2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

An Elegant scoreboard eco-system deploying UVM Callbacks, Parameterization for Multimedia designs from Imaging perspective

Chakravarthi Devakinanda Vurukutla, Samsung Semiconductor India Research, chakravart.d@samsung.com

Sahana Ranganathan, Samsung Semiconductor India Research, sahana.ranga@samsung.com

Devendra Satish Bilaye, Samsung Semiconductor India Research, dev.bilaye@samsung.com

Vivek Kumar, Samsung Semiconductor India Research, vivek9.kumar@samsung.com

Karthik Majeti, Samsung Semiconductor India Research, k.majeti@samsung.com

***Abstract*— One of the principal features driving mobile market is the camera specification. Based on the needs of consumer market, there is a demand for camera sensors from the entry level to premium ones. An Image Sensor SOC will have many image processing sub-blocks to capture a high quality image. A CMOS image sensor has an Analog component which captures light, converts it to digital value and feeds it to digital pipeline. There are noises involved in Analog operation and manufacturing defects during fabrication. The digital pipeline has to mitigate these noises and defects for a high quality image output. Finally, the high quality image is sent to the host via Industry standard serial protocol. For a high quality image, advanced image processing IPs and algorithms are developed, resulting in increased design complexity. Additionally, the ever increasing need for higher resolution Image Sensors leads to larger designs. Due to market demand and aggressive schedules, the timelines for Design Verification shrink along with emphasis on quality validation. The key component for verification of any ISP (Image Signal Processing) algorithm is scoreboard, where we compare RTL output against golden reference model. A robust, configurable scoreboard architecture is vital to meet the verification quality in a short time frame. This architecture needs to be reusable across all sub-blocks. This paper details the practices followed in scoreboard implementation to achieve high quality verification within the stipulated time.**

Keywords*— *scoreboard; callbacks; verification; parameterization; image sensor; multimedia; adaptability; scalability; reusability;

I. INTRODUCTION

A typical CMOS Image Sensor (Figure 1.) has Analog and Digital components. Analog components consist of the Active Pixel sensor(APS) and Analog-to-Digital converters(ADC). An APS can be considered as 2d-array with rows and columns. Each entry in an array is a photo-diode. This photo-diode along with analog circuitry is called as pixel. Pixel is the building block of Image Sensor which captures light and converts to electric signal. These electric signals are converted to digital values using ADCs. The timing block in digital logic generates the control signals to Analog, extracts the information from pixel array and sends the data to digital pipeline. Different kinds of Analog noises emerge during the operation of pixels and ADCs. Defects like dead-pixel arise while manufacturing. The digital pipeline, which has the image processing algorithms has to alleviate these noises and defects for a high quality image output. The processed image data is sent to the host via industry standard serial protocol. Every complete readout of APS is termed as a Frame. Sending out first pixel indicates the start of the frame and the last pixel, as end of the frame. Based on the application and use-case, multiple frames can be readout. The number of such frames captured in a second is called FPS (Frames per Second).

There are multiple sub-blocks involved in image processing to mitigate different kinds of noises and defects. To validate the functionality of these image processing blocks, we use the reference models in our scoreboard. A typical scoreboard architecture has been depicted in Figure 2.

