A scalableVIP component to increase robstuness of co-verification within an ASIC

Scalable and automated gray-box method

Mario de Matteis, ON Semiconductor, ASG/MCC, Milan, Italy (mario.dematteis@onsemi.com)

Matteo Barbati, ON Semiconductor, ASG/MCC, Milan, Italy (matteo.barbati@onsemi.com)

***Abstract*—In ASIC development the firmware might be verified by a standalone step on a FPGA emulator without other direct co-verification methods to signoff the complete DUT application. This strategy is commonly accepted for analog-on-top ICs, but it might hide bugs or weaknesses of the device since it doesn't really stimulate and monitor the DUT with the firmware executing. A full-chip co-verification approach should be used: a testbench where the full-chip DUT (digital and analog logic plus firmware) is instantiated and a fully-featured UVM environment is used to qualify the entire device. Several co-verification techniques are available that vary from firmware-centric co-verification approaches to hardware-centric co-verification strategies. In this paper we propose a method based on a scalable VIP that monitors the interactions between firmware and other parts of the design. The component (a.k.a FW_VIP) provides a bridge that supports complex HW/FW interactions, like checking HW/FW overall behavior and synchronizing scoreboards with firmware by capturing specific events with minimal impact on the classic firmware and hardware workflows.**

Keywords—HW/FW co-verification, SystemVerilog, UVM, VIP, firmware, coverage, ASIC, power, SoC.

I. INTRODUCTION

The increasing complexity of power ASIC products, such as multi-phase controller, is pushing platforms to include SoC-like architectures which includes microcontrollers, complex protocol interfaces and security features. The microcontroller is typically used to implement features that requiring more flexibility than RTL code. Hardware designers are required to develop a design that provides the minimum set of capabilities that can be use by firmware to address the required functionalities.

In this kind of context, the target of the verification requires that also the firmware is properly verified. The lack of firmware verification may hide possible mis-behaviors, weakness and/or bugs.

In theory HW/FW co-verification is the right strategy to follow in these cases, but HW/FW co-verification requires that both the verification and firmware teams work heavily together for several reasons. The verification engineers need support from firmware engineers to have reliable code to use in their simulations, and the firmware engineers need help from verification engineers to be able to properly use complex verification environment.

The state of the art is that usually the firmware engineers test their own code on FPGA emulator [2]. This is fine, but for the target of the co-verification is not enough since the DUT's behavior is checked in few cases without the possibility to fully understand what is happening inside the device, but only at the external interfaces. For this reason, this strategy works fine for firmware engineers and provides a way to perform some performance analysis but from a verification point of view is not reliable to validate a design.

On the other hand, another approach, proposed in [1] is to create a "fake" register map that allows mapping of verification components functionalities (also with randomization) in firmware world and to develop verification tests in firmware. This approach works to reduce the gap between the firmware and verification worlds but requires verification engineers to convert their approach from usually System-Verilog UVM [3] to firmware languages. This means verification engineers need to become experts of some aspects related to firmware development (scatter files, compilator options and so on).

Other strategies are available to address proper HW/FW co-verification but are too invasive in terms of changes to the usual design workflow. In [4] a SW-centric Design approach is described where the entire development flow is driven by SW development. In this case, software is used not only to provide Design features but also to perform system architecture analysis that is refined in the following steps to obtain the desired device, with a verification step that is based on the comparison between the implemented Device and the SW-model used in the architecture analysis definition. In [5] a UVM-centric approach is described where an extension to UVM library based on D-language is used in order to emulate FW through an Instruction Set Simulator.

In order to close the gap between verification and firmware worlds we propose a new strategy that minimizes the changes to the firmware and verification workflows. The idea is to create a way to simplify the interactions between the two teams, having the target firmware code with few modifications inside a classic UVM environment, in order to address the limit of the state-of-the-art co-verification approaches outlined previously.

Figure 1 describes on the left the typical development workflows for both firmware and verification teams, while on the right it describes the proposed workflows. The main differences are related to the need for "exit labels" inside significant firmware functions that need to be linked to specific verification environment capabilities and to a dedicated Verification IP (FW_VIP) required to manage automatically generated inputs from the firmware, to synchronize and check the code with the verification environment.

