The goal of this practical exercise is to understand how the SystemC works. In this exercise, we use the Open SystemC Initiative (OSCI) version of SystemC, see http://www.systemc.org.

1 SystemC Simulation Basics

To model basic systems and set-up a basic simulation in SystemC, a few basic SystemC API functions are needed. Here is a list of the fundamental SystemC specific APIs:

- **SC_MODULE**: SystemC modules are declared with macro `SC_MODULE`. It provides an easy and readable way to describe a module. The equivalent C++ code is:

```cpp
1 class module_class_name : public sc_module {
2     // Module body
3 }
```

- **SCCTOR**: The macro `SCCTOR` is the constructor for a `SC_MODULE`. It does the following:
  - Create hierarchy
  - Register functions as processes with the simulation kernel
  - Declare sensitivity lists for processes

The equivalent C++ code are:

```cpp
1 module_class_name() : sc_module()
2     // constructor body
3 }
```

- **SC_THREAD**: Also known as thread processes, `SC_THREADS` are module methods with their own thread of execution, in the sense that they execute concurrently from the SystemC kernel's point of view.

- **sc_main**: Simulation instructions are usually located inside of a function called `sc_main`. The `sc_main` function is equivalent to the more conventional `main` function of C++.

This `sc_main` function will execute simulation specific commands such as setting the simulator’s resolution, channels to be traced, top level instances, the trigger of the start of the simulation, i.e. by calling the function `sc_start`, and more.

- **sc_start**: The simulation is actually invoked by the `sc_start` function.
1.1 Administration

The disk space allocated to you (only applicable if you use SystemC exercise logins) is 250MB. You are responsible to managing your disk space on your own. Please make sure that you remove all un-necessary files after you logout, and if required, backup to another account. To see how much disk space your account is using, execute the following command on the command line:

>quota -s

1.2 The Shell

All instructions below assume that you will be working in the bash shell. To see which shell you are running, execute on the command line:

>echo $SHELL

If you are in the bash shell, the output should be:

>/bin/bash

Change to the bash shell if required, or modify the following instructions suitably.

1.3 Download the Exercise Package

(a) Go to your home folder

>cd ~

(b) Download “SystemC_Exercise.tar.bz2” from the class website http://www.tik.ee.ethz.ch/education/lectures/hswcd/ to your home folder:

>wget http://www.tik.ee.ethz.ch/education/lectures/hswcd/exercises/SystemC_Exercise.tar.bz2

This is 7.1 MB file, please make sure that you have correctly downloaded it.

(c) Unzip the contents of the zipped folder:

>tar -jxvf SystemC_Exercise.tar.bz2. This will give you two subfolders inside SystemC_Exercise:
Exercises/ systemC-2.3.0-src/

1.4 Basics Structure of the Downloaded Package

The Directories

- SystemC-2.3.0-src directory contains the sources required to build your own SystemC v.2.3 installation. A simple build script is already provided to you: SystemC_Exercise/systemC-2.3.0-src/build-SystemC.sh. See more on building SystemC-2.3 in Section 1.5.

- Exercises contains the base code which you will start with. This is basically solution 1.1 to your exercise. You are expected to browse through it, understand it and attempt to build it. Within the

The Makefile

The sample Makefile, included in the Exercises folder, contains instructions on building your code to an executable. Read about Makefiles online, if possible. Here is the explanation of what this makefile does (refer to Makefile):
Line 1 and 2 set the compiler to g++. We need a C++ compiler to build SystemC applications.

Line 4 sets the variable SYSTEMC_INC which indicates the location of the SystemC header files.

Line 5 sets the variable SYSTEMC_LIB which indicates the position of the precompiled SystemC library. Modify it suitably according to your machine’s architecture.

Lines 7 and 8 set up compiler flags. Look up the gcc manual to know what these mean. For instance, you use -g flag to enable debugging.

Lines 10, through 17 specify the actions to be taken when makefile is called with an argument. For example

```bash
>make clean
```

will remove all object files, trace files and binary from the directory. make clean and make sc_application will do the same thing. You may modify the makefile, if you want to.

### 1.5 Building SystemC

In this exercise we will be using SystemC version 2.3. A simple build script is also available for you to use at SystemCExercise/systemC-2.3.0-src/build-SystemC.sh. Remember to set executable permission to the script:

```bash
chmod +x ~/SystemCExercise/systemC-2.3.0-src/build-SystemC.sh.
```

Make sure that you correct any paths, if necessary.