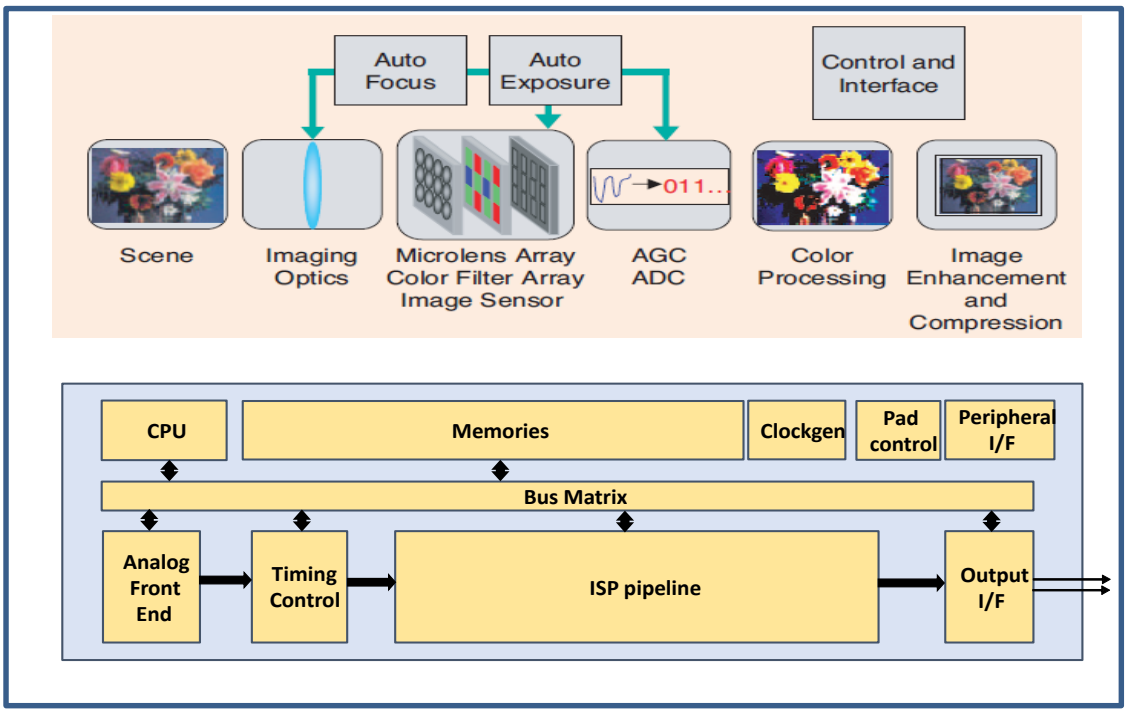

Figure 1. A typical CMOS Image Sensor

A high level language is used to implement the image processing algorithms, which serve as golden reference models. Generally, the reference model is written in 'C'. In scoreboard, the RTL is qualified using the C-models. Each sub-block has an individual scoreboard. For the industry standard output protocol, we use VIPs (Verification IPs) to verify the functionality.

