VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Bao Vu

INTEGRATION OF DEVICE DRIVER SW AND SYSTEMC HW MODEL

Technology and Communication
2022

ACKNOWLEDGEMENTS

I would like to express my most gratitude to Dr. Chao Gao for his great help and guidance on not only my thesis work but over the years of my study at VAMK. Without his valuable support, encouragement, and supervision I could not have gone this far. It has been always an honor to work under his supervision and be his student.

I would also like to extend my sincere appreciation to Mr. Simo Knaapi, my line manager, for facilitating the precious opportunity to work at the Nokia SoC SW team and giving me thoughtful guidance and immediate help.

I am extremely thankful and wish to pay my greatest gratitude to Mr. Petri Vaipuro, my thesis instructor, for his huge and effective support during my thesis progress. He did not only guide me through every stage of my thesis work but carry me through the real work at Nokia. I am honestly grateful to work with him.

I would like to pay my gratitude to Mr. Tero Maaranen, SoC SW Technical Leader, Ms. Deepa Naik, SoC SW Technical Specialist, and all the members not only in the SoC SW L1Low squad3 team but the fellows from the different teams in the Nokia SoC Organization for their support and the chances I could learn from them.

From the bottom of my heart, I would like to send my love and thanks to my family in Vietnam, they are always my inspiration. I would also like to spread my thanks to all the teachers, staff, and friends I made at VAMK and Finland for their endless support during my time studying and living abroad.

Bao Vu

Vaasa, Finland

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Quoc Bao Vu
Title	Integration of Device Driver SW and SystemC HW Model
Year	2022
Language	English
Pages	47
Name of Supervisor	Dr. Chao Gao

The thesis aims to use SystemC TLM in Nokia SoC IP (HW and SW), focusing on hardware and software integration work. The thesis presents the implementation process of an application that was used to integrate the Nokia L1Low driver SW with the Nokia SystemC HW model in the early development phases.

An application named TLM bridge was carried out using SystemC TLM with the interconnectivity features that allow the L1Low SoC SW driver to access the registers of the SystemC HW model. The idea and internal resources access right used for this thesis were granted by Nokia Corporation.

Thanks to the features of the TLM bridge application, the integration work of the Nokia L1low SW driver and SystemC HW model was successfully achieved. Furthermore, the thesis identified drawbacks in Nokia SoC development with SystemC TLM and proposed prospects for the improvement of current HW/SW integration work. The thesis work contributed to HW/SW integration process in the Nokia L1Low SoC SW team.

Keywords SystemC TLM, TLM bridge, HW/SW integration

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF LISTINGS

LIST OF ABBREVIATIONS AND ACRONYMS

ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
API	Application Programming Interface
BBU	Baseband Unit
BTS	Base Transceiver Station
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
IP	Intellectual Property
L1	Layer 1 / Physical layer
MN	Mobile Network
OSCI	Open SystemC Initiative
SoC	System on Chip
SW	Software
TLM	Transaction Level Modeling

1 INTRODUCTION

Today, the notion of System on Chip (SoC) is a reality and fully functioning system that includes complex processors, peripherals, digital signal processors (DSP), multi-layer buses, multiple memories, and other blocks that might be separate ASICs. With an SoC, we can fulfill the primary goals such as overhead saving, energy waste reduction, and minimizing the system dimension which gives us a powerful system within a small single processor /1/.

However, SoC is a highly integrated system with both HW and SW, the development is complicated and time-consuming, and especially the SW development can be only conducted when HW is ready, thus simulation is necessary. Furthermore, Nokia SoC products are massive in design, fast in speed, and large in complexity. A design is better to be described at higher levels of abstraction so that it enables HW/SW integration and faster simulation.

The SystemC was chosen as one of the potential approaches for certain Nokia SoC IPs simulation in the early phases of SoC development. The thesis with the interconnection work between the SoC software driver and the SoC SystemC hardware (HW) model, which is a crucial step in the development process.

The thesis includes five chapters. The overview of SystemC and Transaction-Level Modeling (TLM) components are introduced in Chapter 2, in which some simple examples using SystemC and TLM are described.

The main topic is presented in Chapter 3 and Chapter 4. Chapter 3 highlights the current problems of SoC development in Nokia and introduces the most essential components that construct a Nokia SoC SystemC design. In Chapter 4, the implementation of the SoC SW driver/SystemC HW model integration approach within Nokia's SoC SystemC design development and the accomplishment is presented.

The last chapter proposes some points that could be better improved and further developed in the future within the Nokia SoC SystemC design.

1.1 Nokia Solutions and Networks Oy

Nokia is a Finnish solutions and networks company founded in 1865. Nokia is recognized as a pioneer in the matter of technology evolution and provides cutting-edge networks ranging in mobile, infrastructure, cloud, and enabling technologies /2/.

In the past few years, the Mobile Industry has remarkably changed. Nokia has been always a leading in mobile technology serving a million customers all around the world. Along with a long history of network services, Nokia is constantly innovating their network technologies. System on Chip is one of the innovations that leverage Nokia to be a top leader in the field. In 2021, Nokia launched the next-generation AirScale 5G portfolio which is empowered by the latest version of ReefShark System-on-Chip chipsets, the product is integrated into baseband, remote radio heads, and mMIMO antennas with modern digital beamforming technology to deliver a massive 5G capacity, coverage range, and easy deployment /3/.

1.2 SystemC

SystemC is an ANSI standard C++ class library for system and hardware design. In the other words, SystemC expands the capabilities of C++ with modeling of HW descriptions enabled by adding a class library to C++. The class library is a library of functions, data types, and other constructs that are valid C++ code without modifying anything of C++. One highlight that can be mentioned is that SystemC does not add new syntax to C++ but simply defines a new C++ class library /4/.

A system-level representation of a design is critical for managing the complexity of large-scale designs, allowing us to easily perform all design optimizations and investigations. SystemC also supports the Electronic System Level (ESL) design and with TLM /5/.

2 THEORETICAL BACKGROUND

This chapter covers the essential concepts of SystemC, design methodology, and TLM. These topics are necessary for one to get started exploring and designing a system with SystemC and TLM. The chapter also presents some examples to indicate the advantages of utilizing SystemC TLM.

2.1 SystemC Features

2.1.1 Module

Class sc_module is the base class for modules in which ports, signals, processes, constructors, and other modules can be initiated. SC_MODULE is a macro that may be used to prefix the definition of a module for convenience and this macro use is not mandatory /6/. The SystemC module syntax is shown in Listing 1.

```
#include <systemc>
SC_MODULE(module_name) {
//MODULE_BODY
};
```

Listing 1. SC_MODULE macro syntax

2.1.2 Constructor

As SC_MODULE is a C++ class, it requires a constructor. SC_CTOR is a macro that provides facilitation for declaring or defining a constructor of a module.

It will only be used at a place where the rules of C++ permit a constructor to be declared and can be used as the declarator of a constructor declaration or a constructor definition. The argument that shall be passed to SC_CTOR is the name of the module class constructed /6/. Listing 2 is an example of a SC_MODULE constructor.

```
#include <systemc>
SC_MODULE(module_name) {
      SC_CTOR(module_name)
      : Initialization // C++ initialization list
      {
      // Subdesign_Allocation
      // Subdesign_Connectivity
      // Process_Registration
      // Miscellaneous_Setup
      }
};
```

Listing 2. SC_CTOR syntax

2.1.3 Port, Signal and Process

Ports pass data to and from the processes of a module. Port mode can be in, out, inout or even the data type as any legal C++ data type, SystemC data type, or user-defined type. A signal is used to connect the port modules. The signal can be considered as the physical wire that interconnects devices to the physical implementation of the design. The data is carried by signals and ports determine the direction of data /6/.

A process is a method registered with the SystemC kernel. A process can have its sensitivity list which contains a list of signals that can trigger the process itself to run or cause other processes to be executed by sending new values to the signals that the other processes are sensitive to. The process type defines how it is invoked and executed. There are three available SystemC processes: SC_METHOD, SC_THREAD, and SC_CTHREAD. Each type has its own unique behavior /6/.

The ports are bound to the processes for input and output. Meanwhile, the signals play the role of transmitting the data between the processes. All the definition of ports, signals, and processes happens in the scope of SC_MODULE which we can consider as a C++ class.

2.1.4 Testbench

The test bench is used to stimulate the design under test and verify the design results they can be implemented in different ways. A stimulus can be conducted in the main program and the results checked in another process. The test bench can be done according to the user application /6/.

The stimulus module provides inputs to the Device Under Test (DUT) and the Results Checking module looks at the device output and verifies the correctness of the results. Figure 1 shows a typical testbench design flow /6/.

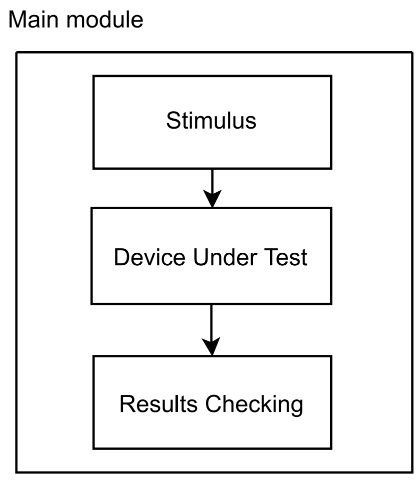


Figure 1. Typical testbench flow

2.2 Design Methodology

2.2.1 Traditional System Design

The simplest and most traditional approach to system design begins by writing a C or C++ model of the system to verify system-level concepts and algorithms. After the verification, parts of the C/C++ model implemented in the HW model are manually converted to HDL for actual HW implementation. Figure 2 shows the traditional system design flow /6/.

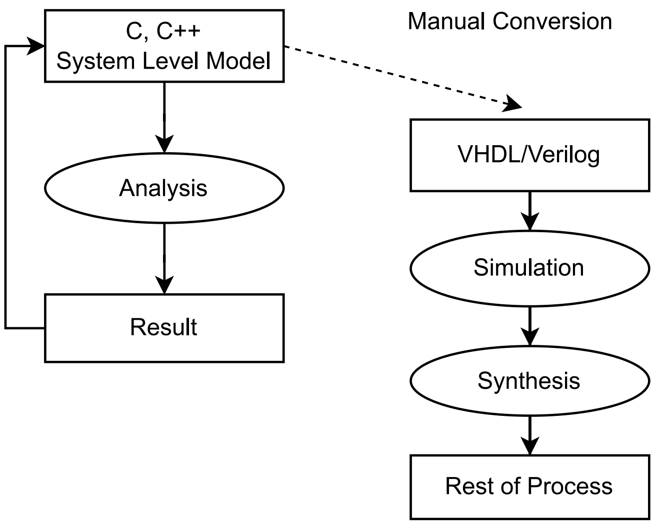


Figure 2. Traditional system design flow

There are obvious problems with this methodology /6/:

- Errors occur in manual conversion from C/C++ to HDL
- C/C++ system model and HDL model disconnection
- Multiple system tests
- Time-consuming

2.2.2 SystemC Design

Due to the problems of the traditional approach, the SystemC design methodology is considered a better methodology for system-level design. The flow of the SystemC design method is shown in Figure 3 /6/.

