SystemC: A Hardware-Software Co-specification, Codesign and Modeling Language: A Tutorial

EE8205: Embedded Computer Systems

1 Objectives

Software This lab/tutorial introduces the SystemC modeling language and provides an insight to employ SystemC for hardware-software codesign of embedded computer systems. It is also demonstrated that SystemC is useful for embedded system co-specification and modeling.

2 Overview

SystemC is a new system modeling language based on C/C++ that can support system level design. The emergence of hardware-software codesign and system-on-chip (SoC) era is creating many new challenges at all the stages of an embedded system design process. At the system level, engineers are dealing with how designs are specified, partitioned into hardware-software modules and then verified. Today, with embedded software engineers programming in assembly and C while their hardware counterparts working in hardware description languages such as VHDL or Verilog, problems arising from the use of different design languages, incompatible tools and fragmented design flows are becoming common.

Momentum is building for the SystemC language and modeling platform as the solution for representing functionality, inter-task communication, software and hardware at various system levels of abstraction. Increasing design complexity that demands very fast executable specifications to validate system concepts, and only C/C++ delivers adequate levels of abstraction, hardware-software integration and performance. Embedded system design also demands a single common language and modeling foundation in order to make a market for interoperable system-level design tools, services and IP (Intellectual Property) a reality. In response to these needs, SystemC has been developed as a standardized modeling language intended to enable system level design and IP exchange at multiple abstraction levels for systems containing both hardware and software components. SystemC is entirely based on C++ and it can also be installed on Windows-XP environment as outlined in the Appendix.

3 Introduction to SystemC

SystemC allows engineers to program software and hardware modules of the same project quite easily. SystemC Libraries are based upon C language and their syntax is similar to C/C++ language we are familiar with. Hence the only things touched upon in this section are the difference and the new features of SystemC. You must consult the SystemC documentation available at https://www.accellera.org/downloads/standards/systemc/ and course web-pages http://www.ee.ryerson.ca/~courses/ee8205/ and documents (doc) available at the department linux system.

(i) Signals

Instead of using variables, *signals* are used when programming hardware modules very close to VHDL or Verilog.

Declaring signals is similar to normal C language as given below.

```
sc_signal<data_type>  sig_name;
```

- The above line would create a signal of type 'data_type' and name it 'sig_name'.
- The 'data_type' could be any valid C++ type, SystemC type or user defined structure type.

Each hardware module has its own *input* and *output ports* to which these signals must be mapped or bound.

(ii) Ports

Ports declaration is similar to declaring signals.
The following line would create a 'port_type' port of type 'data_type' with name 'prt_name'.

```
port_type<data_type>  prt_name;
```

- 'port_type' defines the port as an input, output, or inout as sc_in, sc_out, or sc_inout respectively.
- The 'data_type' could be any valid C++ type, SystemC type or user defined structure type.

There are two types of port binding: *named and positional.*

- Named binding: It binds each port to a signal (one to one) by using their name. e.g. block.input(signal01);
  where 'block' is the name of the module, 'input' is the name of one of the ports of 'block', and it is being bound to 'signal01', which is a signal.
- Positional binding: It binds the ports by their position. This is quite unsafe as one change in the order might result in unforeseeable effects, and the engineer might not even know where the problem is?
  e.g. block(signal01, signal02);
  where 'block' is the name of the module and 'signal01' and 'signal02' are two signals, which are being bound to the first two ports of the module.

Vector ports and *signals* are used for arrays and they must be bound by name.
Reading and writing to ports and signals is performed by using read() and write() functions e.g. "signal01.read()" would read the signal 'signal01', while "signal01.write(1)" would write '1' onto 'signal01'. Similarly, "port01.read()" and "port01.write(1)" will read from 'port01' and write a '1' onto 'port01' respectively.

(iii) Modules

A module is the basic object in SystemC that includes ports, constructors, data members, function members and maybe internal memory storage and internal functions. A module can be thought of as a process or a box in the hardware block diagram. It has two types of syntax. Firstly, “SC_MODULE(module)” declares the ‘module’ as a SystemC module. Secondly, “struct module : sc_module { … };” that performs the same function.

Modular instantiations follows the following syntax:

```
module_name module_shortname(“module_name”);
```

The module_shortname could be any undeclared name, which is used to bind the ports to the signals later on.
Once the modules are instantiated, the ports *must* be bound.

(iv) Constructors

Each module should include a constructor. Constructors identify processes as methods using the SC_METHOD macro as explained here.
SC_METHOD(function) identifies the function or process ‘function’ as a SystemC method.