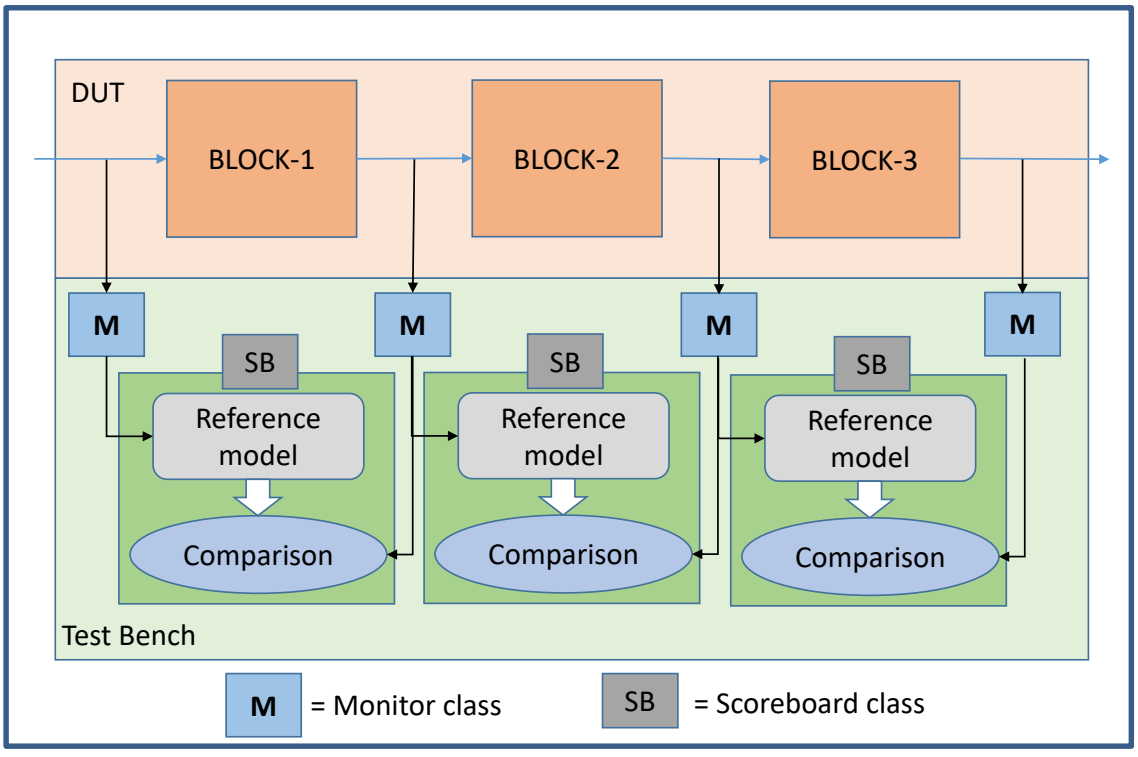

Figure 2. A typical Scoreboard architecture

II. CHALLENGES IN VERIFICATION

With short design timelines and time to market, verification team need to have stable base verification environment which can be tuned easily for new designs. The scoreboard architecture needs to address the following challenges:

A. Scalability and Adaptability

The end consumer is expecting DSLR (Digital Single-Lens Reflex Camera) like performance in Mobile Camera. To meet such emerging market trends, combined with the ever increasing demand for higher resolutions, the CMOS image sensor for mobiles has become very sophisticated. The rise in number of blocks associated in enhancing the image quality as well as their increasing algorithm complexity adds to the challenge. As an illustration, there are more than 40 blocks(IPs) in a premium Image Sensor. Verification team has to ensure that all the blocks are thoroughly verified with their respective reference models. Due to aggressive timelines, the scoreboard setup for all the IPs has to be brought up in quick time. Therefore, the scoreboard structure should be generic to be easily scalable for all the blocks. Further, we are witnessing the growth in market for Image Sensors in Automotive Industry and other security applications. The design of Image Sensors for these industries will be different from that of Mobile phones. The scoreboard structure should be easily adaptable to cater to different design needs.

B. Reusablity and Portability

We have seen the increase in resolution of Image Sensors from 5MP(Mega Pixel) to 200MP. With the increase in resolution of Image sensors, the simulation times have shot up from hours to days. Now verification teams have to use Emulation (Simulation Acceleration) along with simulation to meet the timelines. Hence, the scoreboard architecture should be reusable across simulation and emulation platforms.

With newer and complex algorithms, need arises to do both IP and sub-system level verification. This IP setup should be seamlessly portable to the SOC testbench to verify the entire pipeline. This becomes easy if we have a standard scoreboard architecture across IP and SOC environments.

C. Resource Management

With larger resolutions, the amount of data to be stored and processed by scoreboard of each block increases. With many blocks, this creates a cascading effect that may lead to simulation crashes. In addition to this, the simulation time is in order of days for such designs. Even though Simulation Acceleration setup on emulation is able to solve the large run-time issue, it is limited by high cost hardware and limited licenses.

Considering aggressive timelines, members in team have to work in parallel on different blocks independently. There is a need to manage the manpower, licenses and run times to meet the product schedule. The scoreboard architecture should be versatile to solve the above concerns.

From the challenges mentioned above, it is evident that lack of proper scoreboard architecture will leave out gaps in verification. So, for high quality functional verification and faster execution, it is necessary to streamline the scoreboard framework.

III. PROPOSED SOLUTION

To address all these challenges, we have developed a Scoreboard eco-system using parameterization, UVM Callbacks, and a file-based monitor.

A. Parameterization

Parameters are like constants local to that particular class instance. The parameter value can be used to define a set of attributes in class. Default values can be overridden by passing a new set of parameters during instantiation. This is called parameter overriding [1]. We will discuss how we used parameterization in UVC (Universal Verification Component) and Scoreboard classes to address the defined challenges.

There are standard design specifications which change across SOCs as well as IPs within a SOC. In an Image sensor, there are multiple sub-blocks, each working with different number of data channels, data bit-width, along

with different sideband signals. Figure-3 illustrates this for few blocks. Typically, there are 10-15 standard design parameters that vary across blocks in the chain.

