A UVM Test-bench Skeleton Leveraging the Event Pool and Sequence Layering

Marcela Zachariasova, Jiri Bartak, Tomas Pehnelt, Jan Riha

ASICentrum, EM Microelectronic, Czech Republic, ({marcela.zachariasova, jiri.bartak, tomas.pehnelt, jan.riha}@asicentrum.com)

***Abstract*— EM Microelectronic focuses on ultra-low power, low voltage integrated circuits for battery-operated and field-powered applications. It means that from the verification perspective, system-level mixed-signal test-benches with many power-aware checks are usually developed. The main contribution of this paper is a highly reusable UVM test-bench skeleton, which can be adapted step by step to verify the blocks of the system, as well as the whole System on Chip (SoC), including the HW-SW co-verification with firmware. This test-bench is top-level based; its unique feature is the possibility of parallel development of individual blocks and their UVCs by different verification engineers at the same time while sharing only a few common structures where these people cooperate: UVM layering (with translation) of system transactions, a global register model, processing of interrupts and a global UVM event pool. This feature allows a significant acceleration of development and is essential for reuse in future projects.**

***Keywords*— SoC test-bench, UVM layering, UVM reuse**

I. Introduction and Contribution

The Universal Verification Methodology (UVM) is currently the most widely used verification methodology [1]. It utilizes SystemVerilog as the primary verification language, providing an object-oriented programming approach and the base classes for all UVM test-bench components in the UVM library. The UVM library allows advanced techniques such as constrained random stimulus generation, coverage-driven verification, and self-checking mechanisms.

However, despite these indisputable benefits, the UVM test-bench can become disorganized, especially when verifying SoCs where many people cooperate to build the test-bench. This happens not due to a lack of experience or disregard for UVM rules but rather because each person has their own programming style and strategy for implementing test-bench blocks, sequences, tests, or reference functionality. Therefore, this paper's idea is to develop a test-bench skeleton architecture that establishes some rules but simultaneously allows a good level of freedom in the implementation without losing a sound structure over time.

EM Microelectronic invested a significant amount of effort into developing a test-bench architecture that is suitable for most of their SoC projects. In addition, this architecture allows a high level of reuse for derivative projects. The company specializes in ultra-low power, low voltage integrated circuits for battery-operated and field-powered applications such as access control, radio frequency identification, alarm and security systems, and sensor signal processing. Therefore, from a verification point of view, they are developing system-level mixed-signal test-benches with many power-aware checks and their own *Universal Verification Components* (UVCs).

This paper demonstrates a verification environment for a typical EM Microelectronic SoC consisting of one central bus, for example, AMBA AHB or AXI, one central processor unit (CPU), several digital blocks, and several analog blocks (in the presented test-bench, the digital portion of the design is covered, for modeling analog behavior SystemVerilog behavioral models are used). The main contribution is a description of the UVM test-bench skeleton architecture (including code snippets), which can be applied quickly from the start of every project, can be easily customized and extended, and encourages the reuse of existing UVCs. Additionally, it can be configured to support:

- block-level verification through dedicated UVCs,
- top-level verification through a group of UVCs,
- top-level verification through firmware developed for a CPU.

Despite the standalone techniques described in this paper not being new (referenced in the related work in Section II), their assembly into a configurable and multi-purpose test-bench is unique. Further sections consequently describe this test-bench. Section III outlines how it is configured to support block-level verification. It describes how to start with an empty test-bench skeleton containing a UVM sequence layering, a global UVM event pool, a global register model, a UVM configuration, and how to add new blocks (into the design as well as into the test-bench) step by step while building a comprehensive SoC in the process. Once the entire Design Under Test (DUT) and the test-bench are fully assembled, the focus is on verifying block interactions. Section IV explains how to check those interactions without the CPU being active - meaning that all CPU actions are substituted with the actions of UVCs. This is beneficial when the intention is to have everything under the UVM test-bench control. On the other hand, sometimes it is useful to check the firmware portion of the system and thus have the CPU active during the verification. In the proposed test-bench architecture, it means a different configuration of UVCs and having the tests divided into SystemVerilog and C parts. This approach is described in Section V. The motivation is a good structure, but also reuse. Therefore, Section VI shares ideas on how to reuse existing UVCs when starting a new SoC project. Section VII concludes the paper.