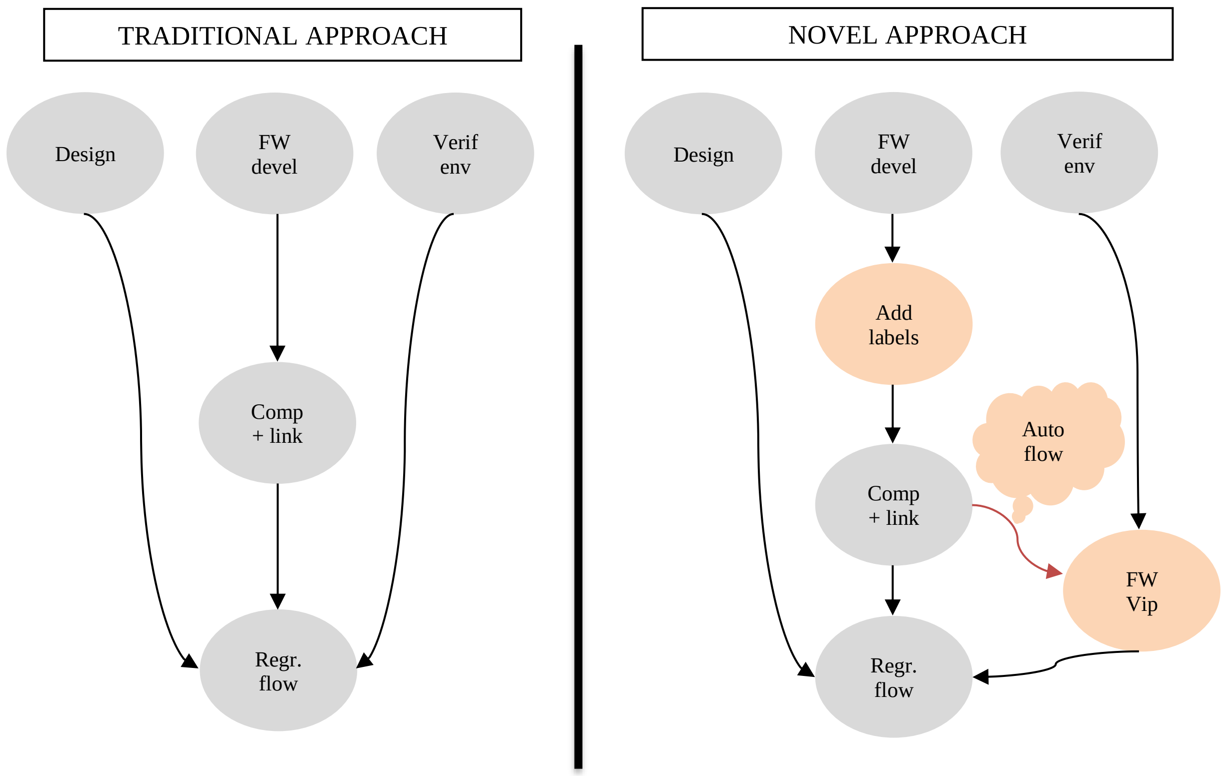

Figure 1 - Workflow changes

The benefit of this proposal is that neither verification engineers nor firmware engineers need to significantly change their usual workflow to reach the target of correct HW/FW co-verification. Additionally, extensive firmware verification is performed during regression runs. From a verification point of view, only two steps need to be addressed to have the flow up and running:

1. Firmware verification IP development, required to manage the synchronization and checking capabilities (functions and tasks triggers, variable monitoring, stack point checks etc.) required by co-verification.
2. Firmware verification IP automation flow, required to process the information needed by the FW_VIP from firmware code.