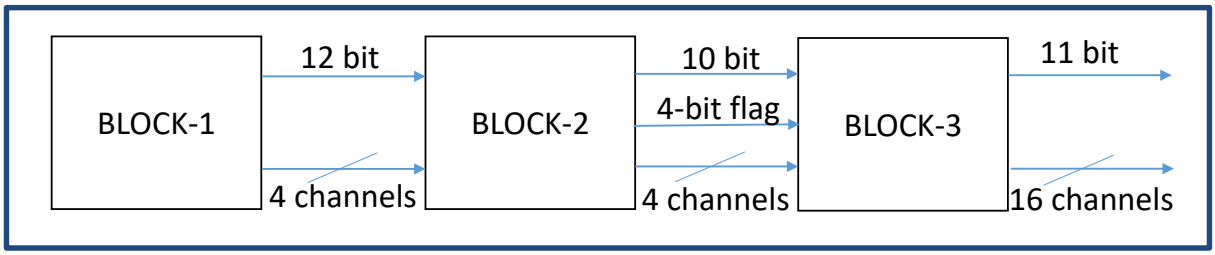

Figure 3. Interface signals across blocks

Further, we need scoreboard for each block to find the source of issue with the help of reference model. Based on its implementation, reference model interface format can be different as compared to RTL. For example, reference model works on 1-channel interface whereas RTL has 4-channel interface. Hence, there is a need to maintain the same format for proper comparison between reference model and RTL.

To address these challenges, in the proposed scoreboard architecture, we have defined 2 types of parameters – UVC parameters and Scoreboard parameters.

- UVC parameters: These parameters are used in UVC logic which constitutes the driver, sequencer, sequences and monitor. UVC parameters can be defined as shown in Figure 4. These parameters are incorporated in logic of UVC to make the code generic and easily reusable. Transaction packet and interface can be defined as shown in Figure 5. In Figure 6, it is shown how monitor logic is parameterized and incorporated in testbench.

```
interface uvc_if#(uvc_params_t params=uvc_default_params);
  logic clock;
  logic [(params.DATA_WIDTH-1):0] data[params.NUM_CHANNELS];
  logic [(params.HADDR_WIDTH-1):0] haddr;
  logic [(params.VADDR_WIDTH-1):0] vaddr;
  logic [(params.FLAG_WIDTH-1):0] flag;
endinterface : uvc_if

typedef struct packed {
  int NUM_CHANNELS; //This parameter captures the info on number of data channels
  int DATA_WIDTH;   //This parameter captures the bit-width of data channels
  int HADDR_WIDTH;  //This parameter captures the horizontal addrress width
  int VADDR_WIDTH;  //This parameter captures the vertical addrress width
  int FLAG_WIDTH;   //This parameter captures the bit-width of flag
} uvc_params_t;

parameter uvc_params_t uvc_default_params = '{
  //Default values using defines
  `NUM_CHANNELS,
  `DATA_WIDTH,
  `HADDR_WIDTH,
  `VADDR_WIDTH,
  `FLAG_WIDTH
};
```

```
parameter uvc_params_t uvc_block1_params =
'{
4,  //NUM_CHANNELS
12, //DATA_WIDTH
16, //HADDR_WIDTH
16, //VADDR_WIDTH
0,  //FLAG_WIDTH
};

parameter uvc_params_t uvc_block2_params =
'{
4,  //NUM_CHANNELS
10, //DATA_WIDTH
16, //HADDR_WIDTH
16, //VADDR_WIDTH
4,  //FLAG_WIDTH
};

parameter uvc_params_t uvc_block3_params =
'{
16, //NUM_CHANNELS
11, //DATA_WIDTH
16, //HADDR_WIDTH
16, //VADDR_WIDTH
0,  //FLAG_WIDTH
};
```

