Verification strategy for pipeline type of design

Djuro Grubor
Vtool LTD, Rajiceva 1, 11000 Belgrade Serbia, djurog@thevtool.com

Introduction: In the world of very complex ASICs, the need for good methodology and proper strategy is continuously increasing. Big combinational logic is too complex for maintain and debug, so instead of this pipeline type of design is taking place.
A common practice is to have a complex IP with pipeline containing multiple sub cores dependent between themselves. The verification process of such IP is rather challenging since the sub cores communicate inside the block, and that communication needs to be followed and verified in verification environment without observing internal DUT signals.

Verification strategy: The strategy for pipeline verification needs to be developed to achieve a modular solution compliant with UVM methodology, so the recommended approach is to create UVM component representing each sub core. Pipeline type of design consists of several almost independent sub cores responsible for one part of the work. Output of one sub core is an input for the next. Depending on the configuration, some sub cores can be bypassed in specific situations. Each sub core has its own set of configuration and status registers and it is independent of other sub cores. Protocol between sub cores is custom solution up to designer, and from the verification perspective it is considered a black box, comprising data structure and handshake protocol. Data flow is a typical pipeline, so when processed by one sub core, the current command is passed to next sub core and new command is taken to be processed. On input and output side of each sub core, there is a fifo buffer, to avoid blocking condition that might occur in case one sub core is faster than the next. Each sub core must take command only when the command is valid, to prevent data corruption, and part of logic should be blocking and part non-blocking. Having all this in mind, main challenges are synchronization and back pressure handling in the system, and main advantages are distributed system and speed up of data and commands processing. UVM methodology provides a very comprehensive set of features allowing us to create suitable strategy for specific DUT. In most of verification environments, standard set of UVM components is used to build coverage driven constrain random testbench. To achieve that, for driving stimulus on DUT interface, protocol specific UVC is used, with set of parameters and configuration. Parameters are used to handle different width of buses and protocol specific part of transaction, and configuration allows simulation of constrained random delays and various stimulus types. Virtual sequencer is utilized to synchronize activity over all UVC sequencers in the environment, and basically it translates higher level items to lower level and provides connection of a UVC sequence to the physical sequencer. Beside that, each UVC is used for coverage collection, to gather stimulus coverage. As a part of DUT interface, environment can extract all information from transaction and use it as separate cover items. All transactions are stored within agent monitor component, and analysis port from monitor is connected to the environment scoreboard. Scoreboard subscribes to all monitors, collecting transactions from all interfaces, and using them to achieve prediction and model expected behavior of the DUT. RAL component with shadow registers is instantiated, and it represents a good way of checking status values, beside storing the configuration registers content. Connection between DUT and verification environment is done via environment interface, and after connecting interface with DUT, env interface handle is passed to environment over uvm_config_db. Uvm_config_db is standard way of passing handles between different UVM components, and it allows to assign exact user of the handle or provide it to a wider range of users. Apart from standard UVM components, which take part in stimulus generation, tracking and checking, some additional UVM components have been utilized to overcome the challenges posed by the DUT architecture.
The list of utilized components includes uvm_tlm_fifo, uvm_blocking_get_port, uvm_nonblocking_put_port, as well as the semaphores. uvm_tlm_fifo provides storage of transactions between two sub core models in the verification environment. Transactions are put into the fifo via one of put_export methods. On the other side, transactions are fetched from the fifo in the order they arrived via one of the get_peek_export methods. Size of the uvm_tlm_fifo is configurable, so it can realistically model fifo between pipes in DUT. Also, there are predefined methods to check size, to check for reached level, whether the fifo is empty or full, and finally, what is very important, to flush the fifo. Flush is critical, because during the reset stage, all components should be reset. If some transaction is left out in uvm_tlm_fifo, after the reset is released, the environment might encounter wrong behavior. UVM's TLM port and export implementations allow connections between ports whose interface signatures are not an exact match. For example, an *uvm_blocking_get_port* can be connected to any port, export or imp port that provides *at the least* an

