2023

Universal Verification Method Test Bench

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 test bench. The hierarchy of the paragraphs describing the hvl match the hierarchy of the actual testbench.

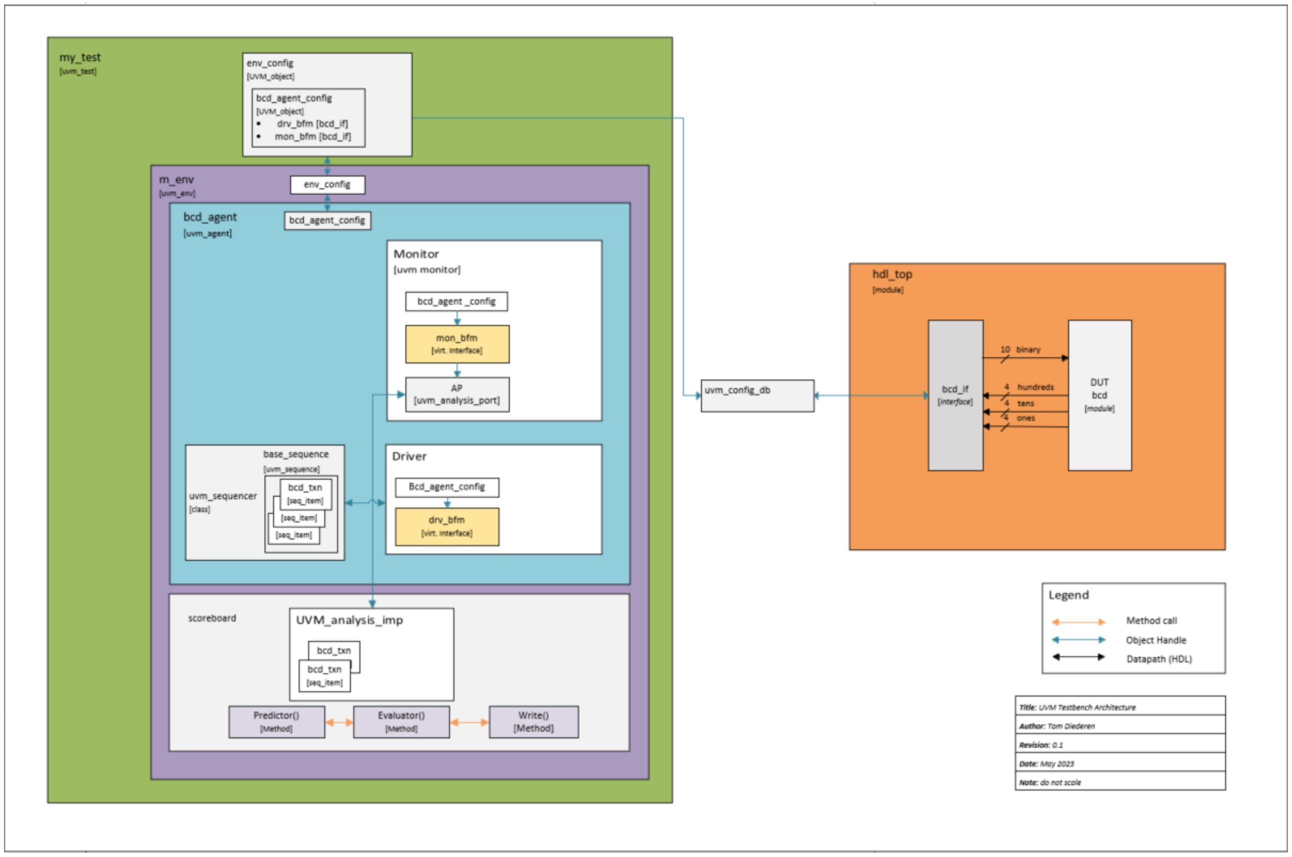


Figure 1 Testbench Architecture

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

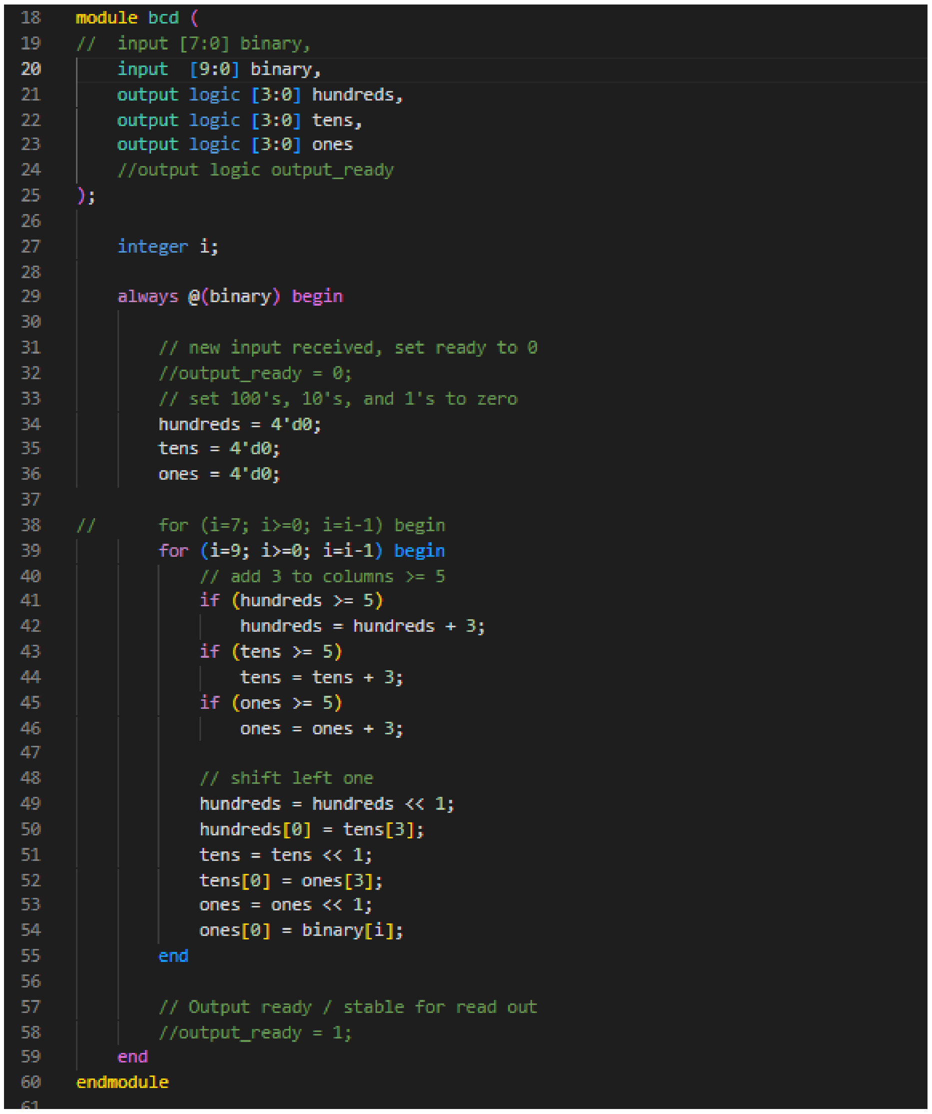
3 RTL

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.

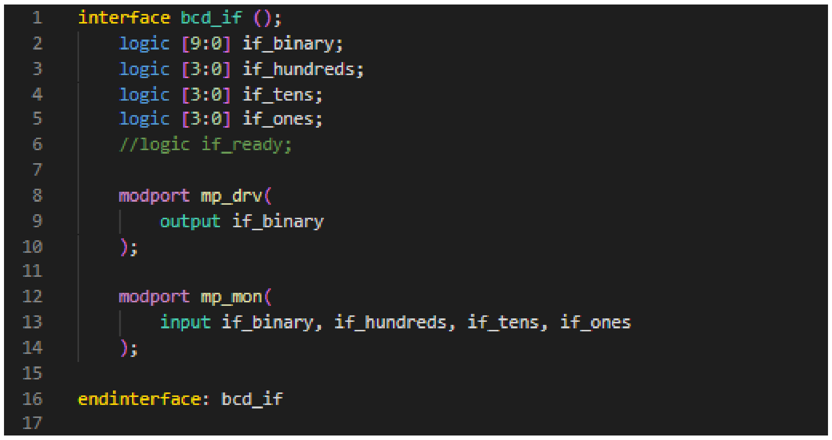


```
module bcd (
//  input [7:0] binary,
    input  [9:0] binary,
    output logic [3:0] hundreds,
    output logic [3:0] tens,
    output logic [3:0] ones
    //output logic output_ready
);

    integer i;

    always @(binary) begin

        // new input received, set ready to 0
        //output_ready = 0;
        // set 100's, 10's, and 1's to zero
        hundreds = 4'd0;
        tens = 4'd0;
        ones = 4'd0;

//      for (i=7; i>=0; i=i-1) begin
        for (i=9; i>=0; i=i-1) begin
            // add 3 to columns >= 5
            if (hundreds >= 5)
                hundreds = hundreds + 3;
            if (tens >= 5)
                tens = tens + 3;
            if (ones >= 5)
                ones = ones + 3;

            // shift left one
            hundreds = hundreds << 1;
            hundreds[0] = tens[3];
            tens = tens << 1;
            tens[0] = ones[3];
            ones = ones << 1;
            ones[0] = binary[i];
        end

        // Output ready / stable for read out
        //output_ready = 1;
    end
endmodule
```

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.

