Embedded Systems Modeling and Design
ECPS 203
Fall 2019

Assignment 5

Posted: October 30, 2019
Due: November 6, 2019 at 6pm

Topic: Test bench model of the Canny Edge Decoder in SystemC

1. Setup:

This assignment is the next step in modeling our application example, the Canny Edge Detector, as a proper system-level specification model which we can then use to design our embedded system target implementation. In this assignment, we will create a SystemC model with a suitable top-level structural hierarchy including a test bench.

We will use the same Linux account and the same remote servers as for the previous assignments. Start by creating a new working directory, so that you can properly submit your deliverables in the end.

```
mkdir hw5
cd hw5
```

As before, our video is available in a shared directory on the server. In addition, we want to use the video frames extracted in Assignment 4. To save disk space, do not copy the data into your new directory, but create symbolic links to it, as follows:

```
ln –s ~ecps203/public/DroneFootage DroneFootage
ln –s ~/hw4/video video
```

As starting point for your SystemC model, you can use your own C++ model which you have created in the previous Assignment 4. Alternatively, you may start from the provided solution for Assignment 4 which you can copy as follows:

```
cp ~ecps203/public/CannyA4_ref.cpp Canny.cpp
```

For your convenience, we also provide a simple **`Makefile`** for use in this assignment which you can copy as follows:

```
cp ~ecps203/public/MakefileA5 Makefile
```

A simple call to **`make`** will then compile your model into an executable, and a call to **`make test`** will simulate the model and compare the generated edge images against the reference images provided in the **`video`** directory.

2. Creating a test bench model with top-level structural hierarchy

Step 1: Create a test bench and platform structure for your SystemC model

The purpose of this assignment is to introduce a proper test bench and overall structural hierarchy into our application model. In particular, we will introduce the top-level module **`Top`**. This will consist of three modules, namely **`Stimulus`**, **`Platform`**, and **`Monitor`**. The **`Platform`** module, in turn, should contain a dedicated input unit **`DataIn`**, an output unit **`DataOut`**, and the actual design under test **`DUT`**.

Specifically, your model should be structured as the following instance tree shows:

```
Top top
|------ Monitor monitor
|------ Platform platform
|       |------ DUT canny
|       |------ DataIn din
|       |------ DataOut dout
|       |------ sc_fifo<IMAGE> q1
|       \------ sc_fifo<IMAGE> q2
|------ Stimulus stimulus
|------ sc_fifo<IMAGE> q1
\------ sc_fifo<IMAGE> q2
```

For communication, we will instantiate FIFO-type channels from the SystemC standard library. Specifically, use the regular first-in-first-out primitive channel **`sc_fifo<IMAGE>`** where template parameter **`IMAGE`** is the type of the data you need to communicate. Since **`IMAGE`** is an array and C++ does not provide an operator for array assignment, however, we need to wrap the array into a proper class with overloaded operators. To simplify this technicality, you may copy the **`class IMAGE`** from this provided file:

`~ecps203/public/Image.cpp`

Since **`sc_fifo`** channels are not well described in the presented Doulos slides, this section summarizes the use of the standard **`sc_fifo`** channel in SystemC. The type of this standard primitive channel is **`sc_fifo`**, so an instance of this channel can be defined as **`sc_fifo ch1;`** To set the size of the buffer in the channel (which defaults to 16), you pass the desired buffer size to the constructor call, for example, **`ch1("ch1", size)`**. For our example, use the value 1 here, which will allow at most 1 image to be stored inside the channel. This will be sufficient freedom for the model to run, while it will not introduce any extra delay stages at the same time.

The **`sc_fifo`** channel offers a number of interface methods, but the main two methods are **`void read(T &data)`** and **`void write(T &data)`**. While it is possible to build your own ports (using **`sc_port`**), there are predefined port types available, namely **`sc_fifo_in`** and **`sc_fifo_out`**. When instantiated, you can communicate via such ports by simply calling **`PortOut.write(myData)`** or **`PortIn.read(myData)`**.

To connect ports to channels, just bind the ports to the channel instance in the constructor of the parent module (or in the **`before_end_of_elaboration`** function). An example looks like this: **`stimulus.PortOut.bind(ch1)`**.

For the above described top-level structural hierarchy, a total of four channel instances will be needed, two at the test bench level (**`Top`** module), and two within the **`Platform`** module.

Specifically, the **`Top`** module should instantiate **`Stimulus`**, **`Platform`** and **`Monitor`** modules in parallel. The **`Stimulus`** module should read the input image from the file system and pass it into the **`Platform`** via the first queue channel. Correspondingly, the **`Monitor`** should receive via the second channel the generated edge image from the **`Platform`** and write it out into an output file.

