1 Introduction to SystemC

SystemC is a system written in C++ that is widely used to simulate architectures. It is a library of C++ classes, global functions, data types and a simulation kernel that can be used for creating simulators for hardware. By using C/C++ development tools and the SystemC library, executable models can be created in order to simulate, validate, and optimize the system being designed. An executable model is essentially a C++ program that exhibits the same behavior as the system when executed.

2 Getting Started

The SystemC library is already configured, installed and accessible on the lab machines in:

/home/rdoumal/systemc

You can copy the following directory which contains a framework for the lab course, which has a Makefile that is configured to use the above SystemC installation:

/home/rbakker3/aca2014/aca2014_assignment1_framework

However, if you want to configure and install SystemC yourself on your own machine, see the Appendix: Installing SystemC.

3 Using SystemC

A SystemC system consists of a set of one or more modules. Modules provide the ability to describe structure. Modules typically contain processes, ports, internal data, channels and possibly instances of other modules. All processes are conceptually concurrent and can be used to model functionality of the module. Ports are objects through which the module communicates with other modules. In C++ terms modules, ports and channels are classes from which objects are
created. Processes are member functions of a module that, when registered as a process to the simulation kernel, are executed every time an event is triggered.

Events are the basic synchronization objects. They are used to synchronize between processes. Processes are triggered or caused to run based on their sensitivity to events. That means that every time a module member function is registered as a process, the events on which it is “sensitive” are also defined.

Access to all SystemC classes and functions is provided in a single header file named “systemc.h”. This file includes other files, but the end user only needs to use this. In order to use the SystemC classes this file must be included and the application must be linked with the SystemC library.

4 Execution Semantics

SystemC is an event based simulator. Events occur at a given simulation time. Time starts at 0 and moves forward only. Time increments are based on the default time unit and the time resolution.

Every C/C++ program has a main() function. When the SystemC library is used the main() function is not used. Instead the function sc_main() is the entry point of the application. This is because the main() function is defined in the library so that when the program starts initialization of the simulation kernel and the structures this requires are performed, before execution of the application code. The main() defined into the library calls the sc_main() function after it has finished with the initialization process.

The sc_main() function is defined as follows:

\[
\text{int sc\_main(int argc, char* argv[])}
\]

which is the same as a main() function in common C++ programs. The values of the arguments are forwarded from the main() defined in the library.

In the sc_main() function the structural elements of the system are created and connected throughout the system hierarchy. This is facilitated by the C++ class object construction behavior. When a module comes into existence all sub-modules this contains are also created. After all modules have been created and connections have been established the simulation can start by calling the function sc_start() where the simulation kernel takes control and executes the processes of each module depending on the events occurring at every simulated cycle. In the first cycle all the processes are executed at least once unless otherwise stated with a call to the function dont\_initialize() when the process is registered.

5 The Memory Module

The following code creates a module which simulates a Random Access Memory. How the code works is explained below.
static const int MEM_SIZE = 512;

SC_MODULE(Memory) {

public:
    enum Function {
        FUNC_READ,
        FUNC_WRITE
    };

    enum RetCode {
        RET_READ_DONE,
        RET_WRITE_DONE,
    };

    sc_in<bool> Port_CLK;
    sc_in<Function> Port_Func;
    sc_in<int> Port_Addr;
    sc_out<RetCode> Port_Done;
    sc_inout_rv<32> Port_Data;

    SC_CTOR(Memory) {
        SC_THREAD(execute);
        sensitive << Port_CLK.pos();
        dont_initialize();
        m_data = new int[MEM_SIZE];
    }

    ~Memory() {
        delete[] m_data;
    }

private:
    int* m_data;

    void execute() {
        while (true) {
            wait(Port_Func.value_changed_event());

            Function f = Port_Func.read();
            int addr = Port_Addr.read();
            int data = 0;
            if (f == FUNC_WRITE) {
                data = Port_Data.read().to_int();
            }

            // Simulate Memory read/write delay
            wait(100);

            if (f == FUNC_READ) {
                Port_Data.write((addr < MEM_SIZE) ? m_data[addr] : 0);
                Port_Done.write(RET_READ_DONE);
                wait();
                Port_Data.write("Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz");
            } else {
                wait();
            }
        }
    }
}
as can be seen from the above code, a definition of a module in SystemC
is the same as declaration of a class in C++ and this is because it is exactly
that. The SC_MODULE statement is just a macro that provides a simple form
of module definition. The macro is internally defined as follows:

#define SC_MODULE(user_module_name) \
  struct user_module_name : ::sc_core::sc_module

As can be seen from the definition of the macro, every module is a sub-class of
the “sc_module” class. SystemC uses many macros like SC_MODULE in order
to simplify definitions and to be similar to SystemC terminology.

