CXL Verification using Portable Stimulus

Ragesh Thottathil
Senior Architect
Vayavya Labs

Karthick Gururaj
Chief Architect and CTO(ESL)
Vayavya Labs

CXL is a low latency fabric which builds upon PCIe interface and enables high-speed, efficient interconnect between CPU, memory and accelerators. What makes CXL distinct is that it maintains coherency between CPU memory space and memory on CXL attached devices. Along with these features comes a set of challenges in verifying these from unit level to Post silicon environments. The objective of this paper is to bring out the effectiveness of using Portable Stimulus Standard (PSS) for the verification of Compute Express Link (CXL).

I. Introduction

CXL provides a low-latency, high-bandwidth path for an accelerator to access the system and for the system to access the memory attached to the CXL device. CXL is made up of 3 distinct set of protocols

- *CXL.io - I/O semantics similar to PCIe (mandatory)*
  This provides a non-coherent load/store interface for I/O devices. Transaction types, packet formatting, flow control etc. follow the PCIe definition
- *CXL.cache - caching protocol semantics (optional)*
  CXL.cache is developed to allow Devices to access and cache Host attached memory. This was not possible with the traditional load-store format of PCIe. CXL.cache works on MESI (Modified, Exclusive, Shared, Invalid) coherence protocol
- *CXL.mem - memory access semantics (optional)*
  This is the CXL memory protocol and allows for memory bandwidth and capacity expansion. This applies for various types of memory like volatile, persistent etc.

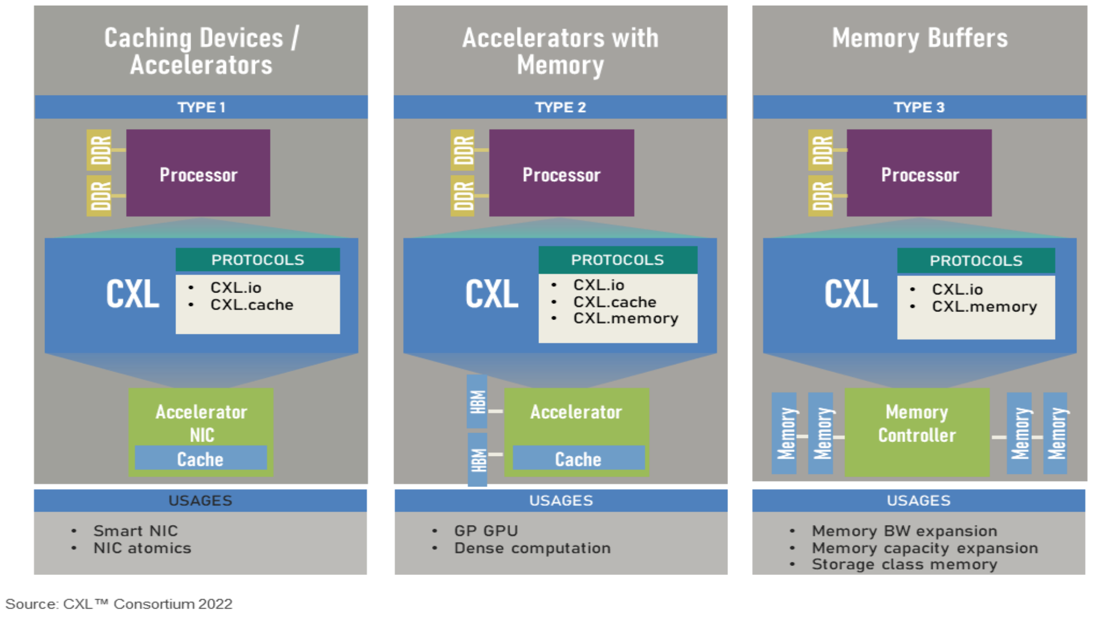

Fig:1 Types of CXL devices

CXL is designed to support 3 different device types with the combination of the protocols defined above. The 3 types of devices are captured in Fig: 1

There are multiple challenges associated with verification of CXL such as

1) *Multiple dynamic and configuration parameters which need crossing*

CXL is feature rich and consists of multiple configuration parameters such as cache state transitions, memory pooling, RAS (Reliability, Accessibility, Serviceability), IDE (Integrity and Data Encryption), power management etc.

