# WEEKLY REPORT

**Prepared for:** Anh Tuan Anh
Anh Nam
Anh Phuong

**Prepared by:** Le Hoang Mai Phuong

October 24th, 2025

# Contents

**1 Week 39 - 40 (26/09 - 01/10/2025) 2**

1.1 RTL Co-Simulation with a Transaction-Level Wrapper in QuestaSim . . . . . 2

1.2 Exploration with Helium Virtual & Hybrid Studio . . . . . 4

# 1 Week 39 - 40 (26/09 - 01/10/2025)

## 1.1 RTL Co-Simulation with a Transaction-Level Wrapper in QuestaSim

**STATUS: In progress**

### 1.1.1 Objective and Workspace setting

The idea for this implementation is:

- I have an RTL block (mini_mem.sv). Then I wrote a C++ DPI wrapper that issues transaction requests (read/write) into the RTL.
- Both RTL and the wrapper run together inside the QuestaSim kernel:
  - RTL executes in the QuestaSim kernel.
  - The C++ code is compiled as a DPI shared object and linked into the same kernel.
  - The DPI wrapper translates high level transactions into RTL signal activity.
- The result is a co-simulation setup where RTL is driven and observed via a transaction level interface, implemented through DPI.

***path: /home/shared/phuonglhm_workspace/TLM_wrapper***
I have a workspace folder containing the following files:

- **mini_mem.sv:** RTL description of the memory block.
- **bridge.sv:** RTL testbench that instantiates the DUT, provides DPI tasks for C++ to perform read/write transactions.
- **rtl2tlm.cpp:** C++ file that implements transaction-level tasks (read/write) using DPI.
- **build.sh:** Build script that compiles the RTL, compiles the C++ file into a shared object (.so), and runs the simulation

### 1.1.2 Issues and Solutions

First, I tried to compile the project with the SystemC library from Accellera (Fast Models) using the command below:

```
export QUESTA_HOME=/home/tools/mentor/questasim/2024.2/questasim
export SYSTEMC_HOME=/home/tools/arm/fastmodels/11.29.027/SystemC/Accellera/SystemC

g++ -std=c++14 \
  -I${SYSTEMC_HOME}/include \
  -I${QUESTA_HOME}/include \
  -L${SYSTEMC_HOME}/lib/Linux64_GCC-10.3\
  -lsystemc \
  -fPIC -shared -o rtl2tlm.so rtl2tlm.cpp
```

The code compiled successfully. But when I ran it with vsim, I got the following fatal error:

```
** Fatal: (vsim-12005) Undefined function 'sc_core::sc_api_version_2_3_4_cxx201402L<&
    sc_core::SC_DISABLE_VIRTUAL_BIND_UNDEFINED_>::sc_api_version_2_3_4_cxx201402L(
    sc_core::sc_writer_policy, bool)' introduced from './rtl2tlm.so' is being called.
    Exiting ...
```

This happened because QuestaSim already has its own SystemC library, and when I linked the Accellera SystemC, the two libraries conflicted at runtime.

To fix this, I decided to build the project only with the SystemC library provided by QuestaSim. The working compile command is:

```
export QUESTA_HOME=/home/tools/mentor/questasim/2024.2/questasim

g++ -std=c++14 \
  -I${QUESTA_HOME}/include/systemc -I${QUESTA_HOME}/include \
  -L${QUESTA_HOME}/linux_x86_64 \
  -lsystemc_gcc103 -lmtipli \
  -fPIC -shared -o rtl2tlm.so rtl2tlm.cpp
```

After unset LD_LIBRARY_PATH, I recompiled the project with the command above. The previous fatal error (vsim-12005) disappeared. However, a new fatal appeared:

```
** Fatal: (vsim-160) bridge.sv(18): Null foreign function pointer encountered when
    calling 'sysc_bootstrap'
```

This happened because I mixed two different ways of binding SystemVerilog and C++ in Questa:

- Foreign interface (-foreign): registers one C function as the entry point.
- DPI-C (import "DPI-C"): SystemVerilog tasks/functions are linked to symbols in a shared library loaded with -sv_lib.

In my setup, I compiled sysc_bootstrap as a DPI-C function, but tried to load it with the foreign interface:

```
vsim -c work.bridge \ -foreign "rtl2tlm_init ./rtl2tlm.so" \ -do "run -all; quit"
```

Questa could not find the symbol in the DPI registry, so the pointer was null and the simulator crashed at time 0.