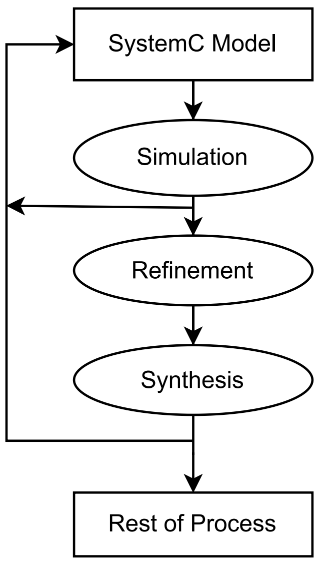


Figure 3. SystemC design flow

The advantages SystemC design over the traditional methodology /6/:

- Refinement methodology: Designs are not converted from C-level descriptions to HDL but are slowly refined in small increments to add the necessary HW and timing structures to the design. Using this refinement method, designers can easily implement design modification and spot errors during refinement.
- Written in a single language: the SystemC facilitates the designer ability to design a system without being a multilingual expert. SystemC supports modeling from system level to RTL if required.

2.3 Transaction Level Modeling

Even though SystemC brings such advantages for system-level modeling, it is not sufficient to create an effective model. Modeling styles and interoperability rules between different models also need to be defined. Transaction Level Modeling is an idea first proposed by the University of California, Irvine, and is now widely recognized as an effective abstract modeling method /5/.

Transaction-level modeling (TLM) is a technique for describing a system in terms of function calls, which define a set of transactions on a set of channels. TLM descriptions can be more abstract and therefore faster to simulate than register transfer level (RTL) descriptions traditionally used as starting points for IC implementations. However, TLM can still be used to define a design in a less abstract and more detailed way /8/. The TLM-based design methodology is described in Figure 4 /9/.

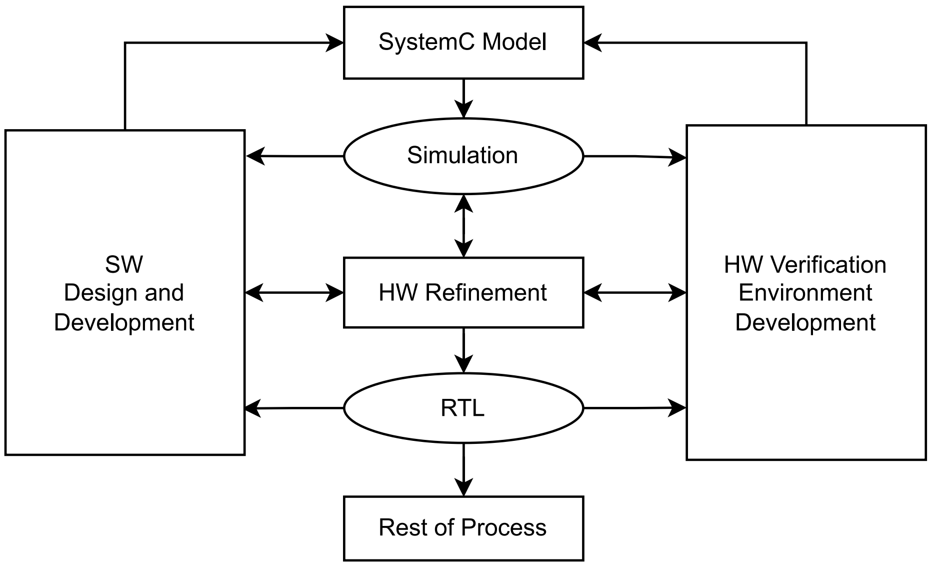


Figure 4. TLM-based methodology

One of the key techniques used in this design process is the modeling of the system at the transaction level. A transaction is a single object that contains the signal and handshake sequences required by system components to exchange data /10/.

In TLM-2.0, four basic components enable transaction transmitted-received between modules /11/:

- Initiators: generate transactions.
- Targets: respond to transactions sent by the modules.
- Transaction: the object encapsulating everything needed for bidirectional communication between modules.

- Sockets: act as the bridge between the initiator module and the target module. Enabling modules to exchange the transaction.

2.3.1 Coding Style

A coding style is a set of programming language idioms that work well together, not a specific level of abstraction or software programming interface. In the scope of the thesis, only two specific named coding styles are discussed: loosely-timed and approximately-timed /4/.

1. Loosely-timed

This coding style takes advantage of the blocking interface which allows only two timing points to be linked with each transaction, according to the call to and return from the blocking transport function /4/.

In other words, loosely-timed means as fast as possible. To achieve speed, the loosely-time coding style supports "temporal decoupling", which allows individual processes to run before the simulation time, minimizing context switches occurring during simulation, which has a significant impact on simulation speed /12/.

2. Approximately-timed

This coding style utilizes the non-blocking transport interface, which is sufficient and appropriate for the architectural exploration and analysis use cases. The added timing information is just accurate enough to perform modeling /12/.

2.3.2 Advantage

The goal of TLM is to significantly increase simulation speed while still providing decent accuracy for design tasks. To achieve this, TLM offers a way to minimize the number of events and the amount of information that has to be processed during simulation /13/.

Noticeable advantages of TLM include:

- Fast and compact
- HW/SW models integration support
- The early platform for SW development, system exploration, and verification
- Accelerates product release schedule.

2.4 Practical Examples

This section provides two examples of how to design a simple HW model using SystemC and TLM to calculate the even parity bit for 8-bit and 32-bit input data. In addition, the perspective about the advantages of SystemC TLM in system design over pure SystemC design is raised.

2.4.1 Pure SystemC Design

The design is implemented using NAND and XOR logic gates. Figure 5 indicates that a XOR logic is achieved using a combination of four NAND gates /14/.

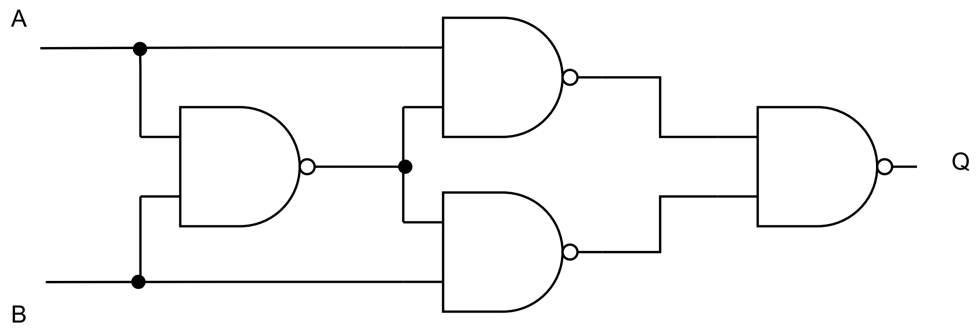


Figure 5. XOR from NAND

The NAND logic is implemented as shown in Listing 3 and XOR logic using the NAND module is implemented as shown in Listing 4.

```
#include "systemc"

SC_MODULE(nand)
{
    sc_in<bool> nA, nB;
    sc_out<bool> nF;

    void do_nand()
        nF.write(!(nA.read() && nB.read()));
    }

    SC_CTOR(nand)
    {
        SC_METHOD(do_nand);
        sensitive << nA << nB;
    }
};
```

Listing 3. NAND logic module definition

In Listing 3, we inspect the *do_nand()* process which is sensitive to input from the ports, which means that any value changes on those ports caused by the XOR trigger the *do_nand()* process.

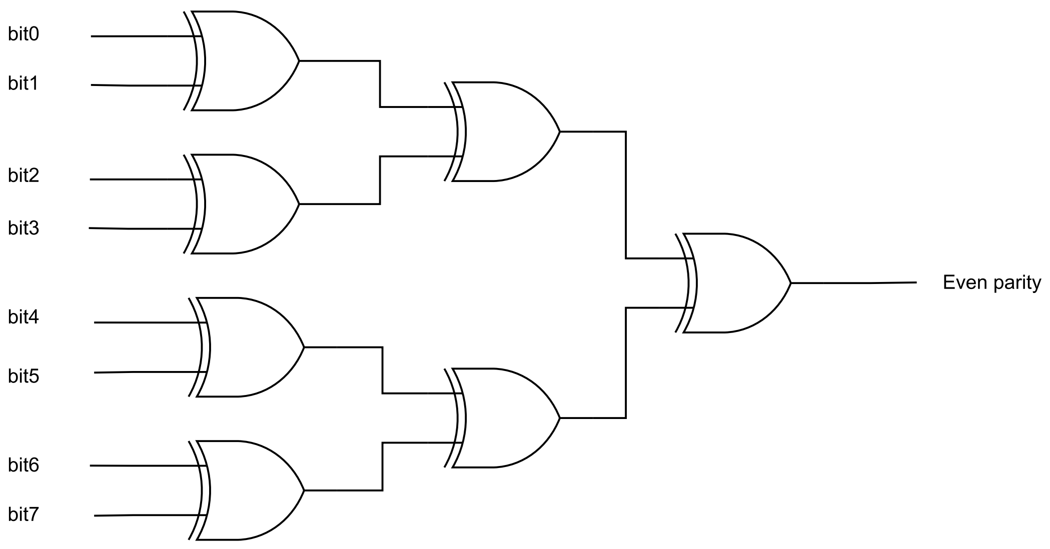


Figure 6. 8-bit word parity bit generator schematic

An 8-bit word even parity bit is generated by combining seven XOR logic gates. Figure 6 shows the generator scheme /15/.

```
#include "systemc"
#include "nand.h"

SC_MODULE(exor)
{
    sc_in<bool> A, B;
    sc_out<bool> F;

    nand n1, n2, n3, n4;

    sc_signal<bool> S1, S2, S3;

    SC_CTOR(exor) :
    n1("N1"), n2("N2"),
    n3("N3"), n4("N4")
    {
        n1.nA(A);
        n1.nB(B);
        n1.nF(S1);

        n2.nA(A);
        n2.nB(S1);
        n2.nF(S2);

        n3.nA(S1);
        n3.nB(B);
        n3.nF(S3);

        n4.nA(S2);
        n4.nB(S3);
        n4.nF(F);
    }
};
```

Listing 4. XOR logic module definition

The implementation is done using the C++ programming language with the SystemC data types. We can notice that SystemC is C++ based language and a designer who is familiar with C++ programming can develop his HW model without using any HDL programming.