2) *Coherency and ordering rules cannot be verified at just unit level*
CXL aims to build heterogeneous and disaggregated compute environments which means memory subsystems and coherency problem is no longer problem of unit implementing it. We will need cache coherency and memory ordering verification to be done at multiple levels

- Unit
- Subsystem (Headless)
- Full system – Software driven system
- Emulation and prototyping
- Post silicon

3) *Scalable topologies and shared resources*
There are numerous topologies possible with CXL bridges in between and verifying multiple configurations across multiple topologies is a great challenge.
As a part of the work different hierarchies were enumerated and CXL transactions were verified

4) *Software world behavior needs to be modelled for verification*
CXL is micro-architecture aware so we need to model their behavior to certain extent in verification environment early to make sure we model the traffic more realistically. Also, most of configuration and enumeration happens in System firmware and it is better to model it in PSS, so that we can leverage it at different hierarchies.

Portable Stimulus Standard defines a specification for creating a single representation of stimulus and test scenarios. A PSS model is a representation of some view of a system’s behavior, along with a set of abstract flows. It is essentially a set of class definitions augmented with rules constraining their legal instantiation. A model consists of two types of class definitions: elements of behavior, called actions; and passive entities used by actions, such as resources, states, and data flow items, collectively called objects. The behaviors associated with an action are specified as activities.

II. WORK SUMMARY

A. *PSS Modelling*

The work consisted of creating portable Stimulus model for a standard CXL host and device controller, which consists of the following

1) *Portable stimulus register model for the cxl.io and cxl.mem registers*
A quick snapshot of the CXL register space that is modelled using PSS is captured in Fig: 2

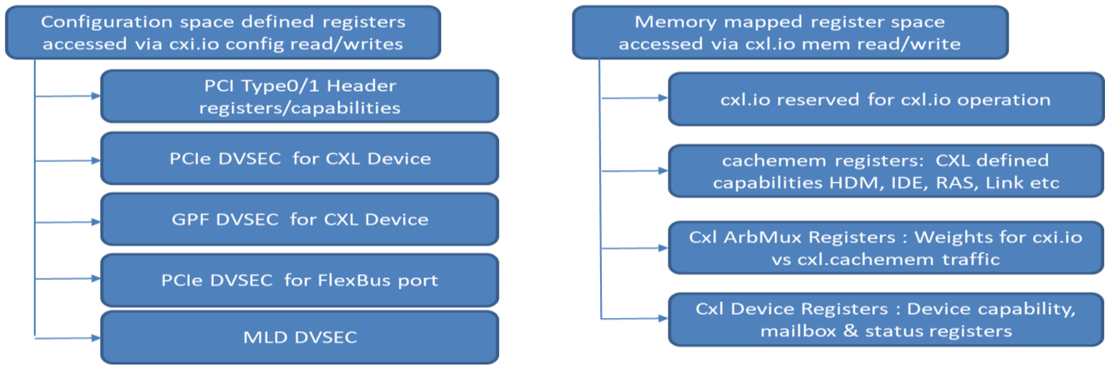
Configuration space defined registers accessed via cxi.io config read/writes
PCI Type0/1 Header registers/capabilities
PCIe DVSEC for CXL Device
GPF DVSEC for CXL Device
PCIe DVSEC for FlexBus port
MLD DVSEC
Memory mapped register space accessed via cxl.io mem read/write
cxl.io reserved for cxl.io operation
cachemem registers: CXL defined capabilities HDM, IDE, RAS, Link etc
Cxl ArbMux Registers : Weights for cxi.io vs cxl.cachemem traffic
Cxl Device Registers : Device capability, mailbox & status registers

Fig:2 CXL register space

As part of our work, these different subsets of registers which are part of CXL controller were modelled using PSS core library as pure components. These registers will be accessed in PSS atomic actions to program the controller. Fig: 3 shows an example of how a register in the configuration space (PCIe DVSEC for CXL device) is represented in PSS

```
struct cxl_dvsec_range1_size_low_s : packed_s <LITTLE_ENDIAN>
{
  bit [1]  mem_info_valid;
  bit [1]  mem_active;
  bit [3]  media_type;
  bit [3]  mem_class;
  bit [5]  desired_interleave;
  bit [3]  mem_active_timeout;
  bit [12] rsvd;
  bit [4]  mem_size_low;
}

pure   component   cxl_dvsec_range1_size_low_c:   reg_c   <cxl_dvsec_range1_size_low_s,
READONLY, 32> {} ;

pure component cxl_reg_grp : reg_group_c
{
...
cxl_dvsec_range1_size_low_c  cxl_dvsec_range1_size_low;
...
```

