Technische Universität München
Lehrstuhl für Integrierte Systeme
Prof. Dr. sc. techn. A. Herkersdorf

SystemC Laboratory

Manual

(ENGLISH)

Version 3.6e

Thomas Wild

Overview

Target and contents of the lab

The SystemC lab aims at conveying a basic understanding to the participants of how to model with the system level language SystemC and how to apply it for the exploration of HW/SW architectures in early phases of the design process. Special focus in this context is given to transaction level modeling (TLM), in respect to both the underlying concepts and the currently used standard.
The lab is basically structured in three parts: First, the participants should get to know basic concepts of SystemC, which is a class library based extension to C++. This part encompasses the structure of SystemC models and major language constructs to model and simulate concurrency.
The second part covers communication concepts in SystemC on different abstraction levels. The starting point is to model communication on signal level similar to hardware description languages. Then interaction through function calls as rudimentary step towards transaction level modeling is considered. Finally, one of the communication mechanisms proposed in the TLM 2.0 standard is applied. This study is done with a very simple system setup consisting of two modules communicating via a FIFO.
In the final part a network processing application, which is mapped onto a bus based computing architecture, is modeled on TLM level and different implementation options are to be explored using the TLM 2.0 mechanisms introduced in the second part.

With the successful completion of the lab and on the basis of the conveyed concepts, the participants should be able to independently write basic SystemC models for architecture exploration.

Realization

The exercises are specified in a way that they can be solved solely with the help of the contents as conveyed in the introductory presentations at the beginning of the lab. The material is made available electronically. Further literature is not needed for carrying out the exercises. However, if you are interested in getting deeper insights you should have a look at the following two books:

- David C. Black, et. al., "SystemC: From the Ground Up", Springer
- T. Grötker, et. al., "System Design with SystemC", Kluwer

As a final reference the Language Reference Manual for SystemC (IEEE 1666-2011 standard) could be used, which is available from IEEE (link can be found in the download section of the Accellera web page **www.accellera.org**). However, as this document is very complex it is recommended and it should be sufficient to use the distributed presentations slides only.

The exercises are carried out independently by each student with free schedule. Besides the three introductory presentations there are no further common appointments. For doing the exercises, each participant gets access to the lab workstations of the Institute for Integrated Systems.
As no commercial tools are used and as the SystemC libraries required in this lab are available for download from the Accellera web page (www.accellera.org) it is also possible to carry out the exercises on a computer at home. (See “use of own computer” below.)

Topics of the final exam are only the language constructs of SystemC and TLM 2.0 as they are actually used in the lab. **Language constructs that have not been used during the lab**

and especially the application example of the network processing application, which serves to demonstrate the usage of SystemC in the lab, **are not topics of the final exam**.
The final grade of the SystemC lab is generated from one or more deliverables and the final exam. Details concerning grading and the submission of the deliverable(s) are announced at the beginning of the lab and online via the news section of the course web page.
Deliverables are part of the academic assessment and thus have to be prepared and submitted individually by each participant. **Submission of (nearly) identical solutions is considered cheating and will be handled appropriately.**

Organizational Matters

Further information on all organizational matters, especially concerning the final exam and grading as well as the available support will be announced in the first introductory presentation.
This and further current news and announcements as well as downloads will be available via the associated web page in the TUM Moodle eLearning portal (https://www.moodle.tum.de). The Moodle page for the respective semester can be found via TUM Online.

General hints for carrying out the exercises on the lab workstations

- **Account**
  Please use your **LRZ account** to log into the LIS lab workstations. Your LRZ account will be enabled after your lab registration has been confirmed in the first lab introductory presentation.

  When you are logged in, start a terminal window by opening the "Applications Menu" in the top left corner of the GUI and then clicking on "Accessories" and then on "Terminal Emulator".

  In the terminal window you have to execute the command

  ```
  lislab load systemc
  ```

  which sets the environment variables for the tools that are used in the lab.

  IMPORTANT: In each new terminal window, you have to execute `lislab load systemc` before the tools of the lab can be used.

  IMPORTANT: The basic account settings on the LIS workstations do not include the current directory as part of the search path. When you want to execute a program stored in the current directory (e.g. the SystemC simulation programs) you have to start it by entering *`./program_name`* (mind the leading ./).

  When you want to finish your session please log off from the workstation via the appropriate entry in the "Applications Menu" on the top left of the GUI.

  Please be sure to log out when you leave the workstation!!

If you leave a workstation with locked screen your session might be terminated by system administrators (except short term locks of course)**.**

- **Initialization**
  Before you can start with the lab you first have to **load the directory structure and the given files of the lab**. To do this, generate a specific directory for the lab (e.g. "mkdir lab_systemc"). Then change into this new directory ("cd lab_systemc") and load the folder structure by entering the command

  ```
  lislab copy systemc all
  ```

  In consequence, for each of the lab exercises a separate subdirectory **ex_*** is generated in the current folder, where this command is executed. There you can work on the respective exercises und you also find given files. Given files with SystemC source code are either complete or have to be complemented by you. In part, source code files have to be newly generated by you (the manual specifies for each exercises what is to be done by you). Each subdirectory also contains a Makefile, which controls compilation and linking of the respective simulation program with the help of the make program.
  If not stated differently, all path specifications in the subsequent lab manual are given relative to the lab directory, where you copied the lab folders.

  In case you want to re-load an exercise to return to the initial status, execute the command

  ```
  lislab copy systemc <ex_1a, ex_1b, ex_2, ex_3…>
  ```

  (The last parameter is a comma separated list of the folders to be copied.)
  Available folders with the given names are not overwritten but renamed; the new copies get the desired names.

The command `lislab`, which helps to setup the environment for the lab and also to set the required environment variables for the used tools, is only available on the LIS lab workstations.

- **Operating system and graphical user interface (GUI)**
  The manual of the SystemC Laboratory is written for carrying out the exercises on Ubuntu Linux installation on the workstations in the LIS lab room (N2135). The workstations are set up to run the XFC window manager.

- **C++ development environment and editor**
  The lab setup as used on the LIS workstations is based on the GNU compiler collection, especially ***g++***, und relies on the ***make*** program, which enables the comfortable generation the simulation programs. For editing the SystemC models you may use an arbitrary editor installed on the Linux workstations, e.g. ***gedit*** or ***nedit***.

- **Some hints for the work on the Linux workstations at LIS**

This section describes some commands that are helpful when working on a command line in a terminal under Linux. (You get a terminal window by opening the "Applications Menu" in the top left corner of the GUI and then clicking on "Accessories" and then on "Terminal Emulator".)

Commands for handling files and directories under Linux

Change to the directory ex_1a:	`cd ex_1a`
Change to the directory one level higher:	`cd ..`
Change to the home- directory:	`cd`
Print the current directory:	`pwd`
List the content of directory ex_1a:	`ls ex_1a`
List the content of the current directory:	`ls`
Extended list the content of the current directory:	`ls -l`
Delete file name_x:	`rm name_x`
Copy a file:	`cp source destination`
Move / rename a file:	`mv source destination`

Printing

If you want to have a nicely formatted printout of your source code from the printer in the lab room, please enter the command

```
a2ps filename_1 [filename_2]
```

on the command line. The parameter of this command is the list of files to be printed. Please print only things that are necessary to save toner and paper. Thank you.

- **Use of own computer**

  The exercises can also be carried out on your own computer or laptop at home. For this purpose, load the SystemC installation from the download section of the Accellera web page (www.accellera.org) and install it locally. The distribution contains information on how to install it under Linux and Windows. **Please be sure to use SystemC 2.3, which also includes the TLM 2.0 library**.

  The other tools used in this lab like the debugger *DDD* and the waveform viewer *gtkwave* are freely available from the Internet as well.

  If you want to use the Makefiles of the individual exercises on your own computer, you have to make sure that the installation path of the SystemC library is stored in the environment variable SYSTEMC.

  Further, the directory structure should be identical to the LIS installation (ex_*, config, npu_common and PCAP_samples subdirectories stored in a common folder).

- **External network access**

  If you want to login into the LIS workstations from the Internet you can do this only with *ssh* (file transfer via *sftp*), however only on one specific computer, which is accessible through our firewall: **`ssh.lis.ei.tum.de`**.

  In any case, **do not edit files and run simulations on `ssh.lis.ei.tum.de`**. Instead, make a further *ssh* connection from the shell you get on `ssh.lis.ei.tum.de` to one of the lab workstations `prakt??` (01 to 18). For this, enter the command e.g. *ssh prakt01* and authenticate yourself. On the lab workstation your home directory of the LRZ account is visible, and thus also the directories containing the files associated to the different lab exercises. Change into the appropriate subdirectory and start there the desired programs.