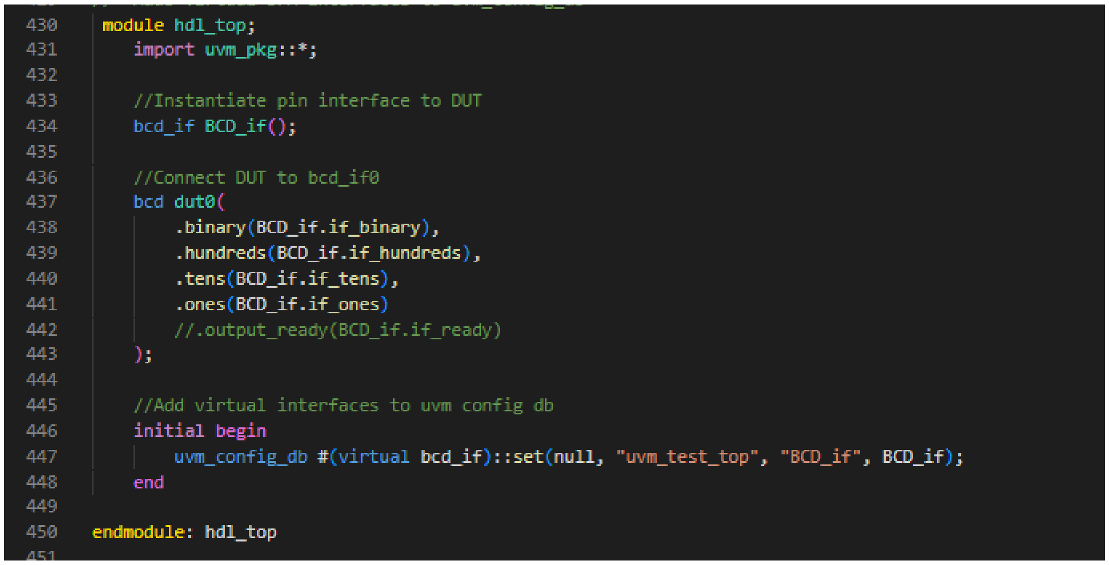


```
interface bcd_if ();
    logic [9:0] if_binary;
    logic [3:0] if_hundreds;
    logic [3:0] if_tens;
    logic [3:0] if_ones;
    //logic if_ready;

    modport mp_drv(
        output if_binary
    );

    modport mp_mon(
        input if_binary, if_hundreds, if_tens, if_ones
    );

endinterface: bcd_if
```

Figure 3 Interface to RTL

3.1.3 HDL Top Layer

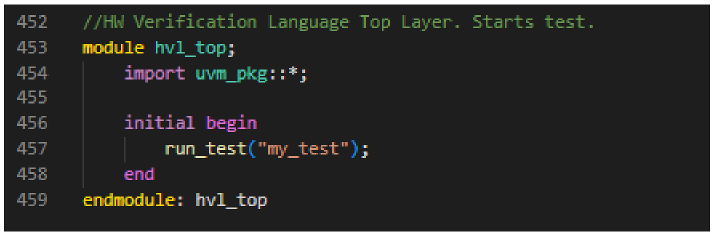
The hardware description language top layer instantiates the interface and DUT, connects them, and adds the interface to the UVM config database



```
module hdl_top;
    import uvm_pkg::*;

    //Instantiate pin interface to DUT
    bcd_if BCD_if();

    //Connect DUT to bcd_if0
    bcd dut0(
        .binary(BCD_if.if_binary),
        .hundreds(BCD_if.if_hundreds),
        .tens(BCD_if.if_tens),
        .ones(BCD_if.if_ones)
        //.output_ready(BCD_if.if_ready)
    );

    //Add virtual interfaces to uvm config db
    initial begin
        uvm_config_db #(virtual bcd_if)::set(null, "uvm_test_top", "BCD_if", BCD_if);
    end

endmodule: hdl_top
```

3.2 HVL

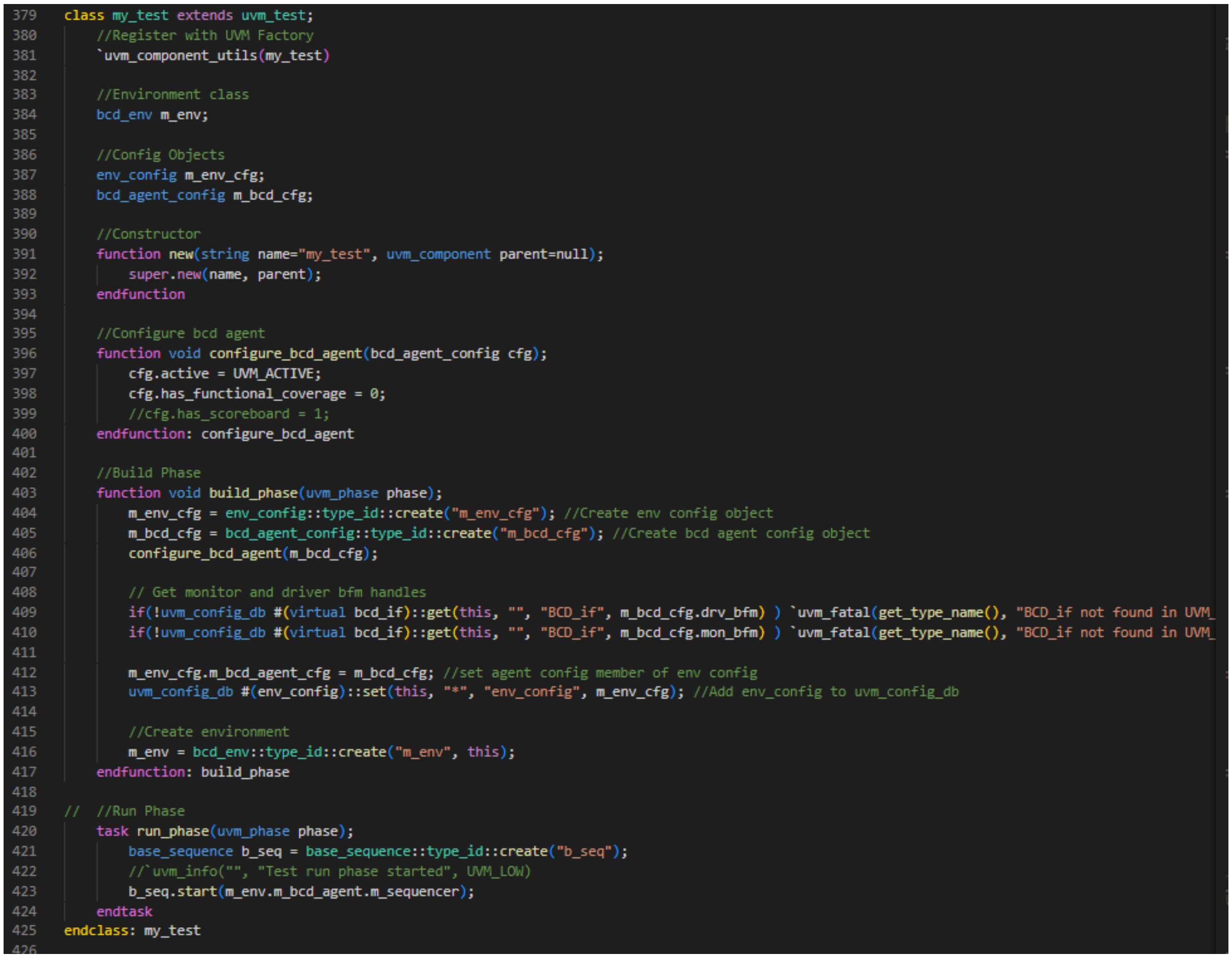
The hardware verification language top layer starts the UVM test, which is at the top of the testbench architecture's hierarchy.



```
//HW Verification Language Top Layer. Starts test.
module hvl_top;
    import uvm_pkg::*;

    initial begin
        run_test("my_test");
    end
endmodule: hvl_top
```