Fig:3 Representation of CXL register space in CXL

2)Flow objects representing the CXL.io topology, DVSECs, Component register space etc.
Flow objects were modelled which captures the CXL.io topology, DVSEC (Designated vendor-specific extended capabilities) and component register space. Fig: 4 shows an example of how the DVSEC register space was modelled

```
/*************************************************************************************
         * struct cxl_dvsec_reg_map_s - Structure to represent the different DVSEC
component registers and
         memeory device base address for each CXL device.

*************************************************************************************/

        struct cxl_dvsec_reg_map_s
        {
                rand bit[64] pcie_dvsec_base;
                rand bit[64] non_cxl_func_dvsec_base;
                rand bit[64] cxl20_port_dvsec_base;
                rand bit[64] gpf_port_dvsec_base;
                rand bit[64] gpf_dev_dvsec_base;
                rand bit[64] pcie_flex_dvsec_base;
                rand bit[64] reg_locator_dvsec_base;
                rand bit[64] mld_dvsec_base;
                rand bit[64] test_cap_dvsec_base;
                rand int reg_locator_size;
        };
```

```
/*************************************************************************************
* buffer cxl_dvsec_reg_map_b - Buffer containing the DVSEC register maps for all the CXL
devices in the hierarchy

************************************************************************************/

 buffer cxl_dvsec_reg_map_b
 {
    rand cxl_dvsec_reg_map_s cxl_dvsec_reg_map[NUM_PCIE_DEVICES];
 };
```

Fig: 4 **PSS flow object representation of CXL DVSECs**

3) *Actions for enumerating CXL.io and CXL.mem registers*

CXL.io enumeration is similar to PCIe enumeration in which the system software discovers the devices in the hierarchies by probing for the vendor ID register in ECAM (Enhanced Configuration Access Mechanism) address space.

Once the CXL.io devices are discovered and BARs are configured the enumeration of the CXL components takes place. A PCIe device is identified as CXL device through the presence of CXL specific DVSEC capabilities. The action provided below depicts the identification of multiple CXL DVSECs which are part of the configuration region. In addition to this the System software enumerate the CXL.mem register space which is memory mapped through the BAR region. Fig:5 shows an example of how the action for enumeration of CXL.io DVSEC register space was modelled

```
extend action cxl_io_c :: cxl_enumerate_dvsec_registers_a {
       int count = 0;
       int dummy;
       exec body
       {
         while (count < NUM_PCIE_DEVICES)
         {
              out_cxl_dvsec_reg_map.cxl_dvsec_reg_map[count]
              =
              comp.pci_find_dvsec_capability(out_cxl_dvsec_reg_map.cxl_dvsec_reg_map[count],
              in_pcie_hierarchy.pcie_hierarchy.dev_list [count], dummy);
              count = count + 1;
         }
       }
}

/*************************************************************************************
* pci_find_dvsec_capability () - PSS native function to find the dvsec capabilities
* @cxl_dvsec_reg_map : Structure containing dvsec registers
* @pci_devices       : Structure containg enumerated bdfs
* @pcie_var          : int variable
* @return cxl_dvsec_reg_map_s
************************************************************************************/
function cxl_dvsec_reg_map_s pci_find_dvsec_capability (cxl_dvsec_reg_map_s
cxl_dvsec_reg_map, pcie_device_s pci_devices, int pcie_var)
{
       bit [16] pos;
       bit [16] vendor;
       bit [16] id;
       bit [32] read_from_handle;
       bit [32] header;
       addr_handle_t config_base;
       addr_handle_t handle;
       bit [64] addr;
```

```
        bdf_s b_d_f;

        pcie_var = pci_devices.pcie_var;
        b_d_f = pci_devices.dev;
        pos = PCI_CFG_SPACE_SIZE;

        config_base = cxl_io_get_handle (b_d_f, 0);
        header = pci_find_ext_capability (config_base, pos, b_d_f);

        while (header != 0)
        {
          handle = cxl_io_get_handle (b_d_f, header + 0x04);
          read_from_handle = read32 (handle);
          vendor = read_from_handle & 0x0000FFFF;
          handle = cxl_io_get_handle (b_d_f, header + 0x08);
          read_from_handle = read32 (handle);
          id = read_from_handle & 0x0000FFFF;

          if (vendor == PCI_DVSEC_VENDOR_ID_CXL && id == 0)
          {
               cxl_dvsec_reg_map.pcie_dvsec_base = config_base + header;
          }
          else if (vendor == PCI_DVSEC_VENDOR_ID_CXL && id == 2)
          {
               cxl_dvsec_reg_map.non_cxl_func_dvsec_base = config_base + header;
          }
          …
          …
           header = pci_find_next_ext_capability (config_base, header);
       }
       return cxl_dvsec_reg_map;
}
```

