver = bcd_driver::type_id::create("m_driver", this);
243                m_sequencer = uvm_sequencer #(bcd_txn)::type_id::create("m_sequencer", this);
244            end
245        endfunction: build_phase
246
247        //Connect Phase
248        function void connect_phase(uvm_phase phase);
249            ap = m_monitor.bcd_mon_ap;
250            m_monitor.m_bfm = m_cfg.mon_bfm;
251
252            if(m_cfg.active == UVM_ACTIVE) begin
253                m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
254                m_driver.m_bfm = m_cfg.drv_bfm;
255            end
256        endfunction: connect_phase
257
258    endclass: bcd_agent
```

Figure 8 The UVM agent

Monitor

The monitor's main task is to observer the signals going into and coming out of the DUT. It will log these into a UVM transaction and send them to the scoreboard.

The monitor has an analysis port attribute which is connected to an "analysis imp" (implementation) at the scoreboard. Implementation in this case means implementation of a write() method that the monitor calls whenever it has a new transaction ready. This connection is an implementation of the observer OOP design pattern. The monitor also has a virtual interface, called m_bfm, to monitor the DUT ports, an agent config object that holds the virtual interface, and a sequence item.

During the build phase, the analysis port is created, and the agent config object is obtained from the UVM config database. The virtual interface that the monitor uses is obtained from this agent config object.

During the run phase, the monitor creates a sequence item and stores the status of the outputs together with the input that provided it in that item. A "real" monitor would check for some protocol information or sequence of patterns but since the DUT is a simple binary to BCD converter such

functionality is not needed for this project. Because of this simplicity, the monitor's behavior is implemented in the monitor class itself instead of a synthesizable interface.

```
170    class bcd_monitor extends uvm_monitor;
171        //Register with UVM Factory
172        `uvm_component_utils(bcd_monitor)
173
174        uvm_analysis_port #(bcd_txn) bcd_mon_ap; //Analysis port
175        virtual bcd_if m_bfm; //BFM handle
176        bcd_agent_config m_config; //Config, contains monitor bfm handle
177        bcd_txn item; //Sequence item
178
179        //Constructor
180        function new(string name="bcd_monitor", uvm_component parent=null);
181            super.new(name, parent);
182        endfunction
183
184        //Build Phase
185        function void build_phase(uvm_phase phase);
186            bcd_mon_ap = new("bcd_mon_ap", this); //Analysis port
187
188            //Get config object
189            if(!uvm_config_db #(bcd_agent_config)::get(this, "", "bcd_agent_config", m_config)) begin
190                `uvm_error("Config Error", "uvm_config_DB #(bcd_agent_config)::get cannot find resource bcd_agent_config")
191            end
192            m_bfm = m_config.mon_bfm; //Set virtual interface handle
193
194        endfunction
195
196        //Run Phase
197        task run_phase(uvm_phase phase);
198
199            item = bcd_txn::type_id::create("item");
200
201            forever begin
202                @(m_bfm.if_hundreds or m_bfm.if_tens or m_bfm.if_ones)
203                    //`uvm_info(get_type_name(), $sformatf("Hundreds: %b", m_bfm.if_hundreds), UVM_LOW);
204                    item.binary = m_bfm.if_binary;
205                    item.hundreds = m_bfm.if_hundreds;
206                    item.tens = m_bfm.if_tens;
207                    item.ones = m_bfm.if_ones;
208                    `uvm_info(get_type_name(), $sformatf("Monitor output: binary: %b hundreds: %b, tens: %b, ones: %b", item.binary, item.hundreds,
209                    bcd_mon_ap.write(item);
210
211            end
212        endtask
213
214    endclass: bcd_monitor
```

Figure 9 The UVM monitor

The Driver

The driver receives sequence items from the sequencer and turns these into signals that are inputs for the DUT (through the interface). The sequencer and driver are connected by the agent during its connect phase.