3.2.1 The test

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



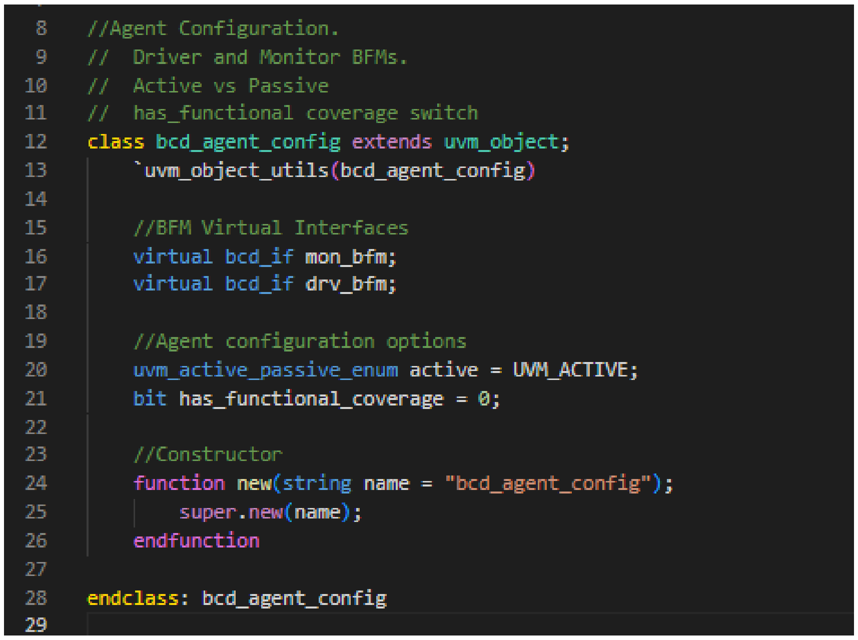
```
class my_test extends uvm_test;
    //Register with UVM Factory
    `uvm_component_utils(my_test)

    //Environment class
    bcd_env m_env;

    //Config Objects
    env_config m_env_cfg;
    bcd_agent_config m_bcd_cfg;

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

    //Configure bcd agent
    function void configure_bcd_agent(bcd_agent_config cfg);
        cfg.active = UVM_ACTIVE;
        cfg.has_functional_coverage = 0;
        //cfg.has_scoreboard = 1;
    endfunction: configure_bcd_agent

    //Build Phase
    function void build_phase(uvm_phase phase);
        m_env_cfg = env_config::type_id::create("m_env_cfg"); //Create env config object
        m_bcd_cfg = bcd_agent_config::type_id::create("m_bcd_cfg"); //Create bcd agent config object
        configure_bcd_agent(m_bcd_cfg);

        // Get monitor and driver bfm handles
        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_
        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_

        m_env_cfg.m_bcd_agent_cfg = m_bcd_cfg; //set agent config member of env config
        uvm_config_db #(env_config)::set(this, "*", "env_config", m_env_cfg); //Add env_config to uvm_config_db

        //Create environment
        m_env = bcd_env::type_id::create("m_env", this);
    endfunction: build_phase

//  //Run Phase
    task run_phase(uvm_phase phase);
        base_sequence b_seq = base_sequence::type_id::create("b_seq");
        //`uvm_info("", "Test run phase started", UVM_LOW)
        b_seq.start(m_env.m_bcd_agent.m_sequencer);
    endtask
endclass: my_test
```

Figure 4 The UVM test

3.2.1.1 Configuration

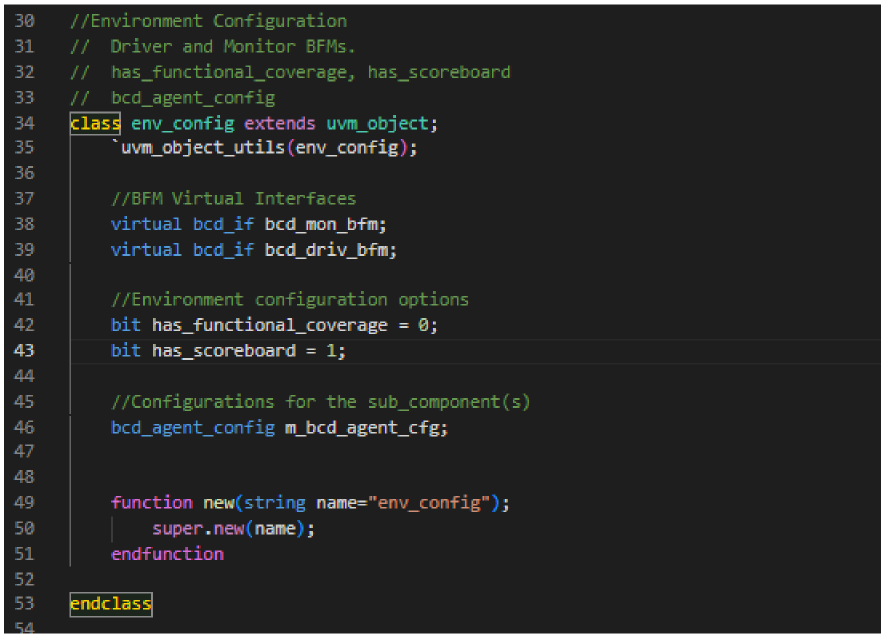
Configuration objects contain options to determine what happens in the build phase. This test bench has one for the environment and the agent. The agent's config object is embedded in the environment config object.

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.



```
//Agent Configuration.
//  Driver and Monitor BFMs.
//  Active vs Passive
//  has_functional coverage switch
class bcd_agent_config extends uvm_object;
    `uvm_object_utils(bcd_agent_config)

    //BFM Virtual Interfaces
    virtual bcd_if mon_bfm;
    virtual bcd_if drv_bfm;

    //Agent configuration options
    uvm_active_passive_enum active = UVM_ACTIVE;
    bit has_functional_coverage = 0;

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

endclass: bcd_agent_config
```

Figure 5 Configuration class used by the UVM Agent



```
//Environment Configuration
//  Driver and Monitor BFMs.
//  has_functional_coverage, has_scoreboard
//  bcd_agent_config
class env_config extends uvm_object;
    `uvm_object_utils(env_config);

    //BFM Virtual Interfaces
    virtual bcd_if bcd_mon_bfm;
    virtual bcd_if bcd_driv_bfm;

    //Environment configuration options
    bit has_functional_coverage = 0;
    bit has_scoreboard = 1;

    //Configurations for the sub_component(s)
    bcd_agent_config m_bcd_agent_cfg;


    function new(string name="env_config");
        super.new(name);
    endfunction

endclass
```

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

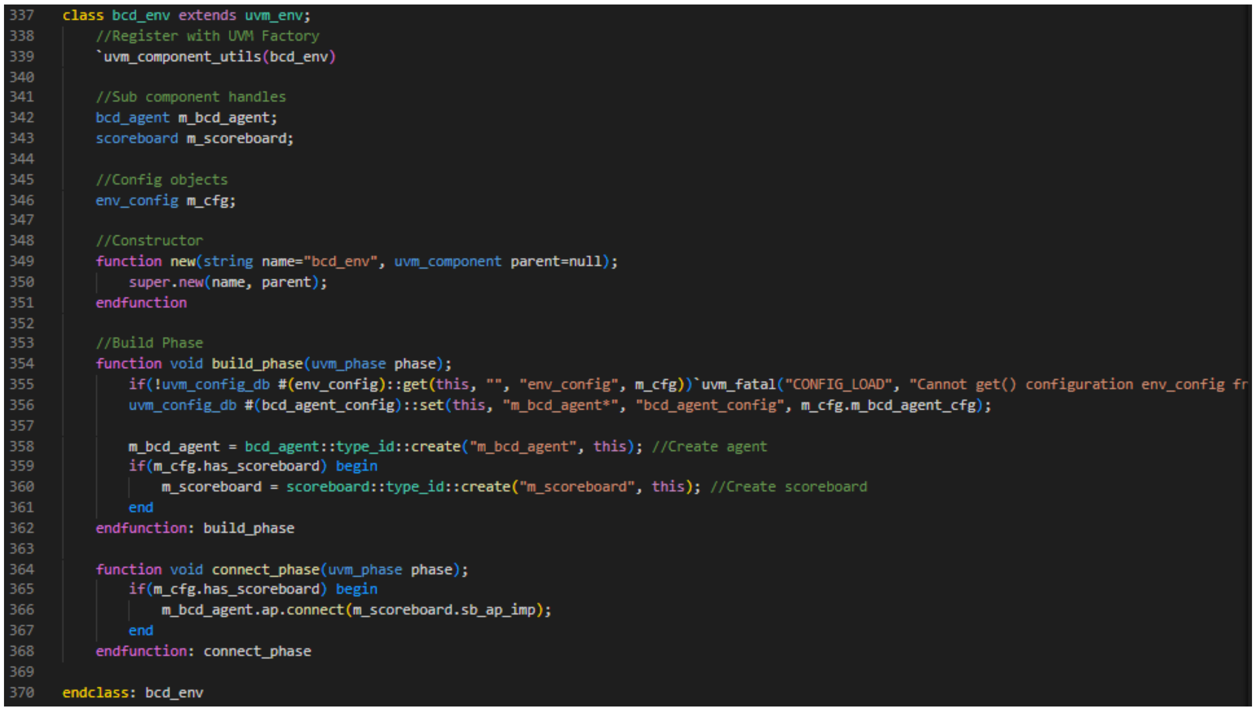
After the test's build phase has created the environment, a sequence is created and started in the run phase of the UVM. Since the UVM build phase works top down, next up is the environment's build phase.