If you have an X11 server running on your local machine, graphical output associated to a program (e.g. the waveform viewer) can be displayed locally. (In this case you must have enabled X11 forwarding in your *ssh* client, which is the default setting of *ssh* on the LIS lab workstations.)
To logout from the lab workstation enter `exit` on the command line, which will return you to `ssh.lis.ei.tum.de`. To close the connection to `ssh.lis.ei.tum.de` enter `exit` there as well, which will log you off the LIS computers.

If you find errors or inconsistencies in this manual, or if information is lacking needed for doing an exercise, please do not hesitate to inform me by sending a mail to thomas.wild@tum.de. This would enable us to both inform your colleagues and correct the manual for the subsequent semesters.
Many thanks in advance!

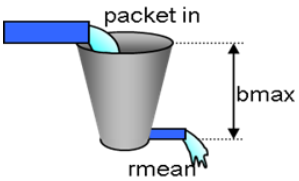
Exercise 1 Basic SystemC

Before starting with the lab, be sure that you have copied the given files into a subdirectory of your home directory, as described above under "Initialization". All paths mentioned in the subsequent manual are relative to this directory.

The first exercise is divided into two sub-parts: In the first part 1a you should gain first hands-on experience with a given SystemC model, which will be compiled and executed. You will also get to know the usage of the debugger *DDD* which can assist you in tracking down problems, if a model does not work as expected. This will be helpful in completing the subsequent exercises when you have to write your own SystemC code. In the second part 1b, you will write and simulate your first own SystemC model (a BCD counter).

In experiment 1a you will simulate a traffic policer module that is used in data communications to limit the flow of data into the network in order to help guarantee quality of service. A policer enforces the traffic sent by a data connection into the network to comply with the parameters defined at connection setup. Excess traffic will be dropped by the policer. This limitation of traffic flow helps to avoid overflow of buffers within the network, which would otherwise also inhibit other connections. The algorithm that is used for this purpose in the policer is the so called "leaky bucket". It is dimensioned for a specific mean data rate and accepts burst traffic up to a certain level. For this purpose the leaky bucket has two parameters r_{mean} and b_{max} and works according to the following principle:

> For each arriving packet, a leaky bucket is filled with the number of bytes corresponding to the packet size. On the other hand, it is drained continuously with a given data rate r_{mean}. As long as the bucket does not exceed its maximum size b_{max}, arriving packets are granted access to the network. If the bucket would overflow with an arriving packet, the packet will be dropped (and the bucket is not filled up).



Leaky Bucket Algorithm

```
for each packet arrival do
  counter -= (curr_time - last_arrival_time) * rmean;
  if(counter<0) counter = 0;
  if(counter + size > bmax) {
    discard = true;
  } else {
    counter += size;
    discard = false;
  }
```

The SystemC model of such a leaky bucket based policer is given in the folder *ex_1a*. It contains the files of the policer module (*polic.h* and *polic.cpp*), a data generator (*data_gen.h* und *data_gen.cpp*), which stimulates the policer with a traffic pattern, and the *main.cpp* file with the top level of the simulation model.
The data generator outputs the size of arriving packets on the output port *dat*. Each new value appearing at this output indicates a new arrival of a packet with the corresponding size in bytes. The size values and the time between the simulated packet arrivals are generated randomly. The parameters in *data_gen.cpp* are set in a way that the packet size is evenly

distributed in the range from 64 to 1500 bytes (corresponding to Ethernet) resulting in a mean value of 782. The inter arrival time is at least the time needed for sending the previous packet.

The policer reads size of the arriving packet on its *size* input performs the leaky bucket algorithm and outputs either true or false on the *discard* output, depending on the result of the calculation.

Please follow the steps given in the following to simulate the policer as described above.

First, start a terminal window (via the applications menu) and change to the folder *ex_1a*. Then, load the given source files in the editor of your choice (e.g. *gedit* or *nedit*) and verify the SystemC constructs as presented in the introductory presentation.

Like in all other exercises of this lab, a Makefile is given in the folder *ex_1a* as well. This Makefile is set up appropriately so that the complete compilation and linkage process can be controlled by only entering *make* commands.

Makefiles and make:

The basic idea behind using Makefiles is that the `make` command can determine from the Makefile which source files have to be compiled and linked to build the final simulation program. Thus, you do not have to call the compiler for each file and link the program with the linker manually. Instead, `make` does this automatically for you. A further advantage is that when changing one of the source files `make` can even determine what other source code files are influenced by this modification and compiles them automatically for you.
As in these Makefiles the names of the involved source code files are specified explicitly, you should **use exactly the file names mentioned in the lab manual**, which makes sure that the given Makfiles really fulfill their purpose. (Otherwise you would have to modify the Makefiles by yourself …)
Some parameters in the Makefile might be of interest for you independent of potential changes:
The name of the executable simulation program that is finally generated is always built from the content of the parameter MODULE and the suffix ".x". If MODULE is set to e.g. *module_name* the executable simulation program will be *module_name.x*.

In addition, the given Makefile also defines the following `make` commands, which can be entered in the terminal window:

`make depend`	determines the dependency between source code files so that make knows what files have to be (re-)compiled. On execution of this command additional lines will be appended to the Makefile. **This command should always be called at the very beginning or when you add/remove files in the subfolder.**
`make`	(without parameter) compiles the simulation program. On success, you should find a program ***module_name.x*** in the folder of the exercise.
`make clean`	removes all compiled object files and the previous simulation program to start a completely fresh compilation when you enter `make`.

Before compiling the model for the policer, be sure that you have set the environment in the terminal window properly for the SystemC lab by entering

```
lislab load systemc.
```

Now, generate the dependencies for the model in the folder *ex_1a* and compile the model by entering the appropriate make commands. Both commands should execute without error and the simulation program *polic.x* should now be present in the sub-directory.

(In case there were errors in the code, the compiler would output error messages indicating the corresponding file and the line number where an error was detected. As in C and C++ a single error may have a bunch of errors as consequence. Therefore, correction of the first error may lead to significant less error messages…)

Now as you have the simulation program *polic.x* start the simulation by entering

```
./polic.x [number_of_us]
```

where you can optionally give a parameter that determines a different simulation duration than the default 200 us.

When the simulation runs, information is written to the console as it is output by the modules of the simulation model. With this you can observe what happens in our policer system.
Now, change some of the parameters of the *data_gen* to modify the generated traffic pattern and observe the different behavior of the *policer*.

In the next step you should get to know how to use the debugger DDD (Data Display Debugger) for the localization of errors in your own SystemC source code.
A prerequisite for debugging is that the program has been compiled with the "–g" compiler option. This can be accomplished by changing the Parameter CFLAGS in the Makefile appropriately. In the given Makefiles usually two lines with a CFLAGS setting are present, one that tells the compiler to optimize the code, the other to compile it for debugging. One is active and the other is commented out. Take care that the one for debugging is activated.

Then *make clean* in order to remove all object code (that might have been compiled without the required option) and compile the model again. Now each time the compiler is executed by the *make* command the option "–g" should appear on the console.
The compiled simulation model is loaded into the debugger with the command

```
ddd polic.x
```

Then two windows should be displayed: the big DDD window with different sections and a small window with several control buttons. The source code of the model is displayed in the center of the big window. To jump to a specific function, which you want to examine with the debugger, proceed as follows:

- First, enter the name of the function without the brackets into the line below the menu. (Please note that you also have to enter the module name in the form `module::function` if you want to lookup a member function or process of a module.)
- Then, press the **Lookup** button aside.

Jump now the code for the process `polic::policer`.

Then set a breakpoint in the source code of this process (best in one of the first commands). For this, point left to the desired line, press the right mouse button and select **Set breakpoint** from the menu. An activated breakpoint will be indicated as a red stop sign and cause the interruption of the simulation program when the execution comes to this line. In case of a process with an active breakpoint this means that the simulation kernel has to activate the process and the execution has to reach the corresponding line of code.

Then start the simulation by either pressing the **Run** button in the small command window, or by entering `run` on the (gdb) prompt in the lower part of the window.

When the debugger has stopped at a breakpoint you can enter the name of different objects in the line on top of DDD window and press one of the buttons at the right to execute the associated commands, e.g. for printing or displaying the contents of objects (or the contents of addresses where a pointer points to) that are visible in the current scope. When you play around with these options you soon will find out what is possible.
To control the further execution of the program use the small control window: To step through the code line by line, press either the **Step** or the **Next** button. (Next is similar to Step, but does not follow called sub-functions, which might lead you into SystemC internal routines). Using these control buttons you can step through the code line by line and observe the control flow and the contents of internal variables. (Signals cannot be observed in this way.)
If you want to continue program execution to go on until the next breakpoint is reached, press the **Cont** button. When pressing the **Finish** button execution will go on until it returns from a called sub-function. The green arrow always marks the line of code that is executed next.