In SystemC, methods are all called at initialization and thus all initializations that need to be performed must be defined inside the constructor. If one does not need initialization then use the “dont_initialize()” command after each method that is to be skipped for initialization. Methods are called similar to normal C implementation and their protocols follow the following syntax:

```
Function_type module_name::function_name(data_type var_name) { … }
```

Methods have to be made sensitive to some internal or external signal. Normally, they are made sensitive to the positive or negative clock edge as shown below respectively.

```
sensitive_pos << clock
sensitive_neg << clock,
```

(v) The Main Program Function

The main function must include all the modules in the project and their ports must be connected to the signals. The argument to the main function is as following.

```
int sc_main(int argc, char *argv[])
```

(vi) Creating the Clock Signal

There are two ways of creating clock signals. A clock is declared using the sc_clock type, i.e. sc_clock clk(“clock”, 20); and then one must “start” the clock with the sc_start(sc_time) macro. This would create a clock signal of name ‘clk’ of period 40 (a default time) and ‘sc_time’ is the total time. It means that the clock would stay low for 20 time periods and high for 20 time periods, and this would continue for ‘sc_time’ length.

The clock can also be declared using the sc_signal type, i.e. sc_signal<bool> clk; then we must manually create the clock signal using a loop and the sc_cycle macro, as given below.

```
for (j = 0; j < sc_time; j++) {
        clk.write(0);
        sc_cycle(20 e –9);
        clk.write(1);
        sc_cycle(20 e –9);
}
```

This would create a clock signal of period 40 nanoseconds of length sc_time.

(vii) Tracing Signals

Tracing is useful to verify that the design of system is working according to the specification and this is done by creating a trace file and then using the built-in signal tracing method to trace signals. A trace file is created by using the following procedure and command.

```
sc_trace_file *tf = sc_create_vcd_file("trace_file");
```

This creates a trace file of extension ***.vcd***, and points it to the file pointer 'tf'.

- The syntax is similar to C decleration but instead of FILE*, a sc_trace_file* is used.
- Other valid file types are ***.wif*** and ***.isdb***, which can be created by using the same command and replacing the ***.vcd*** by an appropriate file extension.

To trace a signal, one needs to follow a syntax given below

```
sc_trace(sc_trace_file*, sc_signal, "signal_name");
    For example, sc_trace(tf, clk.signal(), "clock");
```

A trace file must also be closed by using the sc_close_vcd_trace_file(my_trace_file); Command that will close the trace file named "my_trace_file.vcd"
Similarly, we can use 'sc_close_isdb_trace_file' or 'sc_close_wif_trace_file' to close ***.isdb*** and ***.wif*** extension trace files.

(viii) Viewing Trace Files

The trace file with extension ***.vcd*** can be viewed using the simvision program available on the department linux systems as explained below.

- Execute *simvision* or 'gtkwave' program
- Once the program has loaded; under the **File** menu, click on **Open Database** and then select the ***.vcd** file created by your SystemC simulation code.
- Select your ***.vcd** file and open it. Confirm OK for translation to SST database.
- Under the 'Scope Tree' frame should be the name of your trace file, click on it to open the 'SystemC' checkbox.

- Click on ‘SystemC’ to open the available traced signals in the ‘Signals/Variables’ frame.
- Right click on the signal name and choose the ‘Send to target Waveform Window’ option. A new window should appear, with the waveform displayed.
- If the waveform does not look as it should be, try to magnify out by clicking the appropriate button on the top right corner of the waveform window.

4 Simulating a BCD Counter: An Example

The example hardware implemented to introduce the working of SystemC is a BCD up-counter, which counts from zero (0) to nine (9) in ten (10) clock cycles. The counter is reset to zero (0) after it reaches nine (9). The example runs for twenty-one (21) clock cycles. To explain the hardware-software co-design of the BCD counter, the counter is implemented in hardware, while a software block checks whether the counter has reached ten (10), and then resets it to zero. You can follow the following steps to model and simulate the counter example using SystemC.

- Copy all of the files from the course directory /ee8205/labs/counter/*.*
  There should be five files namely: main.cpp, counter.cpp, counter.h, makefile.sun, makefile.defs as explained below:
  - counter.cpp is the counter block, which is completely modeled in hardware.
  - counter.h defines the ports and the internal variables and functions used in the counter hardware block.
  - main.cpp is the main routine and it also contains the software block, where the program checks the count for ten, and resets it to zero when it reaches ten.
- Compile the program using the make command as given below.