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.



```
class bcd_env extends uvm_env;
    //Register with UVM Factory
    `uvm_component_utils(bcd_env)

    //Sub component handles
    bcd_agent m_bcd_agent;
    scoreboard m_scoreboard;

    //Config objects
    env_config m_cfg;

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

    //Build Phase
    function void build_phase(uvm_phase phase);
        if(!uvm_config_db #(env_config)::get(this, "", "env_config", m_cfg))`uvm_fatal("CONFIG_LOAD", "Cannot get() configuration env_config fr
        uvm_config_db #(bcd_agent_config)::set(this, "m_bcd_agent*", "bcd_agent_config", m_cfg.m_bcd_agent_cfg);

        m_bcd_agent = bcd_agent::type_id::create("m_bcd_agent", this); //Create agent
        if(m_cfg.has_scoreboard) begin
            m_scoreboard = scoreboard::type_id::create("m_scoreboard", this); //Create scoreboard
        end
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        if(m_cfg.has_scoreboard) begin
            m_bcd_agent.ap.connect(m_scoreboard.sb_ap_imp);
        end
    endfunction: connect_phase

endclass: bcd_env
```

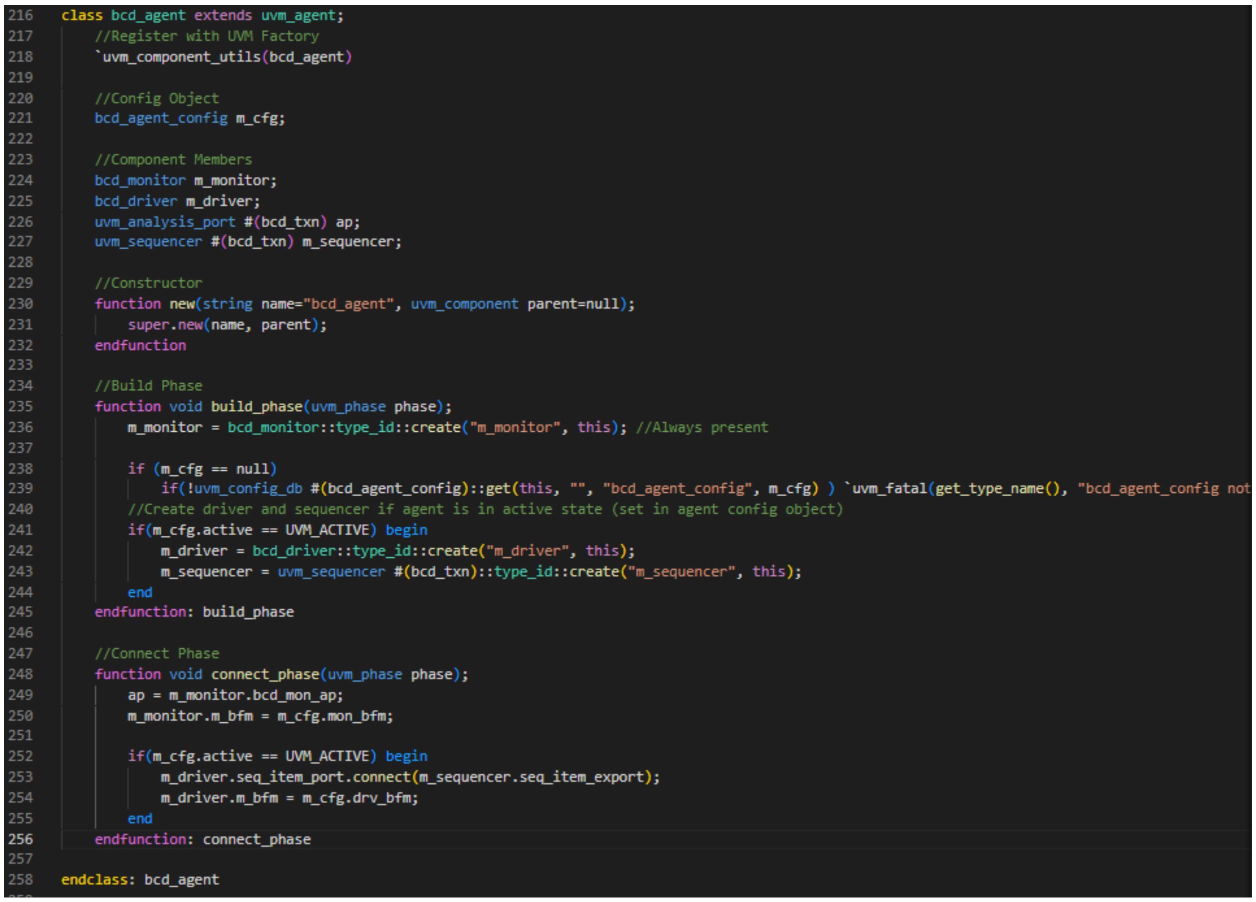
Figure 7 The UVM Environment

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.



```
class bcd_agent extends uvm_agent;
    //Register with UVM Factory
    `uvm_component_utils(bcd_agent)

    //Config Object
    bcd_agent_config m_cfg;

    //Component Members
    bcd_monitor m_monitor;
    bcd_driver m_driver;
    uvm_analysis_port #(bcd_txn) ap;
    uvm_sequencer #(bcd_txn) m_sequencer;

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

    //Build Phase
    function void build_phase(uvm_phase phase);
        m_monitor = bcd_monitor::type_id::create("m_monitor", this); //Always present

        if (m_cfg == null)
            if(!uvm_config_db #(bcd_agent_config)::get(this, "", "bcd_agent_config", m_cfg) ) `uvm_fatal(get_type_name(), "bcd_agent_config not
        //Create driver and sequencer if agent is in active state (set in agent config object)
        if(m_cfg.active == UVM_ACTIVE) begin
            m_driver = bcd_driver::type_id::create("m_driver", this);
            m_sequencer = uvm_sequencer #(bcd_txn)::type_id::create("m_sequencer", this);
        end
    endfunction: build_phase

    //Connect Phase
    function void connect_phase(uvm_phase phase);
        ap = m_monitor.bcd_mon_ap;
        m_monitor.m_bfm = m_cfg.mon_bfm;

        if(m_cfg.active == UVM_ACTIVE) begin
            m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
            m_driver.m_bfm = m_cfg.drv_bfm;
        end
    endfunction: connect_phase

endclass: bcd_agent
```

Figure 8 The UVM agent

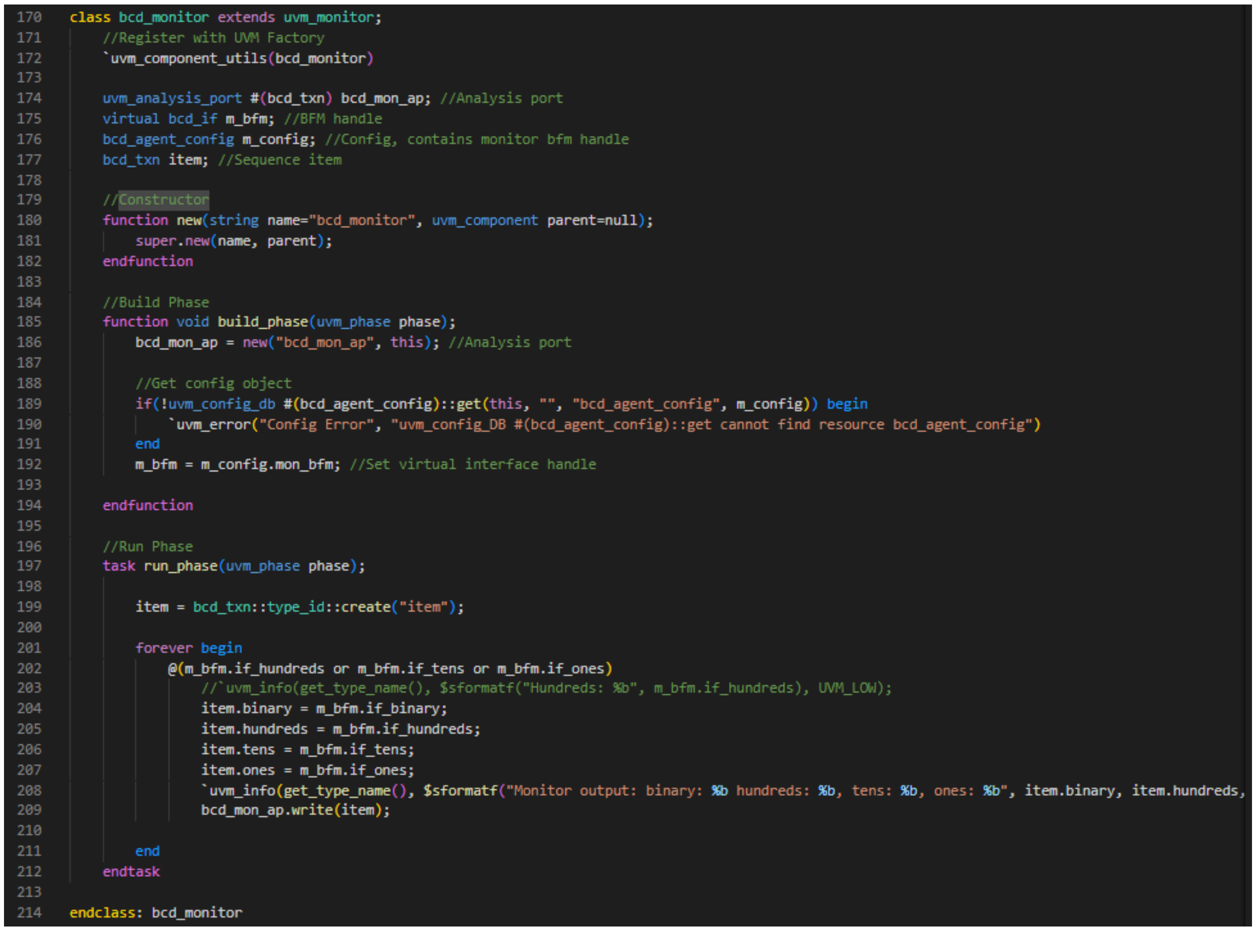
Monitor

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.