Figure 4. UVC parameters

```
//UVC INTERFACE
interface uvc_if#(uvc_params_t params=uvc_default_params);
  logic clock;
  logic [(params.DATA_WIDTH-1):0]
data[params.NUM_CHANNELS];
  logic [(params.HADDR_WIDTH-1):0] haddr;
  logic [(params.VADDR_WIDTH-1):0] vaddr;
  logic [(params.FLAG_WIDTH-1):0] flag;
endinterface : uvc_if
```

```
//TRANSACTION PACKET
class uvc_data_packet_c#(uvc_params_t params = uvc_default_params)
extends uvm_sequence_item; //{
  rand bit [(params.DATA_WIDTH-1):0]
data_channel[params.NUM_CHANNELS-1:0];
  rand bit [(params.HADDR_WIDTH-1):0] haddr;
  rand bit [(params.VADDR_WIDTH-1):0] vaddr;
  rand bit [(params.FLAG_WIDTH-1):0] flag;
endclass: uvc_data_packet_c //}
```

Figure 5. UVC Interface and Transaction packet

```
//UVC_MONITOR
class uvc_monitor_c#(uvc_params_t params=uvc_default_params) extends
uvm_monitor;
uvc_data_packet_c#(params) packet_collected;
virtual interface uvc_if #(params) vif;

function void build_phase(uvm_phase phase);
super.build_phase(phase);
assert(uvm_config_db#(virtual interface pvi_if#(params))::get(this, "",
"vif",vif));
packet_collected = uvc_data_packet_c#(params)::type_id::create
($sformatf("packet_collected"),this);
endfunction : build_phase

virtual task run_phase(uvm_phase phase);
forever begin
@(posedge vif.clock)
if(CONDITION) begin //{
packet_collected.haddr <= vif.haddr;
packet_collected.vaddr <= vif.vaddr;
for(int i=0; i<params.NUM_CHANNELS;i++)
packet_collected.data_channel[i] <= vif.data[i];
for(int i=0; i<params.FLAG_WIDTH;i++)
packet_collected.flag[i] <= vif.flag[i];
end //}
end
endtask : run_phase
endclass: uvc_monitor_c
```

```
//TESTBENCH TOP
module testbench_top();
  virtual interface uvc_if#(uvc_block1_params) uvc_block1_if;
  virtual interface uvc_if#(uvc_block2_params) uvc_block2_if;
  virtual interface uvc_if#(uvc_block3_params) uvc_block3_if;

  dut dut_inst(); //DUT instantiation
initial
begin
  uvm_config_db #(virtual interface
uvc_if#(uvc_block1_params))::set (null, "*",
"vif",uvc_block1_if);
  uvm_config_db #(virtual interface
uvc_if#(uvc_block2_params))::set (null, "*",
"vif",uvc_block2_if);
  uvm_config_db #(virtual interface
uvc_if#(uvc_block3_params))::set (null, "*",
"vif",uvc_block3_if);
end
//Example for block1/block2/block3 connections
assign uvc_block1_if.data[0] = `DUT_BLOCK1.data_channel0;
assign uvc_block1_if.data[1] = `DUT_BLOCK1.data_channel1;
assign uvc_block2_if.flag    = `DUT_BLOCK2.flag;
assign uvc_block3_if.haddr   = `DUT_BLOCK3.haddr;
assign uvc_block3_if.vaddr   = `DUT_BLOCK3.vaddr;
endmodule
```

Figure 6. Monitor parameterization example

- Scoreboard parameters: These are used to keep the attributes configurable across different blocks in the scoreboard framework. Parameters can be defined as in Figure 7.

```
typedef struct{
  int INPUT_DATAW; //Reference model Input data width
  int OUTPUT_DATAW;//Reference model Output data width
  int NUM_OF_IN_CHANNEL;//Reference model data channels
  int NUM_OF_OUT_CHANNEL;//Expected data channels for
comparison
  string REF_OUT_FILE_HIER; // Directory name.
  int ADDR_CHECK_EN; //To enable address checks
  int FLAG_CHECK_EN; //To enable flag checks
} sb_params_t;


parameter sb_params_t sb_params_default_params = '{
10,//INPUT_DATAW            ;
10, //OUTPUT_DATAW
4,  //NUM_OF_IN_CHANNEL   ;
4,  //NUM_OF_OUT_CHANNEL   ;
"DEFAULT", //REF_OOUT_FILE_HIER;
1,   //ADDR_CHECK_EN
1   //FLAG_CHECK_EN
};
```

```
parameter sb_params_t sb_params_block2 = '{
12,//INPUT_DATAW            ;
10, //OUTPUT_DATAW
4,  //NUM_OF_IN_CHANNEL   ;
4,  //NUM_OF_OUT_CHANNEL   ;
"BLOCK2", //REF_OOUT_FILE_HIER;
1,   //ADDR_CHECK_EN
1   //FLAG_CHECK_EN
};

parameter sb_params_t sb_params_block3 = '{
10,//INPUT_DATAW            ;
11, //OUTPUT_DATAW
4,  //NUM_OF_IN_CHANNEL   ;
16,  //NUM_OF_OUT_CHANNEL   ;
"BLOCK3", //REF_OOUT_FILE_HIER;
1,   //ADDR_CHECK_EN
0   //FLAG_CHECK_EN
};
```

