UVM/SystemVerilog based infrastructure and testbench automation using scripts

Prakash Parikh,
Aquantia Inc.,
700 Tasman Drive,
Milpitas, CA
pparikh@aquantia.com

Abstract :

Universal Verification Methodology (UVM) and SystemVerilog based testbenches are widely used in many of the new ASIC starts. Quite a few legacy Verilog based testbenches, Vera, Testbuilder, Specman "e" and SystemC based testbenches are being migrated to UVM based testbenches. Developing UVM based testbenches and infrastructure from scratch is time consuming and requires a good understanding of Object Oriented Programming from the entire team responsible for UVM based verification. Automation for block and top level verification testbenches based on Perl or C shell scripts can provide powerful methods to get the block and top level infrastructure ready in a short period of time while various verification team members get familiar with new concepts of UVM based verification methodology. This allows extra time for verification engineers to focus more on actual block verification aspects instead of spending more cycles on infrastructure development.

This paper discusses how the DSP (Digital Signal Processing) verification environment for the 10GBase-T chip was generated using the perl scripts. Further, it discusses how the driver, monitor, sequences, sequencer, agent, env and test classes generated from scripts. Finally it discusses how the verification is performed and how coverage data are generated.

I. INTRODUCTION:

As UVM and SystemVerilog become more widely adopted in the industry, companies need to schedule significant amounts of time for their engineers to master the learning curve inherent in UVM, SystemVerilog, and even object-oriented programming. Worse, it is easy for engineers to make mistakes when coding up the first project in a new language. These mistakes cost debug time and can affect testbench coverage and quality. Also, whenever a testbench is composed of checkers coded up by more than one engineer, top level hookups can be inconsistent and error prone.

One way to mitigate some of the schedule and quality issues is to use a scripting based approach to create the top level and even some of the lower level structural code used in the system. This ensures a uniform coding style, consistent hookups, and enables the verification engineers to spend more time focusing on the actual block/system verification instead of infrastructure development.

There are two options available at the beginning of the project to create a verification environment. The first approach is to create individual block level environments hand coded from scratch and then stitch individual block level environment to create the top level environment. None of the code can be reused in this approach. The time it takes to create such an environment is proportional to the number of

blocks in the environment. The second approach is to take advantage of similarity in the different blocks functionality that need to be verified and use scripting methods to generate various UVM components template codes and create uniform block level environment that is easily scalable to top level environment. The approach of generating infrastructure using scripts was followed for creating a DSP based verification environment.

II. TOP LEVEL DSP SYSTEM:

The following diagram (Figure 1) shows a typical block diagram of the DSP based system for which the UVM infrastructure was developed.

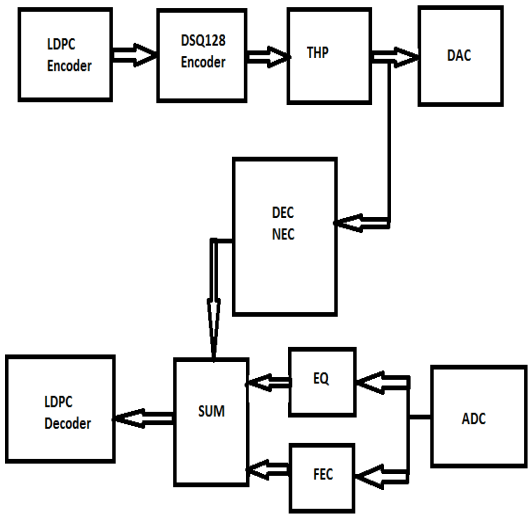

Figure 1: Block Diagram of the DSP system

Data Flow for the system shown in Figure 1 above is described below.

► The LDPC (Low density parity check) encoded data gets mapped to DSQ (Double Square Constellation) symbols.

► THP (Tomlinson Harashima Precoder) operation is performed on the mapped symbols.

► The transmit data is sent out after the DAC (digital-to-analog converter) operation.

► On the received side, the received data after ADC (analog-to-digital conversion) passes through EQ (Equalization) and FEC (Far end cross talk cancellation).

►DEC (digital echo cancellation) and FEC (Far end cross talk cancellation) filtering is performed and output of DEC and NEC is added to the received data that undergo the EQ (Equalization) and FEC (Far end cross talk) filtering.

► The received data after DEC/NEC/EQ/FEC cancellation get decoded by LDPC decoder.

III. UVM TESTBENCH FOR DSP SYSTEM:

The following diagram (Figure 2) shows UVM agent for individual FFE/FEC/DEC/NEC filter blocks. This follows standard UVM methodology for block level testbench and infrastructure development.

