

A UVM Multi-Agent Verification IP architecture to enable Next-Gen protocols with enhanced reusability, controllability and observability

Prathik R – Samsung Semiconductor India R&D, Bangalore

Ramesh Madatha – Samsung Semiconductor India R&D, Bangalore

Girish Gupta – Samsung Semiconductor India R&D, Bangalore

Tony George – Samsung Semiconductor, Inc.

SAMSUNG

Agenda

1. Next Gen Multiplexed Protocols and Verification
2. Proposed Architecture using Multi-Agents
3. Configurability of the Multi-Agent
4. Synchronization between the Multiplexed Protocols
5. Debug-ability of the Data/Control Flow in the VIP
6. Observation & Results
7. Conclusion

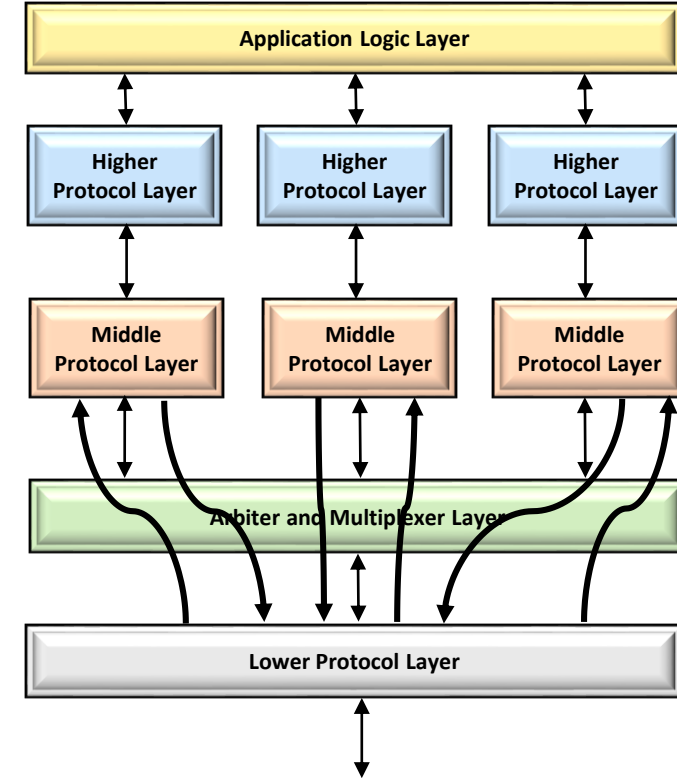
Next Gen Multiplexed Protocols and Verification

What are next gen multiplexed protocols?