Figure 7. Scoreboard Parameters

B. UVM_CALLBACKS

Callbacks are empty methods with a call to them. UVM provides a set of classes, methods and macros to implement callbacks [2]. In this section, we will discuss how we used UVM callbacks in our scoreboard architecture. In the proposed scoreboard architecture, there are two main components - Scoreboard Callback class and Generic Scoreboard Class. We will discuss how we utilized these two components along with parameterization to make the scoreboard easily scalable.

- Scoreboard callback class: In this class, we have the implementation details of packing logic and comparison logic between reference model and RTL outputs, as demonstrated in Figure 8. A base callback class is defined first and callback class for each block is extended from the base callback class. In this base callback class, we implement the packing and comparison logic in common tasks. As per the requirement of dedicated packing and comparison logic for different blocks, this can be achieved by using function override in their respective callback classes.

```
//SB_CALLBACK_CLASS //{
class sb_callback_base#(sb_params_t params_sb =
sb_default_params,uvc_params_t input_params = uvc_default_params,
uvc_params_t output_params = uvc_default_params) extends uvm_callback;

  virtual function void pack_input_data();
  //callback usage
  //Implement the input packing needed for Reference model
  endfunction: pack_input_data

  virtual function void pack_output_data();
  //callback usage
  //Implement the output packing needed for comparison
  endfunction: pack_output_data

  virtual function void pack_flag_data();
  //callback usage
  //Implement the flag packing.
  endfunction: pack_output_data

  virtual function void run_ref_and_compare();
  //callback usage
  //Implement the reference model call and comparison logic
  endfunction: run_ref_and_compare

  virtual task reset_sb();
  //callback usage
  //Implement the reset logic
  endtask: reset_sb
endclass: sb_callback_class
```

```
//INDIVIDUAL BLOCK CALLBACK CLASS
class sb_callback_block2#(sb_params_t params_sb =
sb_default_params,uvc_params_t input_params =
uvc_default_params,uvc_params_t output_params = uvc_default_params)
extends
sb_callback_base#(params_sb,input_pvi_params,output_pvi_params);

  function new(string name="sb_callback_block2");
    super.new(name);
  endfunction: new
  virtual function void pack_input_data (); //{
 //FUNCTION OVERRIDE
  endfunction : pack_input_data //}

  virtual function void pack_output_data ();  //{
//FUNCTION OVERRIDE
  endfunction : pack_output_data  //}

  virtual function void run_ref_and_compare ();  //{
//FUNCTION OVERRIDE
  endfunction : run_ref_and_compare  //}

endclass : sb_callback_block2

class sb_callback_block3#(sb_params_t params_sb =
sb_default_params,uvc_params_t input_params =
uvc_default_params,uvc_params_t output_params = uvc_default_params)
extends
sb_callback_base#(params_sb,input_pvi_params,output_pvi_params);
endclass : sb_callback_block3
```