II. Related Work

As stated in the introduction, existing techniques are assembled into a unique concept. And because some of them are not widespread, a brief introduction is provided, including a reference to a more detailed description.

A. UVM sequence layering

Many protocols have a hierarchical definition. For example, PCI Express or USB 3.0 have Transaction, Transport, and Physical Layer. Alternatively, a protocol-independent layer on top of a standard protocol is needed to create protocol-independent verification components (for example, TLM 2.0 over AMBA AHB). All these cases require the deconstruction of transactions (UVM sequence items) of the higher-level protocol into transactions of the lower-level protocol to stimulate the bus and, in the analysis data path, the reconstruction of higher-level transactions from the lower-level transactions [2]. Such transformations are needed in the proposed test-bench skeleton for translating/reconstructing transactions of UVCs (including register transactions) into bus transactions and vice versa.

B. UVM events and UVM event pool

A pool of UVM events is available and shared between all components in the UVM environment. Whenever an event is needed, it is possible to pick one of the events and call it by name. Once the event is obtained, it can either be triggered immediately or waited upon [3] [4]. In the proposed test-bench skeleton, the UVM event pool is used for interrupt processing and, in cases where multiple parts of the system are involved, register fields prediction. The UVM event pool simplifies the integration and reuse of UVCs but requires attention to the UVM event naming scheme to avoid a collision resulting in unwanted behavior.

III. Block-level Verification

Even when verifying a single sub-block, the objective is to think about the DUT and the test-bench from the top-level perspective. This is an essential prerequisite for all further work. The concept will be demonstrated on a series of block diagrams and code snippets, starting with a block diagram in Figure 1. It illustrates a simplified SoC consisting only of two blocks: *Block A* and *Block B*, but the principles remain the same also for more complex systems. The further description focuses on *Block A* and its *UVC*; therefore, only these two are highlighted in orange in Figure 1.

According to the specification, *Block A* should be connected to a bus via a bus interface. It is also known that the configuration of this block is register-based. Whenever a particular computation is completed, an interrupt for the CPU is generated through the IRQ interface. Eventually, this block can have a digital-analog interface for communicating with its analog counterpart.

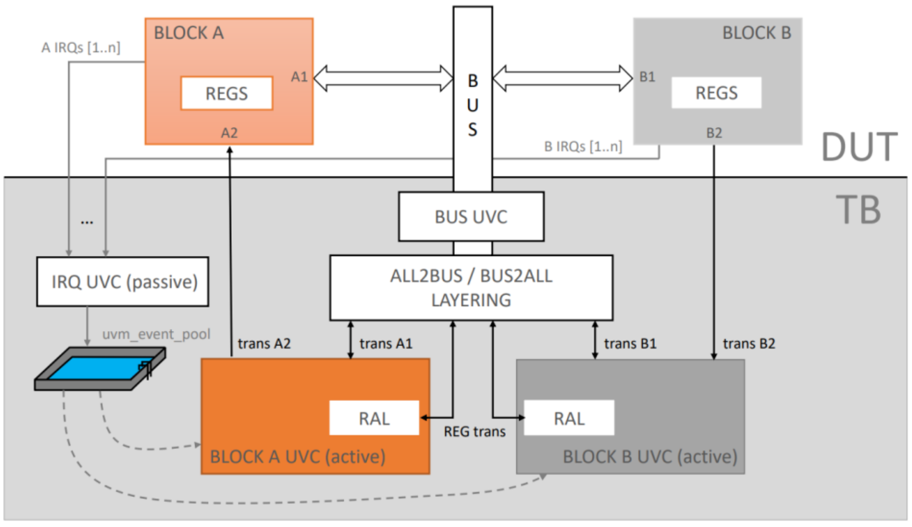


Figure 1: Block-level control and data flow example for *Block A*.