```
130   class bcd_driver extends uvm_driver #(bcd_txn);
131       //Register with UVM Factory
132       `uvm_component_utils(bcd_driver)
133
134       //Constructor
135       function new(string name="bcd_driver", uvm_component parent=null);
136           super.new(name, parent);
137       endfunction
138
139       //Virtual interface handle
140       virtual bcd_if m_bfm;
141
142       //Run Phase
143       task run_phase(uvm_phase phase);
144           bcd_txn item;
145
146           forever begin
147               //`uvm_info(get_type_name(), $sformatf("Driver run phase started"), UVM_LOW);
148               seq_item_port.get_next_item(item);
149               `uvm_info(get_type_name(), $sformatf("Driver received sequence item. Binary: %b", item.binary), UVM_LOW);
150               m_bfm.if_binary = item.binary;
151               seq_item_port.item_done();
152
153           end
154       endtask: run_phase
155
156   endclass: bcd_driver
```

Figure 10 The UVM Driver

The sequence

There isn't much to sequence for this simple testbench. Just binary numbers to be sent to the DUT. The free version of Questa that I used did not allow usage of .randomize() which obviously was a big disadvantage. I chose a simple for loop that sends binary numbers to the DUT instead. I tried to send 0 to 1023 to cover every input but free Questa stopped after about 30 sequence items. I figured this too may be a limitation of the free version.

```
85    //Base Sequence
86    //  sends sequence item seq_item n_times.
87    class base_sequence extends uvm_sequence #(bcd_txn);
88        `uvm_object_utils(base_sequence)
89
90        bcd_txn seq_item;
91        int n_times = 3;
92
93        //Construnctor
94        function new (string name="base_sequence");
95            super.new(name);
96        endfunction
97
98        task body();
99            //Raise objection
100           if (starting_phase != null) begin
101               starting_phase.raise_objection(this);
102           end
103
104           seq_item = bcd_txn::type_id::create("seq_item");
105
106           //Send a sequence item "n_times"
107           for (int i = 0; i < n_times; i++) begin
108               start_item(seq_item);
109               seq_item.binary = i; //10'b00_0110_1111: Decimal 111. DUT outputs hundreds, tens, and ones should be '0001'
110               //`uvm_info("body", $sformatf("Sequence item: %b", seq_item.binary), UVM_LOW)
111               finish_item(seq_item);
112           end
113
114           //Drop objection
115           if (starting_phase != null) begin
116               starting_phase.drop_objection(this);
117           end
118       endtask
```

Figure 11 The UVM Sequence

3.2.1.2.2   Scoreboard

The Scoreboard gets sequence items from the monitor, predicts expected outputs and compares the predicted values to the ones in the sequence item. The monitor and scoreboard are connected through an analysis port at the side of the monitor and an analysis implementation on the side of the scoreboard. Implementation is done through the write() method of The Scoreboard.

Scoring correctness is done by using two methods: a predictor and an evaluator. The evaluator takes a sequence item as input argument and passes it to the predictor which then calculates and returns what the outputs should be. The evaluator compares the output of the predictor to the values in the sequence item and counts the correct ones. The process is repeated for each item in the sequence until the UVM run phases are done. In the report phase, it sends a message printing the amount of correct and incorrect results.

```systemverilog
240    class scoreboard extends uvm_scoreboard;
241        //Register with UVM Factory
242        `uvm_component_utils(scoreboard)
243
244        //Create uvm_analysis_imp(lementation) for monitor's ap write() function
245        uvm_analysis_imp #(bcd_txn, scoreboard) sb_ap_imp;
246
247        //Variables to track correct and incorrect transactions
248        int correct = 0;
249        int incorrect = 0;
250
251        //Constructor
252        function new(string name="scoreboard", uvm_component parent=null);
253            super.new(name, parent);
254        endfunction
255
256        //Predictor.
257            //Input: binary number stored in uvm sequence item of type bcd_txn.
258            //Converts input to decimal, then calculates 4 bit value for 'hundreds', 'tens', and 'ones' of that number.