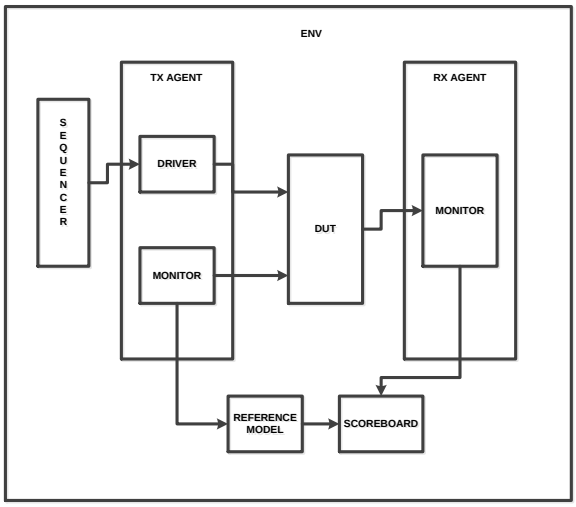

Figure 2: Block level Environment

Each block level environment instantiates both a transmit agent and a receive agent. The Transmit Agent instantiates driver and transmit monitor. The Receive Agent instantiates receive monitor. Data collected by the transmit monitor gets passed to the reference model and the reference model output data is provided to the scoreboard. The data collected by the Receive Agent monitor is also provided to the scoreboard. The scoreboard does the comparison

between the reference model data and the Receive Agent monitor data.

Following (Figure 3) shows a picture of top level env that instantiates various block level envs shown in Figure 2 above.

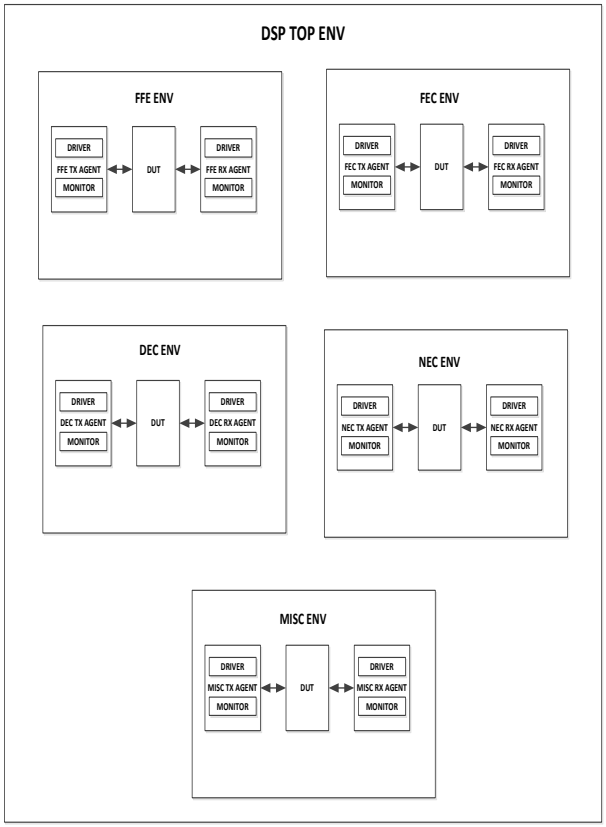

Figure 3: Top Level Environment

As shown in Figure 3 above, the top level DSP system instantiates FFE (Equalizer) env, FEC env, DEC env, NEC env and other miscellaneous env. For developing UVM infrastructure environment for the above complex system with strict deadlines, writing all the UVM driver, monitor, sequence, agent, etc. code for each of the block was not practical. So the following perl based methodology was used to generate each block UVM infrastructure code.

The script takes various arguments for the block specification and generates templates for the UVM drivers, monitors, sequences, sequencers, scoreboards, coverage analysis, TX and RX agents, environments with default implementations of build, connect, run phases for each of these UVM components. As seen in the top level environment Figure 3 above, the system consists of various filters, FFE/FEC/DEC/NEC. These filters are all FIR filters but may vary in tap lengths and data and coefficients widths. The stimulus needed to verify these blocks with directed and random patterns could be similar. All the filters can use Impulse, DC, Incrementing, random, etc. patterns. Using perl scripts it is easy to generate templates for specific filters with parameters indicating name of the filter, transaction type and data and coefficients widths. The template code generated for driver, sequences and other UVM components is described in next sections.