- This will create the executable counter file. To run this executable, type the name of the executable file.
- You should see the counter moving from 0 → 9 and then a message being displayed showing the counter was reset to 0. There should be two separate cycles, corresponding to 21 clock cycles
- Lines 12 to 19 of the main.cpp are the commands used to declare a trace-file, creating a trace-file and defining the signals to be traced. For more details on tracing, consult the **Tracing Signals** sub-section described earlier.
- The counter example creates “counter_tracefile.vcd” which can be viewed using the simvision software installed on the department Sun-Unix workstations.

Appendix

MS-Windows Installation of SystemC

- You need both the Cygwin and SystemC packages to work in a Windows-XP environment as SystemC will be running everything under Cygwin.
- Download the Cygwin setup program from www.cygwin.org

Recommendation: Select "Download from Internet" to save the files to a local directory. You can then put this directory to a CD to install on other systems, or simply to archive them for later installation.

- Open up the devel section and make sure to add the following packages:
    - Autoconf
    - Automake
    - gcc
    - gcc-g++
    - gdb
    - make
- Open up the Doc section and add *cygwin-doc* package.
- Open up the Web section and add *wget* package.
- When you want to install, run the setup program again and select the "Install from Local Directory" to install from the archives you saved earlier
- Don't forget to add the packages above. They have been downloaded but are still not part of the base installation

Recommendation: Do not install Cygwin to the same directory that contains your downloaded files.

- Download the SystemC sources from https://www.accellera.org/downloads/standards/systemc/.
- Unzip the install file into a folder of your choice e.g. 'C:\Temp'
- Read carefully the instructions found in the file called "INSTALL" to compile the SystemC libraries. You can open this file in notepad or wordpad.
- Make a directory 'objdir' (e.g. C:\Temp\SystemC-2.0.1\objdir)
- Change to the ‘objdir’ directory, in Cygwin, and configure SystemC as following:

```
../configure --prefix=/usr/local/systemc
```

or

```
../configure --prefix=/usr/local/systemc --host=i386-pc-cygwin
```

- This will place the installed files (once you install SystemC) into the directory '/usr/local/systemc'
- The configure command should not return any errors. If you encounter any error, please double check and try again. There could be errors in your cygwin installation, for example files you forgot to add gcc or g++.
- Compile the library using 'make'
- Then install the library and examples using 'make install'
- The compiled library and examples should now be sitting in the directory that you specified earlier (e.g. /usr/local/systemc/)

Configuring SystemC for MS-Windows System

- Each example and application of SystemC must include the two files namely: Makefile and Makefile.defs
- Makefile.defs can be copied from the SystemC folder/examples
- Makefile can be copied from the SystemC folder/examples/any of the folders
- It will be named with an extension of .gcc, .linux, .hp, .sun
- All of these are similar but customized for different systems
- Copy appropriate file for your system and rename it Makefile
- In your SystemC project directory (you should make one for every system design you are working on), you will need, Makefile, Makefile.defs and your source files (*.cpp or *.c) and source libraries (*.h)
- Modify Makefile.defs as:

  SYSTEMC = **…/.../...**

  This line should point to the location of compiled SystemC libraries
- The rest of the file can be left as unchanged
- Modify Makefile as follows:
  - TARGET_ARCH = gccsparc05
    - This line will be different depending for which of the Makefile sample you have copied
    - Modify the string to correspond to the host architecture you are running on, and should point correspond to the '/usr/local/systemc/lib-xxx' folder, where xxx is the host system
    - Under cygwin it will be '/usr/local/systemc/lib-cygwin'. Make sure that a ~1.9MB library file 'libsystemc.a' should exist there
    - therefore the line should look like … TARGET_ARCH = cygwin
    - Not all architectures are supported. If you are running on a non-supported architecture, simply use 'linux' or other compatible architecture. This string is used to differentiate a few tools and will not have a big effect on the system.
  - MODULE = run
    - This line reflects the name of the compiled executable
    - It can be left as-is or modified. Left as-is, the compiled executable will have a name of 'run.x'
  - SRCS = main.cpp display.cpp
    - This line must include the names of all the files in the project, not including the custom libraries, which were included from elsewhere
  - Do not remove the 'include **..**/Makefile.defs' line
  - Change it to 'include Makefile.defs' if the Makefile.defs is in the same folder as Makefile file
- The Makefile ***must*** be in the same folder as the rest of the project files
- Now, you can compile and run the examples as follows:

  make –f makefile
  - This will compile the executable file (*.x), which is ready to be executed by typing ***./****.x* or **.x*. This is your SystemC simulator.