A verification engineer will usually implement a dedicated block-level test-bench for *Block A* while providing stimuli to all the needed interfaces by UVM drivers and monitoring the responses by UVM monitors/subscribers. However, when thinking from the top-level test-bench perspective, a different approach needs to be applied, and more implementation time to be invested at the beginning into the test-bench skeleton. This will save time in the future when verifying other sub-blocks and the entire system. he following three parts are essential for the test-bench skeleton:

First, *BUS UVC* must be implemented. It is responsible for handling bus transactions. At the same time, this UVC is a mediator between the DUT and the test-bench to ensure consistent communication between UVCs and their counterpart DUT blocks. *BUS UVC* is depicted in all Figures describing different verification stages: block-level verification (Figure 1), top-level verification without CPU (Figure 9), and top-level verification with CPU (Figure 10). An example of *BUS UVC* can be *AHB-Lite UVC*, containing standard UVM components such as AHB-Lite virtual interface, sequence item, sequences in the sequence library, driver, monitor, agent, environment, or base test.

Second, the translation of all UVC transactions to bus transactions and backward translation of bus transactions to UVC transactions must be executed to achieve independence of the interfaces of the UVCs from the test-bench skeleton. For this purpose, a block called *ALL2BUS / BUS2ALL layering* was implemented (visible in all Figures describing different verification stages).

Figure 2 and Figure 3 demonstrate some code from the layering, in particular, the translation of the AHB-Lite transaction's size to the register transaction's size and vice versa.

```
/**
  * Transforms AHB HSIZE into number of bits
  * @param size - AHB HSIZE
  * @return number of bits in transaction
  */
function integer ahb_2_register_subscriber::translate_size(ahb_lite_trans_size_t size);
   case (size)
     AHBL_BYTE : translate_size = 8;
     AHBL_HALFWORD : translate_size = 16;
     AHBL_WORD : translate_size = 32;
     default : translate_size = -1;
   endcase
 endfunction : translate_size
```

Figure 2: Size translation between the AHB-Lite transaction and the register transaction.

```
/**
  * Translate number of bits into corresponding AHB HSIZE value
  * @param n_bits - number of bits in transaction
  * @return AHB HSIZE value
  */
function ahb_lite_trans_size_t register_2_ahb_sequence::translate_size(int n_bits);
  int n_bytes = (n_bits-1)/8+1;

  case (n_bytes)
    1 : translate_size = AHBL_BYTE;
    2 : translate_size = AHBL_HALFWORD;
    3 : translate_size = AHBL_WORD;
    4 : translate_size = AHBL_WORD;
    default : translate_size = AHBL_MAX_SIZE;
  endcase;
endfunction : translate_size
```

Figure 3: Size translation between the register transaction and the AHB-Lite transaction.

Third, a global register model (*Register Abstraction Layer*, RAL) should be implemented. It contains the global register block, the register adapter, and the register environment. When the register block for a UVC is available, it is registered as a sub-block to the global register model. See function `connect_reg_block` in Figure 4. Thanks to this inclusion, it is possible to access all registers hierarchically while implementing the tests.

```
/**
 * Connect - connecting block-level uvm_reg_block to the proj_reg_block
 * @param phase - UVM phase
 */
function void proj_reg_block::connect_reg_block(uvm_reg_block reg_block,
                                                uvm_reg_map reg_map,
                                                string hdl_path  = "",
                                                bit[31:0] offset = 0);
  //Connecting register block to global register block, and fixing HDL paths
  reg_block.clear_hdl_path("ALL");
  reg_block.add_hdl_path(hdl_path);
  reg_block.set_parent(this);
  add_block(reg_block);
  //Adding reg_map from reg_block to project register map
  m_reg_map.add_submap(reg_map, offset & 32'hFFFF_FFFF);
endfunction : connect_reg_block
```

Figure 4: Function for connecting the UVC's register block to the global register block.

In general, UVCs may communicate various types of transactions towards DUT, monitor responses and compare them with reference data from predictors, or just analyze communication, e.g., to do coverage analysis. In the following text, it is described how such communication can look like while being abstract enough to give a reader an option to adapt these ideas to different verification scenarios.

Figure 5 illustrates a detailed organization of *Block A UVC*, which communicates the following transactions:

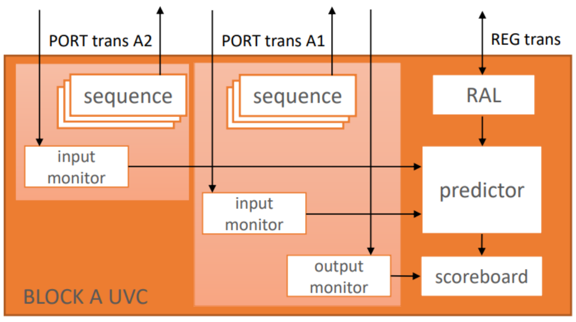


Figure 5: *Block A* details.