IV. DRIVER/MONITOR TEMPLATE CODE GENERATION :

Following shows the template of the UVM driver code generated using the script for the DSP chip block. Figure 4 below shows portion of perl script that is used for generating the driver code. The script can take block name, transaction type, bit widths for different inputs and outputs as arguments for the driver template generation.

```
#!/usr/bin/perl
$block_name = $ARGV[0];
$tx_type = $ARGV[1];
…
gen_template($block_name);
sub gen_template {
  create_header($block_name);
  create_uvm_class($block_name, $tx_type);
};
sub create_header {
   printf"\n/*************************************";
   printf "\n* Description :UVM driver for block
$block_name";
   printf"\n*************************************/\n";
};
sub create_uvm_class {
  printf "\n\nclass ${block_name}_driver extends
uvm_driver # ($tx_type); \n\n" ;
  printf "\n
`uvm_component_utils(${block_name}_driver)";
  printf "\n $tx_type tx_data;";
  printf "\n virtual interface ${block_name}_if_data_input
dif_data;";
```

```
  printf "\n virtual interface ${block_name}_if_ctrl_input
dif_ctrl;";
  printf "\n uvm_analysis_port #(${tx_type})
driver_port;\n";
  create_new($block_name);
  create_build_phase($block_name);
  create_run_phase($block_name);
  create_drive_dut($block_name);
  create_bind_vis($block_name);
  printf "\n\nendclass : ${block_name}_driver \n\n" ;
};
sub create_new {
  printf "\n function new (string name =
\"${block_name}_driver\", uvm_component parent);";
  printf "\n\n super.new(name, parent);";
  printf "\n\n endfunction \: new \n\n";
};
sub create_build_phase {
  printf "\n function void build_phase(uvm_phase phase);";
  printf "\n\n string name;";
  printf "\n super.build_phase(phase);";
  printf "\n driver_port = new(\"driver_port\", this);";
  printf "\n\n endfunction \: build_phase\n\n";
};
sub create_run_phase {
  printf"\n task run_phase(uvm_phase phase);";
  printf"\n\n super.run_phase(phase);";
  printf"\n @(posedge dif_data.reset_n_i);";
  printf"\n @(aq_dsp_config::fw_init_done);";
  printf"\n //Drive data now";
  printf"\n forever";
  printf"\n begin";
  printf"\n @(posedge dif_data.clk_dsp);";
  printf"\n seq_item_port.try_next_item(tx_data);";
  printf"\n drive_dut();";
  printf"\n driver_port.write(tx_data);";
  printf"\n seq_item_port.item_done();";
  printf"\n end";
  printf"\n\n endtask : run_phase";
};
sub create_drive_dut {
  printf "\n\n function void drive_dut();\n";
  printf "\n dif_data.a = tx_data.a;";
  printf "\n\n endfunction \: drive_dut\n\n";
};
sub create_bind_vis {
  printf "\n\n function void bind_vi_data(virtual interface
${block}_data_input vif );\n";
  printf "\n dif_data = vif;";
  printf "\n\n endfunction \: bind_vi_data\n\n";
  printf "\n\n function void bind_vi_ctrl(virtual interface
${block}_ctrl_input vif );\n";
  printf "\n dif_ctrl = vif;";
  printf "\n\n endfunction \: bind_vi_ctrl\n\n";
};
```

Figure 4: Perl code for generating UVM template

Perl script takes argument as block name, for ex. FFE and then generates the template code for driver, monitor, sequence, sequencer, TX agent, RX agent, etc. Similarly infrastructure code is generated for other UVM modules.

```
class aq_dsp_ffe_driver extends uvm_driver #
(aq_dsp_ffe_transaction_in);

`uvm_component_utils(aq_dsp_ffe_driver)

aq_dsp_ffe_transaction_in tx_data;
virtual interface aq_dsp_ffe_if_data_input dif_data;
virtual interface aq_dsp_ffe_if_ctrl_input dif_ctrl;
uvm_analysis_port #(aq_dsp_ffe_transaction_in)
driver_port;

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

function void build_phase(uvm_phase phase);
  string name;
  super.build_phase(phase);
  driver_port = new("driver_port", this);
endfunction


function void bind_vi_data(virtual interface
aq_dsp_ffe_if_data_input vif);
  dif_data = vif;
endfunction

function void bind_vi_ctrl(virtual interface
aq_dsp_ffe_if_ctrl_input vif);
  dif_ctrl = vif;
endfunction

task run_phase(uvm_phase phase);
  super.run_phase(phase);
  @(posedge dif_data.reset_n_i);
  @(aq_dsp_config::fw_init_done);

  //Drive data now
  forever
    begin
      @(posedge dif_data.clk_dsp);
      seq_item_port.try_next_item(tx_data);
      drive_dut();
      driver_port.write(tx_data);
      seq_item_port.item_done();
    end
```

```
endtask : run_phase

function void drive_dut();
  dif_data.rdl_ffe_rx_data_i[0] =
tx_data.rdl_ffe_rx_data_i[0];
endfunction

endclass: aq_dsp_ffe_driver
```