The SystemC-based implementation of the parity bit generator (as shown in Listing 5) is following the block diagram in Figure 7 to generate an even parity bit for 8 input bits as “par_A0-7”.

```
#include "systemc"
#include "exor.h"

SC_MODULE(pargen)
{
    sc_in<bool>
    par_A0, par_A1, par_A2, par_A3,
    par_A4, par_A5, par_A6, par_A7;

    sc_out<bool> P;

    exor exor1, exor2, exor3, exor4,
         exor5, exor6, exor7;

    sc_signal<bool> par_S1, par_S2, par_S3,
                    par_S4, par_S5, par_S6;
    SC_CTOR(pargen) :
    exor1("EXOR1"), exor2("EXOR2"), exor3("EXOR3"),
    exor4("EXOR4"), exor5("EXOR5"), exor6("EXOR6"),
exor7("EXOR7")
    {

        exor1.A(par_A0);
        exor1.B(par_A1);
        exor1.F(par_S1);

        exor2.A(par_A2);
        exor2.B(par_A3);
        exor2.F(par_S2);

        exor3.A(par_S1);
        exor3.B(par_S2);
        exor3.F(par_S3);

        exor4.A(par_A4);
        exor4.B(par_A5);
        exor4.F(par_S4);

        exor5.A(par_A6);
        exor5.B(par_A7);
        exor5.F(par_S5);

        exor6.A(par_S4);
        exor6.B(par_S5);
        exor6.F(par_S6);

        exor7.A(par_S3);
        exor7.B(par_S6);
        exor7.F(P);
    }
};
```

Listing 5. Parity bit generator module definition

The process of generating parity bits is shown in Figure 7. In main.cpp, we have objects of three modules: Stimulate, Parity Generator, Monitor. Additionally, we bind the signals connecting these three modules to their ports.

The module named Stimulate module creates ports connecting to the signals declared in main.cpp. This module is responsible for feeding values into the Parity Generator module.

The Parity Generator module takes the input from Stimulate module and passes it to the XOR module to generate a parity bit.

The Monitor module undertakes the task of tracking and printing the results.

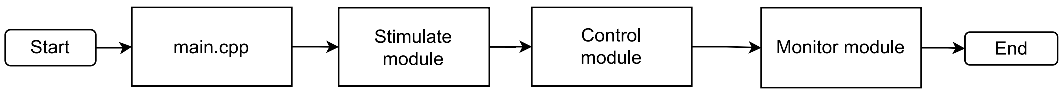
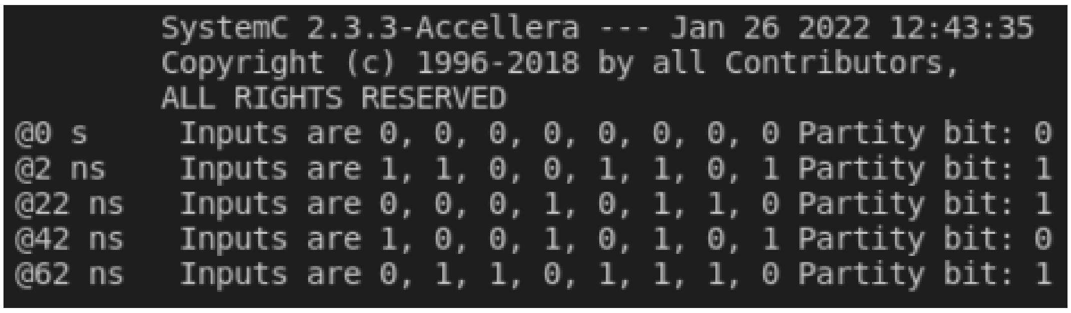


Figure 7. Parity generator SystemC flow

The result, shown in Figure 8, shows that the correct even parity bits are generated. Correct 8-bit word even parity bits are achieved using a combination of NAND, XOR logic designed in SystemC.



```
        SystemC 2.3.3-Accellera --- Jan 26 2022 12:43:35
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
@0 s     Inputs are 0, 0, 0, 0, 0, 0, 0, 0 Partity bit: 0
@2 ns    Inputs are 1, 1, 0, 0, 1, 1, 0, 1 Partity bit: 1
@22 ns   Inputs are 0, 0, 0, 1, 0, 1, 1, 0 Partity bit: 1
@42 ns   Inputs are 1, 0, 0, 1, 0, 1, 0, 1 Partity bit: 0
@62 ns   Inputs are 0, 1, 1, 0, 1, 1, 1, 0 Partity bit: 1
```

Figure 8. Even parity bit SystemC results

2.4.2 SystemC TLM Design

In this example, SystemC TLM involves the use of function calls to communicate between SystemC processes. The focus of TLM is on communication between processes.

This approach is implemented using a generic payload transaction supported by SystemC TLM to illustrate the process of generating parity bit.

The design is built with two registers address:

- PARITY_REG_IN_ADDR (0x04): stores the unsigned int 32-bit word data input
- PARITY_REG_OUT_ADDR (0x08): stores the parity bit output.

This design has three main modules that are top-level module, HW module, and Control module. The top-level module initializes the HW module and the Control module. It takes action to bind the initiator socket on the Control to the target socket on the HW, which deliver payloads with the information needed for bidirectional communication. This is a convenience feature supported by TLM sockets. The initiation is identical to C++ programming, the exclusive thing is the socket binding shown in Listing 6.

```
#include "systemc"
#include <tlm>
#include "Module.h"
#include "testbench.h"

SC_MODULE(Top)
{
  Module    *hw_module;
  Control   *control;

  SC_CTOR(Top)
  {
    hw_module    = new Module("module");
    control    = new Control("control");
    control->initiator_socket_control.bind(hw_module->tar-
get_socket_control);
  }
};
```

Listing 6. Top-level module

The initiator and target sockets need to be declared and constructed plainly within the Control and HW modules.

In Listing 7, we examine the initiator socket, which is declared using the namespace *tlm_utils*. All declarations in TLM-2.0 are in one of the two C++ namespaces *tlm* or *tlm_utils*. Socket types are *simple initiator* and *target* because they are straightforward to implement. They are classes derived from the two underlying socket types *tlm_initiator_socket* and *tlm_target_socket*. The socket template argument states the *typename* of the parent module /11/.

The SC_THREAD process macro is used to register the associated function *PayloadGen()* function with the kernel so that the SystemC scheduler can call back the function during simulation /4/. The *PayloadGen()* member function generates a stream of payload transactions that are sent to the HW module over the sockets and the data carried by the payload is used to calculate and store parity bit.

```
SC_MODULE(Control)
{
  tlm_utils::simple_initiator_socket<Control> initia-
tor_socket_control;

  SC_CTOR(Control) : initiator_socket_control("initia-
tor_socket_control")
  {
    SC_THREAD(PayloadGen);
  }
  void PayloadGen();
};
```

Listing 7. Control module

In the thread process member function, the default transaction type for the socket is *tlm_generic_payload* declared in *tlm* derived namespace. The generic payload plays a vital role in interoperability between transaction-level models. The transaction encapsulated in the generic payload is then sent over the socket using the TLM-2.0 blocking transport interface b_transport which transmits the transaction argument by reference without a return value /11/.

Listing 8 shows the standard set of attributes /4/:

- set_command: Two kinds of commands supported that is read and write.

- set_address: The address to which data is read or written.
- set_data_ptr: The pointer to a data buffer within the Control module.
- set_data_length: The length of the data array in bytes.
- set_streaming_width: The width of a streaming burst. For non-streaming transactions, the stream width equals the data length.
- set_byte_enable_ptr: The pointer to the byte enable array is set to the value passed as an argument.
- set_dmi_allowed: The method set the DMI allows an attribute to the value passed as an argument.
- set_response_status: The method indicates the current status of the transition. The response status should be initialized as TLM_INCOMPLETE_RESPONSE to keep track of the transaction status.

The blocking transport call has the timing annotation to state the time when the transaction is processed.

```
void Control::PayloadGen()
{
tlm::tlm_generic_payload* PLtrans = new tlm::tlm_generic_pay-
load;
...
PLtrans->set_command( cmd );
PLtrans->set_address( addr );
PLtrans->set_data_ptr(reinterpret_cast<unsigned char*>(&data));
PLtrans->set_data_length( 4 );
PLtrans->set_streaming_width( 4 );
PLtrans->set_byte_enable_ptr( 0 );
PLtrans->set_dmi_allowed( false );
PLtrans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

initiator_socket_control->b_transport( *PLtrans, delay );
...
}
```

Listing 8. Generic payload transaction attributes set

A target socket is declared and constructed in the HW module. The blocking transport method is implemented inside the constructor by registering the socket with a callback method called register_b_transport. Listing 9 declares the functions and variables used for parity bit calculation and extraction of a generic payload transaction.

```
SC_MODULE(Module)
{
  tlm_utils::simple_target_socket<Module> target_socket_con-
trol;

  SC_CTOR(Module) :  target_socket_control("target_socket_con-
trol")
  {
    target_socket_control.register_b_transport(this, &Mod-
ule::b_transport_control);
  }
  virtual void b_transport_control(tlm::tlm_generic_payload&
PLtransExt, sc_time& delay);
  bool pargen(uint32_t data_param);
  void read(uint32_t* t_data, uint32_t t_address);
  void write(uint32_t t_address, uint32_t* t_data);
```

Listing 9. HW module

The HW module uses the get method to extricate the payload transaction attributes set by the Control module. In Listing 10, the extraction of the six most important attributes is performed and checked to ensure that the Control module does not utilize the characteristics that are not supported by the HW module.

The set_response_status is set to *TLM_OK_RESPONSE* to notify the transaction was successful and the target module received the payload without errors.

```
void Module::b_transport_control(tlm::tlm_generic_payload&
PLtransExt, sc_time& delay)
{
  tlm::tlm_command cmd = PLtransExt.get_command();
  uint32_t*        ptr = (uint32_t*)PLtransExt.get_data_ptr();
  uint32_t         addr = PLtransExt.get_address();
  unsigned int     len = PLtransExt.get_data_length();
  unsigned char*   byt = PLtransExt.get_byte_enable_ptr();
  unsigned int     wid = PLtransExt.get_streaming_width();
  uint32_t offset = addr/4;
  uint32_t Par;
  if (byt != 0 || len > 4 || wid != 4)
     SC_REPORT_ERROR("TLM-2",
                    "Target does not support given"
                              " generic payload "
                    "transaction");
  if (addr == PARITY_REG_IN_ADDR) { ...
  else if (addr == PARITY_REG_OUT_ADDR) ...
  PLtransExt.set_response_status(tlm::TLM_OK_RESPONSE);
}
```

Listing 10. Generic payload transaction attributes get

If *cmd* is set to *TLM_WRITE_COMMAND,* the parity bit is generated by a function named *pargen()* which takes 32-bit input data and generates an even parity bit. The generator algorithm for even parity bit is given in Listing 11 /16/.

```
bool pargen(uint32_t data_param)
{
  int y = data_param ^ (data_param >> 1);
      y = y ^ (y >> 2);
      y = y ^ (y >> 4);
      y = y ^ (y >> 8);
      y = y ^ (y >> 16);

      if(y & 1)
          return 1;
      else return 0;
}
```

Listing 11. Definition of *pargen()*

Function *pargen()* uses a simple algorithm with a bit-shifting technique to compute and return the even parity bit. The working flow of SystemC TLM design is depicted in Figured 9.