implementation of the blocking_get interface, which includes the *uvm_get_** ports and exports, *uvm_blocking_get_peek_** ports and exports, and *uvm_get_peek_** ports and exports. A blocking interface conveys transactions in blocking fashion; its methods do not complete until the transaction has been successfully sent or retrieved. Because delivery may be time-consuming, the methods in such interface are declared as tasks. A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and cannot accept the request), the methods may return a failed status. A combination interface contains both the blocking and non-blocking variants. Because SystemVerilog does not support multiple inheritance, the UVM emulates hierarchical interfaces via a common base class and interface mask. The *put* interfaces are used to send, or *put*, transactions to other components. Successful completion of a put guarantees its delivery, but not execution. The *get* interfaces are used to retrieve transactions from other components. The *peek* interfaces are used for the same purpose, except the retrieved transaction is not removed from the fifo; successive calls to *peek* will return the same object. Combined *get_peek* interfaces are also defined. The UVM provides unidirectional ports, exports, and implementation ports for connecting components via the TLM interfaces. Ports are instantiated in components that *require*, or *use*, the associate interface to initiate transaction requests. Exports are instantiated by components that *forward* an implementation of the methods defined in the associated interface. The implementation is typically provided by an *imp* port in a child component. Imps are instantiated by components that *provide* or *implement* an implementation of the methods defined in the associated interface. In this approach, uvm_blocking_get_port will be used, because each sub core component should wait from command on its input, and then proceed with execution, and once it is finishes that execution, it will look back on uvm_blocking_get_port and wait for the next item. This component also performs synchronization. On the other side, we will use uvm_nonblocking_put_port, because uvm_tlm_fifo is controlling fifo depth. High-level and easy to use synchronization and communication mechanism are essential to control the kind of interaction that occurs between dynamic processes used to model a complex system or a highly reactive verification environment. Beside static object synchronization with -> and @ operators, SystemVerilog provides additional techniques to synchronize dynamic processes: semaphore, mailbox and events. Semaphore is used for lock/unlock of a commonly shared resource. Depending of the state of semaphore, if unlocked, the resource can be used, and if locked resource can't be used and the source should wait for the semaphore unlock. Mailbox is a mechanism to exchange messages between processes. Data can be sent to mailbox by one process and retrieved by another. Fifo can be used in mailbox if needed. SystemVerilog events hold Verilog functionality events, plus time-step in which the event is triggered and events act as handle to synchronization queues, so they can be passed as an argument to task and compared. In this approach, semaphore will be used to synchronize all sub core components. Semaphore supports get() and put() methods, which are used for locking and unlocking resource. Additionally, try_get() method can be used for querying information whether a resource locked or not. Each sub core component needs to be synchronized itself, when got from external uvm_tlm_fifo item, and start processing, it need to wait for access to external memory to fetch data, and then to proceed with execution and processing.

This main goal of the strategy is to remove all dependencies between design and verification components, such as clock accurate model and peek into internal signals etc., and also provide reusable environment for top-down verification. In this approach, each sub core component is completely independent, so we could just create one UVC and replace uvm_tlm_fifo on input and output, and drive real stimulus. We have self checking component, all interfaces to external memories, all logic and coverage. Each uvm_component can be used for standalone sub core(pipe) verification, but they can also be seamlessly integrated into top level verification as passive reference model. Also for future use of this environment, it is maintainable since any change request is related to specific sub core, therefore only one uvm_component is affected and subjected to change. Also, if one of the components is used on another project, it is very easy to integrate and reuse environment for just that one.