259            //Returns those results concatenated
260        function bit[11:0] predict(bcd_txn item);
261            bit [3:0] b_ones;
262            bit [3:0] b_tens;
263            bit [3:0] b_hundreds;
264
265            int d_in = int'(item.binary);
266            int d_ones = d_in % 10;
267            int d_tens = $floor((d_in % 100) / 10);
268            int d_hundreds = $floor((d_in % 1000) / 100);
269
270            b_ones = 4'(d_ones);
271            b_tens = 4'(d_tens);
272            b_hundreds = 4'(d_hundreds);
273            //`uvm_info(get_type_name(), $sformatf("Predict output: hundreds: %b, tens: %b, ones: %b", b_hundreds, b_tens, b_ones), UVM_LOW);
274            return {b_hundreds, b_tens, b_ones};
275        endfunction: predict
276
277        //Evaluator
278            //Input: binary number stored in uvm sequence item of type bcd_txn.
279            //Calls predict() and compares results
280        function void eval(bcd_txn item);
281            bit [11:0] prediction = predict(item);
282            if (prediction == {item.hundreds, item.tens, item.ones}) begin
283                correct += 1;
284            end
285            else begin
286                incorrect += 1
287                `uvm_info(get_type_name(),"Incorrect result detected by scoreboard.eval()", UVM_LOW);
288                `uvm_info(get_type_name(), $sformatf("scoreboard.eval() prediction: hundreds: %b, tens: %b, ones: %b", prediction[11:8], prediction
289                `uvm_info(get_type_name(), $sformatf("scoreboard input from monitor.analysis_port: hundreds: %b, tens: %b, ones: %b", item.hundreds
290            end
291        endfunction: eval

292
293        //Build Phase
294        function void build_phase(uvm_phase phase);
295            sb_ap_imp = new("sb_ap_imp", this); // Create the uvm_analysis_imp for the scoreboard (links to .write() of the analysis port of the mo
296        endfunction: build_phase
297
298        //Implementation of monitor.ap.write()
299        function void write(bcd_txn item);
300            `uvm_info(get_type_name(), $sformatf("Scoreboard input: binary: %b hundreds: %b, tens: %b, ones: %b", item.binary, item.hundreds, item.
301            eval(item);
302        endfunction
303
304        //Report Phase
305        function void report_phase(uvm_phase phase);
306            `uvm_info(get_type_name(), $sformatf("Scoreboard results: incorrect: %d, correct: %d ", incorrect, correct), UVM_LOW); //Store incorre
307        endfunction
308
309
310    endclass: scoreboard
```

Figure 12 The UVM Scoreboard

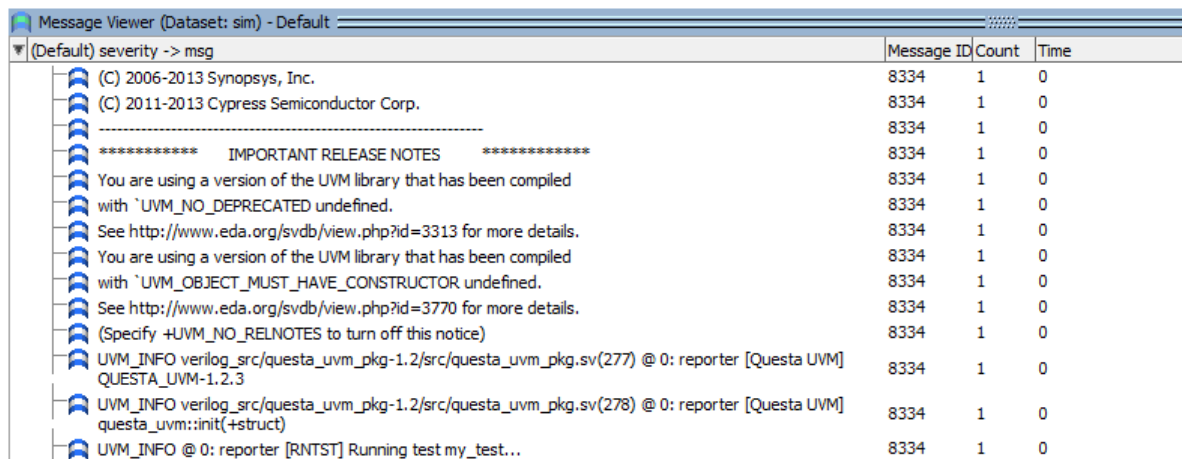# 4 VERIFICATION (SIMULATION)

4.1 EDA TOOLING

Questa, the free Intel FPGA Starter edition, was used for simulation purposes. It comes with a UVM package pre-installed. Since running the simulator and adding waveforms got repetitive quickly, I created a .do file, in TCL, to speed up the process.

