Multicore Architectures

Embedded Systems Architecture

Technische Universität Berlin

Einsteinufer 17, 10587 Berlin, Germany

SystemC Tutorial

1. Introductionto 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. GettingStarted

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

```
/afs/tu-berlin.de/units/Fak_IV/aes/tools/mca/systemc-2.3.0
```

You can copy the *part1_srcs* directory which contains a framework for the lab course, which has a Makefile that is configured to use the above SystemC installation:

However, if you want to configure and install SystemC yourself on your own machine, see the Appendix: Installing SystemC. The SystemC path above also contains precompiled binaries for Mac OS X (PPC and x86), Linux (32 and 64 bit x86) and Solaris (sun4u), so you could also copy that to your own machine. The Makefile provided in the framework automatically picks the correct library for the correct platform.

3. UsingSystemC

Modules: 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. They are defined as:

```
SC_MODULE("module_name")
{
//module body
}
```

Ports: 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. They are declared with in, out, or inout. Examples:

```
sc_in<bool> reset;
sc_inout<int> data;
```

Signals: Ports are used for the external communication and signals are used for the communication inside the module. Example:

```
sc_signal<bool> reset;
```

Process: 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.

Event: 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.

Constructor: Same as C++ constructors and is defined as:

```
SC_MODULE(module_name)
{
      SC_CTOR(module_name)
      {
            //body
      }
}
```

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. ExecutionSemantics

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 into the library so that when the program starts initialization of the simulation kernel and the structures this requires to be 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:

```
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.

4. Hello World example

```
// All systemc modules should include systemc.h header file
#include "systemc.h"
// Hello_world is module name
SC_MODULE (hello_world) {
  SC_CTOR (hello_world) {
    // Nothing in constructor
  }
  void print_hello() {
    //Print "Hello World" to the console.
    cout << "Hello World.\n";
  }
};

// sc_main in top level function like in C++ main
int sc_main(int argc, char* argv[]) {
  hello_world hello("HELLO");
  // Print the hello world
  hello.print_hello();
  return(0);
}
```

In *part1_srcs* directory, you will find the above hello world example under */src/hello_world.*
From your terminal, go inside *part1_srcs* directory, and type *make hello_world* in order to compile this example.
Successful compilation will result in a file called *hello_world.bin*
to run it, type *./hello_world*

notice the different parts of the hello world example, and notice your outputs.

5. 4-bit Counter example

A more comprehensive example is to design a 4-bit counter. Its design includes two files, the module description (*first_counter.cpp*), and the test bench (*first_counter_tb.cpp*).
Both files can be found under *part1_srcs/src/counter_4bit*

Going through the source code of first_counter.cpp:

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;      // Clock input of the design
  sc_in<bool>   reset ;      // active high, synchronous Reset input
  sc_in<bool>   enable;      // Active high enable signal for counter
  sc_out<sc_uint<4> > counter_out; // 4 bit vector output of the counter
  //------------Local Variables Here---------------------
  sc_uint<4>      count;
```

The above snippet of the code declares the module and defines its inputs and outputs. Adjacent comments explains the corresponding ports.
The final line of code defines a local variable of type 4-bit uint.

Next snippet shows the functional part of the counter.

```
  //------------Code Starts Here-------------------------
  // Below function implements actual counter logic
  void incr_count () {
    // At every rising edge of clock we check if reset is active
    // If active, we load the counter output with 4'b0000
    if (reset.read() == 1) {
      count =  0;
      counter_out.write(count);
    // If enable is active, then we increment the counter
    } else if (enable.read() == 1) {
      count = count + 1;
      counter_out.write(count);
      cout<<"@" << sc_time_stamp() <<" :: Incremented Counter "
        <<counter_out.read()<<endl;
    }
  } // End of function incr_count
```

The above lines of code declare one function called *incr_count()* .
SystemC signals can be read or written by using *<signal_name>.read()* and *<signal_name.write()>* functions respectively.

The above function writes 0 to the counter output port if the reset signal is active. Otherwise, it increments the count value and writes it to the output port.
It also prints out when did the incrementation operation take place, by using the function *sc_time_stamp()* .

```
 // Constructor for the counter
  // Since this counter is a positive edge trigged one,
  // We trigger the below block with respect to positive
  // edge of the clock and also when ever reset changes state
  SC_CTOR(first_counter) {
    cout<<"Executing new"<<endl;
    SC_METHOD(incr_count);
    sensitive << reset;
    sensitive << clock.pos();
  } // End of Constructor

}; // End of Module counter
```

The above snippet shows the declaration of the module constructor. The constructor defines the sensitivity list of each of the module functions, i.e., when should a certain function be executed. In the above case, the incr_count method is sensitive to both the reset signal, and the positive edge of the clock signal.