***Fig: 5* PSS action for enumeration of DVSEC register space**

4) Actions for performing mailbox communication from CXL host to device

CXL implements a command/response interface over mailbox. CXL host can send a command to devices which is identified through a 2-byte opcode. Opcodes 0000h-3FFFh describe generic CXL device commands and falls into 4 categories – Events, Firmware update, Timestamp and logs. Opcodes 4000h-BFFFh describe Class Code specific commands such as memory device commands. CXL controller uses the mailbox register interface to send device commands and payload and read back the response. Fig: 6 show the action modelled for sending a mailbox command to CXL device

```
extend action cxl_mem_c::cxl_send_command_a {

        addr_handle_t h1;
        exec post_solve{
                h1 = make_handle_from_claim(mem_claim);
        }

        exec body {
                MAILBOX_CAPABILITIES_REG_s mb_cap_reg;
                MAILBOX_CONTROL_REG_s mb_control_reg;
                MAILBOX_STATUS_REG_s mb_status_reg;
                MAILBOX_COMMAND_REG_s mb_cmd_reg;
                MAILBOX_BACKGROUND_COMMAND_STATUS_REG_s mb_back_status_reg;
                MAILBOX_PAYLOAD_REG_s mb_payload_reg;

                //Caller reads MB Control Register to verify doorbell is clear.
                mb_control_reg = comp.MB_grp.MAILBOX_CONTROL_REG.read();
                //Caller writes Command Register.
                //Caller writes Command Payload Registers if input payload is non-
empty
                if (mb_control_reg.DOORBELL == 0){
                        bit[64] cmd_reg = 0;
                        bit[16] opcode = in_mb.mb_b.op;
                        cmd_reg |= opcode;
                        mb_cmd_reg.COMMAND_OPCODE = cmd_reg;
                        //Copy  the payload to PAYLOAD DATA
                        . . .

                        comp.MB_grp.MAILBOX_COMMAND_REG.write(mb_cmd_reg);

                        //Caller writes MB Control Register to set doorbell
                        mb_control_reg = comp.MB_grp.MAILBOX_CONTROL_REG.read();
                        mb_control_reg.DOORBELL =1;
                        comp.MB_grp.MAILBOX_CONTROL_REG.write(mb_control_reg);
                        comp.cxl_display ("doorbell is set");

                   //Caller wait for doorbell to be cleared and reads MB Status
Register to fetch Return code
                        if (mb_control_reg.DOORBELL == 0){
                                mb_status_reg =
comp.MB_grp.MAILBOX_STATUS_REG.read();
                                bit[16] rc = (mb_status_reg.RETURN_CODE >> 32 ) &
0xFFFF;
                                if (rc == 0){
                                        comp.cxl_display ("command executed
successfully");
                                }
                                else{
                                        comp.cxl_display("command failed");
                                }
                        }
                        //If command successful, Caller reads Command Register to
get Payload Length
                        mb_cmd_reg = comp.MB_grp.MAILBOX_COMMAND_REG.read();
                        bit[21] payload_size = (mb_cmd_reg.PAYLOAD_LENGTH >>16) &
PAYLOAD_SIZE_MASK;

                        //If output payload is non-empty, host reads Command Payload
Registers
                        . . .
        }
}
```

Fig: 6 **PSS action for sending a mailbox command to CXL device**

5) Actions for performing CXL.mem transactions

CXL.mem transactions work in the same was as normal PCIe mem transactions once the CXL HDM (Host Managed device memory) is configured. Once the HDM ranges and sizes are configured, the CXL device memory will appear as normal memory to the system software. Any reads/writes within the HDM range will result in a CXL.mem transaction on the bus.

```
extend action cxl_io_mem_wr_a {
       exec body {
         addr_handle_t mem_h = mem_write_buff_in.mem_addr;
         int offset = 0;
         addr_handle_t mem_h1 = make_handle_from_handle(mem_h, offset);
         mem_h = mem_write_buff_in.mem_addr;
```

```
        . . .

        repeat (i: (mem_write_buff_in.data_size / 4)) {
            write32(mem_h1, (mem_write_buff_in.data[i]));
            offset = offset+1;
            mem_h1 = make_handle_from_handle(mem_h, offset);
        }//then the rest
        repeat (i: (mem_write_buff_in.data_size %4)) {
            write8(mem_h1, (mem_write_buff_in.data[i]));
            offset = offset+1;
            mem_h1 = make_handle_from_handle(mem_h, offset);
        }
    }
}
```