If you have displayed an object and leave the scope where this object is visible, the data element will be marked as disabled. When execution returns into its scope again, the debugger either automatically re-activates the displayed data or you can do this by double-clicking on the data element. Double-clicking data that is symbolized wit "…" often opens up the contained data and lets you explore more details of data structures.

Please experiment with stepping through the code and displaying or printing data to make yourself familiar with the features of DDD to be prepared for searching potential errors in own code that you will write in subsequent exercises.

During exercise 1b you will now model your first own modules in SystemC. After simulating the modules, you will review the generated waveforms with a waveform viewer.

SystemC provides a specific function to generate a trace file of your simulation:

sc_trace_file sc_create_vcd_trace_file("t_file_name")*

It creates the trace file *t_file_name.vcd* and returns a pointer to an *sc_trace_file* object
The pointer to the *sc_trace_file* object is used to add the traced objects by using the function

*sc_trace (sc_trace_file *, signal_or_port_or_variable, "trace_name")*

Note: The *trace_name* string **must not contain spaces**!

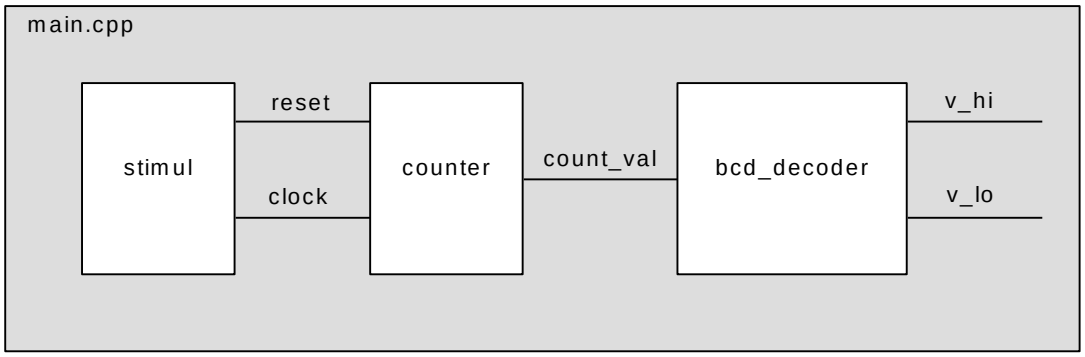
The creation of the trace file (using *sc_create_vcd_trace_file*) and the definition of the traces (using *sc_trace*) is usually done in the main program *sc_main*, although it is also possible to do this inside modules, e.g., in the module constructor.
With progress of the simulation time (such as by starting the simulation for a certain period with *sc_start* in *sc_main*), the signals, ports or member variables are traced, i.e., their changes are stored in the trace file. Finally, the function *sc_close_vcd_trace_file(sc_trace_file *)* closes the specified trace file and writes it to disk. The resulting trace can then be inspected with a waveform viewer.
The following code snippet of a simulation program *main.cpp* demonstrates how the signals *sig_x* in the highest model level, the signal *signal1* in the *m1* instance of a module *module1*, and the member variable *m_var2* of the instance *m2* of a module *module2* are recorded to a file *traces.vcd*.

```
// main.cpp
...
sc_signal<int> sig_x;
// Module
module1 m1("inst1");
module2 m2 ("inst2");
...
sc_trace_file *tf = sc_create_vcd_trace_file("traces");
sc_trace(tf, sig_x, "sig_x_name");
sc_trace(tf, m1.signal1, "m1_s1_name");
sc_trace(tf, m2.m_var2, "m2_var2_name");
...
sc_start(100, SC_NS);
sc_close_vcd_trace_file(tf);
...
```

Now you should write your first own SystemC model and use the wave form tracing as described above to verify the correct function of your module. As an application example we use a synchronous counter, which is connected to a BCD converter. Finally you will inspect the waveform of the outputs of the modules.

The model is written on Register Transfer Level (RTL) and primarily uses the modeling techniques which are similarly available in hardware description languages.

The setup is as follows: A given module *stimul* generates a *clock* with a 10 ns period and a *reset* signal (both signals are of *bool* type).
Your module *counter* should synchronously (i.e., with rising edge of *clock*) count from 0 to 99 (on overflow it should restart with 0). Furthermore it should be possible to reset the counter synchronously (reset is active low, i.e., when *reset==0* the counter will reset to 0). The output of the counter is required to be of *unsigned short int* data type.
The *bcd_decoder* is again a given module that converts the counter value to BCD. The unit position on the output *v_lo* and the decade *v_hi* are both of type *char*.



The modules *stimul* and *bcd_decoder* are readily available in the *ex_1b* directory (files *stimul.h, stimul.cpp, bcd_decoder.h* and *bcd_decoder.cpp*). It is your task to model the behavior of the *counter* module. In a first step think about what type of SystemC process to use.
The top level file *main.cpp* is partly provided. You need to instantiate and connect your *counter* module here. Furthermore, you need to trace the signals as described before.
The *Makefile* to build the simulation program is also provided. Verify the file names in the Makefile before compilation and continue as in the previous assignment.

A waveform file *trace_file_name.vcd* should be created during simulation. You can open this file with a waveform viewer. In the lab the open source tool *gtkwave* is used. Execute

```
gtkwave trace_file.vcd
```

to open the file in the viewer. In order to display the signal traces select the menu „*Search – Signal Search Tree*“ (Shortcut Shift+Alt+T) to display the list of the available signals. Click on the signal hierarchy named “SystemC”, select the signals to be displayed and click on “Insert” or “Append”.
(*Note*: *gtkwave* does not distinguish between signals and variables. Hence, in the context of the waveform viewer the term signal denotes the trace of a signal or variable.)
s
You can modify the displayed time frame with the button on top of the viewer window. The menu „*Markers*“ allows you to set markers in the viewer and measure time differences.
Please make yourself comfortable with the viewer. You can get help via the „*Help – WAVE Help*“ menu (short Ctl/Strg+H). The help window displays information regarding the clicked functionalities, so that you need to close the help window to continue working.

Finally, verify that your counter and the whole system work as expected.

Beside the signals in the model, try also to trace variables, such as member variables, or temporary variables (e.g., as declared in the implementation of an SC_METHOD). What happens respectively?

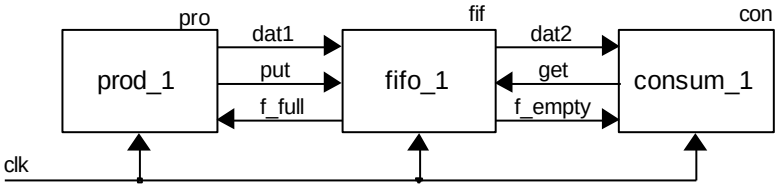
The second part of exercise 1 finishes the introduction that gave you a first impression of SystemC modeling and of the used tools. In the following exercises you will learn about different mechanisms to capture communication within SystemC models.

Exercise 2 Communication on Signal Level

This and the two subsequent exercises address modeling communication within SystemC models on different levels of abstraction, starting from signal level then looking at communication via *sc_interface* and finally addressing TLM 2.0. In the associated experiments a very simple application example will be used. It consists of a producer and a consumer, which communicate via a FIFO module. The following figure shows the configuration.

In this exercise, the interaction between the different function blocks is modeled via signals, like they are known from hardware description languages. For carrying out the first part of this experiment, please change into the directory *ex_2*.

The modules of the first simulation model are working synchronously to the clock *clk*, i.e. all actions are carried out with rising edge of *clk*. In the following, all data signals are assumed to be of type *int*, all control signals to be of type *bool*.



The **producer**, described in the files *prod_1.h* and *prod_1.cpp*, is completely given. This module contains two processes, which communicate via the signal *send*. At particular points in time, the SC_THREAD process *send_trigger()* sets *send* to high (*true*) and thus initiates the SC_THRAD process *produce()* to execute a write action with rising edge of the clock if the FIFO is not full, i.e. if *f_full* is low (*false*). It outputs a certain value on the output port *dat1* and sets *put_data* to high (*true*). The FIFO takes the value with the subsequent rising clock edge and stores it internally. If it becomes full it sets the signal *f_full* to high (*true*).
In case two values v1 and v2 should be written in subsequent clock cycles and the FIFO has only one free position, the following situation occurs:

- The producer starts the first write operation by putting v1 to its output and setting *put* to high.
- With the next rising clock edge, the FIFO takes v1 and then sets *f_full* to high. With the same clock edge the producer puts v2 at its output and keeps *put* in high state (for the producer it is not recognizable that the FIFO comes into full state when writing v1).
- With the subsequent rising clock edge, the producer can recognize that the write operation for v2 failed as at that point in time both *f_full* and *put* are still asserted. To avoid data loss v2 has to be restored in such a case and a new write attempt has to be started. (In the example at hand, the data stream to be written consists of incremented values. Therefore, in case of an unsuccessful the counter that generates the data is decremented.)