Once the flow described above is in place the FW_VIP facilitates the interaction between firmware and verification environment since it automatically translates the firmware events into UVM transactions which will be passed to scoreboards and checkers. Figure 2 is depicting some types of events which can be monitored:

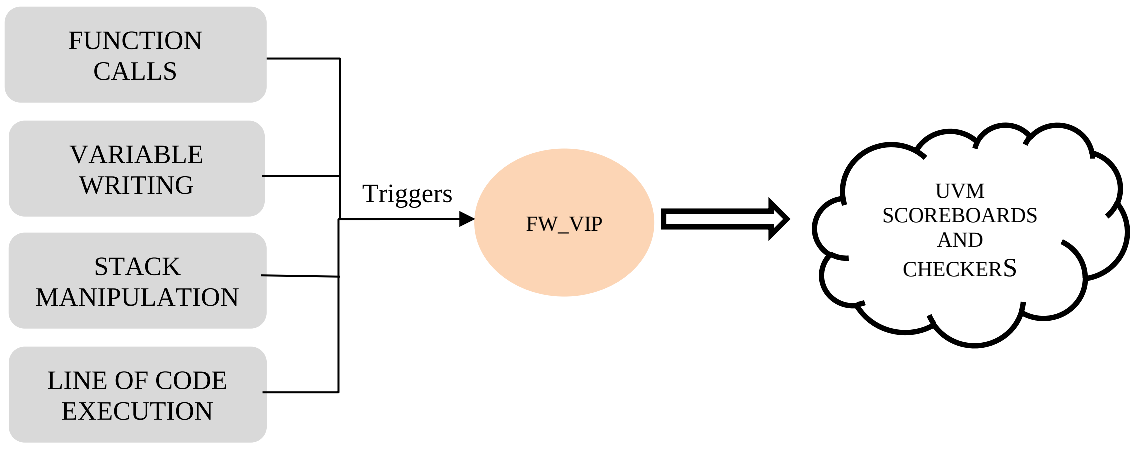

Figure 2 - FW-VIP Monitored events

At the end the quality of coverage of the entire system is improved since also the firmware and its interaction with hardware is monitored and checked by a fully-featured UVM environment and with little changes to normal development flow. The level of results cannot be matched either in an environment with separated verification step for firmware and hardware or in UVM environment with only hardware VIP instantiated where there is no access to the internal firmware events.

II. FIRMWARE VERIFICATION IP

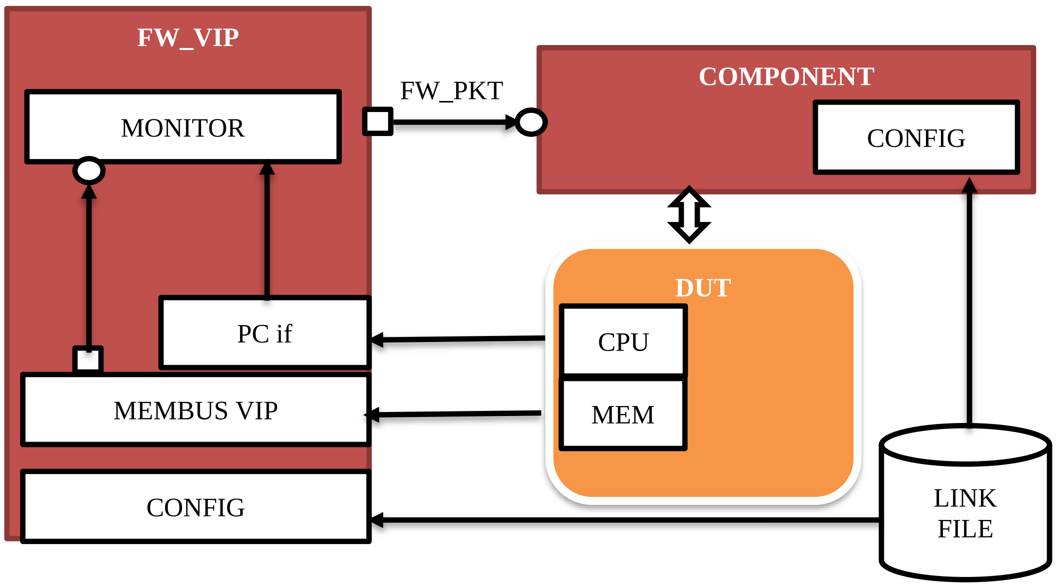

Figure 3 - FW_VIP block description

The Verification IP provides a mechanism to monitor specific events at the DUT memories and Program Counters and to trigger the other components of the verification environment that are able to update their behavior or to perform the proper check accordingly.