***Fig: 7* PSS action for performing CXL memory write**

6) *Test scenario*

Top level test scenario instantiates other leaf level actions and the action instances are scheduled inside an activity. Fig:8 shows an example for a top-level scenario which performs injecting poison to a set of memory locations and read back the list of poisoned locations from a CXL memory device.

```
        /*********************************************
            scenario to get the poisoined list
        **********************************************/
        action get_poisonlist_a {
                cxl_mb_cmd_c::inject_psn_input_a inject_psn;
                cxl_mb_c::send_inject_psn_cmd_a send_inject_psn;
                cxl_mb_cmd_c::psn_input_a psn_list_input;
                cxl_mb_c::send_psn_cmd_a send_psn_list;
                cxl_mb_cmd_c::cxl_poisonlist_out_a get_psn_list;

                activity {
                        inject_psn;
                        send_inject_psn;
                        psn_list_input;
                        send_psn_list;
                        get_psn_list;
                        bind inject_psn.out_mb send_inject_psn.in_mb;
                        bind psn_list_input.out_mb send_psn_list.in_mb;
                }
        }
```

***Fig: 8 PSS* scenario action for poisoning memory location and retrieving the poisoned list**

B. *Test generation*

PSS models described in the above section are compiled using PSS tool. The PSS tool processes the models and randomizes the variables which are provided as random based on the constraints provided and generates C test cases for the scenarios modelled. The tool also generates results of the coverage achieved on the randomization of the variables.

Multiple scenarios were created during the project for verification of CXL system. Below is a sample list of actions which were implemented as PSS scenarios

- Enumeration of CXL DVSEC registers
- Enumeration of component registers
- CXL.io and CXL.mem transactions with randomized payload,
- Various mailbox communication between CXL host and device

Figure 9 below shows a sample test scenario that was executed

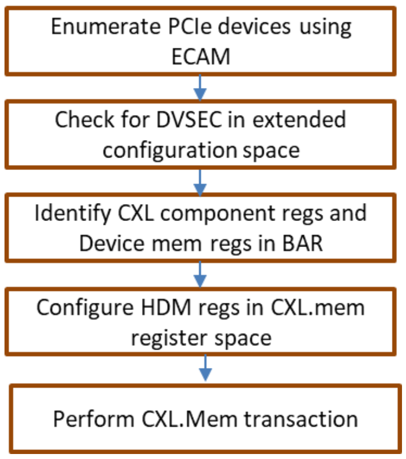
Enumerate PCIe devices using ECAM
Check for DVSEC in extended configuration space
Identify CXL component regs and Device mem regs in BAR
Configure HDM regs in CXL.mem register space
Perform CXL.Mem transaction

Figure 9. Representative PSS scenario used for test generation

C. *Test Execution*

Qemu was used as the test environment to validate the generated test cases. Qemu is an open-source emulator which emulates processor and peripheral models. A test setup which consists of an x86 processor connected to a PCIe RC and which in turn is connected to a topology of CXL root ports and devices is emulated on Qemu. Figure.11 below shows a pictorial representation of the test environment built on Qemu.
The test code generated from the PSS tool in the above section was compiled and built into a binary(.bin) file which can be loaded on the processors emulated on Qemu.