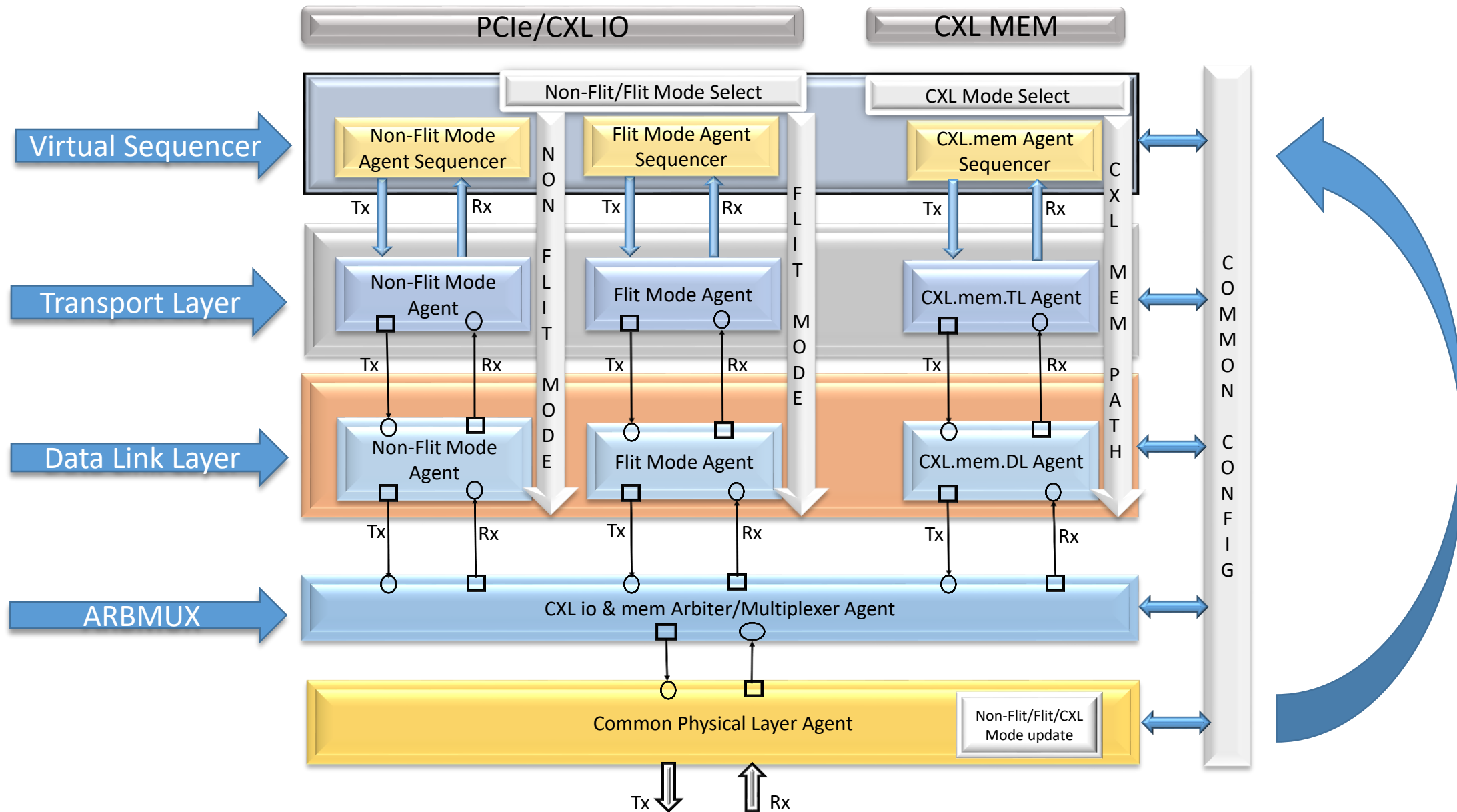
- Transfer multiple types of semantics
 - By transforming the type of semantic based upon the generation (Ex: PCIe6.0)
 - Or, transfer different semantics in parallel (CXL)
- A layer to maintain the synchronization of traffic via multiplexing and arbitration

Verification requirements:

- Ability to dynamically decide on,
 - Type of semantics to be transferred
 - Need of parallel layers
 - Synchronize the parallel traffic

