A novel approach to create multiple domain based DV architecture to address typical Verification challenges, for the DUT with mutual exclusive functionalities, using UVM Domains

Author-1
Subham Banerjee
Xilinx Asia Pacific Pvt. Ltd.
5 Changi Business Park
Singapore – 486040

Author-2
Keshava Krishna Raja
Cisco Systems India Pvt. Ltd.
Cessna Business Park
Bangalore – 560103

1. Introduction

Designs with mutually exclusive functional islands are around for quite some time. For an example, multiple reset-island, mutually exclusive Ethernet/Interlaken pipes within MAC Layer, etc. These mutually exclusive design-islands can be reset, configured or re-configured independently.

To be more specific, our DUT which is a Line-Side-Gigabit-Transceiver, can support 8 Ethernet lane, and each lane can be configured in different line-rate and can be used as isolated, or combined together to make 10G to 100G Ethernet pipe. So for a DUT like this, lanes can be configured and reconfigured at any given time without disturbing other lanes.

To support this mutual exclusivity within a DUT, the challenges in Design Verification increases. With ever increasing complexity of data processing and dynamic traffic handling, it becomes harder to address all verification requirement, without a structured approach.

The proposed method addresses these challenges by devising a fully automated, scalable and reusable environment based on UVM_DOMAIN and runtime UVM_PHASES. UVM_DOMAIN, creates and controls the verification environment in such a way that basically mimics the mutual exclusivity in DUT.

UVM_DOMAIN is the inbuilt UVM base class, which works like a bridge between all the UVM_COMPONENTs and UVM_PHASE class. By default all the UVM_COMPOENTs are part of a default domain. As a result, all components are in sync as per as phasing in concerned; which means ‘next_phase’ of any component will only start when ‘current_phase’ of all the components are completed. Till that time, the components which already completed the ‘currect_phase’; can’t proceed further in phase execution.

The proposed methodology recommends to create multiple of such domains, based on the required mutual exclusivity of the specific DUT. This will give an advantage to control phasing of different portion of the testbench in asynchronous manner to each other. This means one portion of testbench which belongs to one

particular domain (say domain0); can proceed, continue or jump across UVM_PHASES without disturbing other components that belongs to another domain (say domain1). Hence the mutual exclusivity within the same testbench is achieved.

2. Concept Description

A typical UVM based environment consists of multiple INTERFACE-UVCs (Interface Verification Component), MODULE-UVCs (Module Verification Component). These INTERFACE-UVCs (IVC) house single or multiple agents with sequencer, driver, monitor and the sequences. Typically for each DUT interface we have one INTERFACE-UVC and they are responsible for creating and driving packets/stimulus into DUT interface. And MODULE-UVCs (MVC) will contain the scoreboards, checkers and DUT configuration class etc. Module-UVCs are responsible for DUT configuration, initialization, and checking the correctness of the logic.

By default there is one domain called 'UVM_DOMAIN'. All the DV components, such as INTERFACE-UVCs and MODULE-UVCs are implicitly mapped to this domain [Figure 2], during build phase.

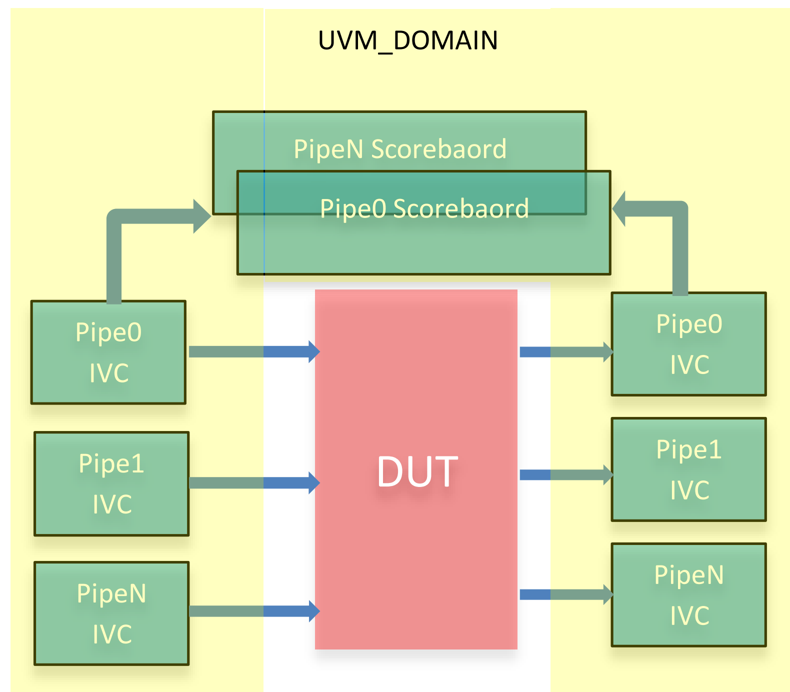

Figure 1 Single Domain Testbench for multidomain DUT

This single domain approach has some inherent shortcomings, when we need DUT-like mutual exclusivity in a verification environment. With single domain, the environment will have a single phase-scheduler;

which will control the phasing of all the components together. But for mutual exclusivity, we need independent phase scheduling, and selective controllability of each phase scheduler.

