UVM/SystemVerilog based infrastructure and testbench automation using scripts

Prakash Parikh
Aquantia Inc.

Background

- Aquantia – Privately held company
- Delivers High Speed Ethernet connectivity solutions for large-scale Data Centers and Cloud computing
- UVM infrastructure was generated for the PHY chips using the scripting approach.
- Chips successfully taped out with the verification done using this UVM infrastructure

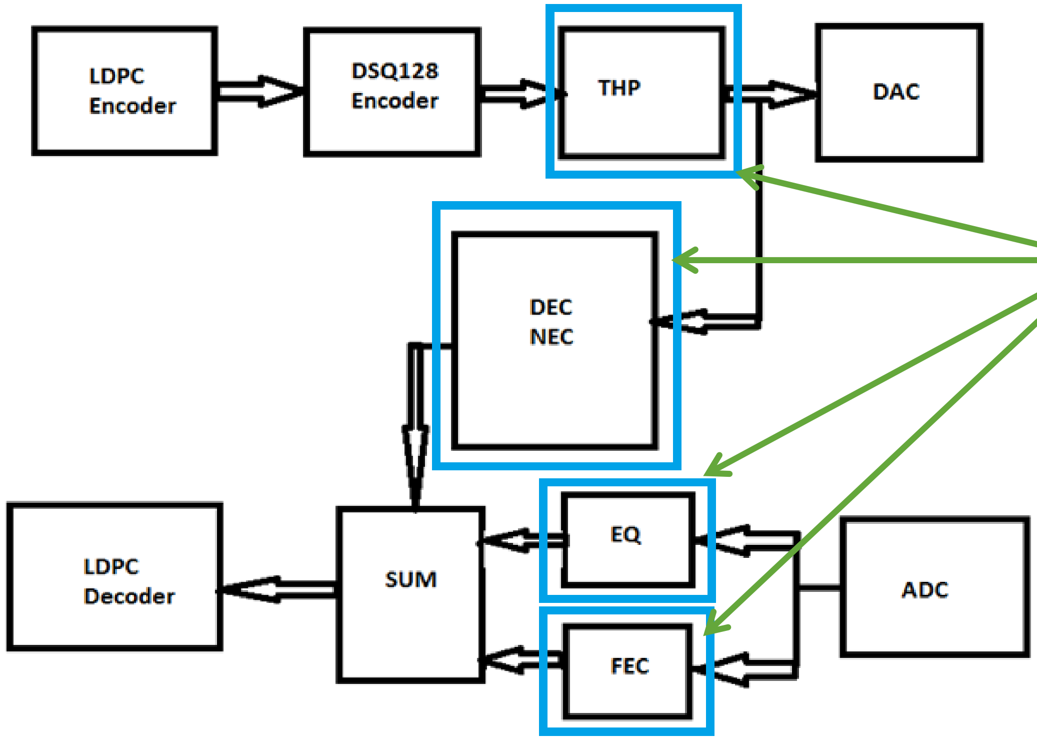
Why script based approach for UVM?

- UVM / SystemVerilog based testbenches gaining popularity
- Legacy SystemC, Vera, testbuilder, Specman “e” based testbenches getting migrated to UVM. Manual approach is errorprone
- Infrastructure development for UVM based testbench is time consuming and requires a lot of OOP knowledge
- Block level env needs to be uniform to be scalable at the top level
- Ensures uniform coding style and consistent hookups