```
Ln#
 1    vsim -c work.hdl_top work.hvl_top +UVM_TESTNAME=my_test -classdebug -msgmode both -uvmcontrol=all
 2    add wave -position end sim:/hdl_top/BCD_if/*
 3    run 0
```

Figure 13 Combining frequent simulator commands in a .do file (TCL).

4.2 SIMULATOR CONFIGURATION

To get familiar with the UVM capabilities of Questa, I referenced a simple UVM example, [3], and the Questa manual to configure the simulator correctly. For example, the vsim flag "-displaymsgmode both" had to be set correctly to relay UVM messages to the Questa message viewer window.

Figure 14 Configuring Questa: The Message Viewer window did not show UVM messages by default.

4.3 UVM HIERARCHY

With the simulator configured correctly, the first step was to create the UVM components (and config objects).
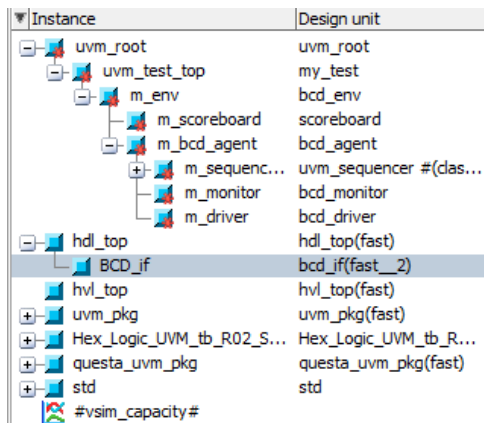
Figure 15 UVM Hierarchy as shown by Questa

4.4 CONNECTING AND RUNNING THE TESTBENCH

The next steps involved connecting all the components, followed by defining the run phases. I took a step-by-step approach. First, I started the sequence and had it send a UVM message. Then the driver, followed by the monitor and finally the scoreboard. This approached helped me debug issues per component before testing the whole testbench.

I sent a few consecutive numbers to the DUT to look at the output which is correct. I'd preferred random numbers but the free version did not support the use of: .randomize().

Figure 16 UVM Messages output by the testbench

This approach is obviously not scalable. For future projects, I intend to use UVM Transaction recording in Questa (if the free version supports it).
