2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Solving verification challenges for complex devices with a limited number of ports using Debugports.

Shyam Sharma, VIP R&D, Cadence Design Systems, San Jose, USA (ssharma@cadence.com)
Shravan Soppi, VIP R&D, Cadence Design Systems, Bangalore, India(ssoppi@cadence.com)

***Abstract*—With multi-fold increase in the complexity of modern IPs (Intellectual Property) and shrinking of the die sizes with the reduction of the technology nodes to sub 5 nanometer, number of pins/ports on the devices has become expensive with general direction of minimizing the number of device pin as much as possible. This introduces a challenge for verification engineers who mostly rely on the pinout signals and waveforms to debug subsystems and IPs for unexpected errors. This presentation talks about a unique and a generic way to expose internal device signals, logic variables and registers on the waveform to assist IP (Intellectual Property) and SoC (System on Chip) verification engineers for dramatic improvement in the debuggability and visibility of internal device signals, commands, and transaction without changes to the pinout of any of the devices.**

Keywords—functional verification; verification IP; memory model; systemverilog, systemC; debugports

I. INTRODUCTION

Verification Intellectual Property (VIP) are behavioral simulation models used to verify all modern IPs. Typical verification test environment testbenches has a set of IP DUT (Device Under Test) with VIPs used for rest of the components in the system. As the complexity of today's devices has increased, verification of these has become an increasingly challenging problem.

Cadence memory models (which are VIPs) are behavioral models representing the actual memory devices that are created in C/C++. They can be used in simulation with all standard simulation tools like VCS, Xcelium, QuestaSim and opensource SystemC simulators. These models can be integrated with DPI/VPI standard interface to connect to designs in Systemverilog/Verilog languages or SystemC interface for designs written using SystemC language. Keeping the main behavioral core in C/C++ helps improve the simulation performance but since it acts as a black box in the simulation, internal details of the device mode are not visible to the user.

One of the hardest challenges for verification engineers is to get enough peek inside the VIPs to review internal signals and variables that are essential to debug protocol failures/analyze the performance issues. Verification engineers typically rely on the waveforms for their work of verifying DUT functionality. This is where viewing the internal signals and logic blocks of VIPs on the waveform, helps with improving debuggability and verification. Traditionally to get this kind of insight, verification engineers relied on a combination of transaction callbacks (function calls made by VIP on things like when a new command is received) and simulation log to analyze the current device stand which can be tedious and error prone to keep track off.

II. STATE OF THE ART

A. Requirements

Before going into details of the solution, let's take a step back and list what are the requirements for solving the visibility challenges in generic way for complex Verification IPs.

Some of the important considerations that the solution should incorporate: -

- Solution should be simulator independent.
  It should work with industry standard commercial simulators like XM/VCS/QuestaSim etc as well as open-source simulators like (OSCI SystemC simulator) when applicable.
- User should be able to access the signal/variable change events in standard simulator debugger.

No special debugging tool should be needed to view these additional variables and signals on the waveform. Standard waveform debugging tools like Simvision/Verisium for Xcellium, Verdi for VCS or even generic VCD/FSDB dump that can be displayed on any simulator.

- Internal component signal representation should have similar look and feel to rest of the device ports.
  These added variables and signals should use standard types which can be enumerations or base Systemverilog types. User shouldn't need to define/include anything else beyond what they are already doing as part of their simulation.
- Added variables in the module ports should be automatically generated.
  Automatic generation of VIP module definition wrappers with these signals and variable included, is an important requirement which not only helps avoid user errors but will make the whole inclusion of the debug variables a seamless process for user.