UVM infrastructure development

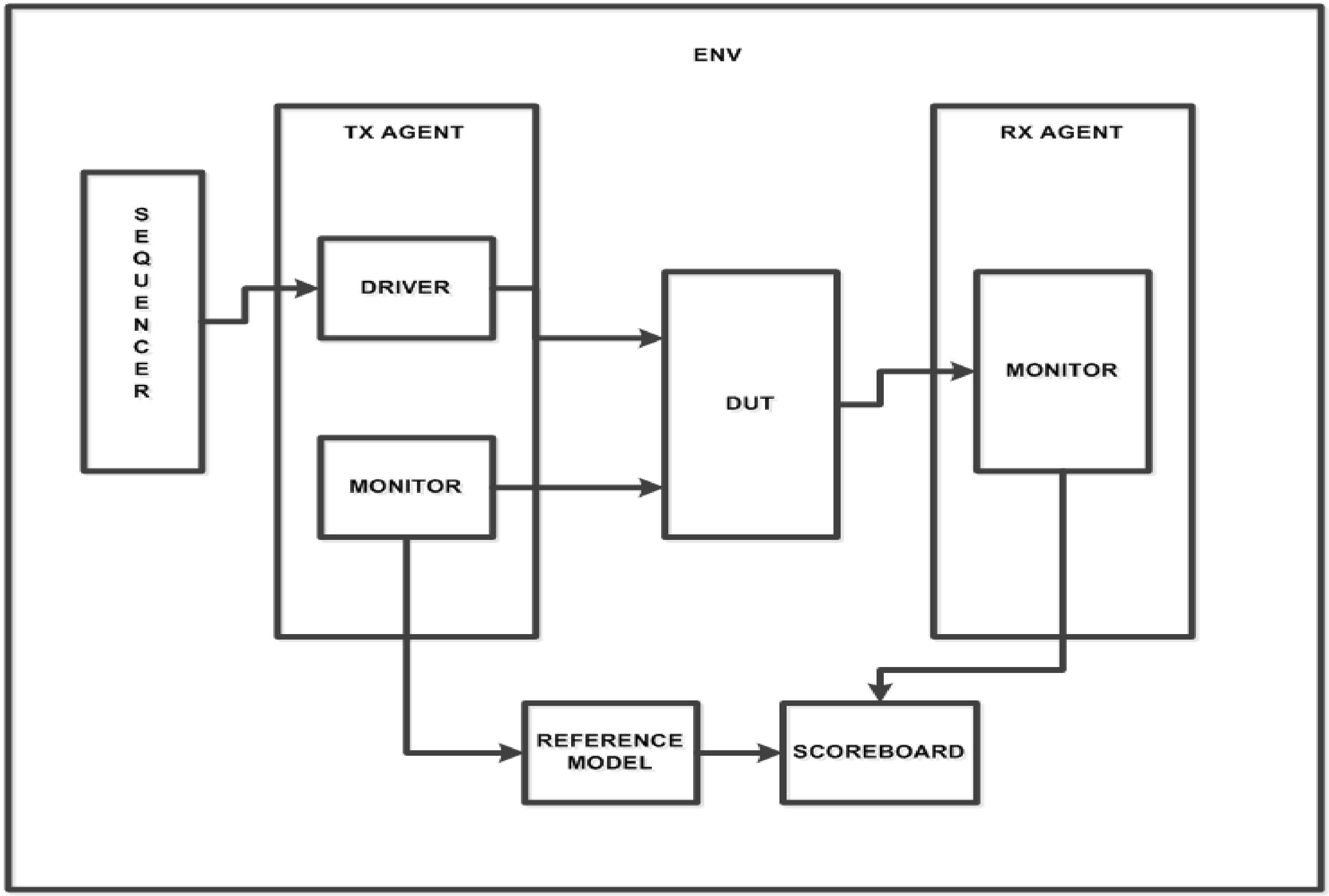
- Option 1 :
  - Develop each block level env from scratch
  - Stitch each block level environment for top
  - Time consuming
  - Not easily scalable since each env is not uniform.
- Option 2 :
  - Take advantage of similarity of the functionality
  - Generate block and top level environment using scripting approach.
  - Advantage : Faster and Scalable approach.