Figure 5: UVM Driver code

As seen in the template code in the Figure 5 above, the "driver" class definition along with build phase, run phase, DUT driver code, etc. are generated through script. Verification engineers can now focus on writing or expanding "drive_dut" function above for the DUT requirements instead of spending time on defining build/run phases, new function, call to `uvm_component_utils.

Similarly, template code can be generated for UVM monitors, UVM sequences, UVM scoreboards, UVM agents, UVM envs, UVM test classes, etc.

V. SEQUENCE TEMPLATE CODE GENERATION :

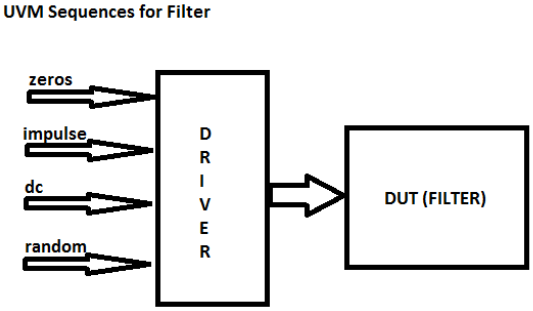

Figure 6: Filter UVM Sequences

For the top level DSP system used, DSP filter blocks have typical sequences such as zeros, impulse, incrementing, DC, decrementing, single tone, random, max positive and max negative. There are sequences needed for developing the testbench with easy debug and with the final goal of reaching towards random stimulus sequence. Since code for all the typical sequences mentioned above gets generated through scripts, it allows enough time to generate sequences for directed tests specific to particular functionality of the block. Following shows an example of code generated using the script. The perl script takes arguments for different sequences to be generated for a given filter and generates the following code. The following example shows DC sequence generated for a DEC filter.

```
class aq_dsp_ffe_sequence_base extends uvm_sequence
#(aq_dsp_ffe_transaction_in);

`uvm_object_utils(aq_dsp_ffe_sequence_base)

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

virtual task body();
endtask : body

endclass : aq_dsp_ffe_sequence_base

// DC scenario
class aq_dsp_ffe_sequence_dc extends
aq_dsp_ffe_sequence_base;

`uvm_object_utils(aq_dsp_ffe_sequence_dc)

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

task pre_body();
  if(starting_phase != null) begin
    starting_phase.raise_objection(this);
  end
endtask

virtual task body();
  repeat(no_of_iterations)
  begin
    `uvm_do_with(req,
  {
    req.rdl_ffe_rx_data_i[0] == ( DC_VAL );
  });
  end
endtask : body

task post_body();
  if(starting_phase != null) begin
    starting_phase.drop_objection(this);
  end
endtask
```

```
endclass : aq_dsp_ffe_sequence_dc
```

Figure 7: UVM sequence code for filter

The script can also be used to generate the base class for all FIRs sequences. This base class can contain all important sequences such as zeros, DC, impulse, incrementing and random. The sequence class template generated from the script can have the class derived from above base class.