The topology is depicted in Figure 3. A MEMBUS VIP monitors the memory transactions at the DUT memory interface and sends transaction to a MONITOR that represent the core of the FW_VIP. In the same way, one (or more in case of multicore) Program Counter interface is used to monitor the Program Counter of the microcontroller. This interface is connected to the MONITOR too. The Monitor collects the information received by MEMBUS VIP and PC Interface in dedicated transactions that are sent to the components (scoreboards, predictors and monitors) connected to the FW_VIP. The MEMBUS VIP can be customized, according to the needs, to manage the proper DUT memory interface (i.e. AHB VIP).

A. Firmware VIP Packet

Two kinds of information are usually required to be able to perform HW/FW co-verification:

- Hardware and firmware synchronization events.
- Firmware variable update events.

The first one, based on the Program Counter value, is used typically to notify the verification environment that an event of interest has occurred in the firmware, for example that a specific function was called or that the interrupt routine associated to a specific interrupt has started.

The second kind of events are related to the monitoring of specific firmware variables that have some effect on the verification environment behavior or checks. For example, knowing the value of a specific variable allows checking the contents of the payload of a specific DUT internal protocol or verifying that the device behavior is in line with the configuration done in firmware.

For these reasons, the interactions between the Verification IP and the other part of the verification environment are managed through the packet of Figure 4.

```
typedef enum {pc,dut_state,variable} event_type;
class fw_packet extends uvm_sequence_item;
    event_type     currentEvent;
    string         pc_event;
    string         variableName;
    bit [31:0]     variableValue;

     `uvm_object_utils_begin(fw_packet)
           `uvm_field_enum(event_type, currentEvent, UVM_ALL_ON)
           `uvm_field_string(pc_event, UVM_ALL_ON)
           `uvm_field_string(variableName, UVM_ALL_ON)
           `uvm_field_int(variableValue, UVM_ALL_ON)
     `uvm_object_utils_end

    function new(string name = "fw_packet");
           super.new(name);
    endfunction
endclass : fw_packet
```

Figure 4 – fw_packet class

The FW_VIP packet (a.k.a. fw_packet) contains infomation regarding:

- The source of the information identified by the enumerated type variable currentEvent notifying if the updated information is coming due to a Program Counter event or due to a Variable Update event.
- The Program Counter event name: in case a Program Counter event happens, this variable stores the name of the raised event.
- The variable name: In case of a variable update event, this variable stores the name of the updated firmware variable.
- The variable value: In case of a variable update event, this variable stores the value of the updated firmware variable.

B. *FW_VIP Monitor*

The core of the FW_VIP is represented by the MONITOR. This component will operate as a bridge between the MEMBUS VIP, the Program Counter interface and the other parts of the verification environment. Only a small subset of memory addresses and Program Counter values need to be managed for the purpose of HW/SW co-verification. For this reason, as reported in Figure 5, two associative arrays, ramAddressList and pcValueList are used to store lists of Addresses and PC values:

- *ramAddressList* defines the keys (integer) representing the memory addresses of the variables (string) to monitor
- *pcValueList* defines the keys (integer) representing the PC values associated with specific function name (string)

```
class fw_monitor extends uvm_monitor;
    fw_packet              pc_pkt;
    fw_packet              ram_pkt;
    ahb3_master_packet     ahb_pkt;

    virtual interface pc_if vif;

    int             pcLogFileIndex;
    string          ramAddressList[integer];
    string          pcValueList[integer];
    string          ramAddressFilePath;
    string          pcValueFilePath;
    bit[31:0]       valueMask= 32'hffffffff;

    uvm_analysis_imp  #(ahb3_master_packet, fw_monitor) ahb_pkt_port;
    uvm_analysis_port #(fw_packet) send_pkt;

    `uvm_component_utils_begin(fw_monitor)
           `uvm_field_object(pc_pkt, UVM_ALL_ON)
           `uvm_field_object(ahb_pkt, UVM_ALL_ON)
           `uvm_field_string(ramAddressFilePath,UVM_ALL_ON)
```

```
            `uvm_field_string(pcValueFilePath,UVM_ALL_ON)

            `uvm_field_int(valueMask,UVM_ALL_ON)

    `uvm_component_utils_end

    …
```

Figure 5 – MONITOR class

These data structures are populated at the beginning of simulation starting from two external files identified through monitor variables *ramAddressFilePath* and *pcValueFilePath*. The content of these file is defined by the verification and firmware engineers together to ensure correct monitoring of the more interesting PC events and firmware variables and are automatically populated through a set of scripts operating on firmware elf file and symbol file.