For an example DUT [in Figure 2], it has four different ENET pipes (each pipe consists of multiple ports). Each pipe can be configured as 10GE, 100GE, or 40GE modes. These configurations can change dynamically for single or multiple pipes. To achieve this, Testbench needs to be very flexible and responsive towards the DUT activities during this transition. Testbench also needs to re-configure DUT/Pipe based on the new modes. Below are the primary steps the Testbench needs to take care of:

1) Need to have the detailed information about old and new modes with the pipe information, which is getting reconfigured.
2) Reset only the logic for the effected pipe.
3) Stop the traffic generation from the agents, which are associated with this pipe.
4) Make a phase jump of the selected domain, which is mapped with this pipe, to reset_phase. All the DV components that are mapped with this domain will start afresh from the reset phase.
5) Agents will get re-initialized and re-configured based on the new mode.
6) Scoreboards, which belong to the effected pipe, will get re-initialized and all the stale data will get flushed out.
7) TB configuration-class (uvm component) will start the reconfiguration of the DUT/pipe based on new mode.
8) Once the configuration is done, the Agents will resume traffic and scoreboarding will start for the reconfigured pipe.

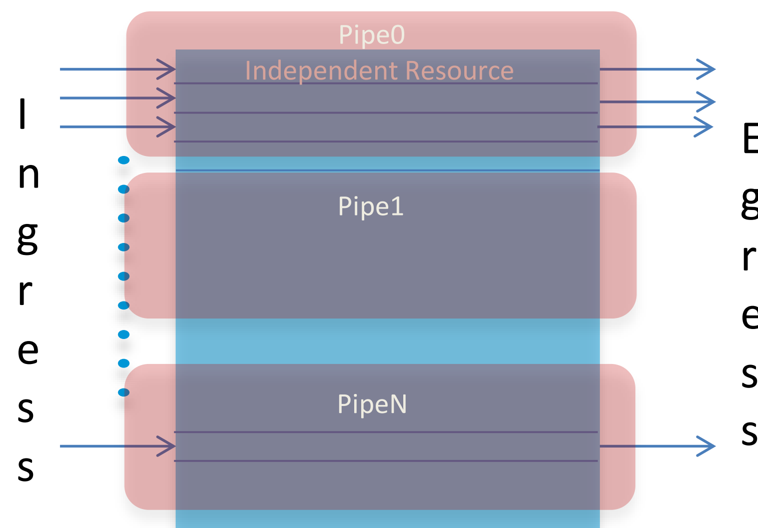

Figure 2 Multiple Independent pipes

The Testbench should be flexible enough to do all the above steps automatically for any pipe, at any time, and for any kind of mode transition. To achieve that, following points should be kept in mind while architecting a multi-domain DV environment [Figure 3].

1) Identify the mutual exclusive domains in DUT.
2) Create those many uvm domains in DV environment.
3) Map DV component with a particular domain based on its associativity with DUT. For an example Pipe0 has one agent, one scoreboard. Those components should be mapped to pipe0_domain. The same should be repeated for other pipes.
4) DUT configuration class which includes all constraint to ensure legal/supported modes for each pipe, will have a separate domain. This needs to be synced with all domains and whenever there's a pipe_domain junp, configuration class also should jump to reconfigure the new modes.