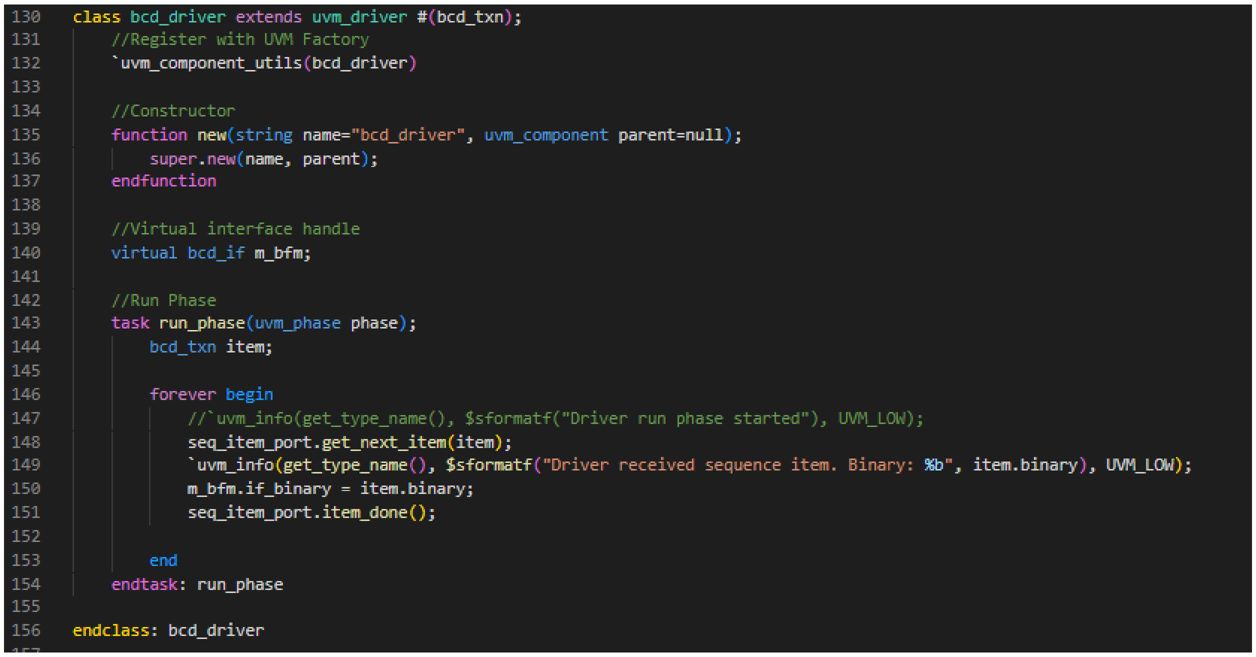


```
class bcd_monitor extends uvm_monitor;
    //Register with UVM Factory
    `uvm_component_utils(bcd_monitor)

    uvm_analysis_port #(bcd_txn) bcd_mon_ap; //Analysis port
    virtual bcd_if m_bfm; //BFM handle
    bcd_agent_config m_config; //Config, contains monitor bfm handle
    bcd_txn item; //Sequence item

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

    //Build Phase
    function void build_phase(uvm_phase phase);
        bcd_mon_ap = new("bcd_mon_ap", this); //Analysis port

        //Get config object
        if(!uvm_config_db #(bcd_agent_config)::get(this, "", "bcd_agent_config", m_config)) begin
            `uvm_error("Config Error", "uvm_config_DB #(bcd_agent_config)::get cannot find resource bcd_agent_config")
        end
        m_bfm = m_config.mon_bfm; //Set virtual interface handle

    endfunction

    //Run Phase
    task run_phase(uvm_phase phase);

        item = bcd_txn::type_id::create("item");

        forever begin
            @(m_bfm.if_hundreds or m_bfm.if_tens or m_bfm.if_ones)
                //`uvm_info(get_type_name(), $sformatf("Hundreds: %b", m_bfm.if_hundreds), UVM_LOW);
                item.binary = m_bfm.if_binary;
                item.hundreds = m_bfm.if_hundreds;
                item.tens = m_bfm.if_tens;
                item.ones = m_bfm.if_ones;
                `uvm_info(get_type_name(), $sformatf("Monitor output: binary: %b hundreds: %b, tens: %b, ones: %b", item.binary, item.hundreds,
                bcd_mon_ap.write(item);

        end
    endtask

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.

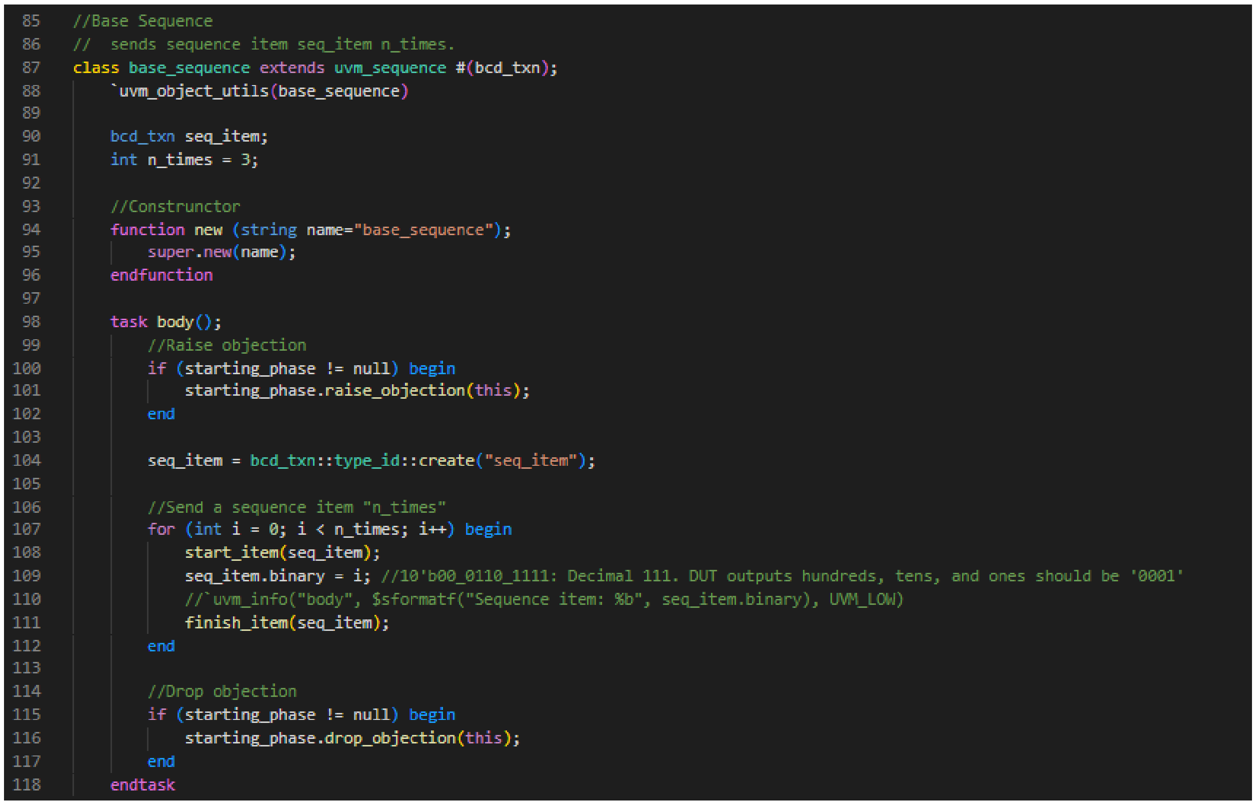


```
class bcd_driver extends uvm_driver #(bcd_txn);
    //Register with UVM Factory
    `uvm_component_utils(bcd_driver)

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

    //Virtual interface handle
    virtual bcd_if m_bfm;

    //Run Phase
    task run_phase(uvm_phase phase);
        bcd_txn item;

        forever begin
            //`uvm_info(get_type_name(), $sformatf("Driver run phase started"), UVM_LOW);
            seq_item_port.get_next_item(item);
            `uvm_info(get_type_name(), $sformatf("Driver received sequence item. Binary: %b", item.binary), UVM_LOW);
            m_bfm.if_binary = item.binary;
            seq_item_port.item_done();

        end
    endtask: run_phase

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.