Figure 8. Scoreboard Callback class

- Generic scoreboard class: In this class (Figure 9), we have the logic to control when to trigger the packing logic, reference model execution and the comparison logic. As the name suggests, this class is generic and

```
//GENERIC SB
class generic_sb_c#(sb_params_t params_sb = sb_default_params,uvc_params_t input_params = uvc_default_params, uvc_params_t output_params =
uvc_default_params) extends uvm_scoreboard;

   `uvm_register_cb(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base)
   // TLM PORTS
   `uvm_analysis_imp_decl(_in_data)
   `uvm_analysis_imp_decl(_out_data)
   `uvm_analysis_imp_decl(_flag_data)

   uvm_analysis_imp_in_data #(transaction_packet#(input_params), generic_sb_c#(params_sb,input_params,output_params)) input_data;
   uvm_analysis_imp_out_data #(transaction_packet#(output_params), generic_sb_c#(params_sb,input_params,output_params)) output_data;
   uvm_analysis_imp_flag_data #(transaction_packet#(output_params), generic_sb_c#(params_sb,input_params,output_params)) flag_data;

   //Trigger via TLM ports
   function void write_in_data (input transaction_packet#(input_params) data_pkt); //{
     `uvm_do_callbacks(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base,pack_input_data())
     `uvm_info(params_sb.BAS_OUT_FILE_HIER, "CAPTURED_INPUT_DATA_WRITE", UVM_LOW)
   endfunction: write_in_data //}

   function void write_out_data (input transaction_packet#(output_params) data_pkt); //{
      `uvm_do_callbacks(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base,pack_output_data())
      `uvm_do_callbacks(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base,run_ref_and_compare())
   endfunction: write_out_data //}

   function void write_flag_data (input transaction_packet#(output_params) data_pkt); //{
      `uvm_do_callbacks(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base,pack_flag_data())
   endfunction: write_out_data //}

   virtual task run_phase(uvm_phase phase);
     super.run_phase(phase);
       fork
         begin //{
           forever @(negedge vseqr.vintf.Rstn_pad or negedge vseqr.vintf.frame_reset) begin //{ Trigger via EVENT
             `uvm_do_callbacks(generic_sb_c#(params_sb,input_params,output_params),sb_callback_base,reset_sb())
           end//}
         end //}
       join_none
   endtask: run_phase
endclass: generic_sb_c
```

Figure 9: Generic scoreboard class

controlled by parameters. The trigger can be done by TLM ports or through events. We register scoreboard callback class with this generic scoreboard class using uvm_register_cb. The generic scoreboard and the callback classes are connected as shown in Figure 10.

```
// sb_callback instances
sb_callback_block2#(sb_params_block1,uvc_block1_in_params,uvc_block2_in_params) callback_block2;
sb_callback_block3#(sb_params_block1,uvc_block2_in_params,uvc_block3_in_params) callback_block3;

//generic scoreboard instances
generic_sb_c#(sb_params_block2,uvc_block1_in_params,uvc_block2_in_params) sb_block2;
generic_sb_c#(sb_params_block3,uvc_block2_in_params,uvc_block3_in_params) sb_block3;

//Callback connections
uvm_callbacks#(generic_sb_c#(sb_params_block2,uvc_block1_in_params,uvc_block2_in_params),sb_callback_base)::add(sb_block2,
callback_block2);
uvm_callbacks#(generic_sb_c#(sb_params_block3,uvc_block2_in_params,uvc_block3_in_params),sb_callback_base)::add(sb_block3,
callback_block3);
```

Figure 10. Callbacks and Scoreboard connections

C. *File-based data handling*

With the help of parameterization and callback-based scoreboard, we have implemented a portable, adaptable, scalable and reusable scoreboard architecture. As the scoreboard is streamlined, the emphasis is on the possible methods to improve simulation speed. Reference models are designed to work on complete frame information. Therefore, the entire frame data needs to be captured before we can trigger the reference model and scoreboard. Monitor uses queues to capture the data from RTL and send it to the scoreboard using TLM ports. With higher resolutions and addition of HDR (High Dynamic Range) like features, monitor has to capture many folds of data in each block monitor. The huge amount of data causes the simulator to slow-down and possibly crashing at times, while transferring the data from monitor to scoreboard. It is observed that in Emulation (Simulation Acceleration) setup, this data transfer takes lot of time. To tackle this, we approached data handling through files. In this section, we will discuss how we used file-based monitor to solve the crash issue and speed up the simulation.

In this approach, we write the data into a file instead of accumulating entire data in queues. At the start of frame, we open a file and the incoming data is collected in it. We adopted a hybrid approach of capturing small amounts of data in a queue and then storing in a file, instead of performing continuous file operations. We achieved better performance with this hybrid approach. At frame end, the file is closed and scoreboards are triggered. As the entire eco-system (Figure 11) is parameterized, the scoreboard knows the file name based on parameter and perform the desired operation. This logic is easily portable for emulation as well.