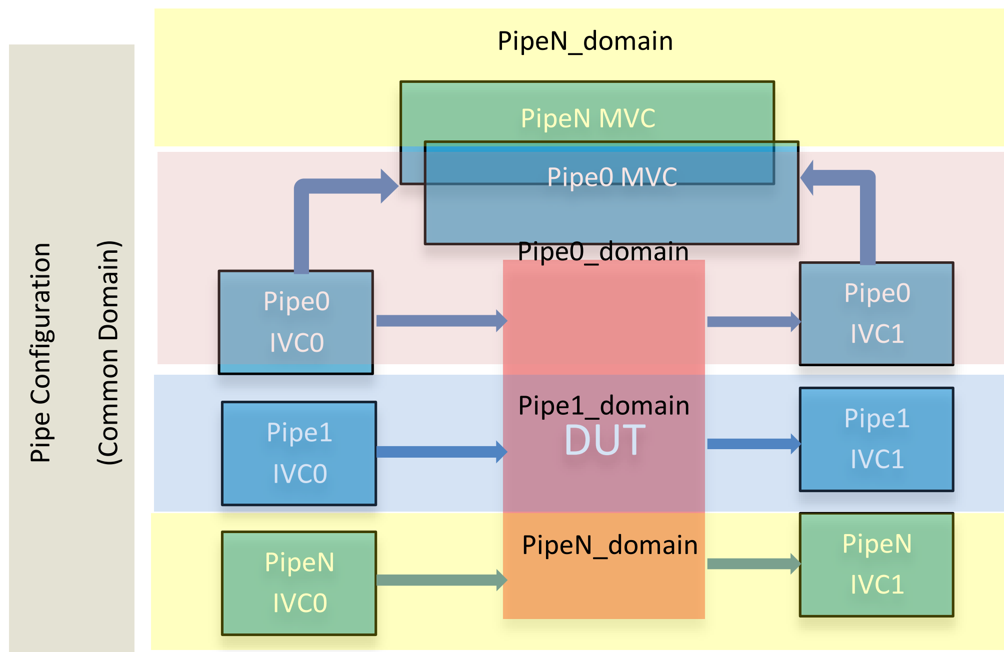

Figure 3 Multi Domain Testbench for multidomain DUT

3. Implementation

In multiple domain approach, each DV component is mapped to a corresponding domain based on its association with DUT. Since each 'Domain' can be associated with one or more DV components, and they can be at any hierarchy in the environment, TEST-class is chosen to create and map the domains.

Since all the domain based mapping can be done from the top level, so this feature can be appended/added to any existing Testbench framework as well, without much hassle. This is a huge advantage, which avoids lot of re-work and code change.

i) As mentioned, the first steps is to create the domains, as shown below.

```
class user_test extends uvm_test;
…..
uvm_domain m_pipe_domain[N];
uvm_domain m_common_domain;

function new(string name, uvm_parent parent);
  ……
  foreach (m_pipe_domain[i])
    m_pipe_domain[i] = new ($sformatf("m_pipe_domain[%0d]",i);
    m_common_domain = new ("m_common_domain");
endfunction
….
endclass
```

ii) Next is to map the DV components to its corresponding domains as shown below:

```
class user_test extends uvm_test;
….
function void connect_phase (uvm_phase phase)
    ……

   //Recursive:
     foreach (m_pipe_ivc[i])
       m_pipe_ivc[i].set_domain(m_pipe_domain[i],1);
     foreach (m_pipe_mvc[i])
       m_pipe_mvc[i]. set_domain(m_pipe_domain[i],1);
   //Non-Recursive (When all the components under ivc is not
   //intended to be mapped):
     foreach (m_usr_ivc[i])
       m_pipe_ivc[i].set_domain(m_pipe_domain[i],0);
     foreach (m_pipe_mvc[i])
       m_pipe_mvc[i]. set_domain(m_pipe_domain[i],0);

    m_config.set_domain(m_common_domain)
endfunction
…
endclass
```

iii) The recursive and non-recursive mapping is solely based on the DV requirement.

iv) There are some scenarios, where two different domains need to be in sync all the time. This means if there's a configuration component which needs a phase-jump whenever any other domain goes through the same. Thus any two or more domain can be in sync.

```
class user_test extends uvm_test;
....
function void connect_phase (uvm_phase phase)
    ......
    ......

   foreach (m_pipe_domain[i]) begin
      m_common_domain.sync (m_pipe_domain[i]);
   end
  //NOTE: There's unsync API as well.

endfunction
...
endclass
```

v) Once the mapping is done, next step is to selectively control the jumping of the phase scheduler of each domain, during a simulation when needed.

```
class user_test extends uvm_test;
....

function void main_phase (uvm_phase phase)
  ......
  foreach (m_pipe_domain[i]) begin
    //If pipe 'i' is getting reconfigured
    if (reconfired_pipe[i])
      m_pipe_domain[i].jump (target_phase::get());
    end
   m_common_domain.jump (target_phase::get());
endfunction
.........

endclass
```

vi) Simulation execution flow before and after 'jump' [Figure 4]. In the following diagram target_phase has been set to reset_phase. Pipe 1 and 3 are jumping to reset phase while other pipes are continuing un-disturbed.