The first elements of the modules declaration are two enumeration types
that the module uses. The five lines that follow are the declarations of the
ports the module has in order to communicate with other modules. In order
to understand the concept of ports, first we must look briefly at interfaces and
channels.

In SystemC, interfaces define a set of member functions a channel that imple-
ments the interface must have. They only provide signatures of the functions and
not the implementation. In C++ terms they are classes with all their functions
being pure virtual functions. Channels define how the functions of an interface
are implemented. They are classes directly derived from the interface(s) they
implement and provide implementations for the functions of the interface(s).
SystemC provides a number of ready defined interfaces and channels which im-
plement them. If those do not provide the required functionality others can be
defined. The most common used interface is the sc_signal_inout_if and the
channel class that implements it is the sc_signal.

Finally, ports are objects through which a module can be connected into
one or more channels. Port objects are directly or indirectly derived from the
template class sc_port. This class is a template so that it can be customized
according to the interface the port can connect. That means that when a port
object is defined using this class, the interface (and consequently the channel) on
which it can connect must also be defined. The following is an example port dec-
laration that can connect to the sc_signal_inout_if interface mentioned earlier.

sc_port<sc_signal_inout<int>,1> port_name
In the above statement we can also see that the `sc_signal_inout_if` is also a template class. SystemC makes extreme use of template classes, because that mechanism gives the flexibility the class needs to be customized for a specific data-type.

In our example though, we have not used such declarations to define the ports of the module. This is because we have used what is called specialized ports. Specialized ports are classes derived from the `sc_port` class, which are customized with a particular (set of) interface(s). These classes also provide additional support for use with a channel or for easier use. The ports used in the example are: `sc_in`, `sc_out` and `sc_inout_rv`, which are ports specific for the `sc_signal_inout_if` mentioned earlier. Again here we see that those classes are also template classes. This gives the ability to define a port that can handle data of any C++ or SystemC data-type.

The code which starts with the statement `SCCTOR` is the code of the constructor of the module. The `SCCTOR` statement is also a macro used for the constructor of the class.

At the beginning of this document it had been mentioned that a module also has processes, which are member functions or threads registered as processes to the simulation kernel and executed by it when an event is triggered. The first line of the constructors code does exactly that. It registers to the simulation kernel that the Memory module has a process which is a thread, the code of which is the execute member function. The `SC_THREAD` is also a macro that makes the code more readable.

Processes have a list of events on which they are sensitive. If an event happens on a process sensitivity list, they get woken up. The second line of the constructors code `sensitive << Port_CLK.pos()` creates that list for the process registered by the previous `SC_THREAD` statement. It specifies that this process will execute on the event when the signal on the Port_CLK port goes positive (i.e. once per clock cycle). Ports trigger events owned by the channel they are connected to, when the value of the port itself changes or the value of another port also connected to the same channel changes.

In simple terms: when the Port_CLK port changes to positive the `execute` process will get woken up. The rest of the constructor code allocates storage for the memory.

The `execute` member function defines the code for the thread that will run for this module, as defined in the constructor. The thread continuously waits for the Port_Func to get written and then serves the request. It reads the address and data if its a memory read, then waits 100 cycles to simulate memory latency and finally writes the result back before waiting on the Port_Func again.

Note that a string of 32 Zs is written to the 32-bit bi-directional data port a cycle after we write the data to the port. This has to be done because bi-directional channels, which have multiple writers, attempt to resolve conflicts when multiple writers are writing values. Since we have now written a value, we need to reset our end of the channel in the next cycle to allow the other end to write values for the next request.
Task 1: Create the Memory Module

1. Enter the code of the example into a new C++ source code file. Do not forget to include the systemc.h file at the beginning of the file.