Script based approach for DSP system

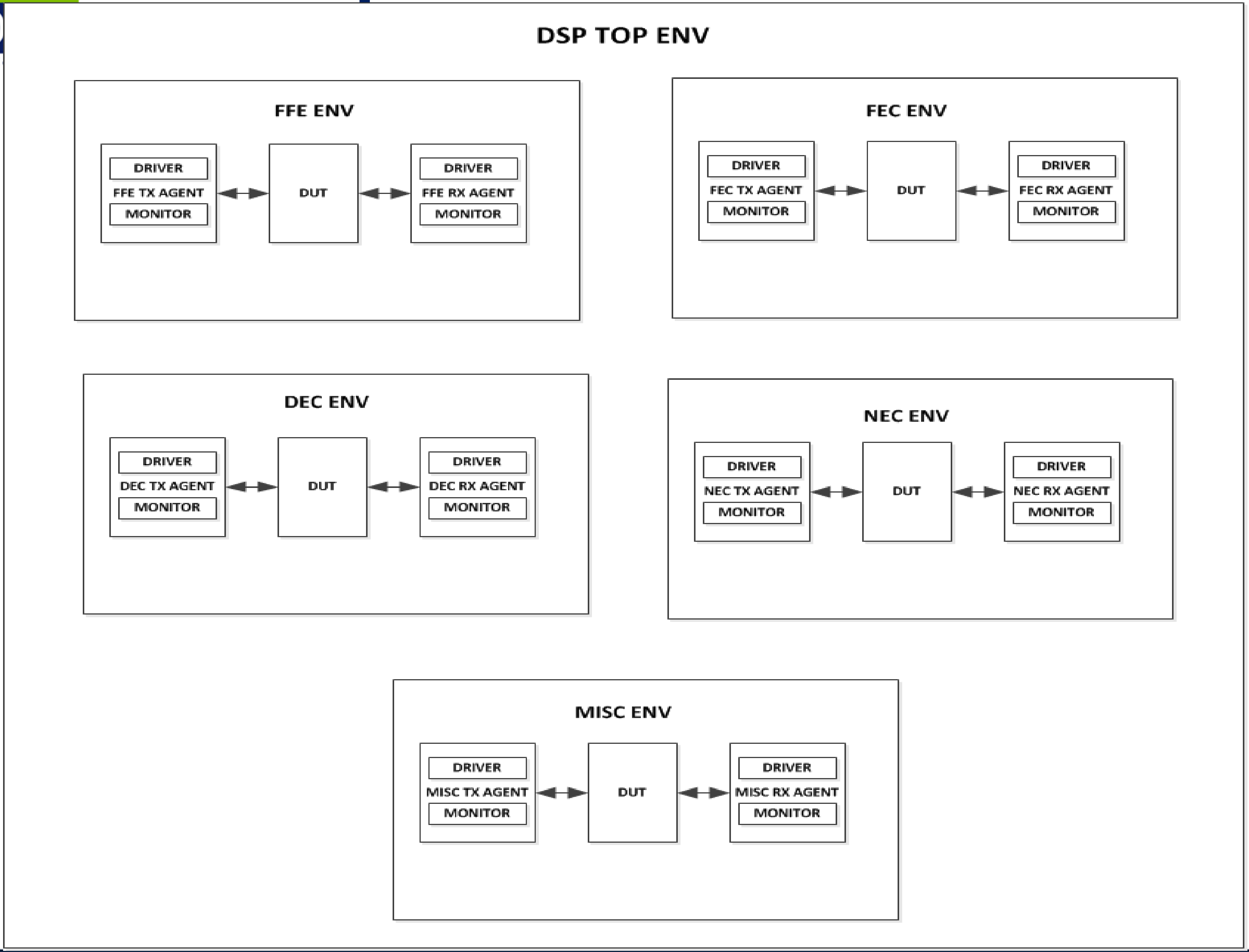


Functionally similar blocks

Standard Block Environment



Top level Environment



Template code for all blocks

- Generate shells for driver/monitor/agent/env/test classes
- Generate build, run, connect phases standard code
- Generate `uvm_component_utils(), data and control interfaces code.

Sample code for scripting method

Same template for all blocks uvm components

- **Class template with generation of functions and tasks definitions.**

```
sub create_uvm_class {
  printf "\nclass ${block_name}_driver extends uvm_driver # ($tx_type); "
  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" ;
};
```

```
$block_name = $ARGV[0];
$tx_type = $ARGV[1];
```

Sample code for scripting method - Continue

```
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 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";
};
```

Similar Driver phases for all blocks of the chip

```
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";
  };
```

Same Data and control interfaces

Similarity of the block level environments

- FFE, FEC, DEC and NEC filters.
- Each filter vary in datapath and coefficients widths
- Verification of each block, with DC, Incrementing, Impulse and random sequences.
- Corner cases same – Max positive, Max negative combinations for data and coefficients
- Saturation cases same