B. Context

This section talks about an example use case of signal visibility challenges for complex Verification IPs. One of the important components of today's SoC (System on Chip) are the memory sub-systems with DRAM (Dynamic Random Access Memory) memories. Server DRAMs (Dynamic Random Access Memory) are typically bundled together to create higher density/bus-width memories called Dual Inline memory Models (DIMM). DIMMs are a good example of complex Verification IPs as these DDR (Double Data Rate) DIMM have multiple sub-components inside each DIMM card. DDR5 (Double Data Rate DRAM Generation 5) DIMM VIP is hierarchical memory model with DDR5 SDRAM, DDR5 Registering Clock Driver (RCD) and DDR5 Data Buffer (DB) modules as sub-components. Figure 1 below shows a typical DDR5 DIMM card and some of its components. Host devices (and by extensions the verification engineers instantiating the DIMM card in their designs) can only see the DIMM connectors as ports and cannot access individual sub-components like DRAMs (Dynamic Random Access Memory), RCD etc. Beyond looking at the component ports, lack of visibility of component logic blocks like DRAM Mode registers, internal write leveling pulses, refresh counters etc. is also a tedious and time-consuming task for verification engineers to figure out.

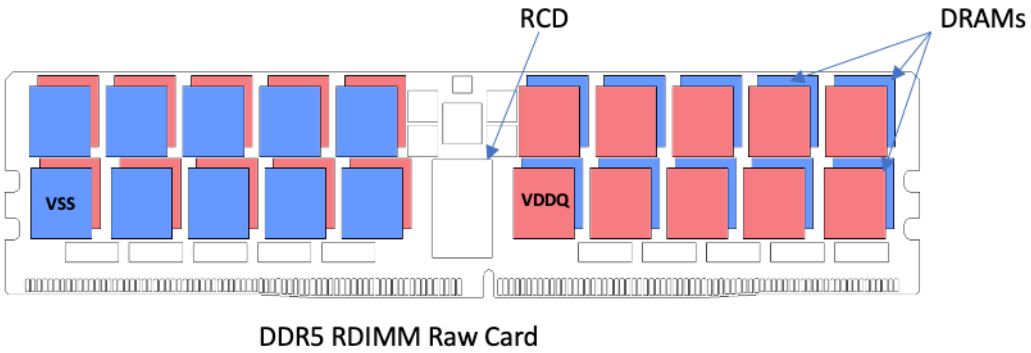

Figure 1: Only top-level DIMM connectors are visible in SV/SystemC ports.

C. Debugports and it's application

Cadence VIP solves this visibility challenge for complex VIPs using what we call "debugports".
A debugport is an HDL (Hardware Description Language) object that accurately reflects internal model state which can be displayed on the waveform like any Systemverilog/SystemC variable. We expose internal objects (which could be signals or variables) as debugports in the wrapper.

Users can treat a debugport just like any other HDL object. Debugports can be,

- Sent to the simulator waveform viewer using standard simulator GUI (Graphical User Interface.) options.
- Added to an event control block like, an always block is Systemverilog.
- Print its value to output using $display or any other standard output functions.

Basic idea behind debugports is using the standard simulator interface library calls to pass on the internal VIP model information to the variables in the model wrapper that can be used by verification engineers to ease their debug. Figure 2 below provides a glimpse of how we handle this using standard Systemverilog LRM defined VPI interface for SV designs.

- Update the wrapper logic variable.
- VPI apis provide very easy access to the variables and update them.

```
handle = vpi_handle_by_name ("rank[0].dram[0].dq", dimm_wrapper);
vpi_put_value(handle, new_dq_val_with_0_delay);
```

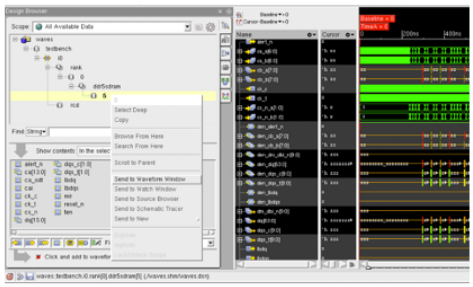

Figure 2: Debugports basics for Systemverilog.