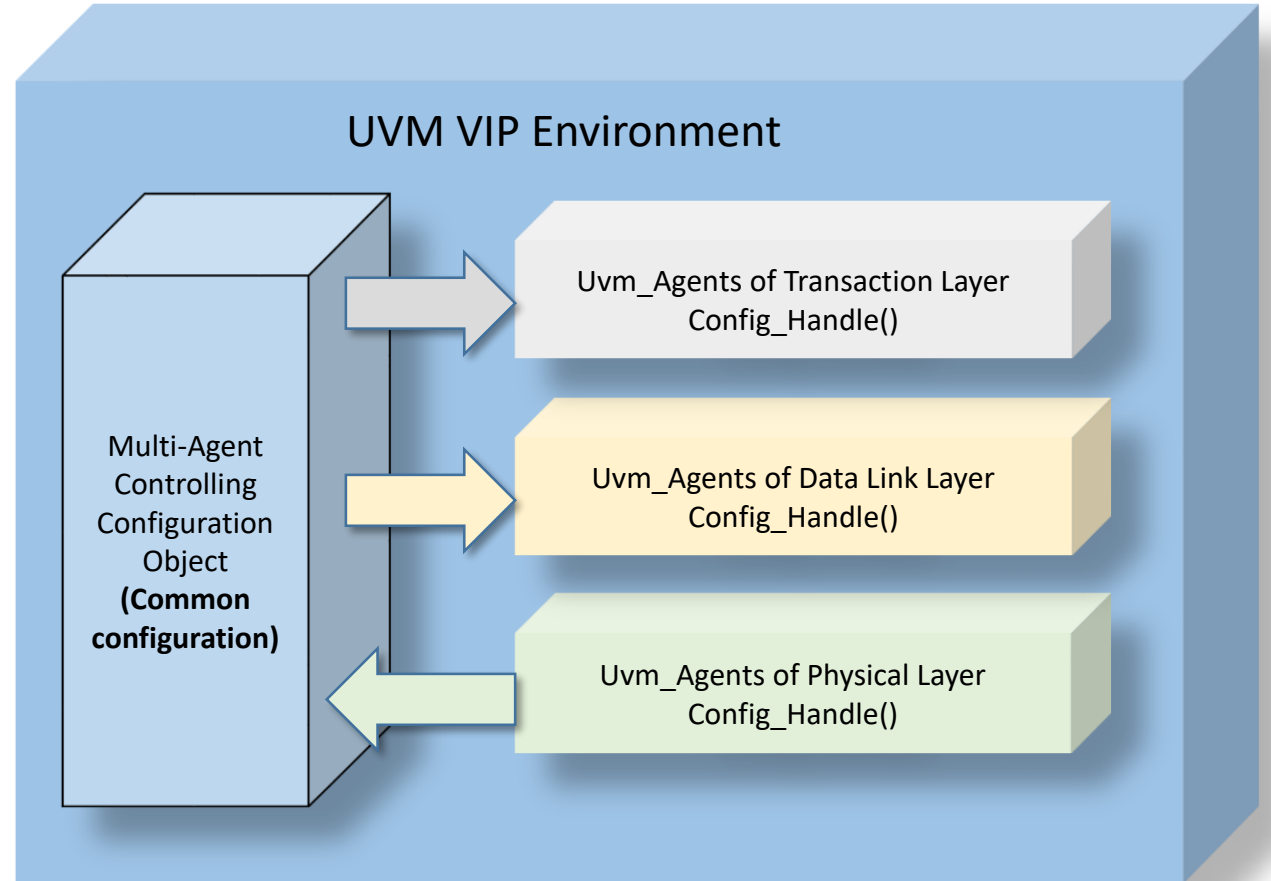


Proposed Architecture using Multi-Agents



Configurability of the Multi-Agents

- Lower Layer Agent updates the negotiated mode in the common configuration object
- Other Layer Agents can see the updates and control the flow of operation
- One single Object is shared across all the layers In-order to reflect the common configuration dynamically



Code Snippet : Configuration Class

Modes to
configure the
state
functionality

```
//Common configuration class
class vip_common_config extends uvm_object;
    //Variables to hold the configurations to be shared across
layers
    bit flit_mode_en;
    bit non_flit_mode_en;
    bit cxl mode;
    `uvm_object_utils_begin(vip_common_config)
        `uvm_field_int(flit_mode_en,UVM_ALL_ON)
        `uvm_field_int(non_flit_mode_en,UVM_ALL_ON)
        `uvm_field_int(cxl_mode,UVM_ALL_ON)
    `uvm_object_utils_end
endclass
```