1. **Register transactions** (REG trans in Figure 5): RAL is used for verifying all the registers of *Block A*. The communication is bidirectional:

    - *Writing to DUT registers* (UVC stimulates *Block A*). WRITE register operations are initiated using RAL, which is a part of the UVC.

- *Reading from DUT registers* (*Block A* monitoring). Using RAL, READ, or MIRROR register operations are initiated to analyze the registers' changes in *Block A*, to check the results of performed operations (if these are stored to registers), or to check various register settings and flags. Monitored data can be automatically compared to the predicted data (a predictor inside the UVC can make the registers' prediction).

All register transactions must be translated to/from bus transactions. For this purpose, *ALL2BUS / BUS2ALL layering* is used.

2. **Over-the-bus port transactions** (PORT trans A1 in Figure 5): They indirectly operate with the port A1 of *Block A* via the central bus. From the *Block A UVC* point of view, bidirectional communication is performed again:

   - *Port A1 stimulation from UVC*. Stimuli originated in the UVC are thanks to the *ALL2BUS layering* translated to the bus transactions, which are sent directly to the port A1 of *Block A*. Furthermore, it can be noticed in Figure 5 that input stimuli can be (directly or via the depicted input monitor) forwarded to the predictor block of the UVC. The predicted reference outputs are sent to Scoreboard.

   - *Port A1 monitoring in UVC*. Outputs of *Block A* (visible as bus transactions) are translated via *BUS2ALL layering* to the output transactions for the UVC and sent via the output monitor to Scoreboard. Scoreboard compares reference and real transactions and reports errors if those two do not match.

   Over-the-bus port transactions allow engineers to perform block-level verification alternatively to RAL (register transactions) or to complement RAL.

3. **Direct port transactions** (PORT trans A2 in Figure 5): They allow for stimulating or monitoring the interface of the DUT directly from the UVC. This is a typical situation when the SoC contains various sensors or receives digital or analog data from the outside world. In this case, the UVC replaces such outside sources and provides data instead. In Figure 1, *Block A UVC* directly stimulates the port A2 of *Block A*. Again, the stimuli can be forwarded via the input monitor to the predictor.

Globally, the predictor of *Block A UVC* processes all stimulus types for *Block A*, either register stimuli or port(s) stimuli. This makes it possible to build a detailed prediction of *Block A* behavior. The recommendation is to pay attention to details while doing the block-level verification and limit the prediction when moving to the top-level.

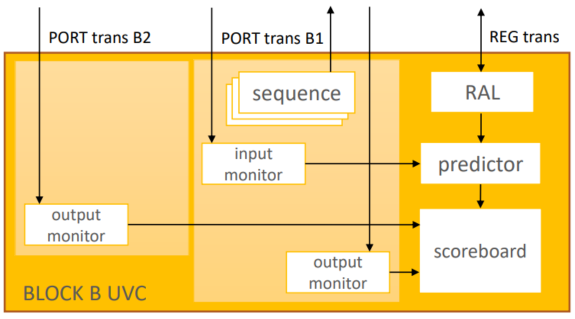


Figure 6: *Block B* details.

Block B UVC in Figure 6 fulfills a very similar function as *Block A UVC*. The only difference is that transactions on the port B2 are not directly driven but directly monitored. Therefore, only the output monitor is present for port B2 in *Block B UVC*. Data from this monitor are sent to Scoreboard for comparison to reference data. An example of such composition is that the DUT is producing data on its primary output ports, and these should be checked, for example, when external devices are driven, or data to the outside world are transmitted.