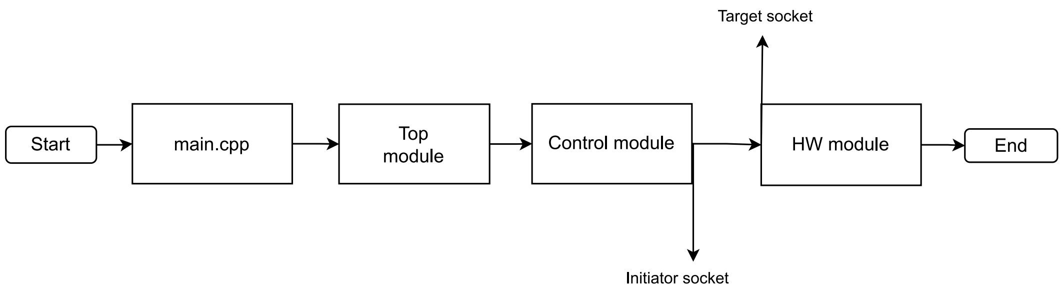
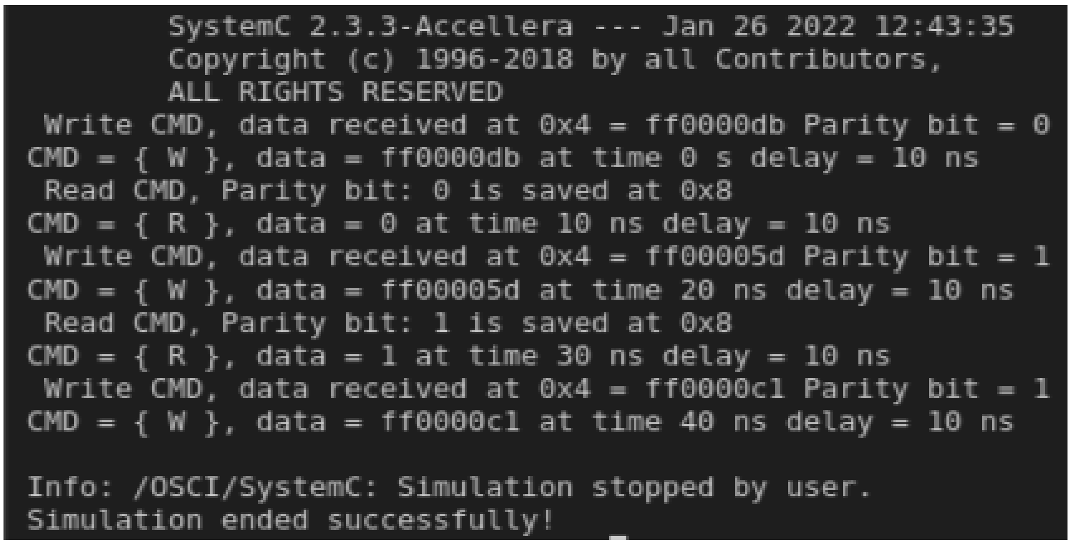


Figure 9. Parity bit generator SystemC TLM flow

The results prove the program using the SystemC TLM generic payload transaction works perfectly, as can be seen in Figure 10. The lines starting with Write or Read CMD come from the HW module and show the data received, and the parity bits generated and stored in the correct address. The other lines are written to the Control module for checking purposes.



```
         SystemC 2.3.3-Accellera --- Jan 26 2022 12:43:35
         Copyright (c) 1996-2018 by all Contributors,
         ALL RIGHTS RESERVED
 Write CMD, data received at 0x4 = ff0000db Parity bit = 0
CMD = { W }, data = ff0000db at time 0 s delay = 10 ns
 Read CMD, Parity bit: 0 is saved at 0x8
CMD = { R }, data = 0 at time 10 ns delay = 10 ns
 Write CMD, data received at 0x4 = ff00005d Parity bit = 1
CMD = { W }, data = ff00005d at time 20 ns delay = 10 ns
 Read CMD, Parity bit: 1 is saved at 0x8
CMD = { R }, data = 1 at time 30 ns delay = 10 ns
 Write CMD, data received at 0x4 = ff0000c1 Parity bit = 1
CMD = { W }, data = ff0000c1 at time 40 ns delay = 10 ns

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation ended successfully!
```

Figure 10. Even parity bit SystemC TLM results

2.4.3 Conclusion

The SystemC TLM design is reusable with other ideas, it is like a skeleton of a HW model with multiple functionalities. In other words, we can use the same program constructed by SystemC TLM for any computation with the same data input just by changing the logic inside the member function in the HW module.

The pure SystemC design is a designated parity bit generator and refactoring of the entire program is inevitable in case using it for other calculation purposes. In the contrast, the TLM design gives us the ability to make some minor modifications in function *pargen()* to perform a different calculation purpose. In general, SystemC TLM provides us with a method to describe a system at an abstraction where we can describe a system using high-level programming techniques such as function calls.

This shows that it is not enough to build efficient models using SystemC alone and that TLM capabilities facilitate designers to describe complex systems dynamically and efficiently using C++-based programming language.

3 SOC SYSTEMC DEVELOPMENT

This chapter outlines the obstacles that currently exist in the development process and the role of the SystemC TLM implementation in Nokia SoC. The chapter provides an overview of the SystemC components, interfaces, and the SW implementation as well as the SystemC HW model.

3.1 Obstacles

Nokia masters both their SoC HW and SoC SW technology, resulting in almost everything needing to be done in-house, and delivery time being a strict part that had to be followed. Launching a fully functional SoC product to market takes several years and requires the work of multiple teams with experienced engineers.

The present difficulties in SoC development can be summarized as slow development cycle, difficulties in debugging, and late HW arrival.

Thanks to the advantages of SystemC TLM, some of the difficulties can be addressed. An early SoC HW model can be provided for SoC SW development. Hence, the development process is accelerated, and the existing resources of Nokia are maximized.

3.2 SoC SystemC Design Components

This section introduces three main components that are constructing a Nokia SoC SystemC design.

3.2.1 SystemC HW Model Library

The IP (HW) model library is the main component that plays the role of the SoC HW containing a memory-mapped registers bank and the SystemC TLM interfaces. Generally, the SystemC HW model is developed to serve the purpose of delivering a virtual HW platform for SW development.

Testbenches are designed to test and evaluate the outputs of the model. The test benches feed the inputs for the HW model to trigger the simulation. For instance, one of the test cases sends a read register access request to the HW model with necessary register access information, then checks and verifies if the HW model returns the right value of the correct register.

In the beginning, test benches could be considered as SW driver simulations in the early phases of development.

3.2.2 TLM Bridge

Basically, a TLM bridge performs the task of intermediary connection, i.e. receives the register access information as a parameter through register read and write operations, then, configures the access information and forwards it to the SystemC HW model through the TLM sockets. In other words, the TLM bridge has inter-connectable functionalities that allow the SW driver to access the SystemC HW model registers.

3.2.3 L1Low SW Driver

L1low SW driver is low-level software implemented based on a predesigned architecture with comprehensible coding and naming conventions regulated by Nokia SoC architects. The L1Low SW driver has two basic levels of APIs and UnitTest built on top of it to access the HW:

- Functional API: The main data structure is specifically created for the HW IP block. It stores the access information to the HW IP block which is divided into two types: access to the regular register and access to memory.
- Register API: the main data structure for storing the register access information is created. It describes the register fields and functions that are called by functional API to get access to HW.

4 TLM BRIDGE IMPLEMENTATION

This chapter describes the implementation of the interfaces and the communication methods between components using the TLM bridge. The general system working process is introduced first, and each part introduces the specific functions and working principles used in each component in detail. The information given in this chapter is the minimum and most essential, it does not comprehensively describe all the functionalities of the components.

4.1 Idea and Goal

Eventually, the SystemC HW model will be used by the SoC SW engineers who design their SW driver utilizing the resources of SoC HW. However, the SW driver cannot connect directly to the SystemC HW model since the main functionality of the SW driver is pure C/C++ based source code built without relation to the SystemC library and designed for the generic register access purpose not specifically for the SystemC HW model. That is the reason for an application that enables the SW driver and SystemC HW model to work together.

The goal of the TLM bridge is to provide necessary operations and interfaces for communication with both the SystemC HW model and the L1Low SW driver thereby creating a complete system.

Figure 11 shows how three main components work as a system.