The procedure described above allows continuous write operations (in every clock cycle) to the FIFO without losing data on FIFO overflow.

(An alternative approach would be to introduce a further control signal, which is asserted by the FIFO when only one free position is available.)

The other modules needed for simulating the shown configuration are not yet complete. The SystemC source code of FIFO and consumer is to be complemented by you,

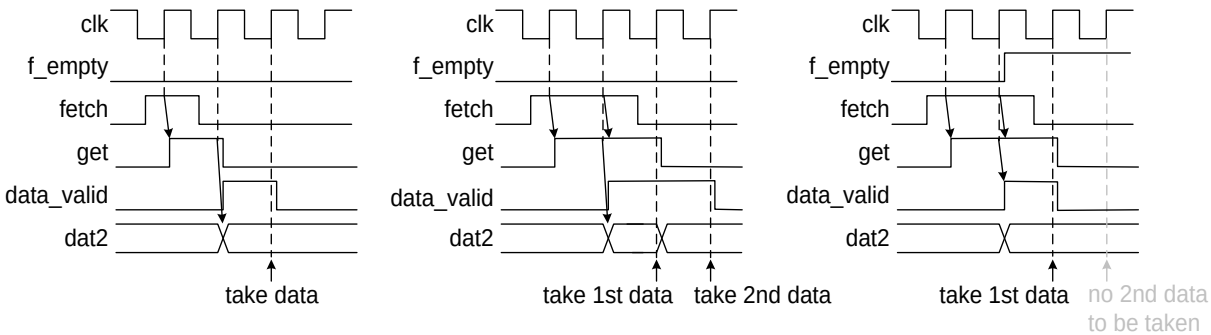
FIFO: The FIFO model is kept very simple. It contains two processes, one for read (*do_read*) the other for write (*do_write*). The FIFO content is stored in a member variable (*integer* array). Further member variables are used for storing the read and write pointers as well as the current fill level.
The FIFO model is mostly complete with the exception of the process registration in the constructor (file *fifo_1.h*). First, have a look at implementations of the two processes in *fifo_1.cpp* to determine their type and then insert the appropriate commands in the constructor for registering the processes with the simulation kernel and specifying the correct sensitivity.

Consumer: Please complement the missing port declarations in the fragmentary given code, (*consum_1.h*).
Further on, the implementation of the process *consumer()* is lacking in *consum_1.cpp*. The consumer should be able to read continuously from the FIFO (as the producer allows continuous write). As can be seen from the header file, similar to the producer two thread processes should be used, which interact via the signal *fetch*. By asserting *fetch* the process *fetch_trigger()* initiates a read operation in the process *consumer()*. Then, the following has to be done within *consumer()*:

- As soon as *consumer()*:recognizes with rising clock edge *fetch* to be asserted and if at the same time the FIFO is not empty, it asserts *get* (set *get* to high).
- With the following rising clock edge the FIFO detects the read command and outputs the value read from its internal memory on the output *dat2*.
- With the subsequent positive clock edge the consumer can read the value via its input port.

I.e. a read operation takes 2 clock cycles until it is completed (c.f. the left timing diagram below.)

To identify within the consumer the correct clock edge for taking the FIFO data, an additional signal *data_valid* is introduced. It basically corresponds to *get*, however, delayed by one clock cycle if the FIFO did not become empty. The timing diagram in the center shows the case of two successful directly subsequent read operations.
If, however, the FIFO becomes empty after the first read operation, the subsequent read cannot be finished successfully. In this case, *data_valid* has to be de-asserted (set *data_valid* to low) to prevent that the value read before (and still available at the FIFO output) is taken again. This situation is shown on the right. Using this mechanism, continuous read from the FIFO is possible.



When writing the *consumer()* process, you have to generate the *get* and *data_valid* signals according to the description above. Further, you should store the read value in the member variable *consumed_data*.
In addition to the pure functionality, please generate messages also in the consumer indicating the simulation time and the read data value when a read operation has been carried out. (C.f. the corresponding commands in the producer.)

Finally you have to complete *main.cpp*: This comprises the instantiation of the three modules making up the top level of the simulation model depicted above and the generation of appropriate waveforms that can be used for observing the proper functionality of the system with *gtkwave*.

Now, compile your completed SystemC model by first entering the command

make depend

and then

make

on the command line of the terminal window.
The first make command updates the dependencies among the individual files of the model (c.f. the description of make above in the manual). The second make command starts the compilation of the files containing the source code and – if successful – links the compiled object code and the SystemC class library together to the executable program *sim_1.x*.

Verify that this program has actually been generated in the directory of the current exercise. Then, run the simulation by entering the command

./sim_1.x

and observe the output in the terminal window.

Verify the functionality of your model by checking the console output and by inspecting the traces, which you have created within *sc_main()*.

(If you want execute the simulation for a different time period than the default value of 5000 ns, start the simulation by ./***sim_1.x sim_dur*** with *sim_dur* specifying the desired number of ns simulation time)

Exercise 3 Communication via *sc_interface*

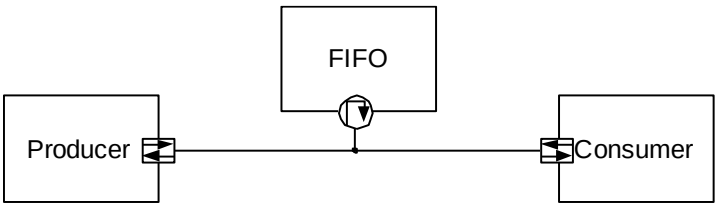
Modeling on signal level as done in the previous experiment is not suited for effectively exploring alternative system architectures because many details of the implementation, especially the communication protocols, e.g. the handshaking between modules, would have to be modeled in detail. These details, however, are not yet known in the exploration phase early in the design flow. In fact, they become available only after several refinement steps of the chosen architecture.
Further on, models that communicate by means of signals are more complex and require much effort for modeling or for modifications. In respect to simulation performance, they also need much more processing power or take longer to execute.
Therefore, SystemC models for exploration purposes are assembled from components that interact with function call (often called transactions). This approach allows to leave it open how the interaction is actually realized in the final implementation.

As a first step, this experiment covers the underlying principles how such transaction based communication can be implemented using the predefined SystemC class *sc_interface*.

A transaction corresponds to calling a function, which is declared in an interface and implemented in a hierarchical channel. The hierarchical channel is therefore derived from this interface (in the sense of inheritance in C++). In this way, processes or member functions within a module with an appropriate *sc_port*, which is connected to the hierarchical channel, can call these interface functions and thus interact with the hierarchical channel. (See the corresponding slides of the introductory presentation.)

In this experiment we will generate and simulate a model that applies these mechanisms for the FIFO system. To accomplish this exercise please change into the directory *ex_3*.

Communication with the FIFO is now realized via the functions *read_fifo* and *write_fifo* to read/write data. These functions are declared in an appropriate interface (*fifo_if*) and implemented in the new FIFO *fifo_2*, which is a hierarchical channel. Therefore, instead of ports of type *sc_in<T>* and. *sc_out<T>* producer and consumer need a port of type *sc_port<fifo_if>*. These ports are connected to the FIFO and allow calling the interface functions of the FIFO from within producer and consumer.



Please first have a look at the model of the FIFO, which is given in the files *fifo_2.h* and *fifo_2.cpp*. Your attention should especially be on the two functions *read_fifo* and *write_fifo*, which replace the processes that have been used in the first FIFO model for writing and reading data. I.e. the new FIFO *fifo_2* does not contain processes any more.

The first task of you is to write the missing interface declaration of *fifo_if* and store it in the file *fifo_if.h*.

Then you have to complete the producer and consumer models. Internally they operate in a similar way as the corresponding modules in the previous experiment. They also use two processes that trigger read and write actions, which are executed in a separate processes. As a difference to the previous experiment, synchronization is done here with events (instead of signals) and the interaction with the FIFO is done using the functions declared in *fifo_if.h*. These functions are executed immediately, without waiting for a rising clock edge as the model does not have a clock any more.

First, the producer module should be completed. For this, generate the header file *prod_2.h* and then complement the synchronization of the process *producer()* with the *send_trigger()* process in the implementation file *prod_2.cpp*.

Then write the process *consumer()*, which reads data from the FIFO, in the consumer module. (You may take the *producer()* process as a sample and modify it appropriately.) The *consumer()* process should generate output on the console in the same way as the producer so that the interaction of the consumer with the FIFO including the read data can be observed. (Use the corresponding section of code as in the *producer()* process to generate the hexadecimal output of the read data.)

Finally, write the file *main_2.cpp* with the instantiation/connection of the modules and the simulation control.

If you have completed the model you should generate the simulation program by entering ***make depend*** and then ***make***. If everything went fine the executable program *sim_2.x* should be available in the directory *ex_3*.