We have created 2 kinds of debugports that are supported with the most used High Level Description languages in the industry today (Systemverilog and SystemC). Both categories of debugports are supported for each of the HDLs (Hardware Description Language) are described below: -

a. Signal Debugports:

This kind of debugports represent an internal signal of the model that is not a port. Some of the examples of these kind of internal signals are: -

- Signals of internal components of a Verification IP (VIP), like DRAM/RCD/DB instances of DDR5 DIMMs as described in Figure 1. Part of the modified DDR5 DIMM Systemverilog wrapper with component signal debugports included is shown in Figure 4.
- Board Delay signals where the signal at device port is different from internal memory die (for example where there are wiring or fly-by delays).
- Internal signals that are useful to see for users like Internal Write Leveling pulses, Internal clocks, Write enable, etc.

b. Variable Debugports:

Variable debugports represent an internal logic block, a register or state of the model that would be interesting for the user to look at. Typically, these variables represent things that Verification IP model already tracks. Some of the examples for the Variable debugports are: -

- Commands
  Protocol defined commands like the set of commands that DRAMs can be issued.
- Data access latencies.
  Latencies are the delay that the device like DRAM memories take for getting to the data phase part of the data access commands like Read or Write.
- Composite timing parameters or equations.
  Many protocols have timing spacing requirements that are dependent on several variables like sequence of commands, mode register settings, input signals timings etc. A good example of this kind of timings are Read to Write or Write to Read turn around timings for DRAMs like DDR5 SDRAM.
- Per bit setup/hold/pulse window duration or errors.

Finding out or keeping track of data valid window (also referred to as data eye) is a significently time consuming task that is required to be done to make sure your pin values are sampled correctly by Host or the devices. Debugports can help represent a visual representation of this easily by highlighting the data valid sample region around the sample strobes.

Before going into the HDL specific solution we've created, let's look at how the debugports enabled simulation will looks like and what are the steps involved (also shown in Figure 3) in supporting such simulation.

Step 1: List out the interesting signal and variable debugports that need to be supported by a verification IP. This can also be an evolving list instead of a static list.
Step 2: Group the related variables together to make it easier for users to find the information for one aspect of device features. This could be for example all debugports that display mode register related information or device state machines as a group.
Step 3: Add xml representation of the debugports that would allow a conversation tool to generate wrappers with debugports included for different languages like Systemverilog or SystemC. See figure 4 for details on tool generation of debugport enabled HDL specific wrapper.
Step 4: Add update functions in the VIP model to send the information across to the debugport variables in the modified wrappers.
Step 5: Run simulation with new device wrapper that has debugports included.

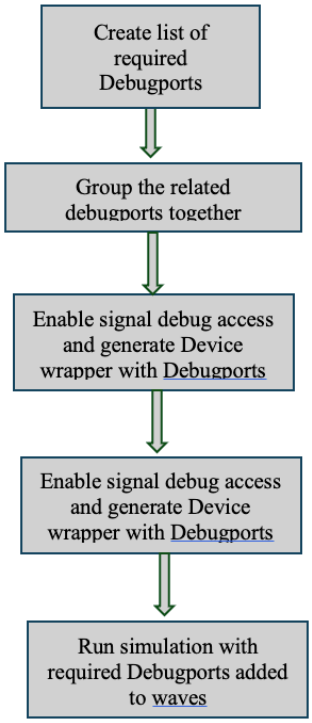

Figure 3: Debugport enabled simulation support flow chart.

The steps 1-4 listed in Figure 3 are internal to VIP that is defined during model development process. Steps 5 will be performed by the verification engineer while using the VIP and integrating with their design before simulation. Figure 4 shows how step 5 can be done by user using pureview tool.

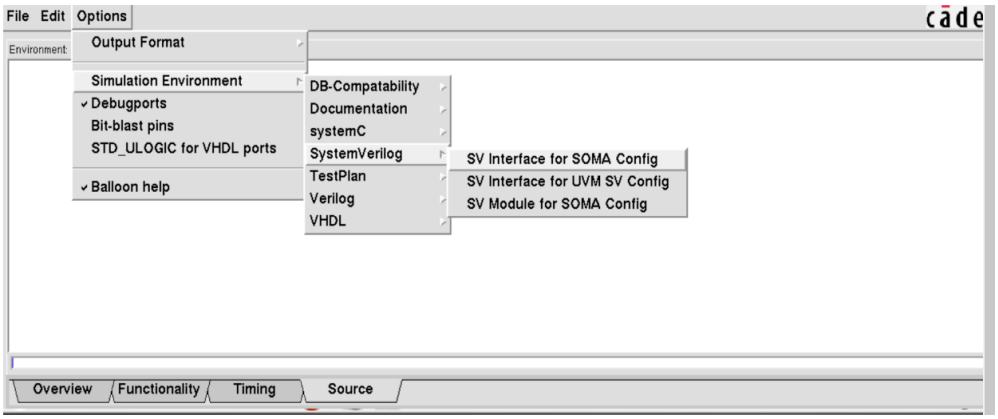

Figure 4: Generation of Debugports enabled wrapper using a tool.