To fix it, I updated rtl2tlm.cpp to use only DPI-C. Then I loaded the library with:

```
vsim -c work.bridge -sv_lib rtl2tlm -do "run -all; quit"
```

This way, SystemVerilog and C++ use the same DPI mechanism, and the conflict is resolved.

But then, my code face a new fatal:

```
** Fatal: (SIGSEGV) Bad pointer access. Closing vsimk. ** Fatal: vsimk is exiting with
    code 211. Exit codes are defined in the "Error and Warning Messages" appendix of the
     QuestaSim User's Manual.
```

This happened because I created SystemC processes too early (inside rtl2tlm_init). In QuestaSim, the SystemC kernel is owned by vsim, and process creation must only happen at simulation time 0 or later, not during library load. To fix this, I updated the SystemVerilog bridge to add a DPI import and call it at time 0. This way, the kernel is fully initialized before any SystemC processes are created.

```
import "DPI-C" context task sysc_bootstrap();

initial begin
    #0;
    $display("[%0t] SV calling sysc_bootstrap()", $time);
    fork
      sysc_bootstrap();
    join_none
end
```

After adding the DPI import and rebuilding, I reran the build script. QuestaSim started successfully and showed the expected messages:

```
# run -all
# [0] SV calling sysc_bootstrap()
# [sysc_bootstrap] Starting DPI sequence in SV context...
# [sysc_bootstrap] WRITE 0x12345678 @0x10
# [sysc_bootstrap] READ  @0x10
# [sysc_bootstrap] READ got 0x12345678
#  quit
```

However, if I just let it run, the simulation froze before executing the write/read tasks. I had to force quit it before it would proceed with the tasks and exit.

This happens because the C++ side (via DPI) and the SystemVerilog side are not yet synchronized correctly. The SystemC scheduler is waiting for a trigger, while the SystemVerilog simulation is also waiting, so both sides end up in a deadlock at time 0.

I am currently working on this issue. The idea is to adjust the synchronization so that the DPI bootstrap function can yield control properly, allowing the simulation to advance without manual interruption.

## 1.2 Exploration with Helium Virtual & Hybrid Studio

**STATUS: In progress**

Besides working on RTL–TLM co-simulation with QuestaSim, I also started studying and working with the Helium Virtual & Hybrid Studio tool. I am following this example, which demonstrates how to use the tlmgen command.

***path: /home/shared/phuonglhm_workspace/helium_practice***

For this example, I used the IP-XACT description of an UART, as shown in above reference link:

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component
  xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009
                      http://www.accellera.org/XMLSchema/IPXACT/1685-2009/IPXACT.xsd">

  <spirit:vendor>example.com</spirit:vendor>
  <spirit:library>peripherals</spirit:library>
  <spirit:name>uartPL011</spirit:name>
  <spirit:version>r1p0</spirit:version>

  <spirit:memoryMaps>
    <spirit:memoryMap>
      <spirit:name>uart_regs</spirit:name>
      <spirit:addressBlock>
        <spirit:name>registers</spirit:name>
        <spirit:baseAddress>0x0</spirit:baseAddress>
        <spirit:range>0x100</spirit:range>
        <spirit:width>32</spirit:width>

        <!-- Your register definition -->
        <spirit:register>
          <spirit:name>uartDMACR</spirit:name>
          <spirit:description>DMA Control Register</spirit:description>
```

```
          <spirit:addressOffset>0x04</spirit:addressOffset>
          <spirit:size>32</spirit:size>
          <spirit:access>read-write</spirit:access>
          <spirit:reset>
            <spirit:value>0x0</spirit:value>
          </spirit:reset>
          <spirit:field>
            <spirit:name>DMAONERR</spirit:name>
            <spirit:bitOffset>2</spirit:bitOffset>
            <spirit:bitWidth>1</spirit:bitWidth>
          </spirit:field>
          <spirit:field>
            <spirit:name>RXDMAE</spirit:name>
            <spirit:bitOffset>1</spirit:bitOffset>
            <spirit:bitWidth>1</spirit:bitWidth>
          </spirit:field>
          <spirit:field>
            <spirit:name>XDMAE</spirit:name>
            <spirit:bitOffset>0</spirit:bitOffset>
            <spirit:bitWidth>1</spirit:bitWidth>
          </spirit:field>
        </spirit:register>