Now lets have a look at the *first_counter_tb.cpp* file, which is a test bench for the 4-bit counter module, where the different input ports are being provided by values, and the output port is being observed.

```
#include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_clock clock("clk", sc_time(1, SC_NS));
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
```

The above piece of code includes the necessary files (*systemc.h* and *first_counter.cpp*), and declares the *sc_main()* function along with some signals. Most important is the clock signal, which defines its period (*1 nsec* in our case).

```
 // Connect the DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);
```

The above piece of code defines an entity of module *first_counter*, and connect its ports to the previously defined signals.

```
 // Open VCD file
  sc_trace_file *wf = sc_create_vcd_trace_file("counter");
  // Dump the desired signals
  sc_trace(wf, clock, "clock");
  sc_trace(wf, reset, "reset");
  sc_trace(wf, enable, "enable");
  sc_trace(wf, counter_out, "count");
```

The above snippet defines a waveform file called counter.vcd, and added the to-be-monitored signals to it. This file can later be viewed by any waveform viewer, *gtkwave* for example.

```
  // Initialize all variables
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable
  sc_start(4,SC_NS);

  reset = 1;    // Assert the reset
  cout << "@" << sc_time_stamp() <<" Asserting reset\n" << endl;
  sc_start(8,SC_NS);

  reset = 0;     // De-assert the reset
  cout << "@" << sc_time_stamp() <<" De-Asserting reset\n" << endl;
  sc_start(4,SC_NS);

  enable = 1;  // Assert enable
  cout << "@" << sc_time_stamp() <<" Asserting Enable\n" << endl;
  sc_start(24,SC_NS);

  cout << "@" << sc_time_stamp() <<" De-Asserting Enable\n" << endl;
  enable = 0; // De-assert enable
  sc_start(4,SC_NS);

  cout << "@" << sc_time_stamp() <<" Terminating simulation\n" << endl;
  sc_close_vcd_trace_file(wf);
  return 0;// Terminate simulation

 }
```

The above piece of code is the final one, and here the generation of actual stimuli takes place. Notice the *sc_start(n, SC_NS)* function, which tells the simulator to simulate the a given (*n, SC_NS*) period of time.

Compile the design by typing: *make counter_4bit*.
Run the simulation by typing *./counter_4bit* and notice the resulting output.
After running it, a *Value Change Dump (vcd)* file is generated, namely *counter.vcd.* Open this file using *gtkwave* waveform viewer, and notice the behavior of the different signals as the clock changes.

6. 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
            {
                if (addr < MEM_SIZE)
                    m_data[addr] = data;

                Port_Done.write( RET_WRITE_DONE );
            }
        }
    }
};
```

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 module's 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 implements 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 implement 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 declaration 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 SC_CTOR is the code of the constructor of the module.The SC_CTOR 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 constructor's 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 constructor's 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 it's 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 Z's 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

- 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.

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

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

- Compile and run the program.

7. 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 module's 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:

```
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("ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ");
            }

            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

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

```
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;
}
```

- 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 module's 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.

8. 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 (*mem_cpu.cpp*) of this example can be found in *part1_srcs* directory.

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.

Appendix: Installing SystemC

The SystemC library can be downloaded from:

http://www.systemc.org/

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.

Linux (should work for MacOS X as well)

- Execute the "`configure`" script.
- Execute the command "`gmake`"
- Execute the command "`gmake 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.

Windows with Visual C++ 6.0

Open the workspace file in the folder:

```
   <SystemC Installation folder>\msvc60\SystemC\
```

Select Build > Build systemc.lib

Windows with Visual C++ 7.1 (.NET)

Open the solution file in the folder:

```
   <SystemC Installation folder>\msvc71\SystemC\
```

Select Build > Build Solution

A file named "systemc.lib" will be created which is the file that must be linked with the application.

Important:
The application that uses the library must be compiled with Run Time Type Information (RTTI) enabled.

If using Visual C++ 7.1 (.NET) the "/vmg" switch must also be set for the application that uses the library. This can be done as follows:

> From the C/C++ tab, select the Command Line Properties and add "/vmg" to the "Additional Options" box.

Further Information about building, installing and using the library can be found in the file "INSTALL".