Let's looks at the example work done for different HDLs below: -

- SV (System Verilog) solution for using debugports.
  This follows the flow described in Figure 3 above. Below (Figure 5) is an example device wrapper for DDR5 DIMM with debugports included. The number of generate blocks is based on the configuration of the DIMM that is used with the design. Each generate block is specifically named to specify what component inside the VIP it represents. If the number of components is more than 1, for loops are used to generate indexed blocks representing each of the components.

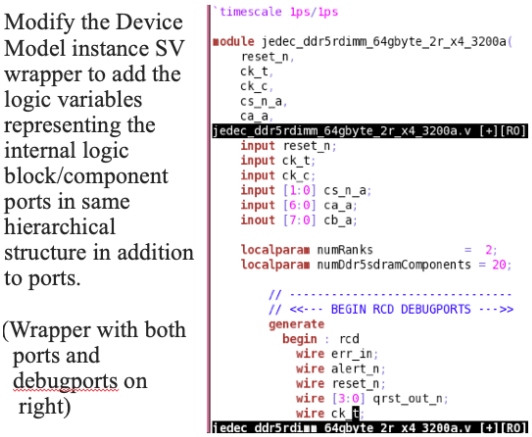

Figure 5: Example DDR5 DIMM wrapper with debugports

- SystemC solution for using debugports
  Debugports for SystemC follows similar flow as Systemverilog but there are added challenges for hierarchal devices like DDR5 DIMM which has several components inside, many of which are of same type as DDR5 SDRAM. We have followed more following steps for enabling debugports for such devices. Figure 6 below shows the changes described here on.
  - Create wrapper for each sub-instances with same naming convention as device internal instances.
  - A unique identifier for each instance is generated based on its full hierarchical path name.
  - The main VIP instance holds a list of all the sub-instances created within its hierarchy. And are created before VIP instance is instantiated.
  - While creating VIP internal instances, the library references the wrapper instance and map the unique identifiers to internal VIP instances using full path names.
  - On internal instance signal update, a callback to device model SystemC interface layer happens with instance identifier and signal details.

- A wrapper instance lookup is done based on the unique identifier and schedule a zero delay debugport wakeup event to that specific sub-instance after adding the signal event to a list of signal updates for that instance.
- On simulator invoking the wakeup, event handler calls the update Debugports function for the sub-instance to update all the signals recorded in the list with new values.

Figure 6: Debugport enabled wrapper for DDR5 DIMM device in systemc.

D. Results

We have been able to provide a simple debugports interface to view internal signals and variables on the simulator waveform using Debugports. Figures 7, figure 8, figure 9 and figure 10, below show how debugports for internal signals and variables cabe be generated by used and how those appear on waveform viewer in a real simulation.