Figure 6 and Figure 7 provide an example of these two files. In the first case a list of variables is reported with the associated memory address. In the second case, the name of relevant functions and exit labels are reported with the associated Program Counter value.

```
vout_max_ra 2000054E

vout_min_ra 20000550

vout_transition_rate_ra 20000556

vout_max_rb 200005A2

vout_min_rb 200005A4

vout_transition_rate_rb 200005AA
```

Figure 6 – ramAddressList configuration file

```
load_configuration_end DEFAULT 00003f78

load_configuration HIDDEN 00003f09

load_user_configuration_from_OTP_end DEFAULT 00002ef6

load_user_configuration_from_OTP HIDDEN 00002ec5
```

Figure 7 – pcValueList configuration file

Snippets of code in Figure 8 and Figure 9 show how the memory accesses and Program Counter updates are managed by the monitor. The strategy is the same for both. Once new data is available, the monitor verifies if the information should generate an event. If true, the monitor updates the relevant information of the FW_VIP transaction and writes to the analysis port.

```
function void write(ahb3_master_packet p);

  $cast(ahb_pkt, p.clone);

  `uvm_info("fw_monitor", $sformatf("AHB packet triggered \n %s",p.sprint()),UVM_FULL)

  if (ahb_pkt.hwrite == AHB3_WRITE) begin

          `uvm_info("fw_monitor", $sformatf("AHB write packet triggered \n

                                  %s",p.sprint()),UVM_FULL)

          if (ramAddressList.exists(ahb_pkt.haddr)) begin

                  ram_pkt.pc_event                             = "NULL";

                  ram_pkt.variableName          = ramAddressList[ahb_pkt.haddr];
                  if ( (ram_pkt.variableName == "vrStateA") ||
                       (ram_pkt.variableName == "vrStateB") )  begin
                         $cast(ram_pkt.dutStateValue,ahb_pkt.hwdata[7:0]);
                         ram_pkt.currentEvent   = dut_state;
                  end else begin
                         ram_pkt.variableValue  = ahb_pkt.hwdata;
                         ram_pkt.currentEvent   = variable;
                    end
                    `uvm_info("fw_monitor", $sformatf("AHB address %8X is in the list \n
                                        %s",ahb_pkt.haddr,ram_pkt.sprint()),UVM_FULL)
                    `uvm_info("fw_monitor", $sformatf("RAM event\n
                                        %s",ram_pkt.variableName),UVM_NONE)
                    send_pkt.write(ram_pkt);
            end
     end
endfunction: write
```

Figure 8 – Monitoring ramAddressList

```
virtual task run_phase(uvm_phase phase);
   forever begin
          @(posedge vif.clk);
          `uvm_info("fw_monitor", $sformatf("program counter triggered: %4X",
                                        vif.program_counter),UVM_FULL)
          if (pcValueList.exists(vif.program_counter)) begin
                 pc_pkt.pc_event        = pcValueList[vif.program_counter];
                 pc_pkt.variableName    = "NULL";
                 pc_pkt.currentEvent    =  pc;
                 `uvm_info("fw_monitor", $sformatf("PC %8X is in the list \n
                                %s",vif.program_counter,pc_pkt.sprint()),UVM_FULL)
                 `uvm_info("fw_monitor", $sformatf("PC event\n
                                %s",pc_pkt.pc_event),UVM_NONE)
                 send_pkt.write(pc_pkt);
          end
   end
endtask: run_phase
```

Figure 9 – Monitoring *pcValueList*

III. Firmware VIP automation Flow

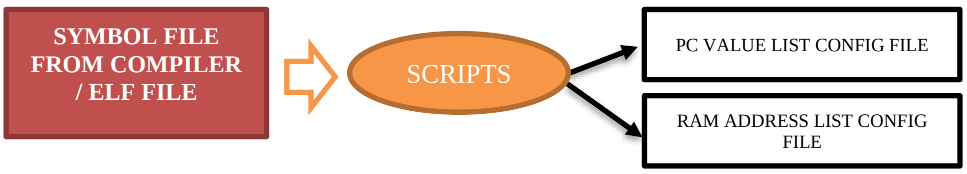
SYMBOL FILE FROM COMPILER / ELF FILE
SCRIPTS
PC VALUE LIST CONFIG FILE
RAM ADDRESS LIST CONFIG FILE

Figure 10 – Automation process

As described in the previous section, the configuration of the FW_VIP monitor is performed using two external files, one for the firmware variable addresses and another one for the possible PC values. These files are strictly related to the current version of the firmware used in the verification flow.