A. *UVM Event Pool*

To complete the description of Figure 1, SoCs usually contain one or more CPUs, which communicate with peripherals via the bus. When peripherals finish their operations, they notify the CPU using interrupts. Of course, when an interrupt system is present in the DUT, interrupts must be processed and checked in verification. An *IRQ UVC* is utilized in the presented test-bench; it collects all interrupts from various peripherals and triggers corresponding UVM events. These events are stored and available for the whole UVM test-bench thanks to the UVM event pool structure. Function `create_uvm_event` in Figure 7 registers newly created events to the global pool, and function `update_irq_event` triggers an event when an associated interrupt occurs.

```
/**
 * create_uvm_event - function for creating the event, protection in case of non-singleton UVM_EVENT
 * and non-singleton UVM_EVENTS cannot be created.
 * @param name - name of the event
 */
function uvm_event interrupt_subscriber::create_uvm_event(string name);
  //Definition of the event, and getting the global event pool handler
  uvm_event new_event;
  uvm_event_pool global_pool = uvm_event_pool::get_global_pool();
  //In case the event doesn't exists, create one, otherwise get it from the database
  if(!$cast(new_event, global_pool.get(name))) begin
    new_event = new(name);
    global_pool.add(name, new_event);
  end
  else begin
    new_event = global_pool.get(name);
  end
  return new_event;
endfunction : create_uvm_event

/**
 * update_irq_event - function for updating actual events, depending on the polarity of the IRQ.
 * @param t - IRQ transaction
 * @param irq_event - UVM event associated with interrupt
 */
function automatic void interrupt_subscriber::update_irq_event(irq_transaction t, uvm_event irq_event);
  if(t.irq_polarity) //If polarity is 1 then the IRQ was caught on posedge
    irq_event.trigger();
  else //If polarity is 0 then the IRQ was caught on negedge
    irq_event.reset();
endfunction : update_irq_event
```

Figure 7: Creating and triggering UVM events for interrupts.

