Introduction to SystemC

COE838: Systems-on-Chip Design
LAB-1

1. Objectives

The purpose of this lab is to introduce students to SystemC and demonstrate how it can be used to model hardware/software components in an event-driven manner. SystemC is a library that may be included in C++ code, consisting of many classes and macros for system development. These classes allow one to design, model, and simulate a system prototype using techniques similar to SystemVerilog and VHDL, but with the programmability of a C-based language. Using this lab, students will become familiar with the SystemC coding paradigm, and obtain an understanding of how it can be used for various different hardware/software designs in SoC modeling.

2. SystemC Overview

Figure 1 provides a SystemC block diagram for a flip flop module and testbench. A brief explanation for each attribute shown in the figure, along with its respective coding technique is provided in the subsequent subsections. The code corresponding to this example can be found in your course directory at /coe838/lab1/tutorial/flipflop. We will first try and understand the SystemC model and code provided before actually executing the code.

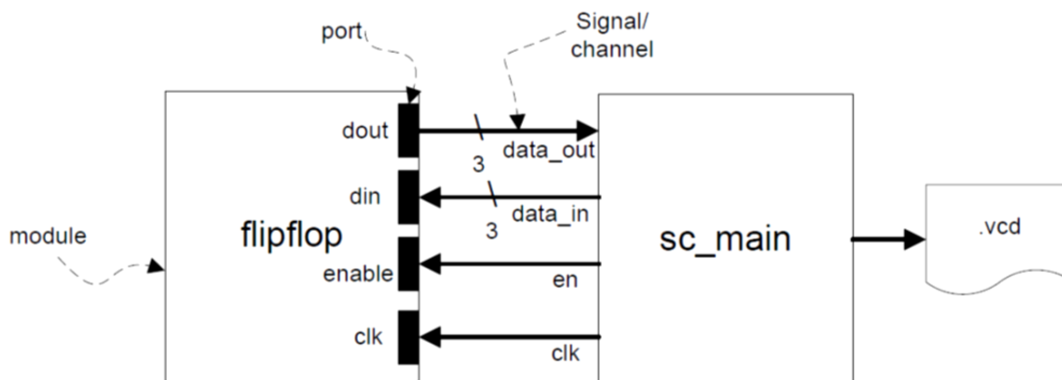

Figure 1: Flip Flop SystemC Example

a. Signals/Channels

A signal is used to connect two or more modules together, providing a means for transferring and communicating data in a system. A signal may be declared and implemented in SystemC as follows:

General Declaration: sc_signal<data_type> signal_name;
Example: `sc_signal<sc_uint<3> > data_in;`

This example declares a signal called data in, which is of type unsigned integer, possessing a datawidth of three (i.e. 3 bits "000", "001" etc). Common data types also include boolean <bool>, integer <int> etc.

Signals are used to connect two different modules using a component's *ports*. Therefore these signals must be bound (or mapped) to module *ports* first to allow for data communication.

b. Ports

A port is an entry way (input, output, or inout) to a module. When a signal is bound to a port (as done in sc_main.cpp), information may be read or written to the module for processing. A port may be declared in a module as follows:

General Declaration: port_direction_type<data_type> port_name;
Example: `sc_in<bool> enable;` or `sc_out<sc_uint<3> > dout;`

Binding signals to ports in the main function (or equivalent testbench) will be described shortly.

c. Modules

A module is a SystemC object which provides various parameters and functionalities to a system. A module declaration is similar to an ENTITY and sensitivity list in VHDL, however must additionally include a brief outline of the module's functionality. A module is declared using a header (.h) file, where its actual functionality is coded in its corresponding .cpp file. In the case of the flip flop, the module declaration may be found in flipflop.h, its functionality in flipflop.cpp, and its ports bound in sc_main.cpp.

As seen in flipflop.h, a module's declaration will often include 1) port declarations, 2) any internal functions that should be executed by the module (SC_METHOD), and 3) a constructor giving specifications of any event-triggered (or level-sensitive) functionality. A basic template example for instantiating a module can be seen in the flipflop.h file as follows:

```
SC_MODULE(flipflop){
      //port declaration
      sc_in<bool> clk;
      sc_in<bool> enable;
      ...

      //function declaration and/or implementation
      void ff_method();

      //constructor
      SC_CTOR(flipflop){
            SC_METHOD(ff_method);
            dont_initialize();
            sensitive << clk.pos();
      }
};
```