Like in the previous experiment, check the messages generated during simulation and verify the correct functionality of the FIFO.

Please note that the scenario used in this exercise is different from that in the previous one. Here not only single words are read or written, but an arbitrary amount of data (in the range between 1 and 16 Bytes). Therefore, the models cannot directly be compared. However, it should become clear that realizing this functionality would lead to significantly more complex models when signal-based communication would be used.

A note on writing header files

If you look at existing header files (*.h files) such as consum_2.h in this exercise, you'll always see this pattern:

```
#ifndef FILENAME_H
#define FILENAME_H
// the code
#endif
```

This pattern is called "include guard". It prevents the header file from being included more than once, which leads to compiler errors such as "redefinition of 'class fifo_if" ...".
It does not matter which define you exactly choose as long as it is unique. Usually the header file name in some form is used. For more information on include guards please look it up on the Internet!

Exercise 4 Communication using TLM 2.0

In this experiment we want to apply also communication on abstract level between modules, however using a method that is compliant with the TLM 2.0 standard. We rely on approximately timed modeling style, which uses explicit timing annotation to synchronize processes within the communicating modules and use for this purpose the variant with 2 phases of the TLM 2.0 base protocol.

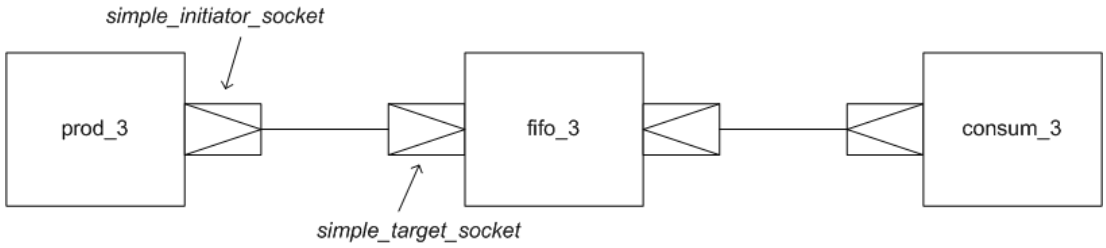
The components of the system model interact via so called transactions, which correspond to the execution of a function. But in this case, the declaration of the used functions is not any more left free to the designer as it was in experiment 3. Instead, the pre-declared functions ***nb_transport_fw*** and ***nb_transport_bw*** have to be used as shown in the third introductory presentation. The function ***nb_transport_fw*** is called by the initiator of the transaction via a so called initiator socket; ***nb_transport_bw*** is called by the target via the target socket. The implementations of these functions are contained in the other module respectively.
The arguments of both functions are:

- a *tlm_generic_payload* instance, which actually represents the transaction,
- a *tlm_phase* value to specify the phase of the TLM base protocol, and
- an *sc_time* value, which is used to annotate the timing of the transaction.

All these parameters are manipulated during the transaction. Therefore the functions are called by reference and not by value.

The sockets that are actually used in this lab are the simple sockets from the TLM utils, an add-on to the TLM 2.0 standard, which make TLM modeling easier than the sockets from the more general TLM 2.0 interoperability layer.

The following figure shows the configuration to be simulated.



In the producer as well as in the consumer a *simple_initiator_socket* has to be included as they enable accessing the FIFO, which purely acts as a slave. The FIFO will thus have two instances of *simple_target_socket*, which will then be bound to the initiator sockets of producer and consumer, respectively (see figure above). The function *nb_transport_fw* will then be called from within the producer to write data into the FFO, and from within the consumer to read data from the FIFO.
According to the mechanisms for the two-phase protocol, the FIFO itself calls the function *nb_transport_bw* (on the backward path). Thus, this function has to be implemented in both the producer and consumer.

Please note that the scenario used in this exercise is identical to the scenario of exercise 3, where an arbitrary amount of data is written/read to/from the FIFO (not only a single word as in exercise 2).

To accomplish this experiment, please change into the directory *ex_4*. There, the producer (*prod_3.h* and *prod_3.cpp*) and the FIFO (*fifo_3.h* and *fifo_3.cpp*) are given in complete form. First have a look at the source code of these models and compare their interaction with the sequence shown on slide 56 of the introductory presentation.

The FIFO provides the implementation of the function *nb_transport_fw*, which is registered with both instances of *simple_target_socket* in the constructor of the FIFO. In the implementation of *nb_transport_fw* the operations read and write are differentiated via the *tlm_command* component of the generic payload. The payload is then put into the appropriate payload event queue. Before the function returns TLM_UPDATED according to the base protocol, the following things are done:

- The delay time is increased by the time needed for a data transfer over a 32 bit wide bus with clock period 50ns and
- The TLM phase is updated according to the base protocol.

The actual read and write are performed in the processes *do_read* and *do_write* after the PEQ have notified them. After making the appropriate modifications in the generic payload instance that belongs to the respective transaction, the *nb_transport_bw* function is called via the corresponding *simple_target_socket*, which concludes the transaction.

Now, complete the simulation model by writing the module declaration of the consumer (new file *consum_3.h* has to be generated) and the missing implementations in the file *consum_3.cpp*. Further on, the top level file *main_3.cpp* is missing as well (no generation of waveforms).

Please note the following hints:

- Model the generation of transactions from the consumer in the same way as before by using two processes, one for triggering the other by notifying an appropriate event. The implementation of the trigger process is already given in *consum_3.cpp*.
- When writing the implementation of the process that executes the read from the FIFO, be sure
  - … to initialize the transaction object each time before *nb_transport_fw* is called.
  - … that the *wait* command, which covers the duration of the respective transaction phase is executed after the *nb_transport_fw* call. (The *sc_time* variable, which had to be initialized to SC_ZERO_TIME before the call, has been updated to the appropriate value.)
  - … not to start a new transaction before the previous transaction has finished its 2nd phase. Use an event that notifies the end of phase 2 from the *nb_transport_bw* function (which is called in the second phase) to the process that executes the read.
- Please write the model in a way that the duration of the simulation can be determined as a command line parameter of the simulation program (number of ns to be simulated). If no value is given the simulation should run for 3500 ns.

After completing the TLM SystemC model you should generate the simulation model by entering ***make depend*** and then ***make***. If everything went fine the executable program *sim_3.x* should be available in the directory *ex_4*.

To run the simulation of the TLM model enter

./sim_3.x

(if you want execute the simulation for a different time period than the default value of 3500 ns, start the simulation by ***./sim_3.x sim_dur***).

All output of this simulation is sent to the terminal window (waveforms are not generated). For each read or write operation the consumer, producer and the FIFO output information on the terminal window concerning the request and also the respective status of the FIFO. Based on this output, verify the correct behavior of the communication between the modules.

This concludes the introduction of SystemC and the different ways of modeling communication in SystemC. You should now have a first insight into some important aspects of modeling in SystemC and TLM modeling. In the final experiments you will apply SystemC TLM modeling for the exploration of a system on chip architecture from the area of network processing.

Section II: Exploration of a Network Processor Architecture

In the experiments of section II a network processor (NPU) architecture will be explored, which is modeled as a SystemC TLM 2.0 model using the approximately timed coding style with 2 phases as introduced in the previous exercise.

In the following, first a short overview of the application and the considered setup is given. Then in the associated exercises the appropriate TLM SystemC model is built step by step. Finally, some architecture alternatives are explored, ranging from modification of some operation frequencies till the introduction of an accelerator module.

Overview of the network processing application

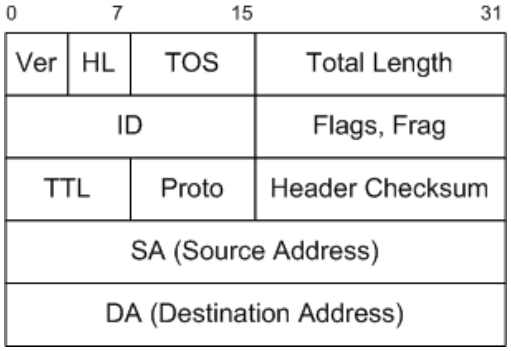
The NPU acts as an internet router with four Ethernet ports and performs simple IP forwarding (IPv4) as its application. I.e. it has to determine for each incoming packet on which of its four ports it has to be sent out in order to reach the final destination, which is identified by the target IP address in the IP Header.

In order to cope with the situation that packets arriving concurrently on different inputs have to be sent out on the same output port, it is necessary to temporally buffer packets until they can be transmitted. Therefore, packets are first stored in the packet buffer (system memory) before they are processed and are then queued (per output port). On the output side, packets are read one after the other from memory and are finally transmitted on the appropriate output port.

In order to make the packets accessible after they have been stored in memory, a descriptor is generated for each packet, which contains the address where the associated packet resides in memory. In our simplified router, packets are stored contiguously in a single 2 k byte buffer, which is sufficient also for maximum sized Ethernet frames.