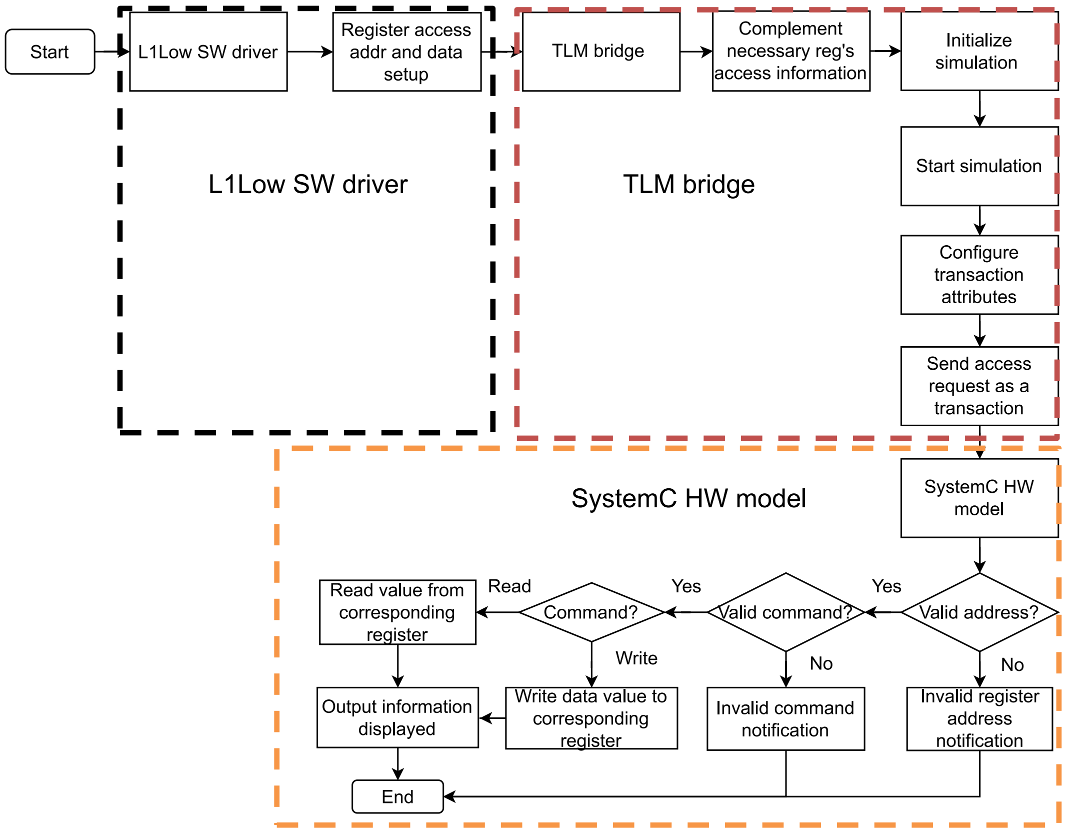

Start
L1Low SW driver
Register access addr and data setup
L1Low SW driver
TLM bridge
Complement necessary reg's access information
Initialize simulation
TLM bridge
Start simulation
Configure transaction attributes
Send access request as a transaction
SystemC HW model
SystemC HW model
Valid address?
Yes
No
Valid command?
Yes
No
Command?
Read
Write
Read value from corresponding register
Write data value to corresponding register
Invalid command notification
Invalid register address notification
Output information displayed
End

Figure 11. General system working diagram

The three main blocks are marked with dash lines to distinguish each block from the other in the entire system.

The SW driver sets register access information, including register addresses and data values, and forwards the information to TLM bridge operations.

The TLM bridge handles the integration process by initializing and starting the simulation. Additionally, the TLM bridge configures register access requests by using the access information set by the SW driver and sends them as a transaction to the HW model.

The SystemC HW model processes register access requests with the necessary verification steps and executes accesses corresponding to the command that can be read from or write to register.

The log message shows all the information about the result of the output of the register, in both cases, whether the register is valid or not.

4.2 Interfaces

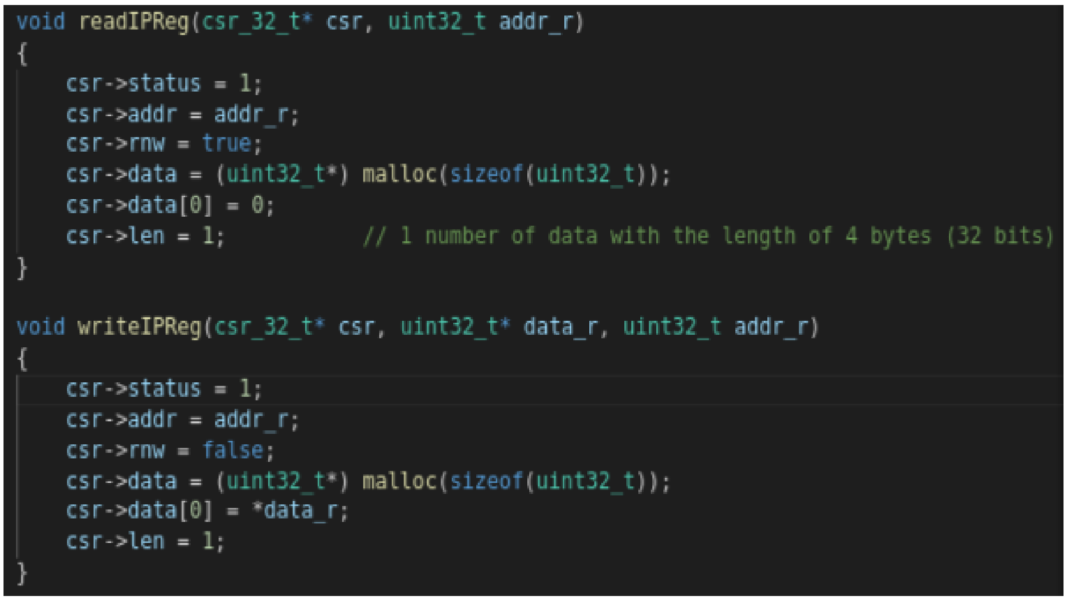
As an intermediary application, the TLM bridge should have interfaces to interact with both the SW driver and the SystemC HW model. This section introduces the feature of the C/C++ interface and the SystemC TLM interface.

4.2.1 C/C++ Interfaces

Two types of operations are designed to take the address and data value set by the SW driver and a variable named *csr* as parameters. The *csr* variable stands for Control/Status Register, which is a struct type variable used to store all the register access information required by the SystemC HW model besides the address and data of register such as:

- Status: indicates the write/read response (1 = OK, 0 = NOT OK)
- Len: indicates the length of data
- Rnw: indicates the command of access. (true = read, false = write)

The operations complement the necessary access information and then assign it to *csr* for later transaction configuration. The complemented access information done by the operations can be seen in Figure 12.



```
void readIPReg(csr_32_t* csr, uint32_t addr_r)
{
    csr->status = 1;
    csr->addr = addr_r;
    csr->rnw = true;
    csr->data = (uint32_t*) malloc(sizeof(uint32_t));
    csr->data[0] = 0;
    csr->len = 1;           // 1 number of data with the length of 4 bytes (32 bits)
}

void writeIPReg(csr_32_t* csr, uint32_t* data_r, uint32_t addr_r)
{
    csr->status = 1;
    csr->addr = addr_r;
    csr->rnw = false;
    csr->data = (uint32_t*) malloc(sizeof(uint32_t));
    csr->data[0] = *data_r;
    csr->len = 1;
}
```

Figure 12. Read and Write operations

4.2.2 SystemC TLM Interfaces

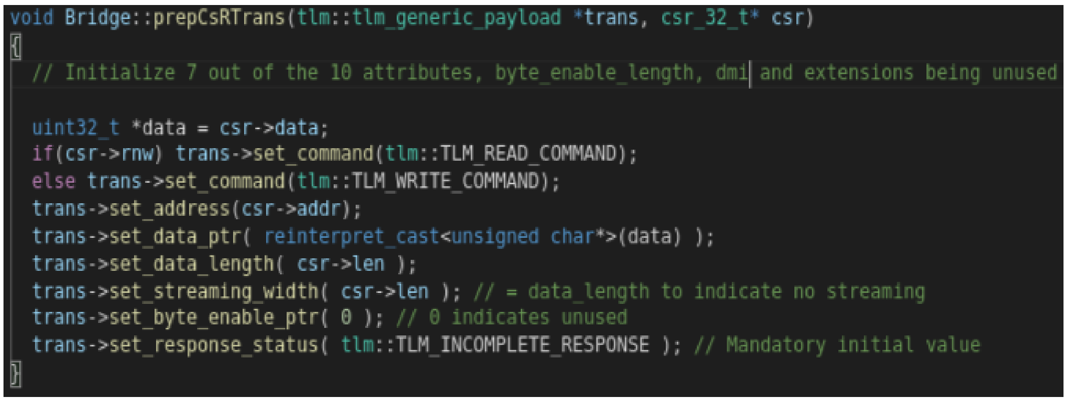
The SystemC TLM interfaces enable the TLM bridge the ability to transform the necessary access information set by *readIPReg()* and *writeIPReg()* operations into transaction attributes that can be sent to the HW model to perform register access.

1. TLM socket

The TLM bridge has an initiator socket named *initiator_socket_control,* which is bound to the *target_socket_control* of the SystemC HW model for transaction exchanges. These sockets are the simple type of TLM sockets.

2. Transaction configure function

A function named *prepCsRTrans()* is based on the necessary access information complemented by operations to initialize the seven attributes of a transaction. The attribute initialization is shown in Figure 13. These attributes are later extracted by the SystemC HW model and used to perform the register access process.



```
void Bridge::prepCsRTrans(tlm::tlm_generic_payload *trans, csr_32_t* csr)
{
  // Initialize 7 out of the 10 attributes, byte_enable_length, dmi and extensions being unused

  uint32_t *data = csr->data;
  if(csr->rnw) trans->set_command(tlm::TLM_READ_COMMAND);
  else trans->set_command(tlm::TLM_WRITE_COMMAND);
  trans->set_address(csr->addr);
  trans->set_data_ptr( reinterpret_cast<unsigned char*>(data) );
  trans->set_data_length( csr->len );
  trans->set_streaming_width( csr->len ); // = data_length to indicate no streaming
  trans->set_byte_enable_ptr( 0 ); // 0 indicates unused
  trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE ); // Mandatory initial value
}
```

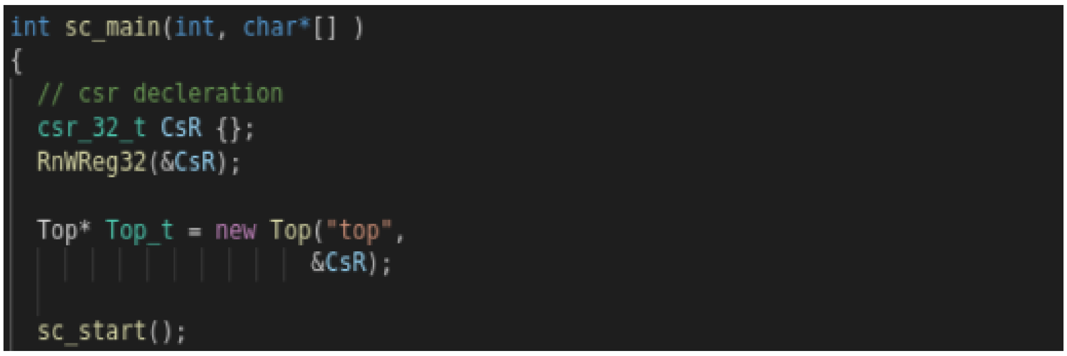
Figure 13. Initialize transaction attributes

4.3 Simulation

In this section, the phases of simulation such as initialization and execution are described. When the operations from the C/C++ interfaces finish complementing the necessary access information, the TLM bridge will start the simulation phases.