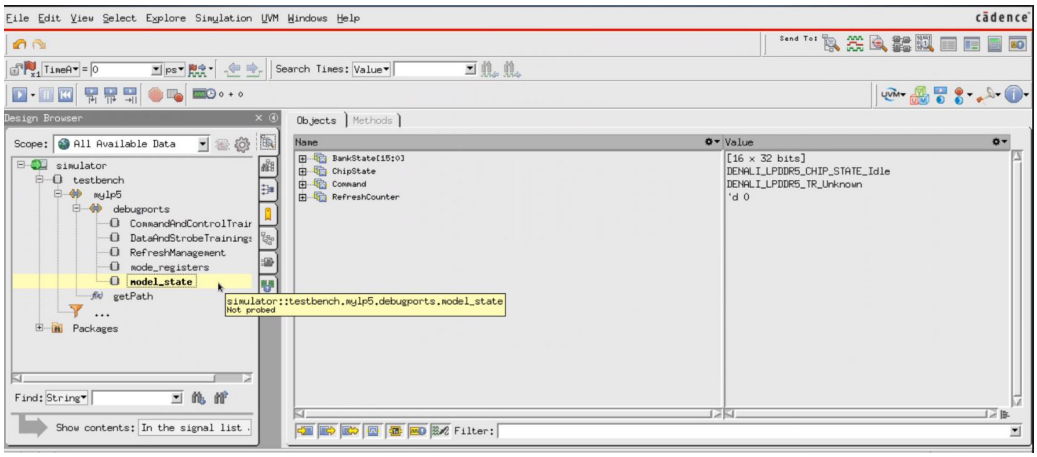

Figure 7: Debugport groups and list showing up under the instance path in simulator design browser.

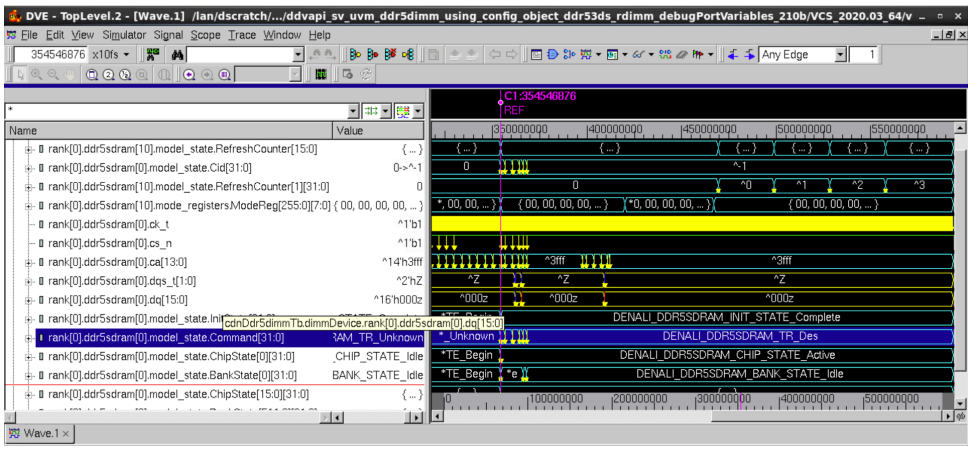

Figure 8: Combination of signal and variable Debugport for DDR5 DIMM devices showing one of the Rank 0 Component 0 DDR5 DRAM debugports. User can scroll down further to look at same information for other Rank and component DRAMs.

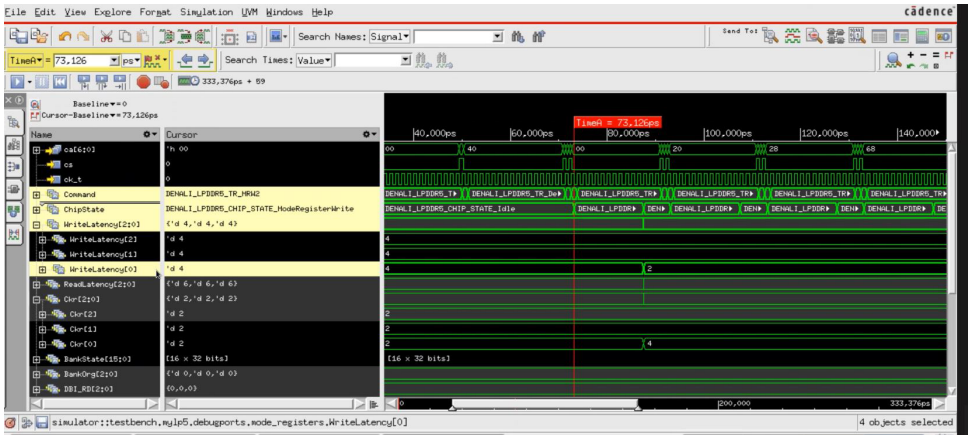

Figure 9: Debugport enabled simulation run that displays variables like received commands, device latencies etc for Lpddr5 VIP.