For IP forwarding only header processing is required. The payload of the IP packet, i.e. headers of higher protocol layers and the actual packet data, is not accessed during processing.

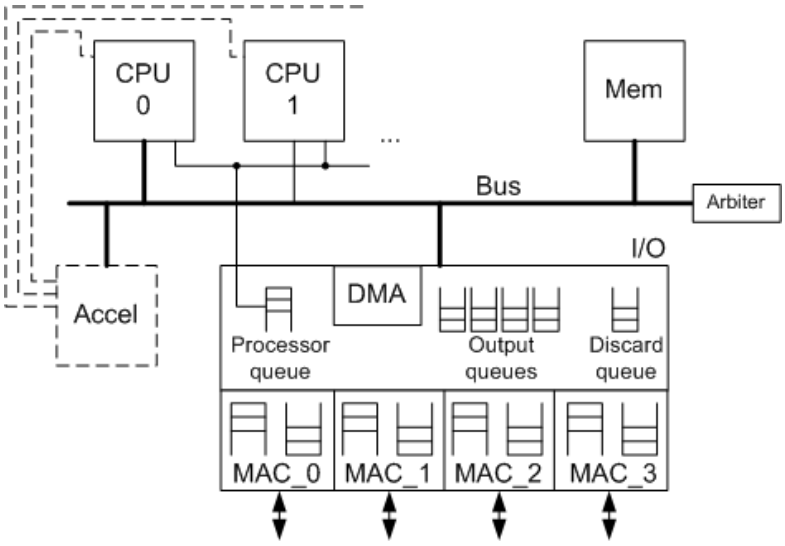
The header of an IP packet (IPv4) consists of 20 bytes and has the following fields:



- Ver: Version des IP Protokolls (4 Bits)
- HL: Länge des IP Headers (4 Bits)
- TOS: Type of Service (1 Byte)
- Total Length: Länge des IP Pakets (2 Bytes)
- ID: Identifier des Pakets (2 Bytes)
- Flags, Frag: Flags und Fragment Offset (2 Bytes)
- TTL: Time to Live (1 Byte)
- Prot: Transport Layer Protokoll (1 Byte)
- Checksum: IP Header Checksumme (2 Bytes)
- SA: Quell-IP Adresse (4 Bytes)
- DA: Ziel-IP Adresse (4 Bytes)

The NPU is based on a system bus that connects the different sub-modules of the architecture. The CPUs are bus masters, the memory and an optional accelerator are slaves on the bus. The I/O module acts both as a master, but also has a slave attachment to the bus.

The I/O module establishes the connection to the four Ethernet ports (MAC, medium access controller), which contain small FIFOs for receiving and transmitting Ethernet frames. The DMA (direct memory access) directly transfers packets between I/O module and the memory without intervention of the processors (therefore the bus master functionality of the IO module).
In receive direction, packets are read from the receive FIFO of the MACs and stored in the memory by the DMA. For each packet a descriptor containing the memory address of the packet is generated and stored in the processor queue. Thus, this queue holds the references to all packets that are waiting to be processed. The availability of descriptors in this queue is indicated by an interrupt line which is connected to all processor cores.
In transmit direction, packet descriptors are read in round robin fashion from the four output queues. The associated packets are then fetched from the memory and written into the transmit FIFO of the corresponding MAC, from where they are output on the physical interface.
The IO module also administers the packet memory, i.e. it assigns addresses where arriving packets have to be stored and frees the memory positions after packet transmission for later reuse. In case a packet has to be discarded (e.g. because of an incorrect header checksum) the corresponding memory position has to be freed as well. For this purpose the I/O module contains a discard queue, where descriptors of packets to be discarded have to be written to.
If the processor queue indicates available packets and a processor is idle it will try to read a descriptor from the processor queue in the I/O module. In case more than one processor tries to do this at the same time, the bus arbiter will grant bus access only to them one after the other. If more processors than available descriptors tried to read from the processor queue the transactions issued from the excess processor(s) will be unsuccessful.



When a CPU has successfully read a packet descriptor it will use the contained packet address to fetch the packet header (20 bytes) from memory. If the the packet header was successful read the following processing steps have to be executed:

1. Verify the IP header by recalculating the checksum. If it is incorrect, discard the packet by writing the packet descriptor to the discard queue in the I/O module. Then wait for a new interrupt from the processor queue. Otherwise go on with step 2.

2. Identify the next-hop, i.e. the output port for the IP packet based on its destination address. For this purpose the next-hop lookup table of the NPU has to be searched. It contains a list of IP address prefixes with a port number associated to each. For the destination IP address the entry with the longest prefix that matches the IP destination address has to be found, which in turn determines the output port of the packet.
3. Decrement the TTL field by 1. If TTL results in 0, then discard the packet. (This mechanism prevents packets from wandering infinitely around in the network. In reality an ICMP error message would have to be created in this case, which is neglected in the lab.)
4. Update the header checksum (as the header was modified).
5. Write back the modified packet header to the memory.
6. Write the packet descriptor into the queue corresponding to the output port.

The following table gives the address map of the NPU and the names of the constants (of type *soc_address_t*) to be used in the SystemC model:

Module	**Name**	**Value**	**Remark**
Memory	MEMORY_BASE_ADDRESS	0x00000000	
I/O module	PROCESSOR_QUEUE_ADDRESS	0x10000000	read only
	DISCARD_QUEUE_ADDRESS	0x10000000	write only
	OUTPUT_0_ADDRESS	0x20000000	write only
	OUTPUT_1_ADDRESS	0x30000000	write only
	OUTPUT_2_ADDRESS	0x40000000	write only
	OUTPUT_3_ADDRESS	0x50000000	write only
Accelerator	ACCELERATOR_ADDRESS	0x60000000	

Please note that the address of the processor queue is identical to the discard queue. However, the I/O module internally differentiates what functionality is accessed as these queues are only read (processor queue) or only written (discard queue).

Modeling the NPU architecture

The parts of the NPU architecture model that capture reception and storage of packets as well as their retrieval from memory and transmission on the output ports are completely given.

The associated functionalities as described above are contained in the given components. They comprise four MAC interfaces with DMA, a bus-based system interconnect and a memory module. Each MAC module reads in recorded packet data from a *pcap* file and sends it as stimulus for the NPU simulation.

The described flow of activity is already completely modeled and the involved modules are given in the subdirectory *npu_common*, which you have copied when you did the lab initialization. You should not have to modify these parts of the simulation model. However, you are invited to have a look at these models.

Those parts of the model that perform the packet processing have to be modeled by you one after the other in the subsequent exercises.

All given data structures and functions required to generate the SystemC simulation model for the rest of the NPU architecture are described in the manual below when needed in the different exercises.

Exercise 5 Packet Loopback

The first task for you in this experiment is to extend the given SystemC model comprising the I/O module, the system bus and the memory by a CPU, which – in this initial version – does not do any packet processing yet, but only loops the arriving packets back to the MAC ports (loopback). In this way you can verify that the packets correctly traverse the system. It should be confirmed that the CPU has access to the packet data on which the processing has to be modeled in the subsequent exercise.

For making the proper extensions to the given model, please switch to the directory *ex_5*. There, the header file *Cpu.h* for the CPU module is already given. Load this file into the editor to see the given declarations.

The corresponding implementation file *Cpu.cpp* is given only in fragments. Your first task is to complete the SC_THREAD process *processor_thread*, which should read a packet descriptor from the processor queue and write it randomly to one of the four output queues (all queues are in the I/O module; see table with addresses above).

When writing the code for *processor_thread* consider the following hints:

- First, make sure that there are actually packets available. Therefore, check if the input *packetReceived_interrupt* is already true. If not, the process has to wait until this is the case.
  For this purpose you may use the method ***value_changed_event()***, which will notify when the signal or port, to which it is applied, changes its value. I.e. a process whose execution stopped at the statement *wait(signal_x.value_changed_event())*, will resume execution as soon as *signal_x* changes its value.
- To read the packet descriptor *m_packet_descriptor* setup the transaction *payload* (declared in *Cpu.h*) appropriately
  As the data pointer in the transaction (*payload*) is an *unsigned char* * you have to cast the address of *m_packet_descriptor* in the function call *set_data_ptr* (see slide 53 of the introductory presentation) to an *unsigned char* *. The data length of a packet descriptor, which has to be set via *set_data_length*, can be determined via the function *sizeof(m_packet_descriptor)*. Also take care to initialize the phase and delay values appropriately, which have to be passed in the *nb_transport_fw* function call.
  Then, call the *nb_transport_fw* function of the CPU initiator socket
- After the function call, verify that
  - …the transaction was successfully carried out
    (i.e. `payload.get_response_status()` has to return `TLM_OK_RESPONSE` or – alternatively –
    `payload.is_response_error()` has to evaluate to *false*. **Only then you may go on with the next steps. Otherwise, the transaction call has to be repeated**. If you do not consider this, you will get errors in the subsequent exercises with architecture configurations containing more than one CPU)
  - … the correct *tlm_sync_enum* value has been returned and that the required phase transition at the end of the 1st phase has been made.
- Wait for the appropriate time resulting from the function call.
- Then wait until the 2nd phase of the transaction is finished. This should be notified via the event *transactionFinished_event* (occurs within the implementation of the *nb_transport_bw* function).