4.3.1 Initialization

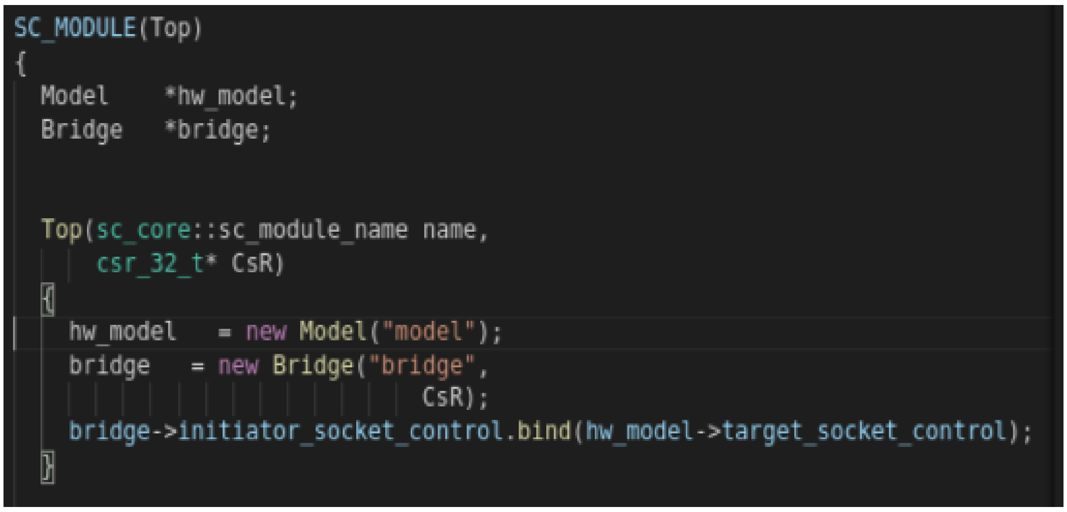
At this stage, the statements prior to *sc_start()* are executed. Particularly, the SW driver sets the address and data value of the register access request and passes it to the TLM bridge operations, and then the TLM bridge operations complement the necessary access information and assign it to the *csr*. After that, the top-level module is initialized which carries the assigned *csr* to later give it to the TLM bridge module. Figure 14 shows the statements executed before *sc_start()*.



```
int sc_main(int, char*[] )
{
  // csr decleration
  csr_32_t CsR {};
  RnWReg32(&CsR);

  Top* Top_t = new Top("top",
                       &CsR);

  sc_start();
```

Figure 14. Statements executed in the elaboration stage

In the top-level module, the modules of the HW and TLM bridge are initialized and the sockets of these modules are bound with each other as shown in Figure 15.



```
SC_MODULE(Top)
{
  Model    *hw_model;
  Bridge   *bridge;


  Top(sc_core::sc_module_name name,
      csr_32_t* CsR)
  {
    hw_model    = new Model("model");
    bridge    = new Bridge("bridge",
                           CsR);
    bridge->initiator_socket_control.bind(hw_model->target_socket_control);
  }
```

Figure 15. Top-level module

The TLM bridge uses the *csr* for later transaction attribute configuration. The process of the initialization phase can be viewed in Figure 16.

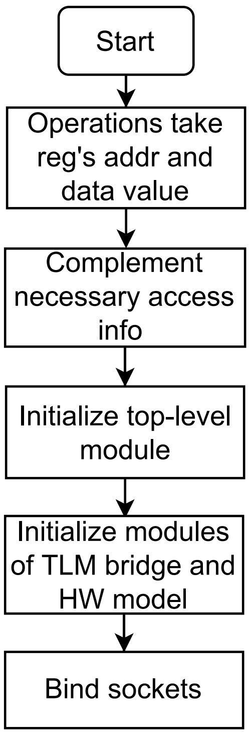

Start
Operations take reg's addr and data value
Complement necessary access info
Initialize top-level module
Initialize modules of TLM bridge and HW model
Bind sockets

Figure 16. Initialization process

4.3.2 Execution

After the initialization phase is completed successfully, the *sc_start()* will start the execution phase. In this phase, the TLM bridge triggers a thread called *TransGen()*, where the *prepCsRTrans()* function configures a transaction with seven attributes based on the complete access information carried by *csr*. The transaction is then sent to the HW model over the initiator socket using the blocking transport method *b_transport* through. Figure 17 shows the implementation of the *TransGen()* thread.

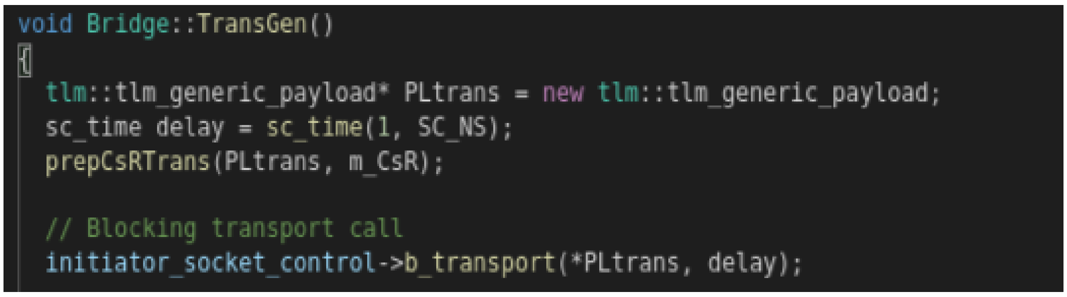

```
void Bridge::TransGen()
{
  tlm::tlm_generic_payload* PLtrans = new tlm::tlm_generic_payload;
  sc_time delay = sc_time(1, SC_NS);
  prepCsRTrans(PLtrans, m_CsR);

  // Blocking transport call
  initiator_socket_control->b_transport(*PLtrans, delay);
```

Figure 17. *TransGen()* thread

The blocking transport method *b_transport_model* implemented in the HW model first takes the transaction through the target socket and extracts all the transaction attributes initialized by the TLM bridge, as shown in Figure 18.

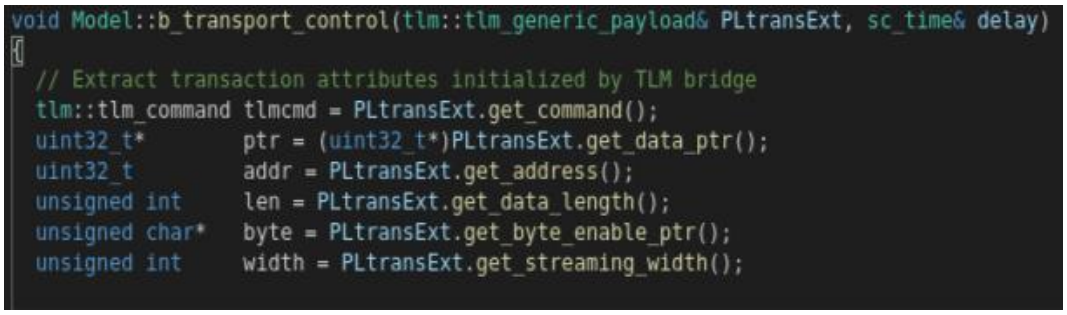
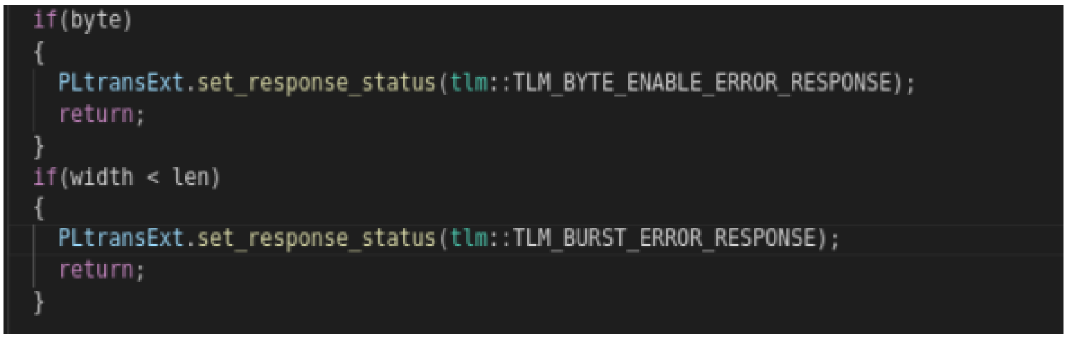

```
void Model::b_transport_control(tlm::tlm_generic_payload& PLtransExt, sc_time& delay)
{
  // Extract transaction attributes initialized by TLM bridge
  tlm::tlm_command tlmcmd = PLtransExt.get_command();
  uint32_t*        ptr = (uint32_t*)PLtransExt.get_data_ptr();
  uint32_t         addr = PLtransExt.get_address();
  unsigned int     len = PLtransExt.get_data_length();
  unsigned char*   byte = PLtransExt.get_byte_enable_ptr();
  unsigned int     width = PLtransExt.get_streaming_width();
```

Figure 18. Transaction attributes extraction

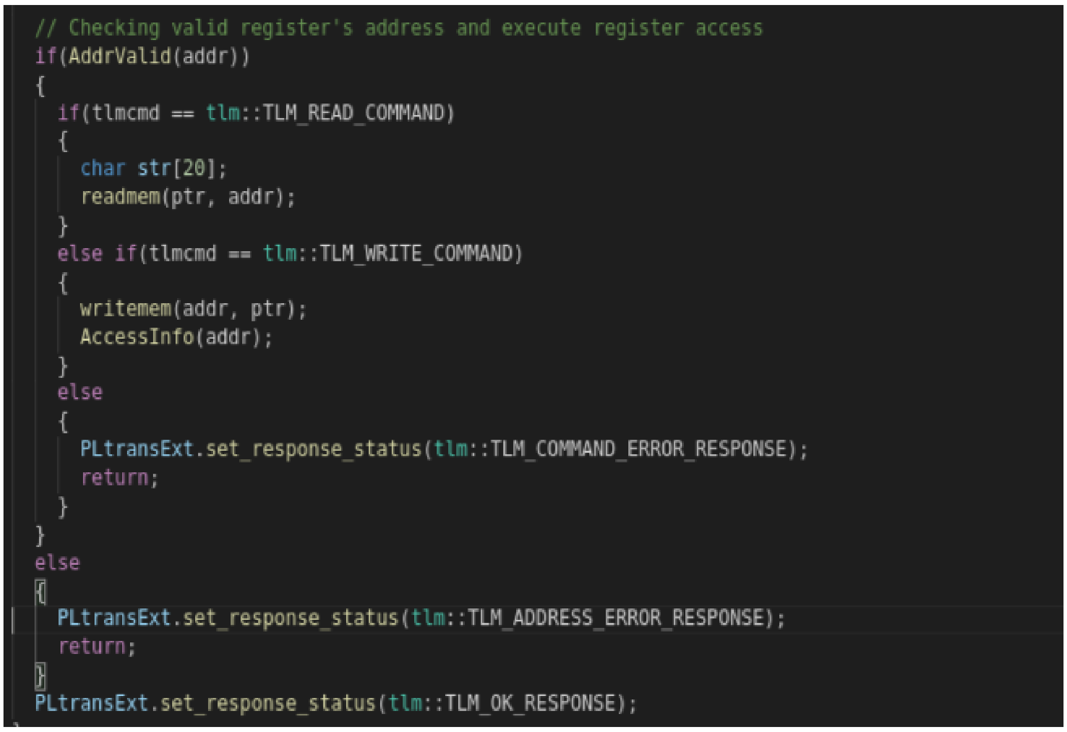
The values extracted from the transaction are assigned to local variables for convenient processing of register accesses. Figure 19 shows some attributes checking will be performed to ensure that the TLM bridge does not attempt to use features that are not supported by the HW model.



```
if(byte)
{
  PLtransExt.set_response_status(tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE);
  return;
}
if(width < len)
{
  PLtransExt.set_response_status(tlm::TLM_BURST_ERROR_RESPONSE);
  return;
}
```

