Embedded System Design and Modeling

EE382V, Fall 2008

Homework #3

System-Level Design with SystemC

Assigned: November 6, 2008
Due: November 21, 2008

Instructions:

- Please submit your solutions via Blackboard. Submissions should include a single PDF with the writeup (do not paste any source code into the PDF) and single Zip or Tar archive for supplementary files (containing the source code, including a README with instructions for compiling and running each model).
- You may discuss the problems with your classmates but make sure to submit your own independent and individual solutions.
- Some questions might not have a clearly correct or wrong answer. In such cases, grading is based on your arguments and reasoning for arriving at a solution.

Problem 3.1: SystemC Compiler and Simulator (20 points)

The goal of this problem is to make you familiar with the SystemC environment, in particular compilation and simulation of SystemC code, using the simple FIFO example included with the SystemC installation.

The SystemC environment is installed on the ECE LRC Linux servers. Instructions for accessing and setting up the tools are posted on the class website:

http://www.ece.utexas.edu/~gerstl/ee382Vf08/docs/SystemC_setup.pdf

In short, once logged in (e.g. remotely via `ssh`), you need to set the $SYSTEMC environment variable (depending on your $SHELL):

```
setenv SYSTEMC /home/projects/courses/…/systemc-2.2.0
```
([t]csh)

or

```
export SYSTEMC=/home/projects/courses/…/systemc-2.2.0
```
([ba]sh)

Next, copy the example found into a working directory for this problem:

```
mkdir hw3_1
cd hw3_1
cp $SYSTEMC/examples/simple_fifo/* .
ls
```

You can then use the provided `Makefile` to compile and simulate the example:

```
make
./simple_fifo
```

Inspect the sources of the example and the included `Makefile` to understand SystemC compilation and simulation process, experiment with the `Makefile` usage and start modifying the example to experiment with different features of the SystemC language:

(a) Briefly explain (in 1-2 sentences) the functionality and structure of the example. Report the output from simulation of the example.

(b) Modify the example to replace the custom `fifo` channel with a corresponding `sc_fifo<char>` channel from the standard SystemC channel library. Simulate the code to verify correctness and submit the modified source code.

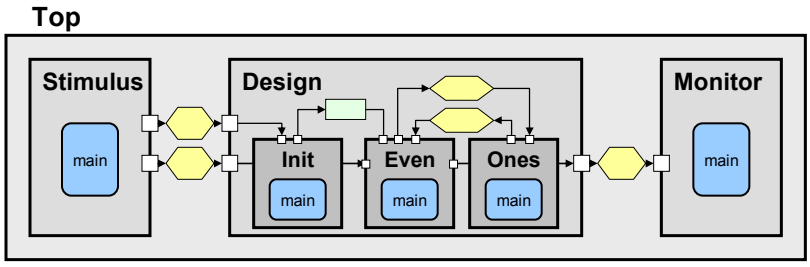
Problem 3.2: SystemC Modeling (40+40 points)

For this problem, we will take the SpecC parity generator/encoder example developed in Homework 2, Problem 2.1 and model it in SystemC. As a reference, you can start from the SpecC code posted as part of the solutions to Homework 2 on Blackboard. Following the ideas outlined in reference [15] on the class webpage, we will convert the SpecC models into equivalent SystemC ones:

(a) Translate the SpecC specification model of the parity checker from Problem 2.1(a) into a corresponding SystemC model at the untimed functional (UTF) level.

In a straightforward manner, the SpecC behavior hierarchy is converted into a matching hierarchy of SystemC modules where each SpecC behavior becomes a SystemC module with exactly one *main* process. As we learned in class, SystemC does not support a serial-parallel composition so we have to convert all behaviors into parallel modules/processes. As such, the extra level of hierarchy around `Even` and `Ones` is unnecessary and we can have a flat `Design` hierarchy. On the other hand, to maintain the proper execution order we have to insert synchronization triggering behaviors only once `Init` is finished. There are different ways of doing that but the easiest solution is to turn the flag/mode variable between `Init` and `Even` into a signal used to synchronize the two (exploiting the fact that `Even` and `Ones` already synchronize properly among themselves, i.e. not needing a trigger between `Init` and `Ones`). Make sure to use a `sc_buffer` for the mode/flag signal between `Init` and `Even` (triggering an event every time the signal is assigned to, not only when its value changes as is the default for `sc_signal`). Lastly, convert all `c_queue` and `c_double_handshake` channels and interfaces into SystemC `sc_fifo<`*T*`>` instances and interfaces of appropriate template type *T*.

Finally, enclose this design into a typical testbench setup. Implement the top level of the example (`Top` behavior) to describe a proper structure consisting of `Stimulus`, `Design` and `Monitor` modules:



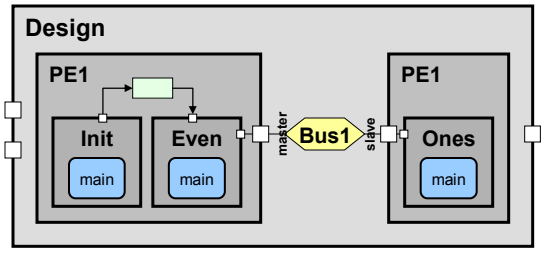
As in Problem 2.1(a), the `Stimulus` and `Monitor` modules should read the mode and the stream of input words from a file, feed them into the design over two input queues (FIFOs), receive the encoded words on an output queue (FIFO), write the result to an output file, and exit the simulation cleanly when the end of the streams has been reached.

Note that there is an alternative way of modeling the specification: recall that SystemC allows multiple processes per module. Hence, the `Design` could be described as a single, flat module containing three communicating processes (where then plain events could be used for synchronization between processes). However, what are some advantages of modeling the specification the way we did, i.e. with one module per process?

(b) Assume again a partitioning where *Init* and *Even* processes are mapped to *PE1*, the *Ones* process is mapped to *PE2*, and a *Bus1* is connecting *PE1* (master) and *PE2* (slave) using a modified double-handshake protocol. Refine the specification model from (a) into a transaction-level model (TLM) where the `Design` module reflects this partitioning.

First, insert an extra layer of modules representing the two PEs and group the processes under these `PE1` and `PE2` modules according to their mapping, exposing inter-module communication channels between `Even` and `Ones` in the process. Again, since SystemC does not support sequential composition for static scheduling, `Init` and `Even` simply remain as separate modules/processes synchronized by the mode/flag signal inside `PE1` (as the only alternative, they could be combined into a single sequential *main* process on PE1 to realize a statically scheduled implementation; but this would require a significant rewriting and modification of the hierarchy).

Next, write a transaction-level channel for *Bus1* that models the modified double-handshake protocol semantics and timing (assuming worst-case delays) from Problem 2.1(c), i.e. re-implementing the SpecC bus TLM in SystemC (again, with master and slave interfaces that contain corresponding `masterRead/Write` and `slaveServeRead/Write` methods). Replace all inter-PE channels with a single instance of the `Bus1` channel and refine all inter-PE communication to go over this bus instance:



Note that in the simplest case, you can just replace all communication calls directly in the processes with equivalent calls to the corresponding bus interface methods. You are not required to implement bus drivers or bus interfaces in the PEs that translate from high-level FIFO interfaces to low-level bus read/write transactions such that no changes in the processes would be required.

Simulate all models to validate their correctness. Turn in the source files for all models and all input and output test files.