      </spirit:addressBlock>
    </spirit:memoryMap>
  </spirit:memoryMaps>
</spirit:component>
```

Then, I generated the SystemC TLM model by running the tlmgen command, which takes the IP-XACT file as input and produces the SystemC TLM output:

```
tlmgen -name uartPL011 -ipxact uart.xml -simple
```

```
tlmgen: Successfully Generated uartPL011_run_include.f
 Include uartPL011_run_include.f in the run script of your virtual plaform assembly.
 To use uartPL011_run_include.f, the virtual platform assembly must reside in a sibling
 directory of the directory containing uartPL011_run_include.f.
tlmgen: Successfully Generated Interfaces
```

Figure 1: Generated result

Besides the initial **uart.xml** file, the generated SystemC code also includes a module skeleton that can be modified to add specific functionality:

```
.
├── tgOutDir
│   ├── build
│   ├── cfg
│   ├── doc
│   ├── inc
│   ├── lib
│   ├── src
│   └── test
├── tlmgen.log
├── uartPL011_run_include.f
└── uart.xml
```

- **tgOutDir:** main folder contains several subdirectories:
  - **build:** build scripts
  - **cfg:** configuration files for the model
  - **doc:** auto-generated documentation
  - **inc:** header files with class and register definitions
  - **lib:** compiled libraries
  - **src:** SystemC source files, including the module skeleton with sockets, registers, and TLM methods
  - **test:** example testbench code
- **tlmgen.log:** generation process
- **uartPL011_run_include.f:** is to be included in the run script of virtual platform assembly that instantiates the model

### 1.2.1 Build the Generated Model

I go to the **tgOutDir/build** directory and use the provided scripts to compile the SystemC TLM code. This build process generates the shared library uartPL011.so, with all required libraries and executables linked automatically.

```
[phuonglhm@cica-venus build]$ ./uartPL011_build.sh
TOOL:   xmsc_run(64)    23.09-s007: (c) Copyright 1995-2016 Cadence Design Systems, Inc.

xmsc_run \
        -DSC_INCLUDE_DYNAMIC_PROCESSES \
        -I.. \
        ../src/uartPL011.cpp \
        -stop link \
        -test uartPL011

$CDSROOT = /home/tools/cadence/installs/XCELIUM2309
$TESTDIR = /home/shared/phuonglhm_workspace/helium_practice/tgOutDir/build

TOOL:   xmsc(64)        23.09-s007
xmsc C++ parameters:
        xmsc -COMPILER $CDSROOT/tools/cdsgcc/gcc/9.3/bin/g++
        -f xcelium.d/xmsc_obj/xmsc.args
        -MANUAL
        -CFLAGS "-DXMSC
                -DNCSC
                -I$CDSROOT/tools/systemc/include_pch
                -I$CDSROOT/tools/tbsc/include
                -I$CDSROOT/tools/vic/include
                -I$CDSROOT/tools/methodology/OVM/CDNS-2.1.2/sc/src
                -I$CDSROOT/tools/methodology/UVM/CDNS-1.1d/sc/sc
                -I$CDSROOT/tools/methodology/UVM/CDNS-1.1d/ml/sc
                -I$CDSROOT/tools/systemc/include/cci
                -I$CDSROOT/tools/systemc/include/factory
                -I$CDSROOT/tools/systemc/include/tlm2
                -fPIC
                -D_GLIBCXX_USE_CXX11_ABI=1 -c
                -x c++  -Wall
                -DSC_INCLUDE_DYNAMIC_PROCESSES
                -I$TESTDIR/..
                -pipe"

xmsc: compiling $TESTDIR/../src/uartPL011.cpp

building library uartPL011.so
xmsc_run: *N,TBLINK: Stopping after link due to -STOP LINK option.
          All libraries and executables are linked.
```

### 1.2.2 Next Steps:

- Add the specific functionality and behavior of UART in the **uartPL011_tlm2.cxx** file located in **/tgOutDir/src/**.

- Modify the testbench in **/tgOutDir/test/** to create and run UART test cases.
- Rebuild the project, then integrate the model into the virtual platform by connecting the target socket to a bus model (AXI or APB) and wiring up interrupts if required.