All uvm_components are connected with uvm_tlm_fifo, so each uvm_component is independent in receiving data from the previous and pushing it into the following pipe in the pipeline. In the figure 1 we can see connection and typical flow of data and order of process. Each sub core is waiting in infinitive loop for command in get_port, and it is blocked there. Once when command is present in tlm_fifo, get_port will pop command, and execution of sub core component will start. At that moment, component will take all part of configuration needed for execution, so execution will be valid for that movement of simulation. Next step will be to get data from external memory, or to write updated data to external memory, depending of the specification. During that period, all other task needs to be blocked, since more transactions are needed in order to collect all data. When we strobe last transaction from expected access, we can used stored data and manipulate with configuration and command, and create prediction and updates. With all of that, after we write in memory data, we are done with this sub core, and command needs to be shifted to next sub component. This is done over uvm_nonblocking_put_port, or over uvm_blocking_put_port if depth of uvm_tlm_fifo is critical point. In that case, we can simulate and have exactly the same behavior as in design, and simulate back pressure inside of design. When put_port finally put command to uvm_tlm_fifo, sub_core component goes back in state where it is waiting on blocking get_port for a new command. Command stored in uvm_tlm_fifo goes on the end of fifo, because order is very important. Next sub core component could be slower the previous one, so uvm_tlm_fifo keep original order of commands. Next sub core is using updated command, so next execution is using results from previous sub core, and it is much easier to debug. In pipeline type of design, some of common issues are related to fetch data synchronization and data processing. Besides that, missing of some data and taking corrupted data is also bi issue. In this approach, environment handle that in consideration, and it is tight with design, every transaction is checked and each operation is performed in same time as in design, couple of clocks is maximum misalignment, but on first transaction of next operation it is synchronized. All access to external memory, or memories, are done over analysis ports, and transactions are filtered out based on address space of each sub component.