Once the code is generated for driver, monitor, agents and sequences, the perl script takes arguments for different block level environments to be instantiated and generates the DSP top environment code. This DSP top environment can either instantiate block level agents or instantiate block level env directly based on the methodology followed. Finally the top env is instantiated in the UVM test code. Following screen shot shows example of DSP top env code generated through script.

```
class aq_dsp_env extends uvm_env;

`uvm_component_utils(aq_dsp_env)

//Instantiate all the agents here.
aq_dsp_ffe_agent_tx m_ffe_agent_tx;
aq_dsp_ffe_agent_rx m_ffe_agent_rx;
aq_dsp_ffe_sb m_sb;
aq_dsp_ffe_coverage m_cov;

aq_dsp_fec_sb m_sb_fec;
aq_dsp_fec_coverage m_cov_fec;
aq_dsp_fec_agent_tx m_fec_agent_tx;
aq_dsp_fec_agent_rx m_fec_agent_rx;

  //Declare all interfaces
virtual aq_dsp_ffe_if_data_input vif_data_input;
virtual aq_dsp_ffe_if_ctrl_input vif_ctrl_input;
virtual aq_dsp_ffe_if_output vif_output;

virtual aq_dsp_fec_if_data_input vif_fec_data_input;
virtual aq_dsp_fec_if_ctrl_input vif_fec_ctrl_input;
virtual aq_dsp_fec_if_output vif_fec_output;

function new (string name = "aq_dsp_env",
uvm_component parent);
  super.new(name, parent);
  `uvm_info(get_name()," Env New ",
UVM_MEDIUM );
endfunction

function void build_phase(uvm_phase phase);
  string name;
  super.build_phase(phase);
   m_ffe_agent_tx =
aq_dsp_ffe_agent_tx::type_id::create("m_ffe_agent_t
x", this);
  m_ffe_agent_rx =
aq_dsp_ffe_agent_rx::type_id::create("m_ffe_agent_r
x", this);
  m_sb = aq_dsp_ffe_sb::type_id::create("m_sb",
this);
  m_cov =
aq_dsp_ffe_coverage::type_id::create("m_cov", this);

  m_fec_agent_tx =
aq_dsp_fec_agent_tx::type_id::create("m_fec_agent_
tx", this);
  m_fec_agent_rx =
aq_dsp_fec_agent_rx::type_id::create("m_fec_agent_
rx", this);
  m_sb_fec =
aq_dsp_fec_sb::type_id::create("m_sb_fec", this);
  m_cov_fec =
aq_dsp_fec_coverage::type_id::create("m_cov_fec",
this);
….

function void end_of_elaboration_phase(uvm_phase
phase);
  uvm_top.print();
endfunction : end_of_elaboration_phase

function void report_phase(uvm_phase phase);
    report_summarize();
endfunction : report_phase


endclass: aq_dsp_env
```

Figure 8: UVM Env code

VI. LIMITATIONS

The scripting method for generating the sequencer code for similar blocks such as FFE/FEC/DEC/NEC worked very well. There were other blocks such as RDL (Receive Delay Line) and AIF (Analog Interface) blocks in the DSP system. RDL block in the DSP system is responsible for sum of DEC, NEC, FFE and FEC filtered data. This RDL block functionality is much different from FIR filter functionality. In AIF block, there is glue logic for passing data from the digital domain to the analog domain. This functionality is also much different than FIR filter functionality. There are no data and coefficients interfaces for RDL/AIF blocks unlike FIR filters. The sequences that need to be generated for RDL and AIF blocks are much different from FIR filter sequences. In this scenario the same scripting code cannot be reused to the same extent as in the FIR filters sequences code.

In the project, there can be requirements to change the infrastructure code once block level verification gets started and new requirements are identified. Knowledge about the infrastructure and corresponding changes could be restricted to a single person owning the infrastructure development scripts and this can become a bottleneck in some cases as compared to distributed knowledge across the entire verification team.

VII. RESULTS :

When there is similarity in functionality and hence testing methodology between different blocks of the top level system such as DEC/FEC/FFE/NEC filters in the DSP system, the script approach to create various templates codes for various UVM components such as driver, monitor, sequences, agents and envs is very useful. Approximately 60% of the code can be generated with this template and it requires filling the remaining 40% of the code with the block specific logic. Overall it saves a significant amount of time in coding.

Generation of block level testbenches takes a day using UVM automation scripts. Most of the randomized and directed tests are generated by these scripts based on the parameters provided to the scripts. The entire infrastructure for the UVM based environments can be developed in a matter of a few weeks instead of few months. Since all the block level testbenches are generated using scripts in an automated way, it imposes consistency in verification methodology for all the blocks of the chip and also between block level and top level verification. Scaling of the block level testbench to top level testbench becomes much easier since each block of the top level system uses the same modeling and verification infrastructure. Time saved in infrastructure development can be effectively used in thorough verification for block/top level verification.

These scripts can be modified further to generate a full chip environment where DSP top environment of the PHY (physical) layer and MAC (Media Access Control) layer top environment can be instantiated for the full chip verification. This allows easily reusable and uniform block level verification environment that can be scaled to DSP top level environment and is further scalable to a full chip PHY-MAC verification environment.

VIII. CONCLUSION:

Block and top level verification infrastructure development can be automated using scripts. This saves a significant amount of time that can be used for actual verification of the design.

IX. REFERENCES:

1) Verification methodology cookbook, UVM, Mentor Graphics
2) UVM Golden Reference Guide, Doulos
3) 10GBASE-T, IEEE 802.3an, IEEE standard document.