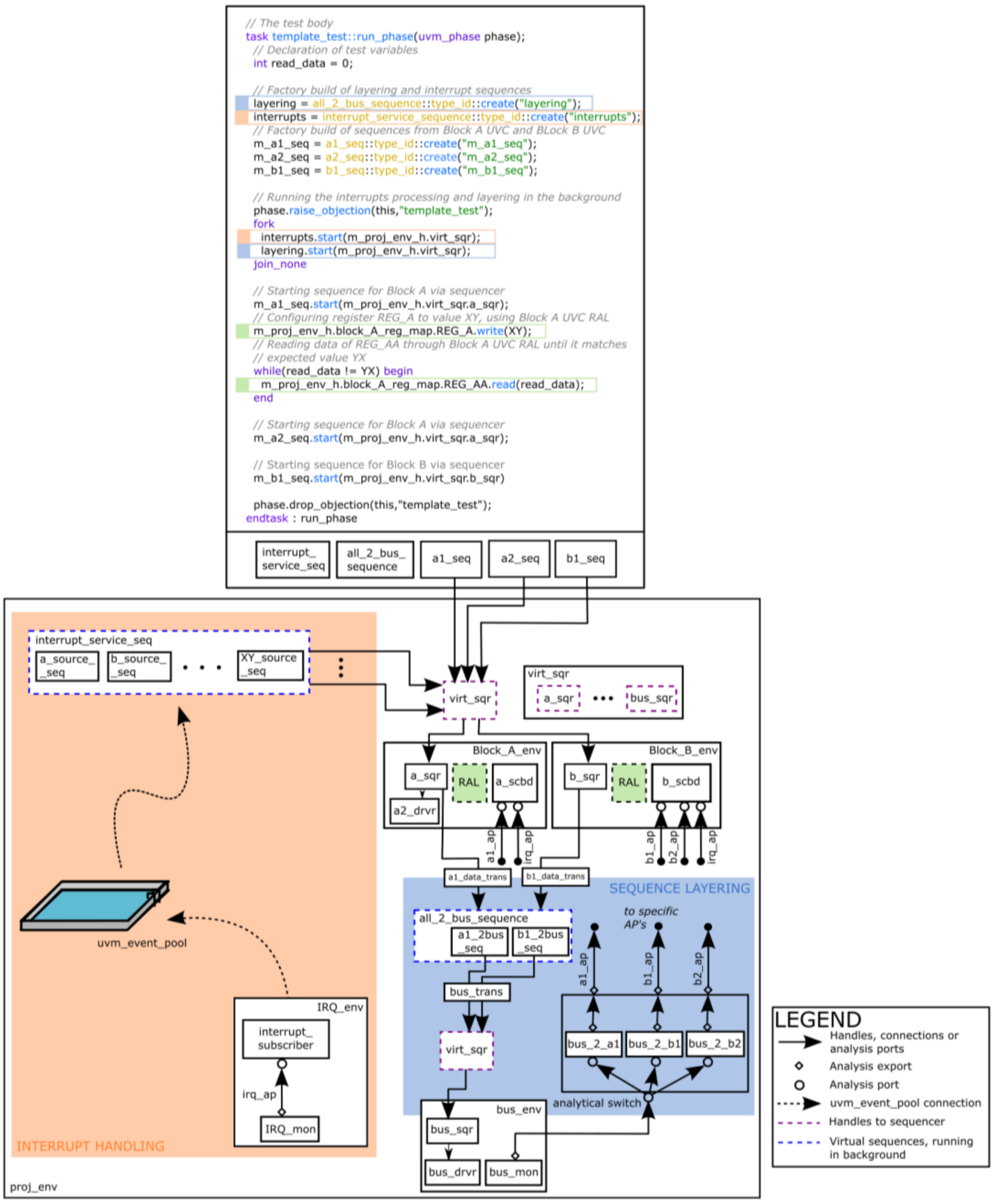
B. *UVM Sequence Layering*

ALL2BUS / BUS2ALL layering was already mentioned in the paper, as well as the idea that block-level verification from the top-level perspective is possible mainly because of the central bus use. It is essential to realize that due to the layering, it is possible to verify blocks of the SoC one after another or in parallel without having all the top-level modules assembled. Even the CPU is not needed now, although it will be the heart of the final SoC. To give a specific example: it is possible to start verifying *Block A* even when *Block B* does not exist yet, or when both exist at the same time, *Block A* can be verified by a different person than *Block B*, and every verification engineer can use his/her style of programming. Of course, both must understand the principle of layering.

C. *Configuration*

Configuration is a vital part of the proposed test-bench. All the UVCs are configured to the active mode at the beginning (while verifying blocks), as they are fully responsible for the verification process. Afterward, when moving to the top-level verification, configuration is needed to accommodate the test-bench and to switch some of the UVCs to the passive mode (CPU is taking a part of the stimulation to itself). As a reminder, an active agent has the driving part (sequence → sequencer → driver) as well as the monitoring part active (monitor → analysis components). A passive agent has only the monitoring part active. One UVC may contain more agents, but for simplicity, the scheme of one agent per one UVC will be used when explaining the concept. Therefore, all the UVCs can be marked as active or passive (see Figures).

Figure 8 provides an example of the test written for the architecture with *Block A* and *Block B*. Parts of the test and the figure with the same color are bound together. To explain the test flow, please follow the code comments.



```
// The test body
task template_test::run_phase(uvm_phase phase);
  // Declaration of test variables
  int read_data = 0;

  // Factory build of layering and interrupt sequences
  layering = all_2_bus_sequence::type_id::create("layering");
  interrupts = interrupt_service_sequence::type_id::create("interrupts");
  // Factory build of sequences from Block A UVC and BLock B UVC
  m_a1_seq = a1_seq::type_id::create("m_a1_seq");
  m_a2_seq = a2_seq::type_id::create("m_a2_seq");
  m_b1_seq = b1_seq::type_id::create("m_b1_seq");

  // Running the interrupts processing and layering in the background
  phase.raise_objection(this,"template_test");
  fork
    interrupts.start(m_proj_env_h.virt_sqr);
    layering.start(m_proj_env_h.virt_sqr);
  join_none

  // Starting sequence for Block A via sequencer
  m_a1_seq.start(m_proj_env_h.virt_sqr.a_sqr);
  // Configuring register REG_A to value XY, using Block A UVC RAL
  m_proj_env_h.block_A_reg_map.REG_A.write(XY);
  // Reading data of REG_AA through Block A UVC RAL until it matches
  // expected value YX
  while(read_data != YX) begin
    m_proj_env_h.block_A_reg_map.REG_AA.read(read_data);
  end

  // Starting sequence for Block A via sequencer
  m_a2_seq.start(m_proj_env_h.virt_sqr.a_sqr);

  // Starting sequence for Block B via sequencer
  m_b1_seq.start(m_proj_env_h.virt_sqr.b_sqr)

  phase.drop_objection(this,"template_test");
endtask : run_phase
```

Figure 8: The test flow with the proposed test-bench.