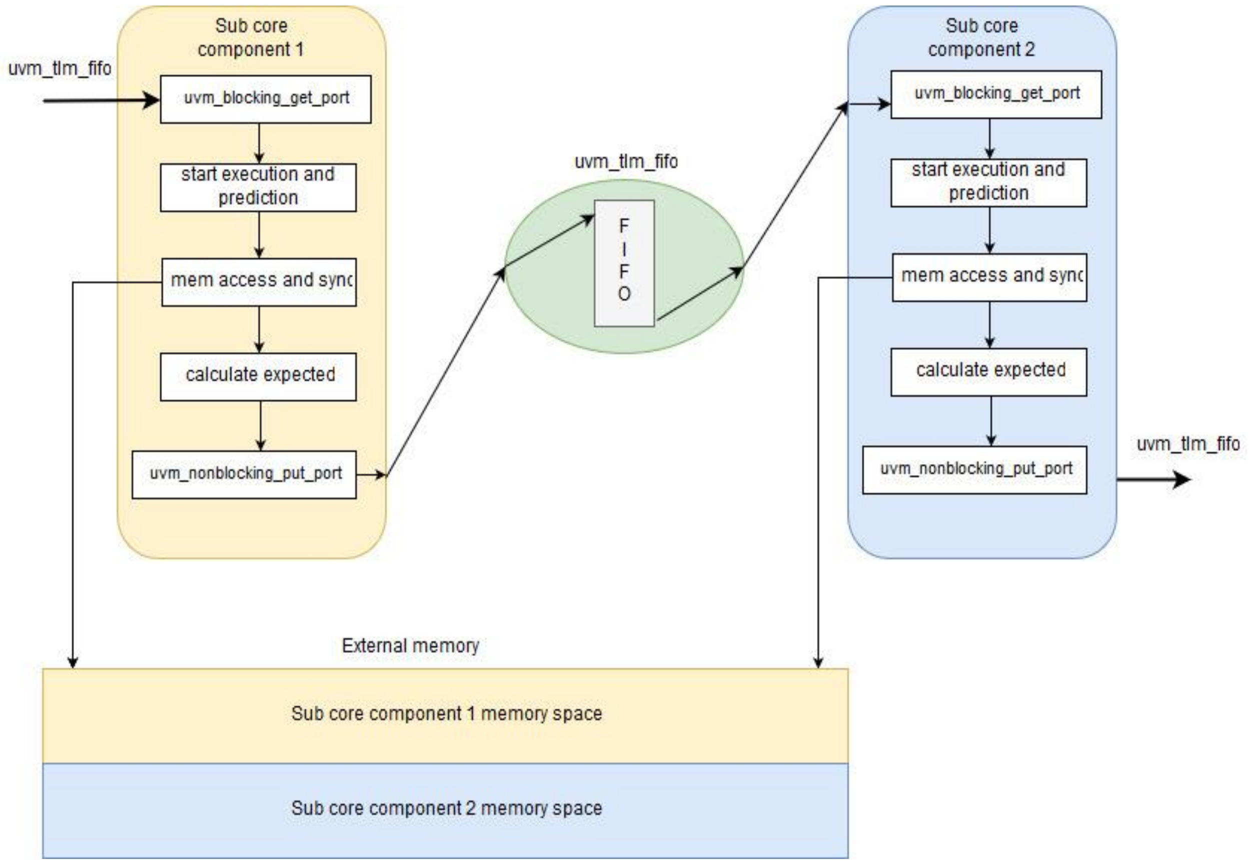

Figure 1. Detailed block diagram for to sub core components

Base code for each component looks like this:

```
class sub_core1 extends uvm_component;

   uvm_comparer                comparer;//used for comparing objects

   virtual env_if env_if;//hook to virtual interface

   ral_block_registers registers_regs;
```

```
    //local fields and variables

    `uvm_component_utils(sub_core1)

    uvm_blocking_get_port #(user_type_trans) core1_get_port;

    uvm_blocking_put_port #(user_type_trans) core1_put_port;

    uvm_analysis_imp_mem#(mem_tr , sub_core1) analysis_port_mem;

   function new (string name, uvm_component parent);

        super.new(name, parent);

        core1_get_port = new("core1_get_port",this);

        core1_put_port = new("core1_put_port", this);

        analysis_port_mem = new("analysis_port_mem", this);

        comparer = new();

    endfunction: new

    function void reset();//custom reset of sub core component

        //reset all parts of component to default state

    endfunction: reset

    function void  build_phase(uvm_phase phase);

        if(!uvm_config_db#(virtual env_if)::get(this, "", "env_if", env_if))

            `uvm_fatal("build_phase",$sformatf("virtual interface must be set for:
%s.env_if",get_full_name()));

        if (!uvm_config_db#(ral_block_registers)::get(this, "", "registers_regs",registers_regs))

        begin

            `uvm_fatal("SB build phase", "registers_regs block is not set ");

        end

    endfunction

    function void write_mem(mem_tr trans_mem);

        comparer.show_max = 1;

        comparer.sev = UVM_INFO;
```

```
      if(trans_imem.mem_addr <= `SC1_END_ADDRES) begin//defined ranges of memory for
                                                           all sub cores
        if(condition_last_trans_access)
            sub_core1_done =1;
  endfunction


  task run_phase (uvm_phase phase);
      while(1)begin
         core1_get_port.get(trans);//blocking port for pull out from fifo, while(1)
         if(bypass_conditions != 1)begin
            strat_working();
            wait(memory_access_done);
            perform_updates();
            if(sub_core1_done)
              core1_put_port.put(updated_ user_type_trans);
   endtask: run_phase
```

As we can see in code, every component has independent logic, but similar structure. So, one can be used as a template, and reused.
All sub core components are instantiated in global scoreboard, also all uvm_tlm_fifos are created and configured in global scoreboard. All connections are performed in connect phase of environment, and it have form like this:

```
sb.sub_core1.core1_get_port.connect(sb.tlm_fifo_core0_core1.get_export);

sb.sub_core1.core1_put_port.connect(sb.tlm_fifo_core1_core2.put_export);
```

Sub cores access the external memories where the necessary data is kept, so each uvm_component consists of an analysis_port as a dedicated interface towards the defined address space. Based on transactions to external memories, environment is synchronized with DUT behavior, so the flow of environment transactions from one uvm_component to uvm_tlm_fifo and finally to another uvm_component is performed in same manner as within DUT. When three sub core components access to the same interface and memory, but different address space, then all have same analysis port, each takes in consideration transactions from address range of interest. Each command requires different number of transactions and access to memory, so very important part of synchronization of this approach is to make good predictions in this field. Any missing transaction or prediction lead to losing synchronization. Some of sub core components can access more memories in parallel or in predefined order. This is easily solved with semaphores, so dead lock and mutual blocking conditions are avoided.
This solves a lot of problems, such as the modelling of the scenario in which each of the sub cores is accessing the external memory. Faulty modelling can lead to synchronization issues, potentially causing data corruption.
All uvm_component could be bypassed, without any time consumption. Blocking get_port will take item from