- Finally, send the read packet descriptor to a randomly selected output queue. For this, execute a further appropriate transaction (i.e. set up the transaction, initialize phase and delay, call function, verify …).

In the next step write the implementation of the *nb_transport_bw* function, which will be called from the bus in the CPU, actually triggered by a slave in reaction to the first transaction phase issued by the CPU. This makes up the second phase and completes the transaction.

Note the following hints:

- First, check whether the phase is set appropriately when *nb_transport_bw* is called. If this is not the case, output a meaningful error message.
- The bus transfer should take one bus cycle (CLK_CYCLE_BUS). Set the delay appropriately.
- Notify the *transactionFinished_event* after this delay to inform the process *processor_thread* about the end of the transaction.
- Set the phase appropriately and return the required *tlm_sync_enum* value to finish the function call.

If logging for the CPU is switched on (i.e. if *do_logging & LOG_CPU* is non-zero) output indicative logging messages in the implementation of the *nb_transport_bw* function and when calling the function *nb_transport_fw* in the *processor_thread* process,
(The global variable *do_logging* can be set appropriately in *main.cpp* to enable logging messages in different parts of the model. See *npu_common/globaldefs.h* for the *LOG_** values.)

After completion of the code for the CPU with the loopback function, compile the simulation program by entering ***make depend*** and then ***make***. Run the generated simulation program by entering ./***loopback.x***.

Verify the number of received and sent packets. The number of simulated packets is set in *main.cpp* to 20 (global variable *unsigned int MAX_PACKETS*). The simulation is immediately stopped upon reception of the specified number of packets. As a consequence, at that point in time there are still packets stored in either the MAC FIFOs or in the system memory, which have not yet been sent out. This can be recognized in a difference between the number of received and sent packets.

As several transactions have to be called for processing a complete packet (which is to be modeled in the next exercise), it is tedious to repetitively write the code for executing the first part of a transaction (i.e. for setting up the payload, the phase and the initial delay value, for calling *nb_transport_fw* etc.). Therefore, it is advisable to write a function for the associated commands. For this, the function prototype ***startTransaction*** is already declared in *Cpu.h*. Write the implementation of this function in *Cpu.cpp* and use it for the loopback functionality in your current implementation of the *processor_thread* process.

Finally verify the modified code and check that the simulation result is the same as before introducing the function *startTransaction*.

Exercise 6 Full CPU-based packet processing

In this exercise you should complement the CPU model with the functionality of the actual packet processing as described in the introduction of section II of the lab manual.
As simple IP forwarding requires the processing of the IP header, only this part of the IP packet has to be read from the memory before processing can start.

For each of the processing steps an appropriate function (member of the CPU module) is given that works on the relevant fields of the packet header. The function prototypes are declared in the given file *ex_6/Cpu.h*, which is extended compared to the version from experiment 5. The implementations of the member functions are given in the file *npu_common/Cpu_proc.cpp*.

As a first step copy the file *Cpu.cpp* from your *ex_5* directory to *ex_6* (execute the command ***cp ex_5/Cpu.cpp ex_6*** in your SystemC lab folder).
Change now into the directory *ex_6* and have a look into the header file *Cpu.h* for the function prototypes (and if you are interested in their implementation in *Cpu_proc.cpp*).

Now modify the *processor_thread* to implement the packet processing as described in the introduction of section II.

After reading the packet descriptor (as in experiment 5) you need the memory address of the associated packet for loading the header. This address is a member of packet descriptor with the following declaration (contained in *npu_common/packet_descriptor.h*):

```
struct packet_descriptor{
     soc_address_t baseAddress;
     unsigned int size;
}
```

All processing steps are performed on a packet header as can be seen from the given function prototypes, even if the parameters are references to IP packets (see table below). Therefore, it is sufficient to read only the amount of data from memory that contains the header of the packet to be processed (including those values that are stored in memory before the header) into the member variable *IpPacket m_packet_header*. All subsequent processing steps after the read are executed on this member variable.

Please note the following hints for reading the header:
The actual packet data as stored in the memory are prepended with a time stamp of the packet arrival time (type *sc_time*) and the packet size (type *uint64_t*). Therefore, these values have to be read in addition to the 20 bytes of a minimum IP header. The data amount to be read is thus

sizeof(sc_time) + sizeof(uint64_t) + IpPacket::MINIMAL_IP_HEADER_LENGTH

(See declaration in *npu_common/IpPacket.h*).

As the ***startTransaction*** function requires an *unsigned char ** as address parameter, the address of the member variable *IpPacket m_packet_header* has to be cast appropriately (in the same way as when reading the packet descriptor).

For processing the IP packet header you can use the following functions (declared in *Cpu.h*, implemented in *npu_commn/Cpu_proc.cpp*):

Function	Description
`bool verifyHeaderIntegrity(const IpPacket &)`	Verifies Header length, TTL (to be not zero), and the correctness of the heaer checksum. Returns *true* if header is correct, else *false*
`unsigned int makeNHLookup (const IpPacket &)`	Performs longest prefix match for IP destination address contained in IP packet. Returns port ID in the range between 0 and 3
`void decrementTTL(IpPacket &)`	Decrements TTL value in header of IP packet by 1.
`void updateChecksum(IpPacket &)`	Updates the Checksum in header of IP packet.

After processing, the modified header has to be written back to the packet memory.On the other hand, the packet descriptor has to be written to the output queue in the I/O module corresponding to the output port that has been identified by the next-hop lookup.

Compile your modified code by entering ***make depend*** and then ***make***, and verify that the program ***processing_cpu.x*** has been generated successfully.

Compare the output of the simulation program *processing_cpu.x* with *loopback.x* from exercise 5. The distribution of the packets to the output ports should now be different as the packets are not randomly sent to one of the output ports but according to the result of the next-hop lookup.

Exercise 7 Timing Annotation and Evaluation

In this exercise the simulation model is to be extended by timing annotations of the different processing steps within the CPU. For this purpose a *wait()* statement with the appropriate duration of the processing has to be inserted after the corresponding call of the processing function (see table in the description exercise 6).

These delays are to be specified as multiples of CPU clock cycles. The variables to be used (*unsigned int*) are already declared globally in *npu_common/globaldefs.h* and are defined in *npu_common/globals.cpp*. The values are:

Processing delay (name of variable)	**Number of CPU clock cycles**
CPU_VERIFY_HEADER_CYCLES	50
CPU_DECREMENT_TTL_CYCLES	5
CPU_UPDATE_CHECKSUM_CYCLES	30
CPU_IP_LOOKUP_CYCLES	350

The CPU clock cycle is also declared as a global variable *sc_time CLK_CYCLE_CPU* in *npu_common/globaldefs.h* and is set to 10 ns (i.e. frequency is 100 MHz) as default value in *npu_common/globals.cpp*.
However, as a feature of the new *main.cpp* file in *ex_7*, this value and some other parameters of the model as described below can be set on the command line of the simulation program.

As a first step copy the file *Cpu.cpp* from your *ex_6* directory to *ex_7* (e.g. when being in *ex_7* execute the command ***cp ../ex_6/Cpu.cpp .***).
Change into the ex_7 directory and add the appropriate WAIT commands as described above.

Then, measurement of the times spent in the CPU for processing and also for communication (i.e. for calling the transactions via the bus) should be added to your model. For this purpose, several things have been prepared in the updated *Cpu.h* file in *ex_7*:

- Two *sc_time* variables, *total_processing_time* and *total_transfer_time* are declared in *Cpu.h* as members of the CPU module. There the time spent for processing and communication should be summed up, which can finally be put into relation to the complete simulation time to get load values.
- To ease these time measurements, two preprocessor macros
  - MEASURE_PROCESSING_TIME(code) and
  - MEASURE_TRANSFER_TIME(code)

  are defined in *Cpu.h* as well, which allow measuring the respective times quite easily. Simply wrap the brackets of the appropriate macro over the relevant instructions *code* in *Cpu.cpp*, whose execution should be measured. During simulation this will automatically update the corresponding *sc_time* variables corresponding to the time spent in the different *code* sections.
- The public member function *output_load()* is added as well. It is called at the end of the simulation from within the *sc_main* function and will then print the accounted CPU load values.

After completing the code compile the simulation program by invoking ***make depend*** and then ***make***. This should generate the executable *processing_cpu2.x*.