Automatic generation of these files is required to reduce the risk of introducing errors into the FW_VIP input. For this reason, a set of scripts is required. In our proposal, we suggest the usage of dedicated scripts that are derived from the symbol file and/or ELF file related to the current version of the firmware and are updated as required upon code changes. These scripts also need some configuration inputs that identify the name of the variables or functions that are required to correctly manage the HW/FW co-verification flow. The list of variables and functions to monitor is defined by both verification and firmware Teams.

IV. Case Study

The proposed approach is currently used in the verification flow of our multi-phase controller. It is used in the following context:

- To trigger self-checking capabilities inside the scoreboards
- To check the correctness of the firmware code

The firmware MEMBUS VIP in the case study is an AHB VIP and the microcontroller is a single-core Cortex M0. The program counter of our micro-controller is connected to the PC SystemVerilog interface of the FW_VIP.

A. *Self-Checking Capabilities Trigger*

In this context, the FW_VIP sends to the scoreboard a transaction with a Program Counter event. The scoreboard checks that the trigger event is one of the events that has some meaning for it and then updates its behavior to verify the firmware is working well.

In the case study this functionality is used to notify the scoreboard to the completion of a set of copy commands from different DUT locations performed by firmware.

```
function void write_pc_pkt(fw_packet p);
    uvm_event start_copy;
    uvm_event copy_finished;
    fw_packet pkt;
    $cast(pkt, p.clone());
    case(pkt.currentEvent)
        pc: begin
            int res [$];
            res = pc_start_labels.find_index(x) with (x == p.pc_event);
            if(res.size() == 1) begin
                copy_t = labels_map[p.pc_event];
                start_copy = ep.get("start_copy");
```

```
                            start_copy.trigger();
                    end
                    res = pc_end_labels.find_index(x) with (x == p.pc_event);
                    if(res.size() == 1) begin
                            copy_finished = ep.get("copy_finished");
                            copy_finished.trigger();
                    end
            end
            dut_state: begin end
            variable: begin end
    endcase
endfunction: write_pc_pkt

virtual task copy_fsm();
    uvm_event start_copy;
    uvm_event copy_finished;
    start_copy = ep.get("start_copy");
    copy_finished = ep.get("copy_finished");
    forever begin
            case(state)
                    'd0: begin
                            int res [$];
                            start_copy.wait_trigger();
                            // Initialize here the addr_list data structures as needed
                            if(copy_list.size() > 0) begin
                                    init_data_structures();
                                    // Check that the copy function is one of the expected
                                    res = copy_list.find_index(x) with (x == copy_t);
                                    if(res.size()!=1)
                                            `uvm_error(get_full_name(), "Unexpected copy
                                                                          function")
                                    else
                                            copy_list.delete(res[0]);
                            end
                    end
                    'd1: begin
                            copy_finished.wait_trigger();
                            // Checks the copied data
                            check_copy();
                            state = 'd0;
                    end
```

```
                endcase

        end

   endtask: copy_fsm
```

Figure 11 – Scoreboard example

Figure 11 shows an example of scoreboard waiting on Program Counter events. FW_VIP notifies the scoreboard to the start of the copy function from location A to location B. The scoreboard prepares the internal data structures used to perform the check and starts to monitor what happens at the two DUT locations. Once the FW_VIP notifies the scoreboard about the completion of the copy function, the scoreboard verifies all the copies were correctly done and if all the required copies were performed by FW.