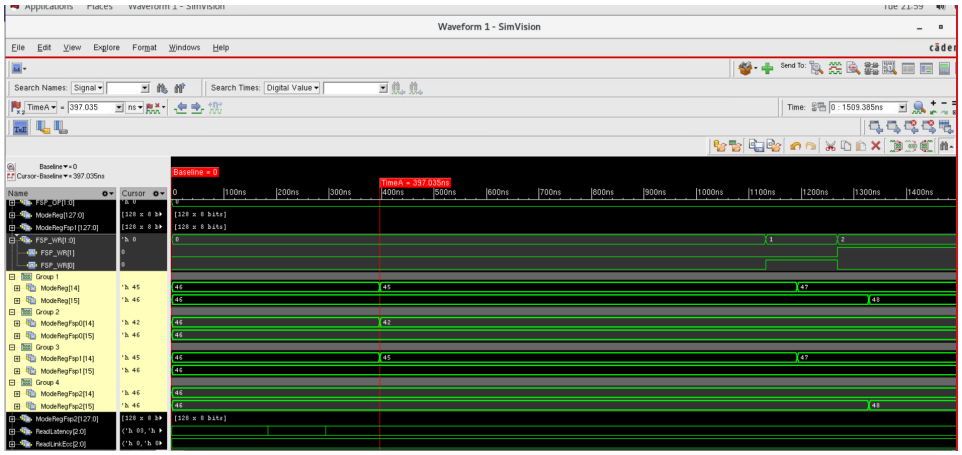

Figure 10: Debugport enabled simulation run showing Mode registers related debugport group for Lpddr5 VIP.

Finally a summary of advantages of using Debugports

- Works in same way as standard device ports.
- Available in the Design hierarchy of instance to ease of locating them.

- Works with standard simulator waveform GUI or waveform dump using simulator specific or generic formats.
- Easy to generate using a Cadence tool called Pureview (with batch mode command or using GUI).
- Supported with SV and SystemC language and works with all market standard simulators out of the box (VCS/Xcelium/QuestaSim etc).
- Can be extended to any type of VIP.
- Minimal performance impact on enabling debugports.
  - Less than 10% performance impact on typical device tests with debugports enabled Vs when debugports are not enabled.
  - Works just like a language construct.

III. CONCLUSION

We have defined and implemented a simple, automotic, modular and generic solution to enhance subsystem or Device model to be able to display internal logic block state/information, registers, timings and signals on the simulation waveform to dramatically improve the verification engineers throughput with enhanced debugability using Debugports with minimal performance impact.

Debugports are widely adapted feature of Cadence memory model Verification IPs like DDR5 DIMM, Lpddr5, DDR4 DIMM.

IV. REFERANCES

1. JEDEC. (2024). *DDR5 SDRAM* (JESD79-5C ed.). https://www.jedec.org/standards-documents/docs/jesd79-5c
2. JEDEC. (2023b). *LOW POWER DOUBLE DATA RATE (LPDDR) 5/5X* (JESD209-5C ed.). https://www.jedec.org/document_search?search_api_views_fulltext=lpddr5
3. *DDR5 Registering Clock Driver Definition (DDR5RCD03)* (JESD82-513th ed.). (2023). JEDEC. https://www.jedec.org/document_search?search_api_views_fulltext=DDR5RCD
4. *DDR5 Buffer Definition (DDR5DB01) - Rev. 1.1* (JESD82-521st ed.). (2021). JEDEC. https://www.jedec.org/document_search?search_api_views_fulltext=DDR5DB&order=title&sort=asc
5. Cadence Xcelium Logic simulator Xcelium Logic Simulator | Cadence
6. Synopsys VCS logic simulator VCS Functional Verification Solution | Synopsys Verification
7. Siemens QuestaSim simulator Questa Advanced Simulator | Siemens Software
8. Cadence memory model VIPs Memory Models | Cadence
9. IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language https://ieeexplore.ieee.org/document/8299595
10. "IEEE Standard for Standard SystemC Language Reference Manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1-638, 9 Jan. 2012, https://ieeexplore.ieee.org/document/6134619