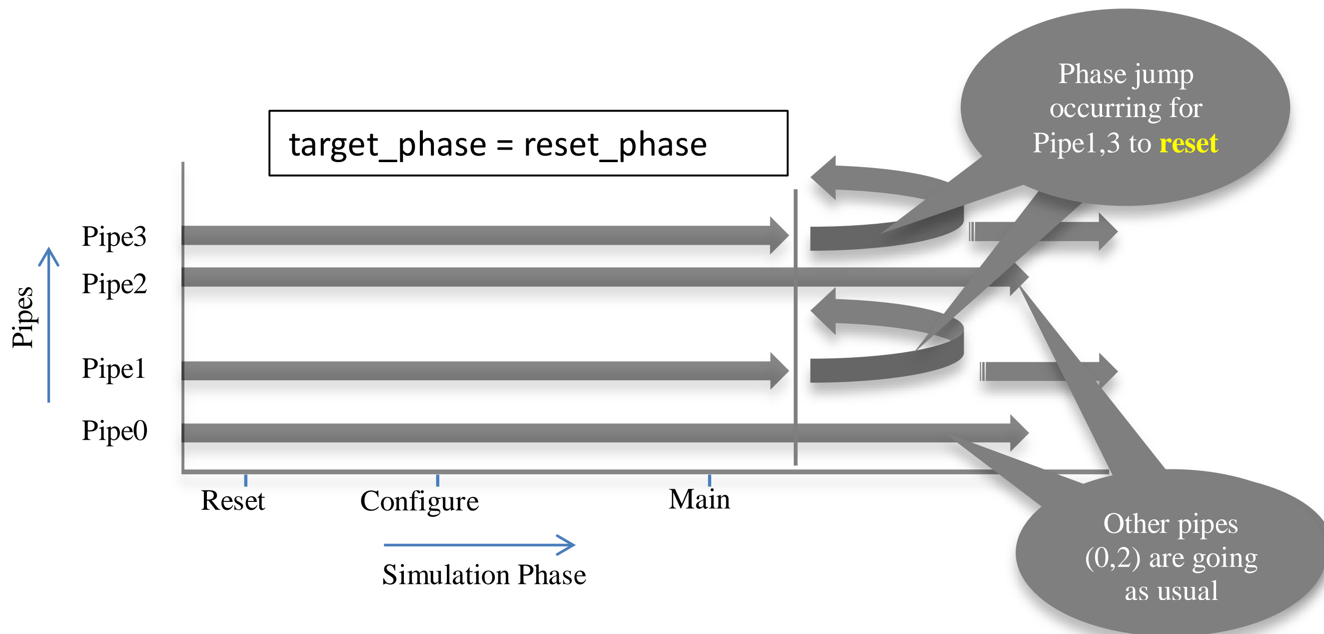

Figure 4 Phase jumping effect on simulation

4. Results & Analysis

i) Once the environment is ready and loaded with all these flexibilities, it's very straight forward to achieve all possible mode transitions for one or more randomly selected pipes, in a fully automated way.

ii) Four tests covered almost 400 odd unique mode transitions in a multiple Pipe Ethernet DUT, leveraging this environment.

iii) Almost 120 odd DV components were handled in fully automated fashion during these transitions.

iv) More common usages of multi-domain DV architecture are, the designs with multiple Reset-Domains. For example, Ethernet framer (MAC+PCS) and the core logic (TCAM look-up, Buff Manager, Queue Manager etc.) will be in two different reset domains.

v) Reset and recovery of each domain is mutually exclusive and multi-domain DV environment can ease our effort in verifying these

5. Summary

This 'UVM_DOMAIN' based methodology to achieve DUT-like mutual exclusivity definitely has a lot of advantages as discussed above.

i) It's very structured way to get the job done
ii) Scenario randomization can happen automatically
iii) Higher functional and transition coverage can be achieved with relatively smaller number of testcases
iv) Easy to integrate with existing testbench.

Having said that, it also carries some disadvantages as well.

i) It will increase to overall complexity of the test execution.
ii) Testbench designers need to be very careful while integrating the flow, otherwise user might end up debugging a lot of hang issues, causing from testbench domains.
iii) Testbenches which doesn't use UVM runtime phases and only relies on 'run_phase', will not get much benefit out of this methodology.
iv) If the execution of the test needs multiple randomization, after every phase jump, then simulation might go out of memory, as the available dynamic memory might not be sufficient for resolver to work seamlessly. So based on my experience, it's better to submit these jobs in bigger machines or 64bit machines.

6. References

[1] IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language
[2] Universal Verification Methodology (UVM) 1.2 Class Reference