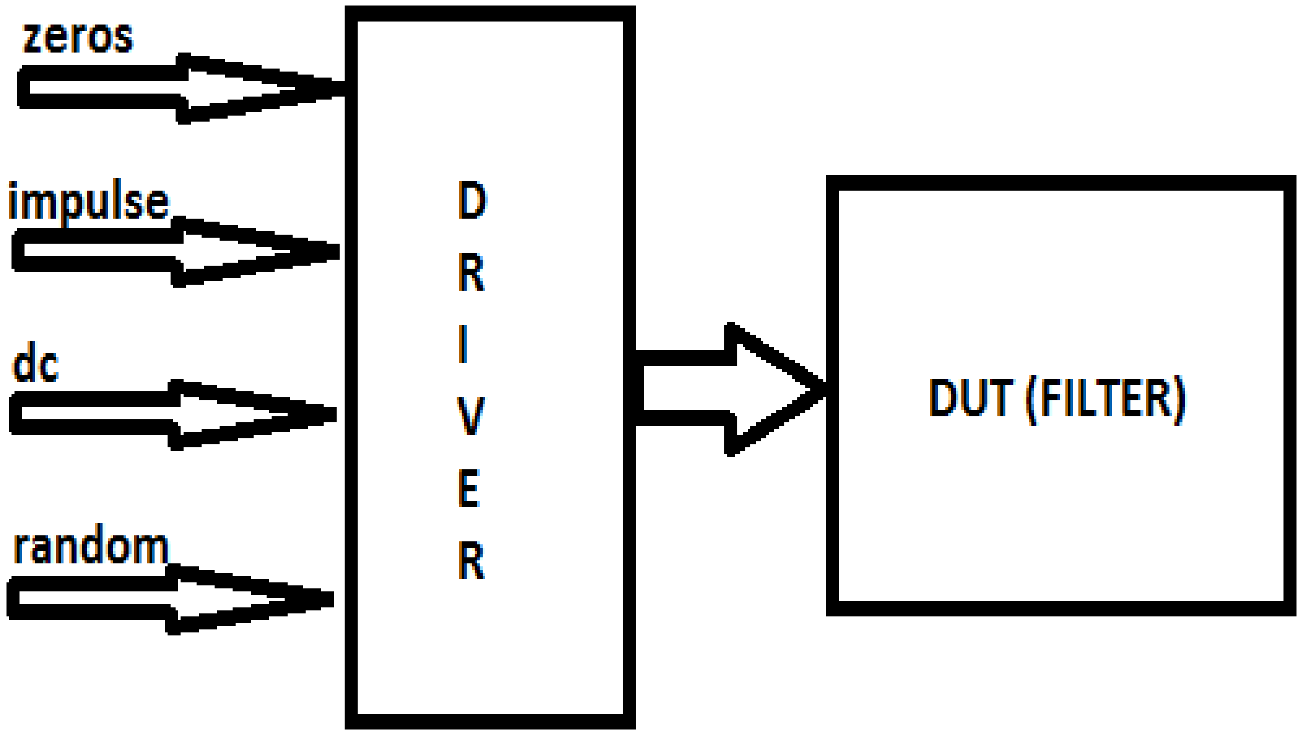
Sample output code from scripts

```
•   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

     task run_phase(uvm_phase phase);
      super.run_phase(phase);
          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
```

FIR-IIR Filter sequences

UVM Sequences for Filter



Sequence code from script – DC / IMP / INCR sequences code reused

```
•   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
```

DC Sequence can be reused for all filters

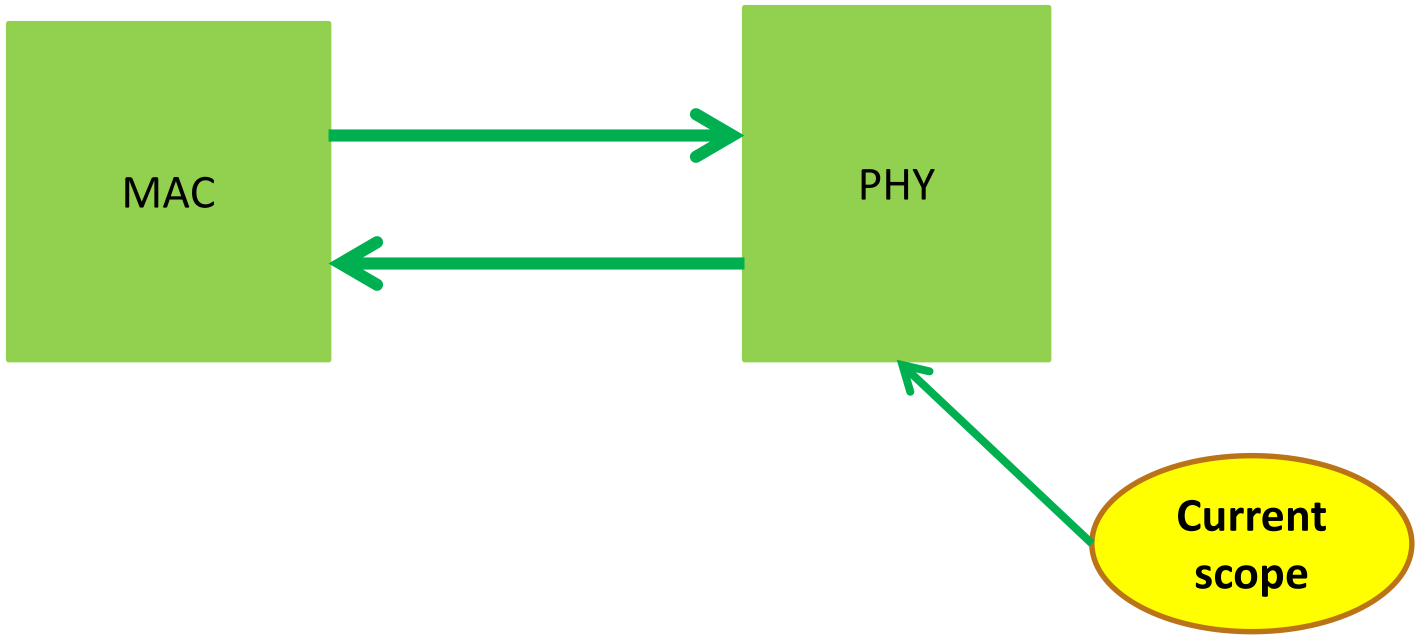
All filters have similar DC/IMP/INCR/SINGLE_TONE/RAND sequences