As seen here, the port declarations are given first, followed by any method to be executed by the flipflop. This is then followed by a constructor, specified with the keyword SC_CTOR(), which states that the module should execute the function ff_method when triggered, with dont_initialize() specifying that the constructor should not execute this method automatically. Rather, the module may only execute ff_method() when an event indicated on its sensitivity list has been triggered. In this case, the module will only execute ff_method when a positive clock edge has triggered, as specified

with the keywords `sensitive << clk.pos()`. The actual ff_method is implemented in *flipflop.cpp* and is coded using standard C++ programming semantics.

d. Main Program (sc_main)

Referring back to Figure 1, we see that the main components and features of a SystemC design have now been covered. However, how do we connect all these components and signals together, and generate test data for execution? The main program, sc_main.cpp (or equivalent testbench file) is used to assemble the system components, bind the system ports, and drive data in and out of the modules. It is also possible to consider sc_main as a system's processor, executing any software functions required, and sending/receiving the "accelerator" (modules) based data.

```
#include <systemc.h>
#include "flipflop.h"    //include any module declaration

int sc_main(int argc, char* argv[]){
	..
	sc_signal<sc_uint<3> > data_in, data_out;           //Declare signals
	sc_signal<bool> en;

	//Create a clock signal of 10ns, with 50% duty cycle
	sc_clock clk("clk",10,SC_NS,0.5);

	//Create an instance (DUT) of the flipflop named "flipflop"
	flipflop  DUT("flipflop");
	DUT.din(data_in);        // Bind the signals to the flipflop module ports
	DUT.dout(data_out);
	DUT.clk(clk);
	DUT.enable(en);
	....

	//Test the module (write) and observe output (based on flipflop.h method)
	en.write(0); //initialize
	data_in.write(0);
	sc_start(9, SC_NS);
	//run the clock for 9ns (for setup/hold time purposes)

	en.write(1);                     //enable and input data
	data_in.write(7);
	sc_start(20, SC_NS);             //run the clock for 20ns
	....

	return 0;
}
```

Note that the keywords .write() and .read() are used to communicate data between modules.

e. Tracing Signals

Aside from using the cout << statement in C++, it is extremely useful to verify and debug SystemC designs using a tracefile for tracing signal execution. A tracefile is created as follows (see sc_main.cpp):

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

The statements above creates a file pointer tf, pointing to a file called *trace_file.vcd* for writing.

```
sc_trace(tf, clk, "clk");
```

The statement above, sc_trace, can trace module signals and write it to the vcd file. The example statement specifies that a trace called "clk" is to be included in the vcd, tracing the clk signal bound to the flipflop module in sc_main.cpp. To specify a certain time unit that the trace file should abide by, one may also use the statement:

```
tf->set_time_unit(1, SC_NS);
```

The trace file should be closed once the main program has finished as follows:

```
sc_close_vcd_trace_file(tf);
```

Once your module has finished executing, you will find a trace file generated in the same folder as your code called "trace_file.vcd".

3. Makefiles, SystemC Execution and Simulation

3.1 Understanding Makefiles

Navigate to the course directory /coe838/lab1/tutorial/flipflop. Copy the contents of the flipflop folder into your home/course directory. This folder contains the files sc_main.cpp, flipflop.h and flipflop.cpp. You will also find a file called Makefile. Makefiles are used to compile a C/C++ project and are extremely useful when it is necessary to include several source files, header files, and/or libraries to successfully create an executable. Thus instead of typing:

```
g++ -I/usr/local/SystemC-2.3.0 -L/....../  -c flipflop.cpp sc_main.cpp -o flipflop.x -lsystemc -lm
```

into the terminal every time you want to compile your lab, a Makefile will allow you to simply type:

```
make
```

The make command will send a command to the Makefile which will execute the g++... line above to compile your project based on the Makefile's specifications. You may also type in

```
make clean
```

which will clean your project, deleting any executables, .o, .cpp~ and .h~ files (where ~ represents old files that have been temporarily saved by Unix as a backup, created every time you re-save your work).