2. Create an sc_main() function in the same file with the following code:

```cpp
int sc_main(int argc, char* argv[]) {
    try {
        Memory mem("main_memory");
        sc_start();
    } catch (exception& e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

3. Compile and run the program.

6 The CPU Module

As you can see, the above program does nothing when run. This is because there are no events triggered. When there are no more events the simulation kernel, started by sc_start(), stops the simulation and the program ends. Actually, in this example errors about port binding are thrown because the modules ports are not connected. These errors are thrown when sc_start is called, and caught and displayed by the try/catch block.

The absence of events is because even though we create an instance of the Memory module we have not connected anything to its ports and no change to the ports values is taking place to trigger any events. In order to test our module we need a second one that will be connected to it and make changes to its ports. Such a module can be created with the following code:

```cpp
SC_MODULE(CPU) {
    public:
        sc_in<bool> Port_CLK;
        sc_in<Memory::RetCode> Port_MemDone;
        sc_out<Memory::Function> Port_MemFunc;
        sc_out<int> Port_MemAddr;
        sc_inout_rv<32> Port_MemData;

    SC_CTOR(CPU) {
        SC_THREAD(execute);
        sensitive << Port_CLK.pos();
        dont_initialize();
    }

    private:
        void execute() {
            while (true) {
```
wait();
Memory::Function f = (rand() % 10) < 5 ? Memory::FUNC_READ : Memory::FUNC_WRITE;
int addr = (rand() % MEM_SIZE);
Port_MemAddr.write(addr);
Port_MemFunc.write(f);

if (f == Memory::FUNC_WRITE) {
    Port_MemData.write(rand());
    wait();
    Port_MemData.write("Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz");
}

wait(Port_MemDone.value_changed_event());

// Advance one cycle in simulated time
wait();

}
}

This module simulates a CPU that has the appropriate ports to connect to our Memory module and make read/write requests for random addresses. First thing to note in the above code is that the CPU module uses a `sc_in` port for every `sc_out` port of the Memory module, and a `sc_out` for every `sc_in`. The constructor and execute member function of the module is similar to the one of the Memory module.

**Task 2: Add the CPU module**

1. Add the above code of the CPU module to your previous program and then modify the `sc_main()` function to have the following code:

```c
int sc_main(int argc, char* argv[]) {
    try {
        // Instantiate Modules
        Memory mem("main_memory");
        CPU cpu("cpu");

        // Signals
        sc_buffer<Memory::Function> sigMemFunc;
        sc_buffer<Memory::RetCode> sigMemDone;
        sc_signal<int> sigMemAddr;
        sc_signal_rv<32> sigMemData;

        // The clock that will drive the CPU and Memory
        sc_clock clk;

        // Connecting module ports with signals
        mem.Port_Func(sigMemFunc);
        mem.Port_Addr(sigMemAddr);
        mem.Port_Data(sigMemData);
```
mem.Port_Done(sigMemDone);
cpu.Port_MemFunc(sigMemFunc);
cpu.Port_MemAddr(sigMemAddr);
cpu.Port_MemData(sigMemData);
cpu.Port_MemDone(sigMemDone);
mem.Port_CLK(clk);
cpu.Port_CLK(clk);
cout << "Running (press CTRL+C to exit)... " << endl;

// Start Simulation
sc_start();
} catch (exception& e) {
  cerr << e.what() << endl;
}
return 0;
}

2. Compile and run the program.

What the code in the sc_main function does is to create instances of the modules, four channels of type sc_signal and sc_buffer and connect the two modules through their ports to those signals. It also creates an instance of the sc_clock class and connects that to the Port_CLK ports of the CPU and Memory modules. The sc_clock class is a predefined primitive channel derived from the class sc_signal and is intended to model the behavior of a digital clock signal. In effect an instance of sc_clock triggers an event in regular intervals (there are constructor overloads that can be used to set certain properties to the clock) so when a modules port is connected to the clock the process sensitive to the port is executed.

Finally the sc_start() statement tells the simulation kernel to start the simulation.

Task 3: Understand!

Although the simulation is now running, there is no output (if everything went ok). So, print details in locations of interest in the code to see how the modules behave and which code gets executed when. When building more complex simulations, these concepts should be basic knowledge.

7 Conclusion

Until now we have seen how a basic simulator can be built using SystemC and the basic aspects of the framework. A complete implementation of this example can be downloaded from:
More detailed information on the topics described here and many others can be found in the document: 1666-2005 IEEE Standard SystemC Language Reference Manual, which can be downloaded from http://standards.ieee.org/getieee/1666/ (via http://www.systemc.org/) or at the assignment site.
A Installing System C

The SystemC library can be downloaded from:

http://www.systemc.org/ or
http://staff.fnwi.uva.nl/r.bakker/teaching/aca2014/

Documentation and User Guides can also be downloaded from the latter site. SystemC comes in source code, thus in order to use it, it must be compiled after you downloaded and extracted the package.

A.1 Linux

- Execute the command: "/configure –enable-shared=no"
- Execute the command: “make”
- Execute the command: “make install”

After doing the above a file named “libsystemc.a” will be created in the directory <SystemC Installation Directory>/lib-linux/. This file must be linked with the application.