Limitations of the scripting approach

- The scripting method for generating the sequencer code for similar blocks such as FFE/FEC/DEC/NEC worked very well.
- Blocks such as RDL (Receive Delay Line) and AIF (Analog Interface) blocks in the DSP system could not reuse the functionality code except template code for various UVM phases.
- Infrastructure knowledge limited to a single person or small team.
- Infrastructure changes late in the design cycle difficult since blocks env have already been developed.

Future work and improvements

- Enhance UVM environment for PHY top by adding MAC where PHY stimulus is generated by MAC.
- Generate MAC Env and DSP Env for PHY from scripts

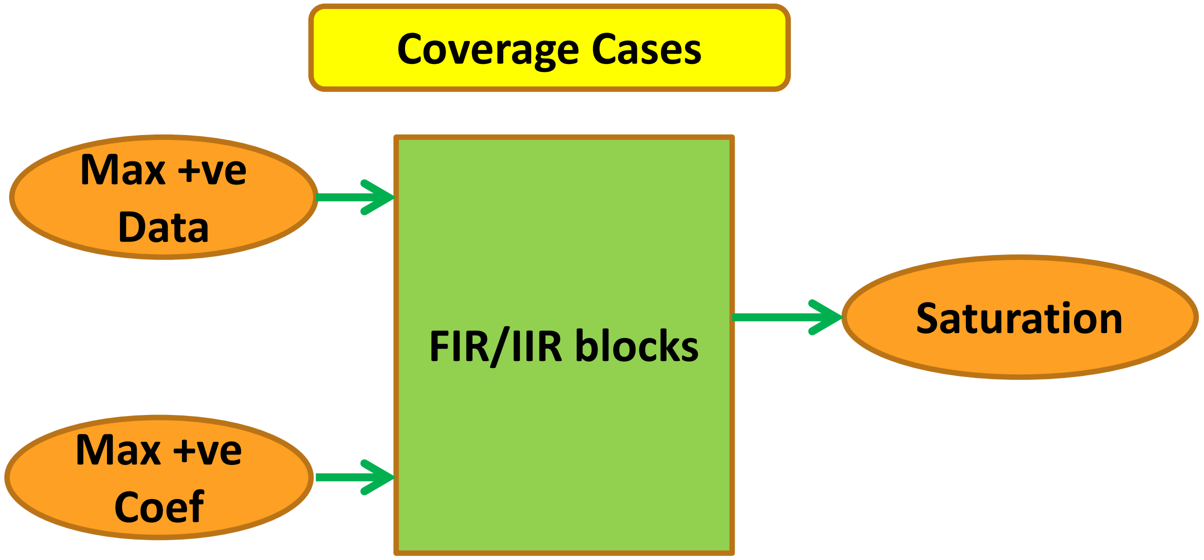


Future work and improvements

- Put more common functionality in the base class for driver, monitor, sequence when using the scripting method.
- Careful study of functionality in base class compared to functionality in different blocks scripts generated code

Future work and improvements

- Generate coverage class using scripting method for all FFE/FEC/NEC/DEC blocks that have similar coverage criterion.



Future work and improvements

- Extend scripts flow to generate Makefile, reference models, DPI interface calls on top of UVM infrastructure files generation.
- Generate templates for UVM infrastructure
- Generate reference code wrappers and DPI calls
- Generate Makefiles for compiling UVM infrastructure, reference code, running regressions
- Automate entire DV regressions infrastructure with scripts

Summary

- Block and top level verification infrastructure development can be automated using scripts
- Significant amount of time saved can be used for actual verification
- Block level environment is easily scalable to top level environment because of the uniformity of all the block level environments.
- Overall, few weeks of time saved in verification effort for the DSP chip.
- Tapeout on time with bug free silicon!