B. *FW_VIP checker*

Another usage of the FW_VIP is depicted in Figure 12. In this case our scoreboard is waiting for specific transaction notifying that a specific firmware variable was updated. According to this kind of information the scoreboard is able to check that the received value is as expected.

```
function void write_fw_pkt(fw_packet fw_pkt);

   fw_packet pkt;

   $cast(pkt, fw_pkt.clone());

   case(pkt.currentEvent)

        pc: begin

        end

        dut_state: begin

        end

        variable: begin

                if(fw_ready) begin

                        case(pkt.variableName)

                           "vout_min_ra" : begin

                                 void'(CHECK_vout_min_var(pkt.variableValue,0));

                           end

                           "vout_min_rb" : begin

                                 void'(CHECK_vout_min_var(pkt.variableValue,1));

                           end

                           "vout_max_ra" : begin

                                 void'(CHECK_vout_max_var(pkt.variableValue,0));

                           end

                           "vout_max_rb" : begin

                                 void'(CHECK_vout_max_var(pkt.variableValue,1));

                           end

                           "vout_transition_rate_ra" : begin

   void'(CHECK_vout_transition_rate_var(pkt.variableValue,0));

                           end

                           "vout_transition_rate_rb"  : begin
```

```
                                          void'(CHECK_vout_transition_rate_var(pkt.variableValue,1));
                                   end
                               endcase
                     end
           end
     endcase
endfunction : write_fw_pkt
```

Figure 12 – Scoreboard example

V. CONCLUSIONS

HW/FW co-verification is becoming more and more important to guarantee the overall quality of a mixed-signal IC. The usage of microcontrollers to implement part of the device capabilities requires an efficient way to perform HW/FW co-verification in the early stage of the project. On the other end, the co-verification requires high interactions between FW and Verification Teams.

In order to mitigate the impact on the two flows, we propose an approach that reduces the number of interactions between the two teams, minimizing changes in their usual workflow. This approach is currently used in our designs and simplifies the interaction between the two teams, increasing the overall quality of the devices in term of verified capabilities and coverage. Table 1 summarizes the main pros and cons related to the state-of-the-art approaches and the proposed one.

	Pros	Cons
FW-Centric (FPGA) approach	Can be used for Performance Analysis/Stress testing on FPGA/HW-emulator	Limited debug capabilities Limited code coverage No self-checking capabilities from UVM world Few capabilities of analog emulation
UVM-Centric (VAL) approach	Full self-checking capabilities from UVM world Easy to debug High Code Coverage	Scenarios written in FW language Verification eng. must manage aspects related to FW development (scatter file, compiler option etc.) Can't be used for Performance Analysis/Stress test on FPGA/HW-emulator
Novel Approach	Full self-checking capabilities from UVM world Easy to debug High Coverage Limited changes to usual FW and Verification workflows	Can't be used for Performance Analysis/Stress test on FPGA/HW-emulator

Table 1 - Co-verification approaches comparison

The verification team is able to verify not only the RTL code but also the entire system represented by digital and analog as well as the firmware code. The firmware team can benefit of this task, focusing on firmware development only and obtaining detailed feedback on the interaction between firmware and hardware.

Furthermore, the approach outlined in this paper is scalable and portable to different platforms with a very small engineering cost. We plan to add new capabilities to the FW-VIP in order to increase the features available for the FW team. In particular, we are already targeting the possibility to keep track of the firmware coverage through the proposed approach.

REFERENCES

[1] A. Allara, F. Brognara: STMicroelectronics. “Bringing Constrained Random into SoC SW-driven Verification”, paper presented at DVCon2013, San Jose February 25-28

[2] Y.B. Liao, P. Li, A.W. Ruan, Y.W. Wang, W.C. Li. State key Laboratory of Electronic Thin Films and Integrated Devices. University of Electronic Science & Technology of China Chengdu, China. “A HW/SW Co-Verification Technique for Field Programmable Gate Array (FPGA) Test”

[3] Universal Verification Methodology (UVM) 1.2 User’s Guide (https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf)

[4] L.Rizzatti, R.Klein, S.Bailey, A.Meier. Rizzatti LLC, Mentor, A Siemens Business. “Application Optimized HW/SW Design & Verification of a Machine Learning SoC”, tutorial presented at DVCon2020

[5] Shilpi Birla, Shikha Sharma & Neeraj Kr. Shukla (2017) UVM-powered hardware/software co-verification, Journal of Information and Optimization Sciences