Code Snippet : Provision for dynamic configurability

Creating
common
configuration
object

Passing the
handle to all
agents and
sequencer

```
//VIP environment to create and pass the handle of config class
class vip_env extends uvm_env;
    `uvm_object_utils(vip_env)
    //Create all layer agents
    //Create the vip_configuration and pass to all layers and
    virtual sequencer
        i_common_cfg =
vip_common_config::type_id::create("i_common_cfg");

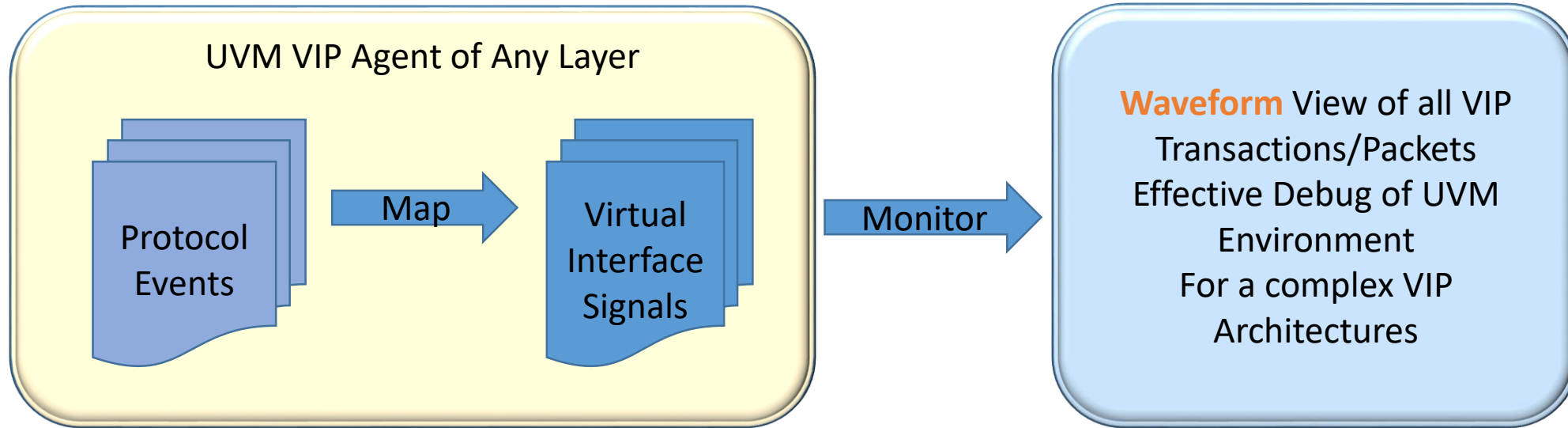
        vip_virtual_sqr.i_common_cfg = i_common_cfg;

        uvm_config_db#(vip_common_config)::set(this,"i_nfm_tl_agent.*",
"vip_common_config",i_common_cfg);

        uvm_config_db#(vip_common_config)::set(this,"i_fm_tl_agent.*",
vip_common_config",i_common_cfg);

        uvm_config_db#(vip_common_config)::set(this,"i_cxl_mem_agent.*"
,"vip_common_config",i_common_cfg);
        //So on for all agents (layers)
        .....
    endclass
```

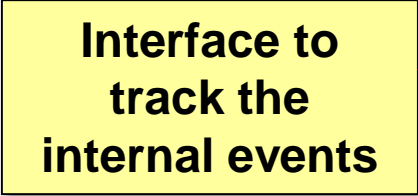
Debug-ability of the Data/Control Flow in the VIP



- Debug Interface enables the user to visualize the flow of packets in waveform and is an effective way of debugging the complex architectures and simple to develop

Code Snippet : Debug interface

Interface to
track the
internal events



```
//Types of ENUM to represent states in ASCII
typedef ltssm_state_e = {POL, CFG, RECV, L0};
typedef dlcmsm_e = {IDLE, INIT, ACTIVE};
typedef tl_state_e = {form_tlp, drive_tlp, cred_updt};

//Interface definition
interface vip_debug_intf;
    ltssm_state_e i_ltssm_state;
    dlcmsm_e i_dlcmsm_state;
    tl_state_e i_tl_state_pattern;
endinterface

//Example of event generation for state map
class dlcmsm_active_state extends dlcmsm_state;
    //Trigger the event
    →dl_active_state_change;
endclass

//Logic to map the state on to interface
//Interface is passed to the agents
class dl_driver extends uvm_component;
    //run_phase
    virtual task run_phase (uvm_phase phase);
        //forever loop to block on to the event and update
        forever begin
            @(dl_active_state_change)
                vip_dbg_intf.i_dlcmsm_state = ACTIVE;
        end
    endtask
endclass
```

Observation & Results

Features	Traditional	Proposed	Gain/Loss
VIP Development time	6 months	4 months	33%
Issue Resolution (TAT)	2 days	1 day	50%
Test Development	1 month	3 weeks	25%
Validation	5 months	4 months	20%
Coverage	2 months	5 weeks	40%

Conclusion

- The motivation for this paper is to analyze and conclude on a Verification IP Architecture which provides full-fledged control without compromising on the simplicity of model development.
- Dynamically modifiable functionality of all layers along with complex test scenario generation is achieved using this methodology.
- ***The proposed architecture has been deployed for live verification project on PCIe6.0 and CXL2.0 protocols.***

Thank You