Once you have confirmed that ~/SystemCExercise/systemC-2.3.0-src/build-SystemC.sh is correct, build SystemC-2.3.0:

```bash
>cd ~/SystemCExercise/systemC-2.3.0-src/
>./build-SystemC.sh
```

The build takes about two minutes. By default, the installation will be done in ~/SystemC-2.3.0-Installation folder.

### 1.6 Build your First SystemC Application

Fig. 1 shows an implementation example, i.e. a producer and a consumer communicating via a FIFO channel. Compile and execute the source code to check the result, by following the next steps:

(a) Build your simple application

```bash
>cd $HOME/SystemCExercise/Exercises/
>make clean all
```

(b) Run your simple application

```bash
>./sc_application
```

Note: This code can be found from the official SystemC source code distribution as well.
1.7 Simulation Monitoring

In order to examine the results of the simulation, the signals of the system under design can be traced and visualized. One of the typical monitoring format is the VCD (Value Change Dump) tracing file.

The SystemC simulation supports the VCD waveform tracing as well. To enable the VCD waveform tracing, mainly three steps need to be done: (1) open the VCD file, (2) select the signals to be traced, which will automatically write to the dumpfile after executing the simulation, (3) close the trace-file. In order to trace the FIFO example in the previous section, additional code need to be inserted in the

```
module fifo : public sc_channel, public write, read
// top (sc_module{name}) : sc_module(name), num_elements(0),
private:
    enum e { max = 10; }
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;
};
```

In the `write` function and in the `read` function as shown in Listings 1, 2, 3, and 4. Modify your source code, recompile it, and check the obtained waveform files.

Notes: The `gtkwave` waveform viewer can enable the graphical display of the VCD trace file by the command `gtkwave trace.vcd`. To show the resulted waveforms, in the graphical interface of `gtkwave`, from the tab `Search->Signal Search Tree`, select and insert the entire SystemC signal tree. If you cannot open the generated `trace.vcd` with `gtkwave`, try first to continue the exercise.

![Figure 1: Simple example of a producer and consumer communicating via a FIFO channel.](image-url)
### Listing 1: The new `sc_main` function.

```cpp
int sc_main (int argc, char *argv[]) {
    top top1("Top1");
    sc_trace_file *tf;  // New inserted */
    tf = sc_create_vcd_trace_file("trace");  // New inserted */
    sc_trace(tf, top1.fifo_inst->reading, "consumer_read");  // New inserted */
    sc_trace(tf, top1.fifo_inst->writing, "consumer_write");  // New inserted */
    sc_start();
    sc_close_vcd_trace_file(tf);  // New inserted */
    return 0;
}
```

### Listing 2: The new attributes of the `fifo` class.

```cpp
class fifo : public sc_channel, public write_if, public read_if {
    public:
        fifo(sc_module_name name) : sc_channel(name),
            num_elements(0), first(0) {}
        bool reading;  // New inserted */
        bool writing;  // New inserted */

    void write(char c) {
        if (num_elements == max)
            wait(read_event);
        writing = 1;  // New inserted */
        data[(first + num_elements) % max] = c;
        ++num_elements;
        writing = 0;  // New inserted */
        write_event.notify();
    }

    void read(char &c) {
        if (num_elements == 0)
            wait(write_event);
        reading = 1;  // New inserted */
        c = data[first];
        --num_elements;
        first = (first + 1) % max;
        reading = 0;  // New inserted */
        read_event.notify();
    }
};
```

### Listing 3: The new `write` method.

### Listing 4: The new `read` method.

### 2 Simulating Time

In the previous points of this exercise, only the untimed functional simulation was executed. However, the major usage of SystemC is for timed simulations, e.g. timed functional simulation. To introduce the notion of time into an untimed simulation, the `wait` function can be used to annotate the source code with a processing delay. One definition of the `wait` function is shown below:

```cpp
void wait(double v, sc_time_unit tu);
```

The caller process will be resumed after that the time given as an argument has been elapsed. The time to be executed is relative to the time at which function `wait` is called. For instance, `wait(1, SC_NS)` will introduce a delay of one nano-second from the moment that it is called during simulation.