To enable an efficient exploration of the NPU architecture with different system parameters without the need to recompile the program each time a value is changed, the *sc_main* function (contained in the updated *main.cpp*) now can evaluate command line arguments.

The following parameters given on the command line of *processing_cpu2.x* can modify the behavior of the simulation model:

Command line switch	**Meaning**
-c <value>	Set the clock period of the CPU(s), value is given as multiple ns, default is 10 [ns]
-b <value>	Set the clock period of the system bus, value is given as multiple ns, default is 20 [ns]
-n <value>	Set the number of CPUs; default is 1
-p <value>	Set the number of packets to be simulated, , default is 100
-v <value>	Set *do_logging* to a bitmask, where each position enables logging for different parts of the model: Bus 0x1 Memory 0x2 IO module 0x4 CPU 0x8 Value has to be specified in hex notation. Default value is 0x0, i.e. logging is disabled.
-h	Print out help information

As a first step before systematic exploration, "play around" with these parameters to verify that the parameters actually change the model behavior.

In the experiment 8, an accelerator module will first be included in the NPU architecture and then the overall exploration will be carried out in the 9th and final exercise.

Exercise 8 Implementation of an accelerator module

Before writing the SystemC code for this exercise, be sure to read the complete description of this exercise.

In this exercise, the next-hop lookup as the most time consuming function in software (see table in previous exercise) should be accelerated and thus be realized as a dedicated hardware module. The accelerator is a slave attached to the bus and can be accessed by writing and reading to/from the appropriate address given in the address map table (see overview of lab part II). It performs the IP lookup significantly faster than the CPU.

With the introduction of the accelerator *processing_thread* within the CPU has to be adapted. Instead of calling the local function *makeNHLookup*, the lookup functionality should be called via the bus from the accelerator.

Modeling of the accelerator
First, you have to complete the model of the accelerator, which is partially given in the directory *ex_8_9*. Thereby, it should be assumed that the algorithm realized in the accelerator may be changed and that the exact duration of a lookup is not yet fixed at the time of design space exploration. Therefore the accelerator should be modeled with configurable processing delay. It should be possible to specify the actual delay value on instantiation of the accelerator.
For this purpose, the member variable ACC_IP_LOOKUP_CYCLES is declared in the accelerator module, which should be set to the desired value via a second parameter in the constructor (in addition to the module name).

Due to the additional constructor parameter, it is not possible any more to use the macro SC_CTOR. Instead a conventional C++ constructor has to be written. Moreover, the macro SC_HAS_PROCESS has to be called as well (see introductory slides).
As a further difference to the previous modules, the header file *Accelerator.h* should only contain the declaration of the constructor. The implementation of the constructor occurs in the implementation file *Accelerator.cpp*, as it is the case for the other member functions and processes.
Accelerator.h is completely given with the exception of the constructor declaration and the SC_HAS_PROCESS call. Complement the missing instructions; especially take care of the appropriate parameters of the constructor.
Then in *Accelerator.cpp* (marked section in the source code), add the still missing parameter list of the constructor and the initialization of ACC_IP_LOOKUP_CYCLES.

Modification of the CPU
As a next step you should extend the *processing_thread* within the CPU in such a way that also the lookup function implemented in the accelerator can be called via a bus transaction.

The code of the CPU implementation should cover both variants, i.e. "CPU stand alone" and "CPU with accelerator". Therefore, the code sections, which are mutually exclusive, have to be separated via `if(use_accelerator){ ... } else{ ... }`, so that both variants can be simulated, depending whether an accelerator is part of the system or not.

For this purpose the now global variable *bool use_accelerator* is introduced, which will be set appropriately in *sc_main()* (the source code for parsing command line arguments). This variable will be set true if an accelerator is present. In this case a corresponding instance is generated and connected as a slave to the bus. Further, it is connected to each CPU with an interrupt line. As described above, the variable should also be used in *processing_thread* to determine how the lookup is carried out.

For interacting with the accelerator a new *struct LookupRequest* is declared in *npu_cmmon/globaldefs.h*, which has the two members *unsigned int destAddress* and *unsigned int processorId*. (Have a look at the code in *globaldefs.h* for the declaration.)
Therefore, a new member *m_lookup_request* is declared in the module declaration of the CPU, which should be used to write the input required for the lookup to the accelerator.

- *destAddress* should contain the IP destination address, for which the lookup has to be done.
  It can be extracted from an IP packet by applying the method *unsigned int getDestAddress()* (see *npu_common/IpPacket.h* for the declaration),
- *processorId* is required within the accelerator to be able to return the result to the correct CPU, as the accelerator may queue requests from different CPUs. Therefore, before writing the lookup request to the accelerator, *processorId* should be set to the member *m_Id* of the CPU, which is a unique identifier for each CPU instance within the system.

Further on, the CPU has a new interrupt input *sc_in<bool> lookupReady_interrupt*, which is connected to the accelerator and which is set to *high* by the accelerator when it has finished a lookup request for the corresponding CPU and the CPU can read back the result..

The interaction should be modeled in the following way:

1. After reading the packet header from the memory, set up the lookup request (set payload appropriately, especially cast the address of *m_lookup_request* correctly) and write it to the accelerator (see address in table above).
2. Then do the remaining processing within the CPU and store the packet header in memory as in the pure CPU based processing.
3. Check whether the *lookupReady_interrupt* is already indicated (value true), if not wait until this happens.
4. Read the lookup result back. It will be delivered from the accelerator as an *unsigned int* within the payload.
5. Use the result for determining on which output queue to write the packet descriptor.

Copy your last file *Cpu.cpp* from your *ex_7* directory to *ex_8_9* (execute the command ***cp ../ex_7/Cpu.cpp .*** if you are in the directory *ex_8_9*) and make the required modifications in the copied file. A new version of *Cpu.h* is already available in *ex_8_9*.
Then modify the code *processor_thread* to support also the accelerator if it is part of the architecture.

Changes in *sc_main*
Finally, complement the instantiation of the accelerator in *main.cpp* (marked area after `if(use_accelerator)`) and thereby set the number of required clock cycles for the accelerator to 40.

An additional command line parameter "–a" can be used with the new *sc_main()* function. It allows to specify the presence and the operation speed of the accelerator:

Command line switch	**Meaning**
-a <value>	Set the clock period of the accelerator, value is given as multiple ns. If this switch is omitted, no accelerator is instantiated.

After making the appropriate changes in the source code, compile the simulation program as before by invoking ***make depend*** and then ***make***. This should generate the executable *processing_acc.x*.

Verify that your executable works. The exploration of the NPU architecture will be done in the final exercise.

Exercise 9 Exploration of NPU-Architecture

In this final exercise the SystemC model as generated in the previous exercises should be used to simulate and evaluate different architecture alternatives of the NPU. The variants differ in the number of CPUs, the CPU frequency, the bus clock frequency and the presence of the accelerator for doing the lookup.

The following architecture alternatives have to be simulated for the CPU frequencies of 100, 200, 250, 333, and 500 MHz (each time for 1000 packets):

Architecture	Characteristics
1	1 CPU, Bus 50 MHz, no accelerator
2	2 CPUs, Bus 50 MHz, no accelerator
3	3 CPUs, Bus 50 MHz, no accelerator
4	1 CPU, Bus 100 MHz, no accelerator
5	1 CPU, Bus 50 MHz, accelerator with 100 MHz

In the directory *ex_8_9* you find the spreadsheet *SysC_ex_9.ods*, which is prepared for you to enter the results you get from the simulation runs. Load this spreadsheet by entering the command ***soffice -calc SysC_ex_9.ods*** into LibreOffice Calc.

Run the simulations with the given settings and enter the results into the prepared tables. The switches to be used on the command line are described in the previous exercises, or call the simulation program with the help option (i.e. *processing_acc.x -h*) to get a list of options.

To ease interpretation, some graphs are prepared in the spreadsheet, which depict the following metrics of the NPU architecture as a function of the CPU frequency:

- packet rate in kpps (k packets per second)
- mean packet latency in ns
- mean CPU load in %
- mean bus utilization in %

Think about the following questions, which should give you some hints on the expected simulation results and thus also whether your model works as expected:

- Why is the bus load of the architecture with the accelerator the highest of all variants?
- Why performs the architecture 2 with 2 CPUs running at 200 MHz better than architecture 1 with a CPU operated at 500 MHz?
- Why are the bus loads of variants A2, A3 and A5 quite soon saturated?
- … (In general, think about the plausibility of your simulation results.)

Finally, save the completed spreadsheet in your directory.

This makes up the end of the SystemC lab. By completing the exercises of the first section of the lab you should have gained insights into the basic concepts of the system level language SystemC and the modeling of communication on different levels of abstraction. In the second section you have used these principles to model and explore different variants of an NPU architecture as an application example. In total, the experiences gained in the lab should enable you to apply SystemC for own architecture exploration tasks.