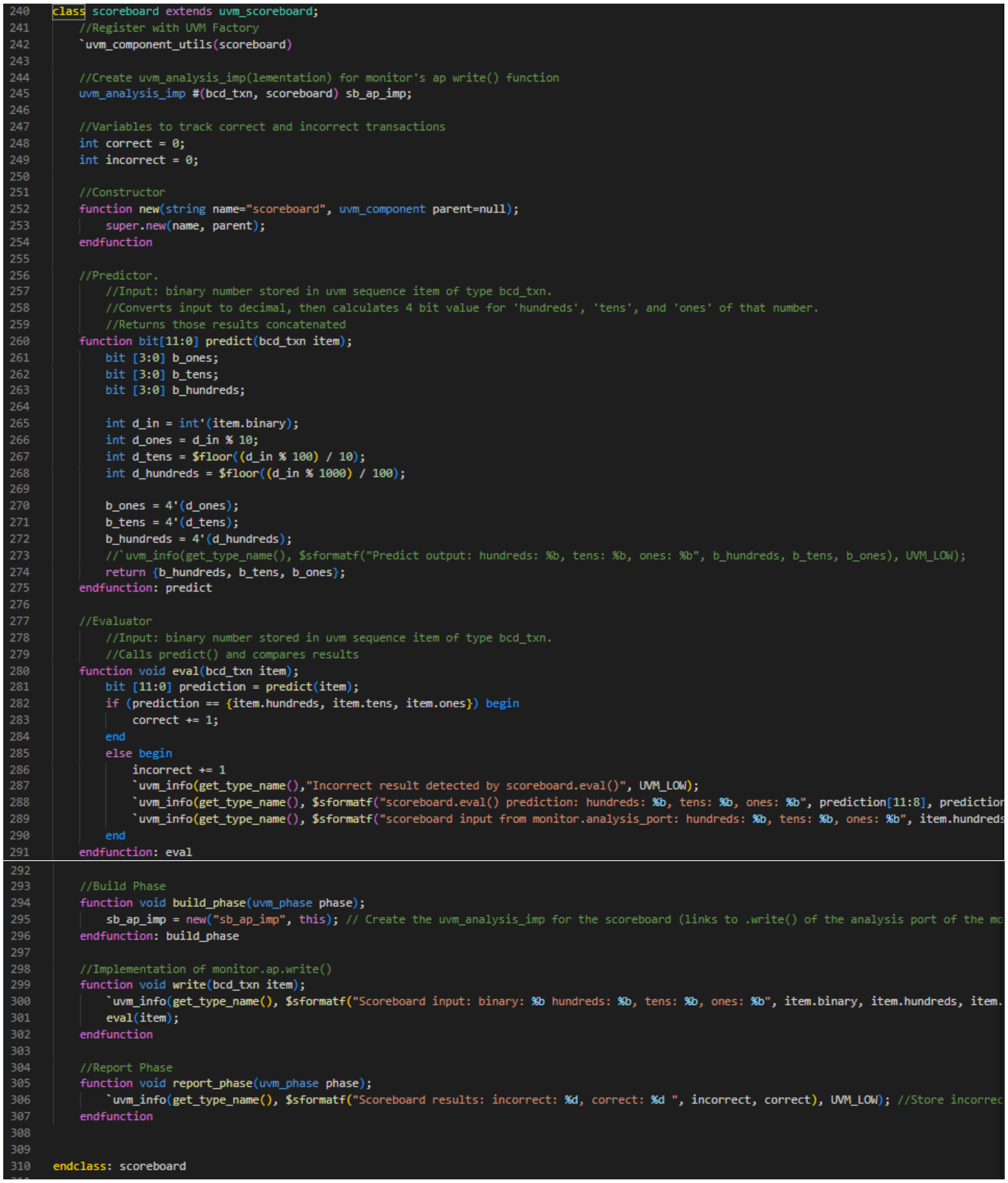
```
//Base Sequence
//  sends sequence item seq_item n_times.
class base_sequence extends uvm_sequence #(bcd_txn);
    `uvm_object_utils(base_sequence)

    bcd_txn seq_item;
    int n_times = 3;

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

    task body();
        //Raise objection
        if (starting_phase != null) begin
            starting_phase.raise_objection(this);
        end

        seq_item = bcd_txn::type_id::create("seq_item");

        //Send a sequence item "n_times"
        for (int i = 0; i < n_times; i++) begin
            start_item(seq_item);
            seq_item.binary = i; //10'b00_0110_1111: Decimal 111. DUT outputs hundreds, tens, and ones should be '0001'
            //`uvm_info("body", $sformatf("Sequence item: %b", seq_item.binary), UVM_LOW)
            finish_item(seq_item);
        end

        //Drop objection
        if (starting_phase != null) begin
            starting_phase.drop_objection(this);
        end
    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.



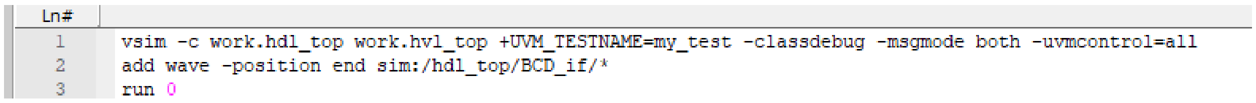
```
class scoreboard extends uvm_scoreboard;
    //Register with UVM Factory
    `uvm_component_utils(scoreboard)

    //Create uvm_analysis_imp(lementation) for monitor's ap write() function
    uvm_analysis_imp #(bcd_txn, scoreboard) sb_ap_imp;

    //Variables to track correct and incorrect transactions
    int correct = 0;
    int incorrect = 0;

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

    //Predictor.
        //Input: binary number stored in uvm sequence item of type bcd_txn.
        //Converts input to decimal, then calculates 4 bit value for 'hundreds', 'tens', and 'ones' of that number.
        //Returns those results concatenated
    function bit[11:0] predict(bcd_txn item);
        bit [3:0] b_ones;
        bit [3:0] b_tens;
        bit [3:0] b_hundreds;

        int d_in = int'(item.binary);
        int d_ones = d_in % 10;
        int d_tens = $floor((d_in % 100) / 10);
        int d_hundreds = $floor((d_in % 1000) / 100);

        b_ones = 4'(d_ones);
        b_tens = 4'(d_tens);
        b_hundreds = 4'(d_hundreds);
        //`uvm_info(get_type_name(), $sformatf("Predict output: hundreds: %b, tens: %b, ones: %b", b_hundreds, b_tens, b_ones), UVM_LOW);
        return {b_hundreds, b_tens, b_ones};
    endfunction: predict

    //Evaluator
        //Input: binary number stored in uvm sequence item of type bcd_txn.
        //Calls predict() and compares results
    function void eval(bcd_txn item);
        bit [11:0] prediction = predict(item);
        if (prediction == {item.hundreds, item.tens, item.ones}) begin
            correct += 1;
        end
        else begin
            incorrect += 1
            `uvm_info(get_type_name(),"Incorrect result detected by scoreboard.eval()", UVM_LOW);
            `uvm_info(get_type_name(), $sformatf("scoreboard.eval() prediction: hundreds: %b, tens: %b, ones: %b", prediction[11:8], predictio
            `uvm_info(get_type_name(), $sformatf("scoreboard input from monitor.analysis_port: hundreds: %b, tens: %b, ones: %b", item.hundreds
        end
    endfunction: eval

    //Build Phase
    function void build_phase(uvm_phase phase);
        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
    endfunction: build_phase

    //Implementation of monitor.ap.write()
    function void write(bcd_txn item);
        `uvm_info(get_type_name(), $sformatf("Scoreboard input: binary: %b hundreds: %b, tens: %b, ones: %b", item.binary, item.hundreds, item.
        eval(item);
    endfunction

    //Report Phase
    function void report_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $sformatf("Scoreboard results: incorrect: %d, correct: %d ", incorrect, correct), UVM_LOW); //Store incorrec
    endfunction


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.