Open the Makefile in the flipflop folder and analyze its contents. For subsequent lab assignments involving SystemC, you may simply copy and paste the contents of this Makefile to your own Makefile. However, the following lines will be of importance:

```
PROGRAM =   flipflop.x
SRCS    =   flipflop.cpp sc_main.cpp
OBJS    =   flipflop.o sc_main.o
```

You will need to change this information based on your module file names, where the last .cpp file name (and .o) must always be sc_main.cpp or equivalent (i.e. you must compile the dependency modules first, prior to compiling the main file). For the interested reader, a more detailed tutorial on Makfiles may be found at: http://www.ecb.torontomu.ca/~elf/ele428/makefile-tutorial.html

3.2 Compiling, Executing, and Simulating the SystemC Modules

i) Copy the folder /coe838/lab1/tutorial/flipflop into your home/course directory.

ii) Open a terminal and *cd* to the directory containing the flipflop tutorial files.

iii) Type *ls* into the terminal. You should see the Makefile and supporting .cpp/.h files discussed in this tutorial. If not, repeat step 2. to find the directory again.

iv) Type *make* in the terminal

v) This will create an executable called *flipflop.x*

vi) To execute the flipflop binary, type *./flipflop.x* in the terminal.

vii) You will see the terminal output stdout and a message (to check the .vcd file produced).

viii) Type *ls* in the terminal. You should see a file called "trace_file.vcd" now present in the folder.

ix) To view this waveform, type the command *gtkwave trace_file.vcd* in the terminal.

x) The program gtkwave should now open a window. On the left side, you will see a module called "SystemC". Double click this field.

xi) This will output the signals we specified for tracing in sc_main.cpp to the bottom left window. Double click each signal to include it in the waveform editor. You may also drag and drop the signals in the waveform editor as necessary.

xii) Examine the waveforms and understand how each signal pertains to the data in the sc_main.cpp and flipflop.cpp files.

4. Additional Examples

The course directory /coe838/lab1/tutorial/ also contains a folder called /barrelshifter. Copy, open, and analyze the files and contents in your own directory. Understand how the barrel shifter works, and how this implementation differs from the flip flop example provided above. Once you have obtained a good grasp of the system's contents, compile, execute, and simulate the barrel shifter using the same steps as Section 3.2.

Another example is given in the folder /coe838/lab1/tutorial/barrelflop. This example amalgamates the flip flop and barrel shifter designs. The user may choose to execute either the flipflop or barrel shifter. Examine the files in the folder to see how they differ in implementation. To execute the flip flop example, in the terminal execute the command:

```
./barrelflop 1
```

To execute the barrel shift example, type:

```
./barrelflop 2
```

To obtain help on the executable's usage, type:

```
./barrelflop - help
```

5. What to Hand In

Using the techniques learned throughout this tutorial, students are to design an Integer Arithmetic-based ALU using SystemC. The ALU may add (A+B) or subtract (A-B) two numbers, where input B is connected to a barrel shifter. Thus if enabled, input B will first be shifted to the left or right (l_r) by shift_amt before being transferred to the ALU's B input. Figure 2 outlines the ALU architecture and respective control signals of the system, whereas Table I dictates specifications of the unit's control signal functionality.

Table I: ALU and Barrel Shifter Functionality

ALU		**Barrel Shifter**		
Op	**Functionality**	**en**	**l_r**	**Functionality**
0	Subtract	0	X	Pass Data through (shifting disabled)
1	Addition	1	0	Shift left by *shift_amt*
		1	1	Shift right by *shift_amt*

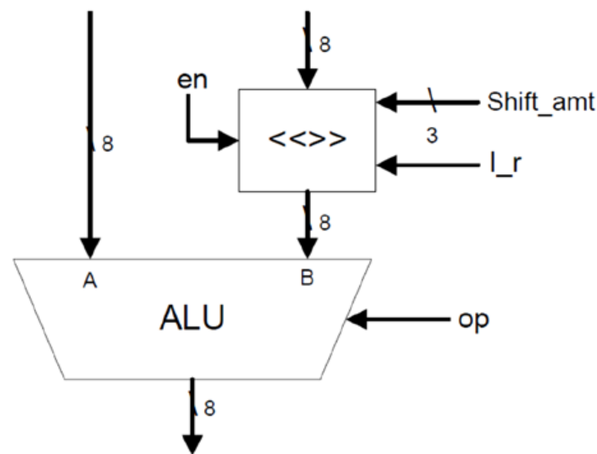

Figure 2: Integer Arithmetic ALU with Barrel Shifter

This lab is due before the start of your lab session in week 3. Please print out and hand in all SystemC code necessary for implementing the Integer Arithmetic ALU design specified above. Test your design using various test case scenarios to ensure that the unit abides by all the functionality outlined in Table I. Include screenshot(s) of the waveforms generated for all test cases. The lab must be handed in with the University cover page, dated and signed. Your lab instructor may also ask you to demo your work during submission. Be prepared.

Note: There are many ways to implement the design of Figure 2. However, it is important that the barrel shift occur before the ALUs computation. Thus the two components may require different sensitivity lists, or may execute in a pipelined fashion to guarantee that the barrel shifter executes in the cycle prior to the ALU.