IV. Top-Level Verification Without Firmware

A strategy in top-level SoC verification can be moving incrementally from the simplest blocks toward the more complex block groups. When the top-level is fully assembled, the strategy can be to verify without the CPU (the verification is driven only from the UVCs), or to include the CPU into the verification process and thus end up with HW-SW co-verification where firmware drivers are involved. Let us have a look at the top-level verification without the CPU first.

Figure 9 shows the same organization of an SoC as in the previous section, but both UVCs (*Block A UVC*, *Block B UVC*) are active at the same time. The real SoCs are much more complex, but the principle remains the same. When a new peripheral should be connected to the bus in the SoC, the steps are the same - an additional block is connected to the DUT, an additional UVC is connected to the test-bench, new interrupt(s) are added, the UVC's register model is included into the global register model, and a translation of new transactions is added to the layering. If the SoC contains more than one bus, the UVM environment is updated with another UVC and transparent layering.

While block-level verification focuses on individual blocks, top-level verification considers how these blocks interact and collaborate. From this perspective, a suitable configuration can deactivate some parts of the UVCs. For instance, deactivating detailed prediction for *Block A*, minimizing the printing of debugging information or coverage measurement, or other measures to accelerate verification by optimizations.

For a particular example, both *Block A UVC* and *Block B UVC* in Figure 9 are marked as active; they drive writing to the registers of *Block A* and *Block B* in the DUT, they drive stimulus to their ports using transactions A1 and A2 for *Block A* and transactions B1 for *Block B*. It is reasonable because the UVCs are responsible for driving the whole verification. They are, in fact, the masters of communication, as the CPU is disconnected now.

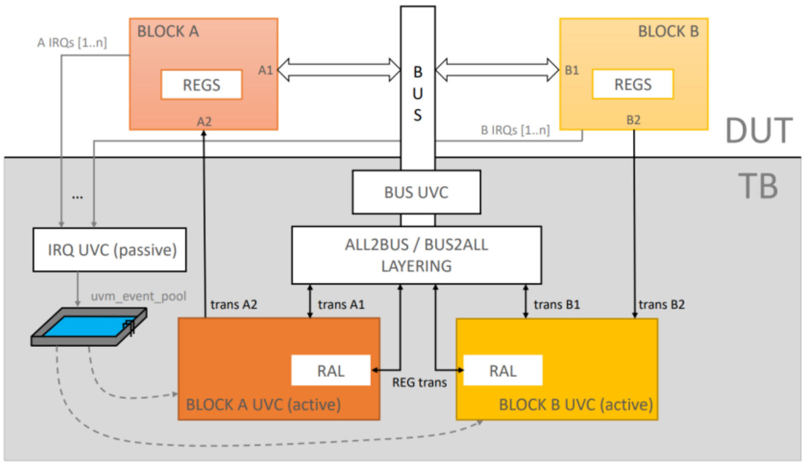


Figure 9: Top-level control and data flow example when CPU is not used.

V. Top-level Verification With Firmware

If one or more CPUs are present in the SoC (let us assume those are already verified in a separate test-bench or delivered as an IP), integration with the HW must be verified, i.e. HW-SW co-verification. Firstly, the interaction of the peripherals with the CPU must be checked, and secondly, it should be verified that the firmware drivers or the product firmware are correct. This, however, requires some adaptation of the test-bench and the test cases.

In this verification phase, the focus is on the interconnection of blocks and their communication, but this time the CPU (or CPUs) is the master of communication. The CPU performs various operations, such as executing a firmware program's instructions, accessing the peripherals' registers, activating their operations, sending data in, and collecting responses and results of the operations based on triggered interrupts. The UVCs and the entire test-bench must be adapted to this situation; the CPU and the UVCs cannot fight for mastership (their commands may collide). The adaptation is done in two ways:

1. The running UVM test is synchronized with the executing firmware. When the firmware initiates commands, the UVM test must wait, and vice versa. Without going into much detail, there are C functions defined for the firmware, such as *c2sv_wait_sync()* or *c2sv_provide_sync()*, and very similar SystemVerilog functions at the UVM side, *sv2c_wait_for_sync()* and *sv2c_provide_sync()*. These four functions can establish a handshake between the test-bench and the executing firmware application. Furthermore, debugging information can be printed, and an exchange of data is accomplished using *set_data()* and *get_data()* functions.
2. Some UVCs are switched to the passive mode. In Figure 10, *Block A UVC* remains active because it still sends A2 transactions to port A2 of *Block A*. However, stimulation using A1 transactions is deactivated together with the writes to registers (only register monitoring remains enabled). The reason is that these operations are now the full responsibility of the CPU - it writes to registers and drives port