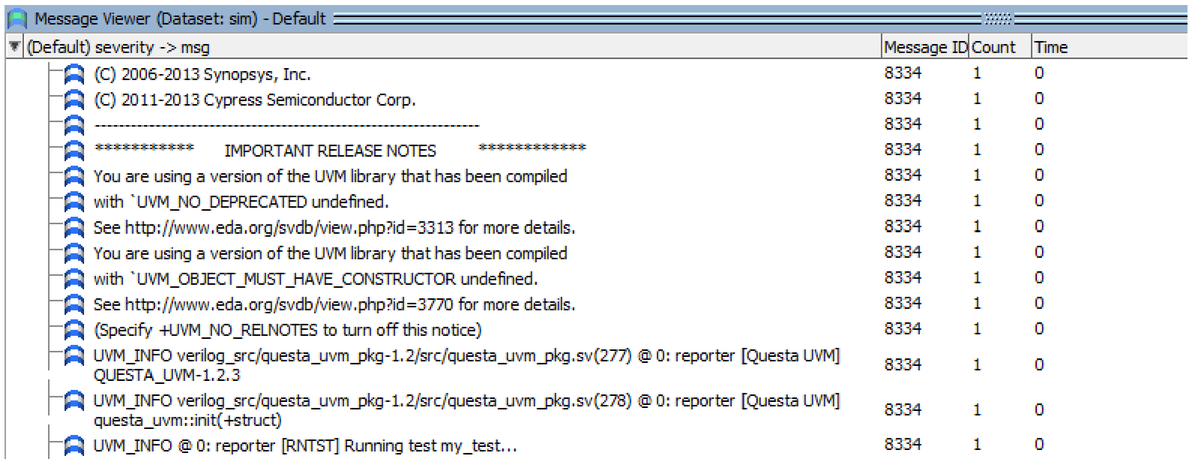


```
vsim -c work.hdl_top work.hvl_top +UVM_TESTNAME=my_test -classdebug -msgmode both -uvmcontrol=all
add wave -position end sim:/hdl_top/BCD_if/*
run 0
```

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.



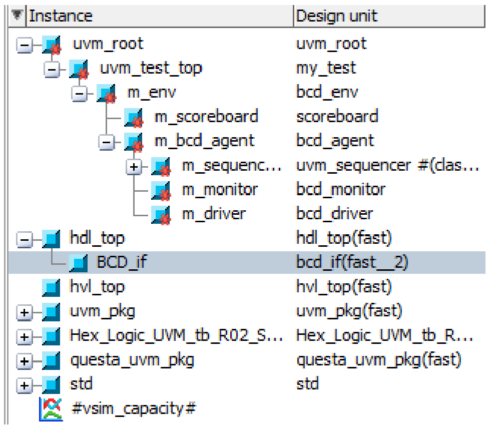
Message Viewer (Dataset: sim) - Default

(Default) severity -> msg	Message ID	Count	Time
(C) 2006-2013 Synopsys, Inc.	8334	1	0
(C) 2011-2013 Cypress Semiconductor Corp.	8334	1	0
--	8334	1	0
************ IMPORTANT RELEASE NOTES *************	8334	1	0
You are using a version of the UVM library that has been compiled	8334	1	0
with `UVM_NO_DEPRECATED undefined.	8334	1	0
See http://www.eda.org/svdb/view.php?id=3313 for more details.	8334	1	0
You are using a version of the UVM library that has been compiled	8334	1	0
with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.	8334	1	0
See http://www.eda.org/svdb/view.php?id=3770 for more details.	8334	1	0
(Specify +UVM_NO_RELNOTES to turn off this notice)	8334	1	0
UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(277) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.3	8334	1	0
UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(278) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)	8334	1	0
UVM_INFO @ 0: reporter [RNTST] Running test my_test...	8334	1	0

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).



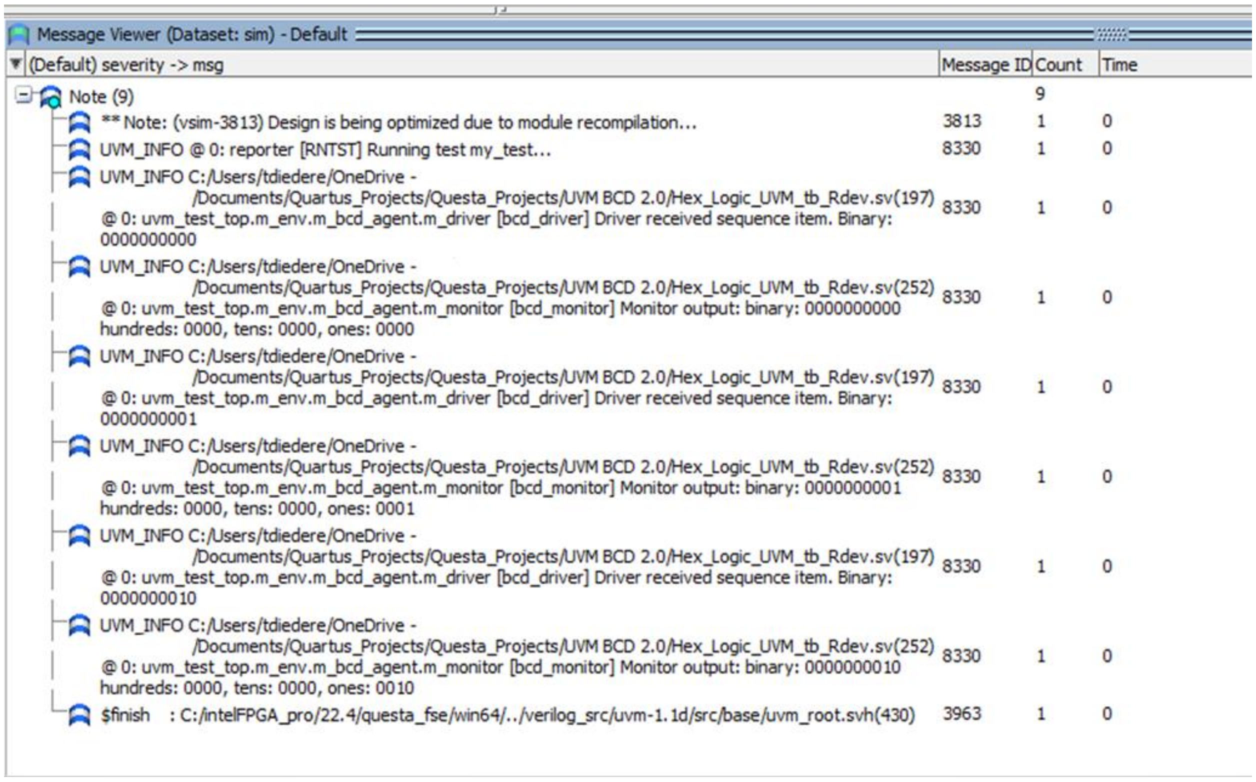
Instance	Design unit
uvm_root	uvm_root
uvm_test_top	my_test
m_env	bcd_env
m_scoreboard	scoreboard
m_bcd_agent	bcd_agent
m_sequenc...	uvm_sequencer #(clas...
m_monitor	bcd_monitor
m_driver	bcd_driver
hdl_top	hdl_top(fast)
BCD_if	bcd_if(fast__2)
hvl_top	hvl_top(fast)
uvm_pkg	uvm_pkg(fast)
Hex_Logic_UVM_tb_R02_S...	Hex_Logic_UVM_tb_R...
questa_uvm_pkg	questa_uvm_pkg(fast)
std	std
#vsim_capacity#	

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().



```
Message Viewer (Dataset: sim) - Default
(Default) severity -> msg                                                                 Message ID Count  Time
Note (9)                                                                                             9
  ** Note: (vsim-3813) Design is being optimized due to module recompilation...            3813      1      0
  UVM_INFO @ 0: reporter [RNTST] Running test my_test...                                   8330      1      0
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(197) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_driver [bcd_driver] Driver received sequence item. Binary:
  0000000000
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(252) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_monitor [bcd_monitor] Monitor output: binary: 0000000000
  hundreds: 0000, tens: 0000, ones: 0000
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(197) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_driver [bcd_driver] Driver received sequence item. Binary:
  0000000001
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(252) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_monitor [bcd_monitor] Monitor output: binary: 0000000001
  hundreds: 0000, tens: 0000, ones: 0001
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(197) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_driver [bcd_driver] Driver received sequence item. Binary:
  0000000010
  UVM_INFO C:/Users/tdiedere/OneDrive -
          /Documents/Quartus_Projects/Questa_Projects/UVM BCD 2.0/Hex_Logic_UVM_tb_Rdev.sv(252) 8330  1      0
  @ 0: uvm_test_top.m_env.m_bcd_agent.m_monitor [bcd_monitor] Monitor output: binary: 0000000010
  hundreds: 0000, tens: 0000, ones: 0010
  $finish   : C:/intelFPGA_pro/22.4/questa_fse/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430) 3963  1  0
```

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).