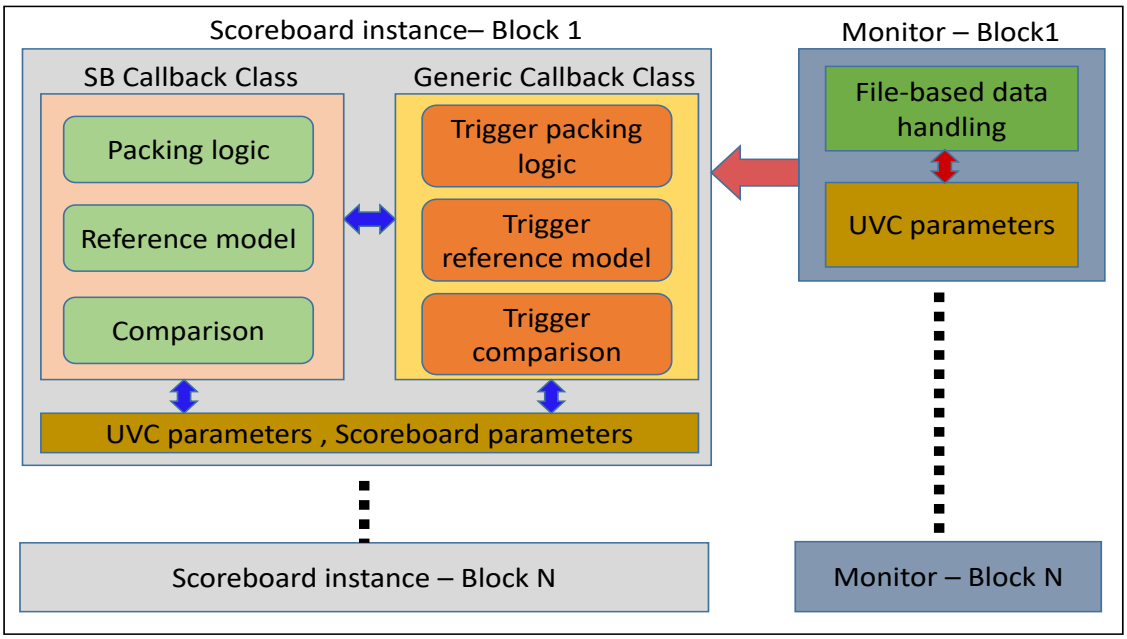

Figure 11. Scoreboard eco-system

IV. CONCLUSION AND RESULTS

1. With this highly reusable setup, scoreboard can be quickly sanitized in Acceleration platform and regressions can be run in simulation. We can leverage the benefits of both simulation and emulation in parallel.
2. Owing to generic implementation and configurability, the setup is easily scalable to large designs and members of the team can work on it independently which helps in faster execution.
3. With the new approach in monitor, gain is achieved in run-time (Table I). A high resolution image sensor simulation which earlier ran for 2 days before facing simulator crash without reference model execution, completed in a day with the proposed updates, with all scoreboard logic executed.
4. The scoreboard setup can be easily ported to other multimedia teams and applications with minimum effort.
5. The readability of code improved a lot and is easy for everyone to ramp up on it.
6. The entire scoreboard architecture is automated and it reduces the bring-up time. A large sensor with 45 blocks is brought up in 2 days.
7. Need for big_mem LSF (Load Sharing Facility) has come down with file-based monitor. This is very helpful as big_mem licenses are limited.

Table I. Improvements with file-based monitor

Category(with scoreboard)	Queue based Monitor	File based monitor	Improvement
Simulation time for 200MP – Single frame	50 hours	28 hours	44%
Simulation time for 200MP – 2 frames	105 hours	45 hours	57%
Simulation time for 12.5 MP – 2 frames	12 hours	10 hours	16%
Emulation run time for 200MP – Single frame	14 hours	10 hours	28%

REFERENCES

[1] https://verificationguide.com/systemverilog/systemverilog-parameterized-classes/

[2] https://verificationguide.com/uvm/uvm-callback/