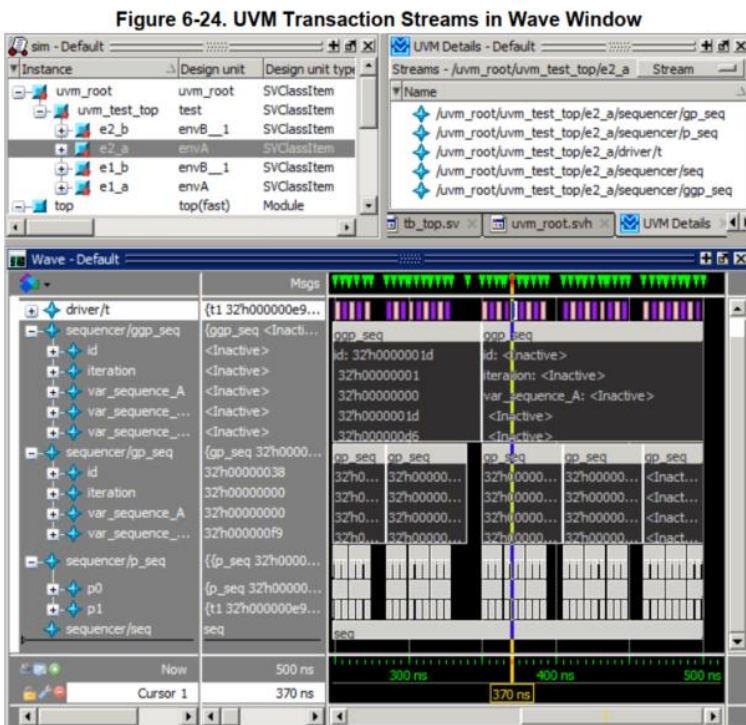

Figure 17 UVM Transaction recording as shown in the Questa User's Manual

4.5 Functional Coverage

To check all possible binary input values, I set the sequencer to go from 0 to 1023. Unfortunately, the simulator stopped after 35 sequence items. I'm thinking this may be another limitation of the free version. Out of scope for this project are values of X and Z. Undefined and tri-state values will be considered for the next project.

A sample result can be seen below. Binary input 00_0010_0010, decimal 34, correctly creates output: zero hundreds, 3 tens, and 4 ones.
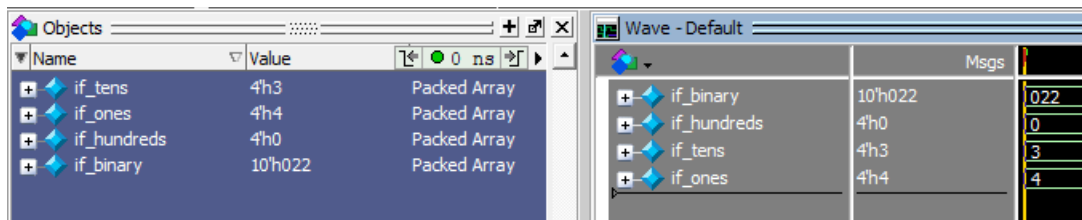
The scoreboard showed all 35 input values were correct:

UVM_INFO C:/Users/tdiedere/OneDrive - Intel
Corporation/Documents/Quartus_Projects/Questa_Projects/UVM BCD Final/UVM_BCD_tb_Final.sv(326) @ 0: 8330
uvm_test_top.m_env.m_scoreboard [scoreboard] Scoreboard results: incorrect: 0, correct: 35

5 Future improvements and Lessons Learned

5.1 Future Improvements

I learned a lot during this project. The main goal was to learn the fundamentals of UVM testbenches and build a base testbench for future re-use during bigger projects. I anticipate that for more elaborate designs, the following features will have to be added or improved:

- Functional coverage checker
- Make code compatible for emulation
- More elaborate TCL / do file to automate simulator commands (e.g. vsim configuration and add wave forms)
- Use UVM's Register Abstraction Layer
- Use TLM viewer of Questa (simple UVM_info prints are not scalable)

5.2 Lessons Learned

- Version Control and text editor. Develop code in VS Code. Push daily copy to remote. Simply paste code in Questa to run.
- Work more Agile. Write down features to include. Use "Kanban/Kaizen?" board. Work in sprints.
- Document more thoroughly throughout the project instead of postponing some of the documentation effort.

6 References

[1] "Cyclone V Logic Solver (Design Example)," [Online]. Available: https://www.intel.com/content/www/us/en/design-example/714798/cyclone-v-logic-solver.html.

[2] Mentor Graphics (Siemens), UVM Cookbook, verificationacadamy.com.

[3] John Aynsley, Doulos, "EDA Playground," Doulos, 2011-2012. [Online]. Available: https://www.edaplayground.com/x/Wzp. [Accessed 05 2023].

[4] Shepherd Tutorials, "Section 4.105," February 2023. [Online]. Available: https://www.udemy.com/course/verilog-hdl-vlsi-hardware-design-comprehensive-masterclass/.