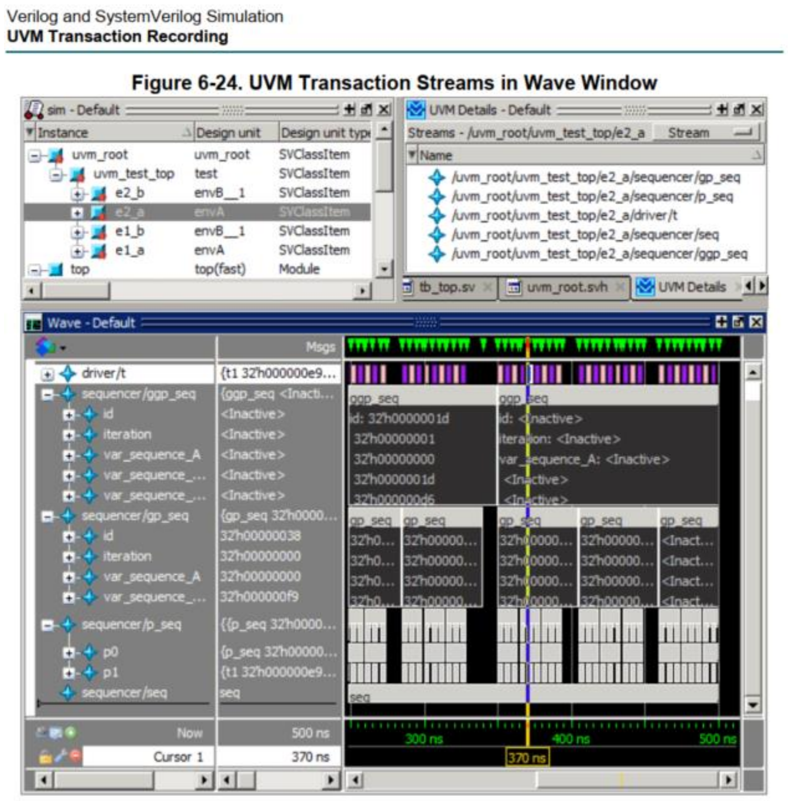

Verilog and SystemVerilog Simulation
UVM Transaction Recording

Figure 6-24. UVM Transaction Streams in Wave Window

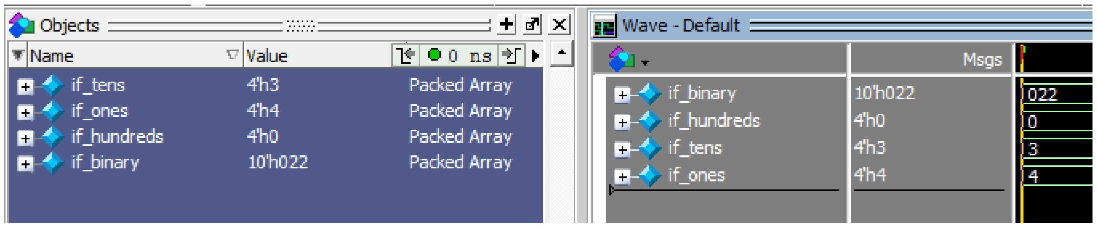
```
Streams - /uvm_root/uvm_test_top/e2_a
/uvm_root/uvm_test_top/e2_a/sequencer/gp_seq
/uvm_root/uvm_test_top/e2_a/sequencer/p_seq
/uvm_root/uvm_test_top/e2_a/driver/t
/uvm_root/uvm_test_top/e2_a/sequencer/seq
/uvm_root/uvm_test_top/e2_a/sequencer/ggp_seq
```

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.

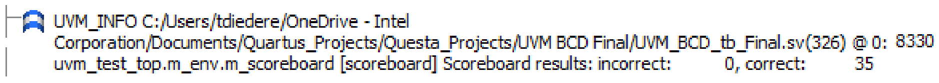

Objects

Name	Value	
if_tens	4'h3	Packed Array
if_ones	4'h4	Packed Array
if_hundreds	4'h0	Packed Array
if_binary	10'h022	Packed Array

Wave - Default

	Msgs	
if_binary	10'h022	022
if_hundreds	4'h0	0
if_tens	4'h3	3
if_ones	4'h4	4

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

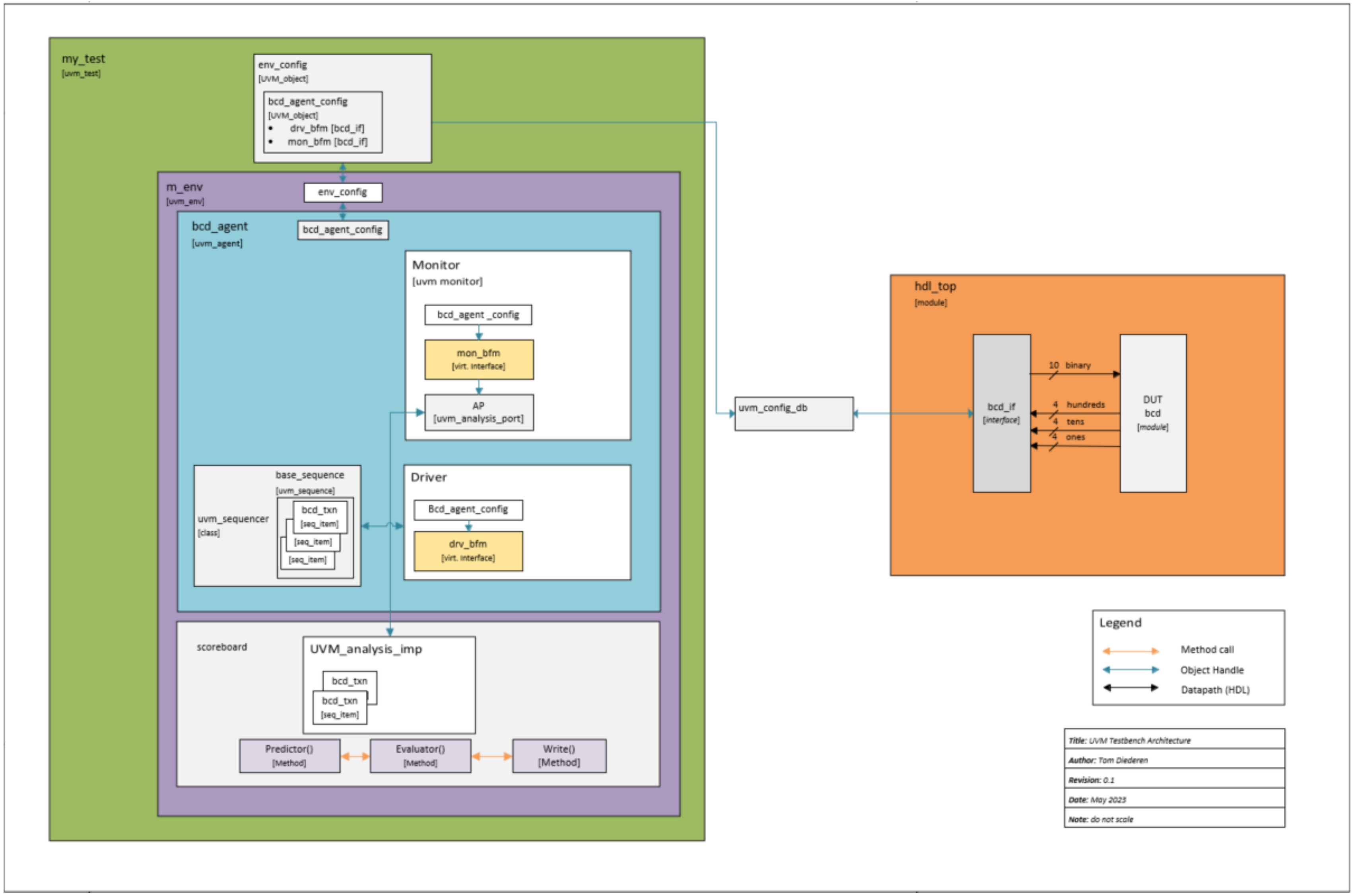
[1] "Cyclone V Logic Solver (Design Example)," [Online]. Available: https://www.intel.com/content/www/us/en/design-example/714798/cyclone-v-logic-solver.html.

[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/.

APPENDIX I TESTBENCH ARCHITECTURE



my_test
[uvm_test]
env_config
[UVM_object]
bcd_agent_config
[UVM_object]
drv_bfm [bcd_if]
mon_bfm [bcd_if]
m_env
[uvm_env]
env_config
bcd_agent
[uvm_agent]
bcd_agent_config
Monitor
[uvm monitor]
bcd_agent _config
mon_bfm
[virt. interface]
AP
[uvm_analysis_port]
base_sequence
[uvm_sequence]
uvm_sequencer
[class]
bcd_txn
[seq_item]
[seq_item]
[seq_item]
Driver
Bcd_agent_config
drv_bfm
[virt. interface]
scoreboard
UVM_analysis_imp
bcd_txn
bcd_txn
[seq_item]
Predictor()
[Method]
Evaluator()
[Method]
Write()
[Method]
uvm_config_db
hdl_top
[module]
bcd_if
[interface]
10 binary
4 hundreds
4 tens
4 ones
DUT
bcd
[module]
Legend
Method call
Object Handle
Datapath (HDL)
Title: UVM Testbench Architecture
Author: Tom Diederen
Revision: 0.1
Date: May 2023
Note: do not scale