uvm_tlm_fifo, and if is set to be bypassed, because of configuration or some type of item, it will just pass item to put port, which is not blocking, and then next sub core component can take item. Since all these actions are non-time consuming, item skips a component in zero time, and then, when there is traffic in proper core on external interface, it is performed for valid item. Each uvm_component is reset by its own corresponding reset. Each component could have implemented reset_phase, or reset function could be implemented, depends on environment. It allows to follow phasing structure or some custom solution. A set of functions is provided to extensively cover behavior of DUT sub core, providing a self-checking environment. A global scoreboard instantiates all uvm_components for each sub core and can interact with any of them, relying upon the data and the flags from uvm_components. To eliminate the undesired dependency between the components, the communication between them is established by using UVM TLM elements, rather than directly between the uvm_components. Each uvm_component contains an instance of uvm_blocking_get_port to get transaction from uvm_tlm_fifo, and uvm_non_blocking_put_port for putting transaction into next uvm_tlm_fifo. All uvm_tlm_fifos are configurable according to design parameters for sub cores fifo on arrival or departure port. During reset, all uvm_tlm_fifo are flushed, so after the reset recovery, the environment behaves expectedly, clear for a fresh start.

Stimulus generation is completely constrained random, and independent from the environment components, and is therefore completely reusable. To apply stress onto the DUT, the provided sequences can be used in parallel with other flows and testing, so the main virtual sequences are built consisting of a number of parallel threads for different modes. All UVCs are randomly configured to introduce delay in responses, so back pressure is present from time to time. This structure is built for this purpose, to be flexible and stay aligned with DUT behavior, based only on external interface observation.

Modules that are sharing same resources are synchronized using semaphores. Also, it is applicable to collect coverage in each uvm_component for each sub core, achieving Metric -driven verification. As a benefit of the solution, the debugging process is facilitated and sped up, because it is localized within the first uvm_component which causes failure. Because of this modularity, the whole environment is easy to document.