#### 2.1 Simulation Time Tracking

To track the simulation time for a specific simulation spot, the `sc_time_stamp` method can be used. Modify the `consumer::main` function, as shown in Listing 5. Explain the obtained result.
```cpp
void main() {
    char c;
    cout << endl << endl;
    while (true) {
        in->read(c);
        cout << c << flush << "\n";
        if (in->num_available() == 1)
            cout << "<1>" << flush << "\n";
        if (in->num_available() == 9)
            cout << "<9>" << flush << "\n";
        cout << "time used: " << sc_time_stamp() << "\n"; /* New inserted */
    }
}
```

Listing 5: The new `consumer::main` method.

Solution: Reading and writing to the FIFO takes 0 time. Therefore, everything happens at time 0.
GTKWave and terminal output shows this behavior.

### 2.2 Simulation Time Advance

Change the simulation code, by inserting different “execution delays”. For each of the following points,
change the source code of the application (do one change at a time and keep the changes from the previous
steps), recompile the source code, check the resulted VCD waveforms, and compare them with the previous
ones. Explain the differences.

(a) Insert the code `wait(1, SC_NS);` between line 5 and line 6 of the functions `read` and `write` in
Listings 4 and 3 respectively.

Solution: FIFO is read and written concurrently. FIFO is never full. FIFO is empty only at beginning
and end. GTKWave plot is shown in Figure 2.

![Figure 2: 2.2a GTKWave plot](image)

(b) Increase the delay of the `wait` function in the `write` method from the previous point, from 1 nano-
second to 2 nano-seconds.

Solution: Read operation is faster compared to write operation. Therefore, FIFO becomes empty
several times during simulation and consumer has to wait for the producer to write element, before
reading. GTKWave plot is shown in Figure 3.

![Figure 3: 2.2b GTKWave plot](image)

(c) Increase the delay of the `wait` function in the `read` method from 1 nano-second to 2 nano-seconds,
while the delay of the `wait` function in the `write` method is set to 1 nano-second.
Solution: Write operation is faster compared to read operation. Therefore, FIFO becomes full several times during simulation and producer has to wait for the consumer to read element, before writing. GTKWave plot is shown in Figure 4.

(d) Change the size of the FIFO, e.g. set the FIFO size to 2 elements, instead of the actual size of 10.

Solution: FIFO becomes full more often compared to 2.2c due to smaller size. GTKWave plot is shown in Figure 5.

(e) Change the notification time, i.e. instead of immediate notification, use a timed notification like for instance notify(1, SC_NS).

Solution: Notification delay adds an extra wait time for reading/writing when fifo becomes empty/full. GTKWave plot along with explanations is shown in figure 6.

Figure 4: 2.2c GTKWave plot

Figure 5: 2.2d GTKWave plot

Figure 6: 2.2e full GTKWave plot along with zoomed version with exaplanations
3 Starting/Stopping the Simulation

In the SystemC standard, the start function is defined as follows:

```c
void sc_start()
void sc_start(const sc_time&)
void sc_start(double, sc_time_unit)
```

The sc_start semantics depends on the function arguments, as follows:

- When the function sc_start is called without any arguments, the scheduler will run until it reaches completion, unless otherwise interrupted.
- When the function sc_start is called with a zero-valued time argument, the scheduler will run for one delta cycle.
- When the function sc_start is called with a time value, the scheduler will execute up to and including the timed notification phase that advances simulation time to the end time (calculated by adding the time given as an argument to the simulation time when function sc_start is called).

Once started, the scheduler will run until one of the following situations occurs: the simulation reaches completion, the application calls the function sc_stop, or an exception occurs.

3.1 Total simulation time

Print the simulation time at the end of the simulation, see Listing 6. Explain the obtained result.

```c
int sc_main (int argc, char * argv[]) {
    top.top1("Top1");
    sc_start();
    cout << "time used: " << sc_time_stamp() << "\n"; /* New inserted */
    return 0;
}
```

Listing 6: The new sc_main method.

Solution: Time used=162ns. The simulator is started, and will run till it reaches the maximum time after which there is no change in signals.

3.2 Simulation for specified time

Execute the same simulation, by invoking the function sc_start with a different argument, i.e. sc_start(50, SC_NS). Check the resulting waveforms and explain the results.

Solution: Time used = 50 ns.