Qemu supports emulation of CXL host controller and CXL memory devices. Multiple hierarchies of CXL Host and devices can be emulated by varying the parameters provided during Qemu. Below is a snapshot of the command provided to create the test setup shown in Figure 10.

```
qemu-system-i386 -drive file=os_image,if=floppy,media=disk -boot order=d  -m
1024,slots=12,maxmem=16G -M q35,cxl=on

-object memory-backend-file,id=cxl-mem1,share=on,mem-path=/tmp/cxltest.raw,size=256M \
-object memory-backend-file,id=cxl-mem2,share=on,mem-path=/tmp/cxltest2.raw,size=256M \
-object memory-backend-file,id=cxl-lsa1,share=on,mem-path=/tmp/lsa.raw,size=256M \
-object memory-backend-file,id=cxl-lsa2,share=on,mem-path=/tmp/lsa2.raw,size=256M \
-device pxb-cxl,bus_nr=11 ,bus=pcie.0,id=cxl.1 \
-device pxb-cxl,bus_nr=222,bus=pcie.0,id=cxl.2 \
-device cxl-rp,port=0,bus=cxl.1,id=root_port13,chassis=0,slot=2 \
-device cxl-type3,bus=root_port13,persistent-memdev=cxl-mem1,lsa=cxl-lsa1,id=cxl-pmem0 \
-device cxl-rp,port=0,bus=cxl.2,id=root_port15,chassis=0,slot=5 \
-device cxl-type3,bus=root_port15,persistent-memdev=cxl-mem2,lsa=cxl-lsa2,id=cxl-pmem1 \

-M cxl-fmw.0.targets.0=cxl.1,cxl-fmw.0.targets.1=cxl.2,cxl-fmw.0.size=4G,cxl-
fmw.0.interleave-granularity=8k
```

Figure 10. Representative command for invoking Qemu with required CXL hierarchy

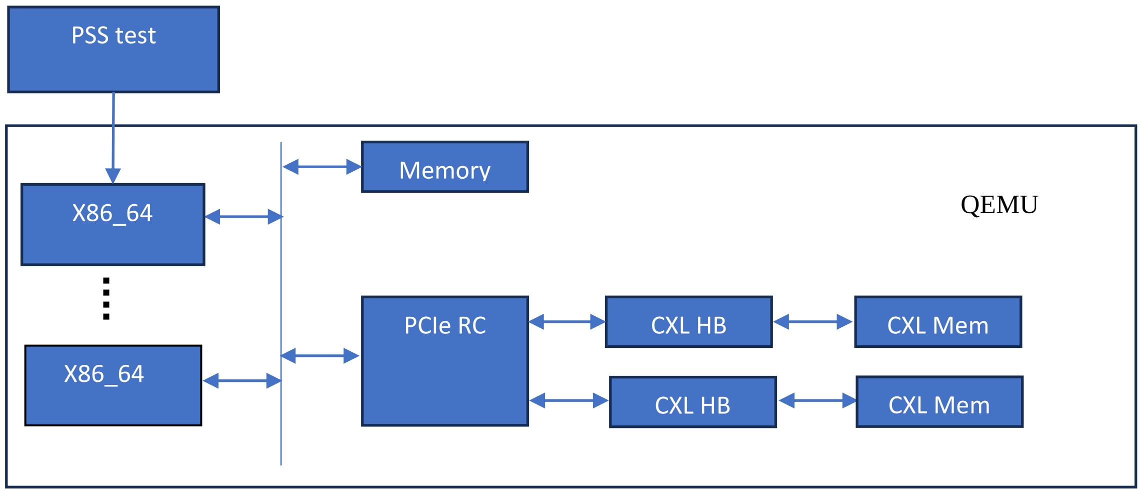

Figure 11. Representative diagram of the test environment on Qemu

III. Results

- Real world software scenarios such as discovery of CXL hierarchy and enumeration of DVSEC registers were validated
- Verified the enumeration of CXL component registers on exposed through BAR region.
- Initiated CXL.mem write/read transactions from multiple CPU cores and verified the results
- Successfully carried out mailbox transactions to the CXL device such as getting memory device info, reading poison list, getting CXL event log etc.

IV. Conclusion

The objective of the work was to model different scenarios applicable to CXL and verify it on a system level platform. Due to the non-availability of hardware platform the testcases generated from the models were validated on Qemu. The current support of CXL in Qemu is limited to CXL.io and CXL.mem features. CXL.cache feature is not yet supported. Hence the focus was on modelling the scenarios pertaining to CXL.io transactions such as discovery and enumeration and CXL.mem transactions including mailbox communication between host and devices.

The PSS model content created as part of this project can be reused to generate test content across multiple platforms from simulation to post-silicon. This helps avoid the duplication of content from one platform to other. Also, offloading of the randomization of attributes to the PSS tool is another advantage. This project proved the concept well that modelling of CXL in PSS provide great advantages in terms of time and effort.

V. References

[1] PSS 2.0 documentation: https://www.accellera.org/downloads/standards/portable-stimulus
[2] CXL™ Consortium 2022
[3] Qemu CXL documentation