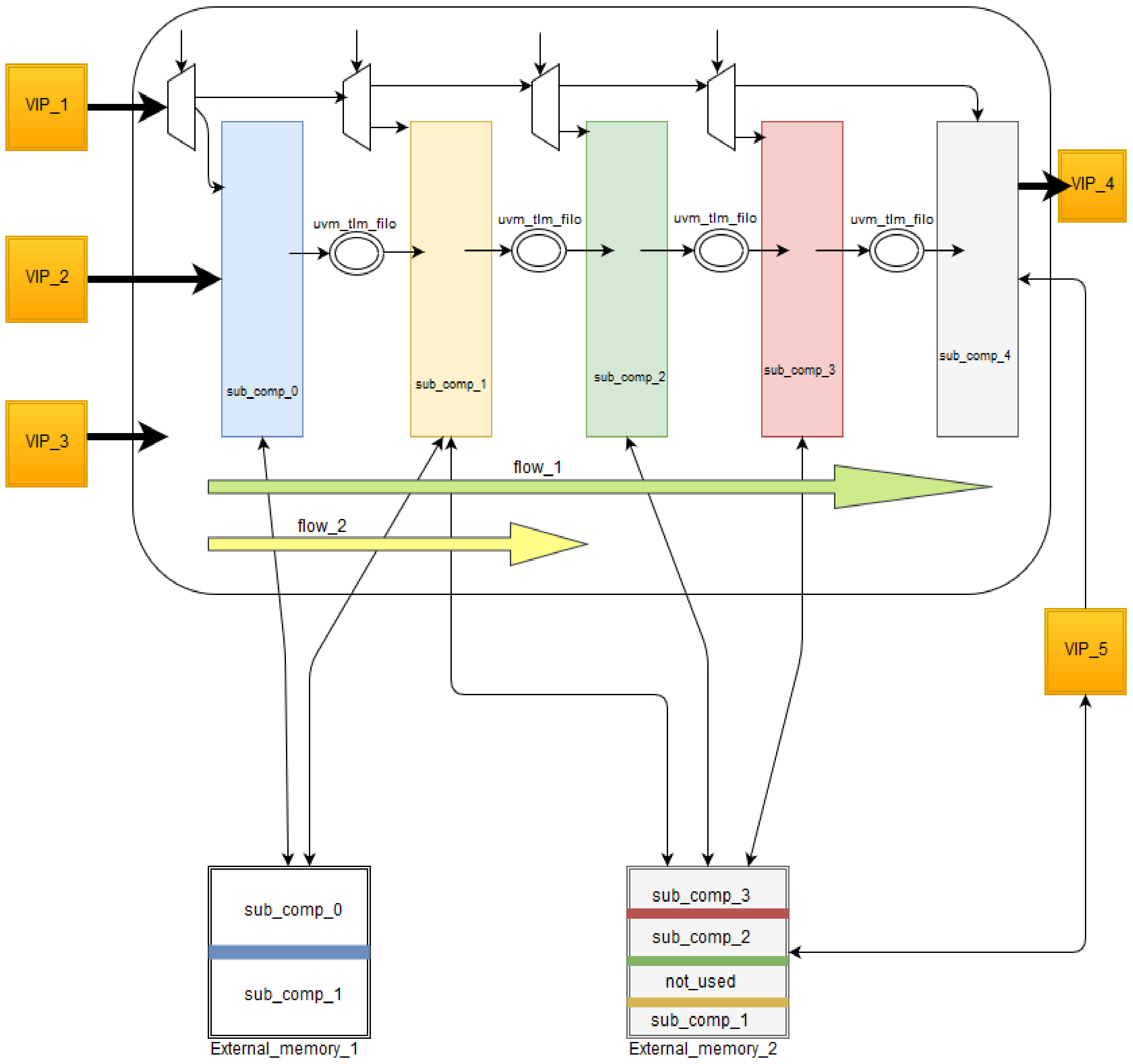

Figure 2. Block diagram of full environment

On Figure 2 is shown block diagram of full environment. Flow selection is controlled with VIP_2, VIP_1 sends logical command, and from that point environment works automatically. Based on configuration, sub_comp_0 will read from memory relevant data for this logical command, and start parsing and updating. When update is done, and sub_comp_0 I done with all operation, command is passed to uvm_tlm_fifo reside between sub_comp_0 and sub_comp_1. At that point of time, sub_comp_0 is ready to take new command and do all operation from beginning, and sub_comp_1 is waiting on uvm_blocking_get_port to take command from uvm_tlm_fifo. Other sub components are also blocked and waiting on get_port. Sub_comp_1 is getting command from uvm_tlm_fifo, and start reading from external memory from processing. In parallel, it reads and updates current state of sub_comp_1 in other external memory. That two operations are synchronized with semaphore, since one is dependent to other, after each part of processing different status is saved and updated. In scenarios like this, it is very important to create good synchronization flow, any issue in this part could be critical. With this approach, debugging is limited to only one sub component, and with localization of problem, is much easier to find the root of problem and solve it, either in design or environment. When sub_comp_1 is done, command goes to next uvm_tlm_fifo, and then next subcomponent. At the end, when last subcomponent is done, all data is check and compared. Common issue is that part of one command reflect other command, some flags stayed leached, or some corrupted data is taken into consideration in wrong moment. In that case, in big environments, tracing is extremely time consuming, so when it is divided into small logical parts, it is much faster and logical. On each write into all external memories, all check is performed, so at first wrong write, address, data, protocol, first error appearance will point out where problem is started, so root will be known. In case of reset, each sub core component should be reset independently, and uvm_tlm_fifos should be flushed so after reset state of env is idle and empty.

Summary: The biggest challenges encountered while developing this strategy is to choose proper get and put port elements for each component, configure all components, and finally take proper approach in handling transactions in each component. At the beginning it was really challenging to get the environment set up and running, but later it was very easy to update and work in it, since the workload was split between components. Even different engineers could develop different components, so whole process of verification could be faster. The main room for potential improvement would be to develop the uvm_components independently and use them for sub cores standalone verification, and only then integrate them into block environment and as a final step, integrate this blocks environment into top level environment.

In comparison to one scoreboard for whole block solution, this solution is very advantageous. Code is much more readable, easier to debug, and maintain, reusable, easier to synchronize and document. With some modification and improvements, this approach of splitting the DUT modelling into units and create a dedicated separate component for each of the units, regardless of the number of inputs and outputs, could emerge into a general solution for complex DUTs with stage part of processing.