A1. *Block B UVC* is switched to the passive mode; it monitors the states of the registers and checks data on the port B2. If needed, it is still possible to have prediction active in both UVCs, even if they do not actively participate in the stimulation of the DUT. Input monitors inside the UVCs may catch transactions initiated by the CPU and forward them to predictors (this is not illustrated in Figure 10).

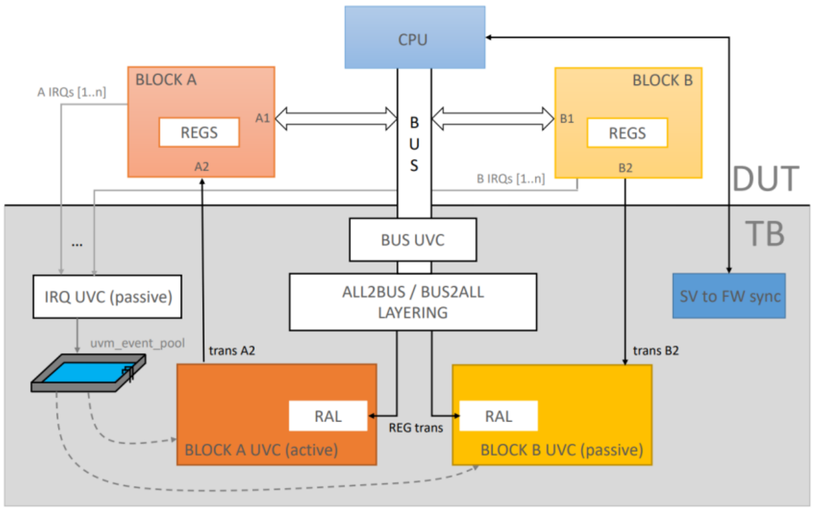


Figure 10: Top-level control and data flow example when CPU is used.

VI. Reuse Options

The potential for reuse is extensive in this type of test-bench. Let us consider a situation when a new test-bench is planned for an SoC containing previously verified blocks and some new ones. What possibilities exist for reusing the old test-bench?

1. The same test-bench skeleton can be used. Based on the central bus (or buses), *BUS UVC(s)* and *IRQ UVC* for processing interrupts are created/reused. If the central bus is different than before, *ALL2BUS/BUS2ALL layering* must be updated, meaning that transactions coming from the UVCs must be translated to the bus transaction and vice versa. The skeleton also contains an empty global register model, where register models of new or reused UVCs must be included.
2. Already verified UVCs from older projects can be reused. It can happen, of course, that their configuration must be adjusted or the functions updated somehow, for example, to extend the supported functionality. If the verification engineers followed UVM and configuration principles from the start, this should be possible, including RAL.
3. If firmware drivers are updated, the test-bench side changes comprise updating the tests. Regardless, the new project usually requires significant updates of tests or at least comprehensive revision. Plus, new test cases are added to the test set.

VII. Conclusion

To summarize, the paper's main contribution is the UVM test-bench skeleton, which can be used and reused to verify SoCs. Thanks to the skeleton, it is possible to develop one test-bench and use it for block-level verification, top-level verification without a CPU, and HW/SW co-verification with a CPU. Furthermore, this test-bench allows parallel development of UVCs (verification engineers may work on different parts of the test-bench simultaneously and even at different levels) and a good level of freedom in UVCs implementation.

References

[1] Wilson Research Group and Siemens 2022, “The 2022 Wilson Research Group Functional Verification Study, Part 6”, https://blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/

[2] Siemens, Layering in UVM, http://verificationhorizons.verificationacademy.com/volume-7_issue-3/articles/stream/layering-in-uvm_vh-v7-i3.pdf, 2022.

[3] DVTalk 2021, How to use uvm_event and uvm_event_pool, https://dvtalk.me/2021/09/20/uvm-event/, 2021.

[4] VerificationGuide.com, UVM event pool example, https://verificationguide.com/uvm/uvm-event-pool/, 2022.