Figure 19. Transaction attributes check

Once the transaction attributes are checked, the HW model proceeds with the register access process by first checking if the address of the register being accessed is in range by a function named *AddValid().* Figure 20 shows the implementation of the register access process handled by the HW model.



```
// Checking valid register's address and execute register access
if(AddrValid(addr))
{
  if(tlmcmd == tlm::TLM_READ_COMMAND)
  {
    char str[20];
    readmem(ptr, addr);
  }
  else if(tlmcmd == tlm::TLM_WRITE_COMMAND)
  {
    writemem(addr, ptr);
    AccessInfo(addr);
  }
  else
  {
    PLtransExt.set_response_status(tlm::TLM_COMMAND_ERROR_RESPONSE);
    return;
  }
}
else
{
  PLtransExt.set_response_status(tlm::TLM_ADDRESS_ERROR_RESPONSE);
  return;
}
PLtransExt.set_response_status(tlm::TLM_OK_RESPONSE);
```

Figure 20. Register access process

If the register address is not supported by the HW model, the access is discarded and notification of an invalid address is prompted. If the address is valid, the access is handled accordingly to the command.

As a result, if the read command is recognized, the HW model will read the current data value in the target register through a function called *readmem()* which reads the data value and saves it to the transaction data pointer attribute. This data value is later received and printed at the TLM bridge.

If the write command is recognized, the HW model writes the desired data value to the target register by a function named *writemem()*. In the case of a write command, the HW model prints the access result through the *AccessInfo()* function which not only displays the current value of the target register but that of other registers. The entire execution process is shown in Figure 21.

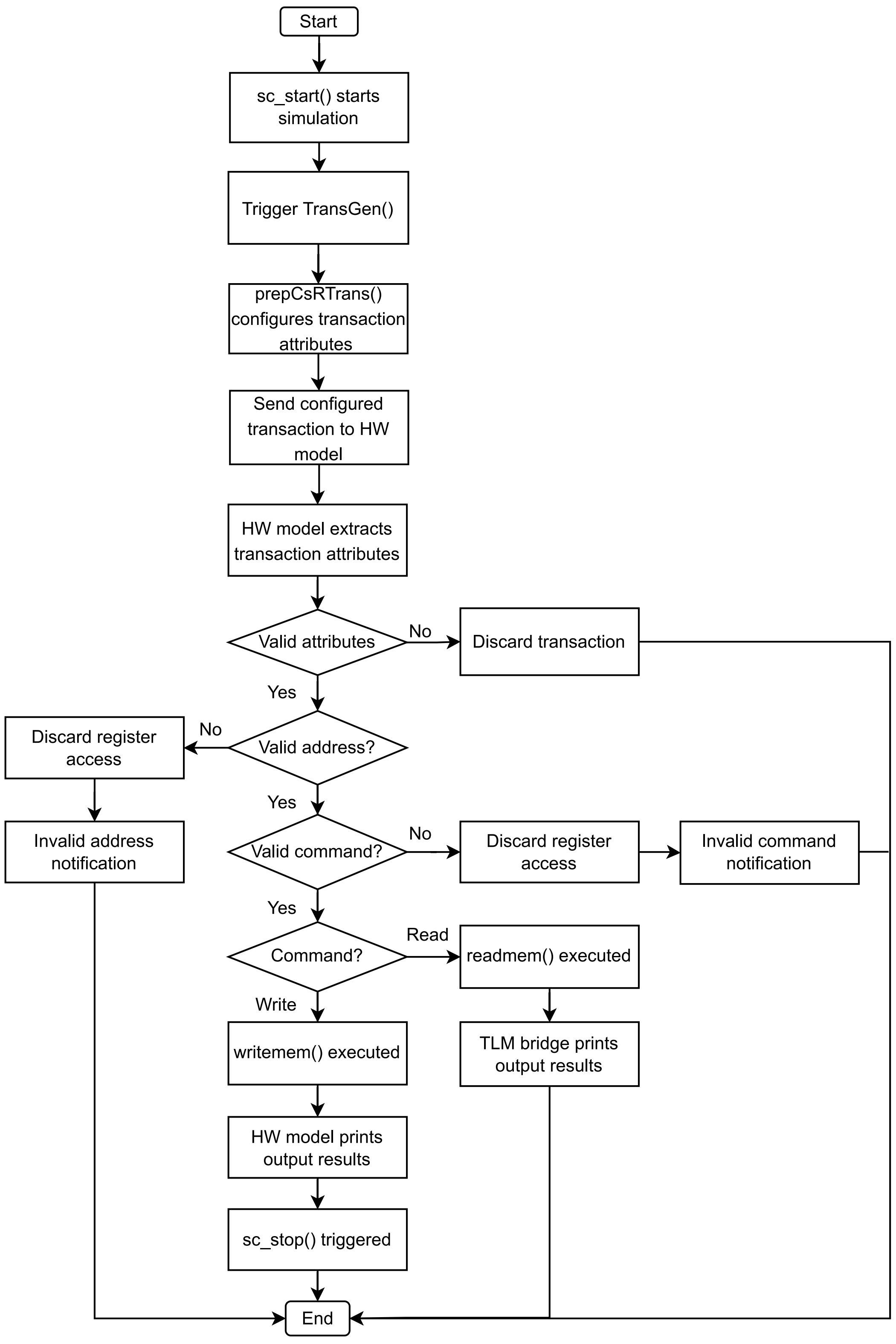

Start
sc_start() starts simulation
Trigger TransGen()
prepCsRTrans() configures transaction attributes
Send configured transaction to HW model
HW model extracts transaction attributes
Valid attributes
No
Discard transaction
Yes
Valid address?
No
Discard register access
Invalid address notification
Yes
Valid command?
No
Discard register access
Invalid command notification
Yes
Command?
Read
readmem() executed
TLM bridge prints output results
Write
writemem() executed
HW model prints output results
sc_stop() triggered
End

Figure 21. Execution process

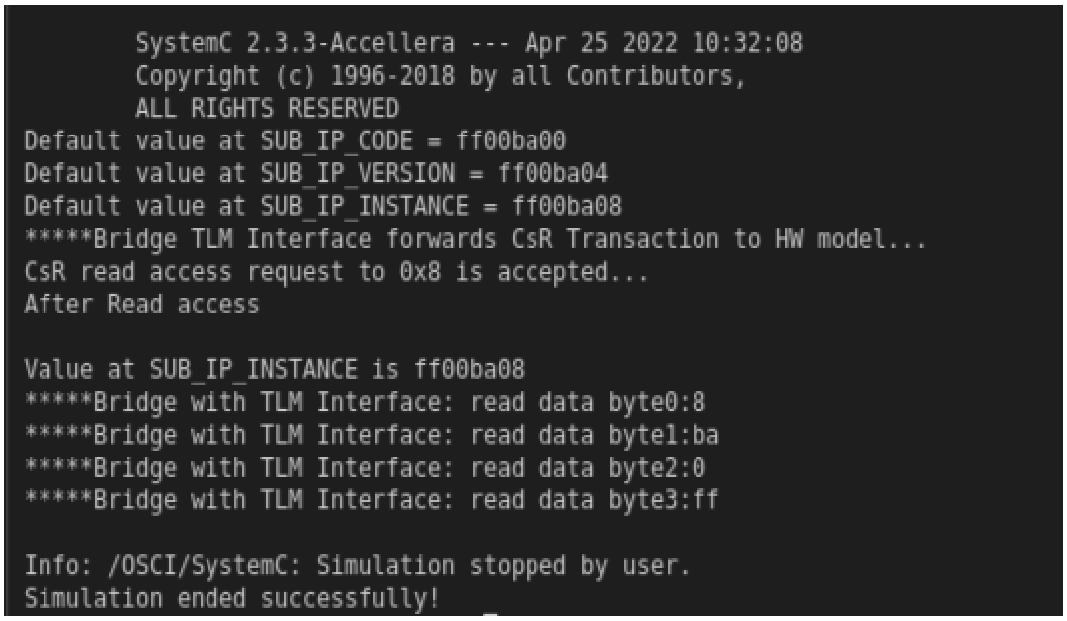
4.4 Outcome

In this section, we examine the results of the entire integration process. The outputs of three registers of the SystemC HW model are discussed. The results prove that the SW driver successfully accessed the SystemC HW model register using the TLM bridge. Register names and outputs do not reflect those of the final product.

The outputs are involved in three 32-bit registers which are SUB_IP_CODE, SUB_IP_VERSION, SUB_IP_INSTANCE, and their addresses are 0x0, 0x4, and 0x8, respectively.

4.4.1 Read Access Result

Figure 22 shows the default values of these three registers, initialized by the HW model printed in the first three lines. If the address and the command are verified and the register access request is accepted, the data value of the target register will be printed byte by byte in LSB (byte) to MSB order. Since this is a 32-bit register, a total of 4 bytes is printed. The target register is SUB_IP_INSTANCE.

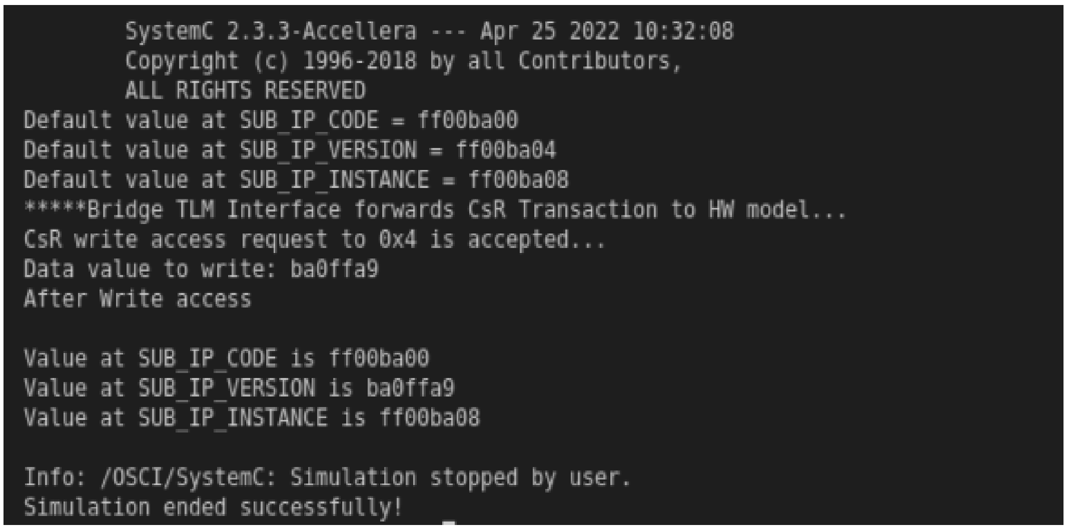


```
        SystemC 2.3.3-Accellera --- Apr 25 2022 10:32:08
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Default value at SUB_IP_CODE = ff00ba00
Default value at SUB_IP_VERSION = ff00ba04
Default value at SUB_IP_INSTANCE = ff00ba08
*****Bridge TLM Interface forwards CsR Transaction to HW model...
CsR read access request to 0x8 is accepted...
After Read access

Value at SUB_IP_INSTANCE is ff00ba08
*****Bridge with TLM Interface: read data byte0:8
*****Bridge with TLM Interface: read data byte1:ba
*****Bridge with TLM Interface: read data byte2:0
*****Bridge with TLM Interface: read data byte3:ff

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation ended successfully!
```

Figure 22. Read access output

4.4.2 Write Access Result

In Figure 23, we inspect the result of a write access request to the target register SUB_IP_VERSION with an address is 0x4. The same address checking and register access processing as the read access is handled by the HW model. After the write access executes, a new value 0xBA0FFA9 is written to the target register.

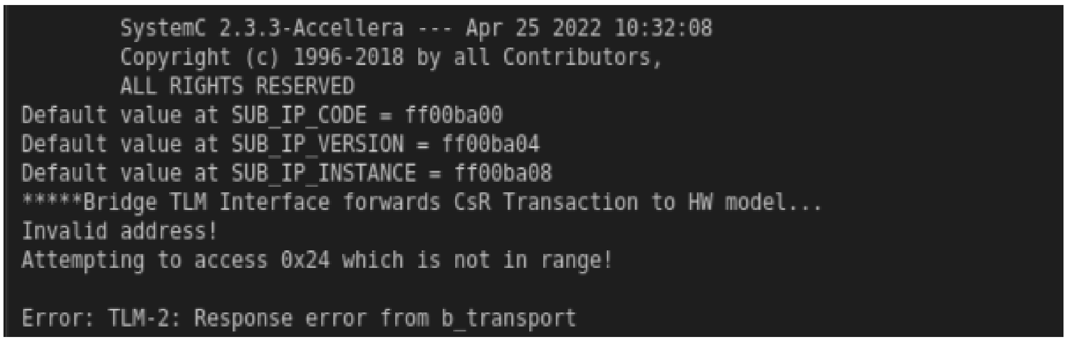


```
        SystemC 2.3.3-Accellera --- Apr 25 2022 10:32:08
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Default value at SUB_IP_CODE = ff00ba00
Default value at SUB_IP_VERSION = ff00ba04
Default value at SUB_IP_INSTANCE = ff00ba08
*****Bridge TLM Interface forwards CsR Transaction to HW model...
CsR write access request to 0x4 is accepted...
Data value to write: ba0ffa9
After Write access

Value at SUB_IP_CODE is ff00ba00
Value at SUB_IP_VERSION is ba0ffa9
Value at SUB_IP_INSTANCE is ff00ba08

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation ended successfully!
```

Figure 23. Write access output

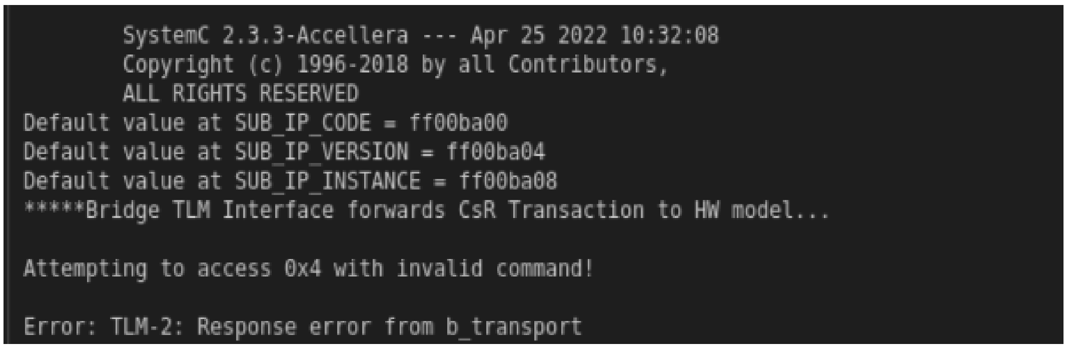
4.4.3 Invalid Register Access Result

The HW model always detects invalid register accesses, which can be either an attempt to access a register that does not exist as shown in Figure 24, or an access attempt without a valid command as shown in Figure 25.



```
        SystemC 2.3.3-Accellera --- Apr 25 2022 10:32:08
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Default value at SUB_IP_CODE = ff00ba00
Default value at SUB_IP_VERSION = ff00ba04
Default value at SUB_IP_INSTANCE = ff00ba08
*****Bridge TLM Interface forwards CsR Transaction to HW model...
Invalid address!
Attempting to access 0x24 which is not in range!

Error: TLM-2: Response error from b_transport
```

Figure 24. Invalid register address access



```
        SystemC 2.3.3-Accellera --- Apr 25 2022 10:32:08
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Default value at SUB_IP_CODE = ff00ba00
Default value at SUB_IP_VERSION = ff00ba04
Default value at SUB_IP_INSTANCE = ff00ba08
*****Bridge TLM Interface forwards CsR Transaction to HW model...

Attempting to access 0x4 with invalid command!

Error: TLM-2: Response error from b_transport
```

Figure 25. Invalid command access

When either invalid register address or command is detected, the HW model will send an error response to the TLM bridge, and the simulation will be stopped.

5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

In SoC development, integration is always a crucial process to enable the full functionality of the final SoC product. As the Nokia SoC product is a complicated system, both HW and SW will be utilized in several telecommunication network devices such as baseband unit (BBU) or base station (BS) serving from thousands to millions of mobile client devices. For that reason, not only SoC HW and SW driver design is focused but the integration of HW/SW is an indispensable process in SoC development.

Due to the lack of real SoC HW at this early stage of development, the SystemC library with TLM features supported is a significant approach to provide an early SoC HW model for SoC SW driver continuous development. Therefore, an application named TLM bridge was developed as an interconnect tool to integrate the SoC SW driver with the SystemC HW model.

In the Nokia SoC SystemC design, the goal of the TLM bridge is to provide an intermediary application that enables communication capability between the low-level SW driver and high-level abstraction SystemC HW model. The application met most expectations and allowed the SW driver fully access the target register of the SystemC HW model without any fatal errors or unexpected accesses.

5.2 Future Work

Despite the flawless functionality of the application, some challenges still need to be addressed to improve the operation of the TLM bridge. The limitation is that if the SW driver sends multiple register accesses with different *csr* configurations, it will cause a module initialization error because SystemC does not allow reinitialization of modules carrying the *csr* during the execution stage. My proposal to solve this problem is to modify the way register access information is passed from the

SW driver to the TLM bridge operations so that the program does not have to initialize the module and refresh the simulation kernel over again.

Another promising alternative is to use a standard Unix UDP socket as a server for the SW driver client exchanging register access information with the SystemC HW model without the constraints of multiple simulation initialization. The TLM bridge maps the access information received from the UDP socket and sends access requests to the SystemC HW model.

REFERENCES

/1/ Anysilicon. What is a System on Chip (SoC)? Accessed 03.04.2022. https://anysilicon.com/what-is-a-system-on-chip-soc/.

/2/ Nokia. Company. Accessed 03.04.2022. https://www.nokia.com/about-us/company/.

/3/ Nokia. Nokia launches next-generation AirScale 5G portfolio powered by ReefShark technology. Accessed 03.04.2022. https://www.nokia.com/about-us/news/releases/2021/06/24/nokia-launches-next-generation-airscale-5g-portfolio-powered-by-reefshark-technology/.

/4/ IEEE 1666-2011. 2011. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Standards Association.

/5/ The European Space Agency. System-Level Modeling in SystemC. Accessed 03.04.2022. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/System-Level_Modeling_in_SystemC.

/6/ Accellera. 2012. SystemC 2.0 User's Guide. Accessed 03.04.2022. https://github.com/accellera-official/systemc/tree/master/docs/sysc/archived.

/7/ Bhasker, J. 2002. A SystemC Primer. Treeline Drive, Allentown, PA. Star Galaxy Publishing.

/8/ Tech Design Forums. Transaction level modeling. Accessed 03.04.2022. https://www.techdesignforums.com/practice/guides/transaction-level-modelling-tlm/.

/9/ Black, D.C., Donovan J., Bunton, B. & Keist A. 2009. SystemC: From the Ground up. 2nd ed. Spring Street, New York, USA. Springer Science & Business Media.

/10/ Hsiung, P.A. 2004. Transaction-Level Modeling in SystemC. Accessed 03.04.2022. https://www.cs.ccu.edu.tw/~pahsiung/courses/soc/notes/SystemC_TLM.pdf.

/11/ Aynsley, J. 2008. Tutorial 1 – Sockets, Generic Payload, Blocking Transport. Accessed 03.04.2022. https://www.doulos.com/knowhow/systemc/tlm-20/tutorial-1-sockets-generic-payload-blocking-transport/.

/12/ Aynsley, J. 2009. What is TLM-2.0? Accessed 03.04.2022. https://www.youtube.com/watch?v=ocniBsPNRwk.

/13/ Debes, E. 2011. Computing Systems for Signal Processing. Accessed 03.04.2022. https://www.lri.fr/~de/Archi%20M2R%20Orsay%20Eric%20Debes%20Part%202.pdf.

/14/ Wikimedia Commons. 2009. A way of building an XOR gate from only NAND gates. Accessed 03.04.2022. https://commons.wikimedia.org/wiki/File:XOR_from_NAND.svg.

/15/ Gate Overflow. 2017. ISRO 2008- ECE Odd parity. https://gateoverflow.in/120122/Isro-2008-ece-odd-parity.

/16/ GeeksforGeeks. Finding the Parity of a number Efficiently. Accessed 03.04.2022. https://www.geeksforgeeks.org/finding-the-parity-of-a-number-efficiently/