In the **`Platform`** module, the **`DataIn`** module should, in an endless loop, receive an input image and pass it unmodified to the **`DUT`**. Similar, the **`DataOut`** module should, also in an endless loop, receive an input image from the **`DUT`** and pass it on. These two instances will be needed later during model refinement. They will allow our test bench to remain unmodified even when later in the design flow the communication to the DUT is implemented via detailed bus protocols.

Finally, the **`DUT`** module should contain the entire Canny algorithm source code. Its main thread will receive an image via the input port, call the **`canny()`** function to process it, and then send out the edge image via the output port. Since our target embedded system will never stop working (unless its power is turned off), this processing will run in an endless loop, similar as the infinite loops in the **`DataIn`** and **`DataOut`** modules.

Throughout your model recoding, ensure that it still compiles, simulates, and generates the correct output images. You are done with this assignment when the hierarchy described above has been created and your code compiles fine without errors or warnings.

In the end, your final model should not contain any global functions (except for **`sc_main`**), neither any global variables, nor any wait-for-time statements. For communication, only standard **`sc_fifo`** channels with proper port connections should be used (no plain events or user-defined channels).

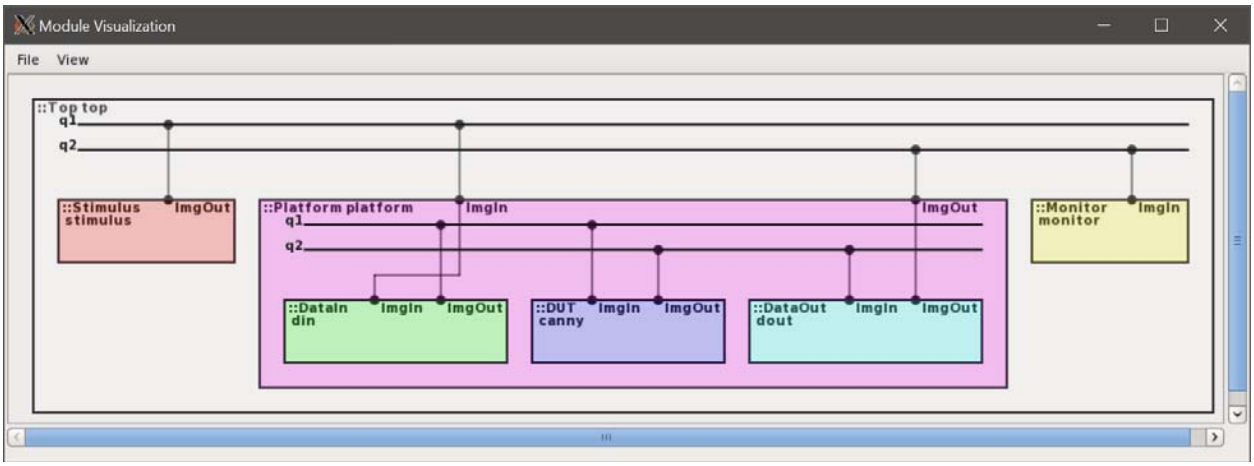
Hint 1: Visualize your model

My research group has created the Recoding Infrastructure for SystemC (RISC) which offers several tools for SystemC model analysis and parallel simulation. For this assignment, the RISC **`visual`** tool is very helpful, as it can automatically render a graphical display of the SystemC model structure.

The RISC v0.5.0 software package has been installed on our department servers. To use the RISC **`visual`** tool, setup your environment and use the tool on your model, as follows:

```
source /opt/pkg/risc_v0.5.0/bin/setup.csh
visual Canny.cpp
```

When properly structured and connected, your model at this stage should look as follows:



Hint 2: Increase stack sizes

As we have already seen in the previous Assignment 4, stack size is an issue that requires special attention due to the large image sizes we now are now stored in local variables. For SystemC models in particular, there are two considerations. First, the stack size of the root thread must be configured the same way as we did for regular programs in Assignment 4.

```
echo $SHELL
```

If you use the **csh** or **tcsh** shell, then adjust your root thread stack size as follows:

```
limit stacksize 128 megabytes
```

On the other hand, if you use the **sh** or **bash** shell, then set your root thread stack size like this:

```
ulimit -s 128000
```

Second, for every **SC_THREAD** in your model, you need to increase its stack size as well. In SystemC, this is accomplished by a statement **set_stack_size(128*1024*1024);** which directly follows the corresponding **SC_THREAD()** statement.

3. Submission:

For this assignment, submit the following deliverables:

```
Canny.cpp
Canny.txt
```

Again, the text file should briefly mention whether or not your efforts were successful and what (if any) problems you encountered. Please be brief!

To submit these files, change into the parent directory of your **hw5** directory and run the **~ecps203/bin/turnin.sh** script. As before, remember that you can use the turnin-script to

submit your work at any time before the deadline, *but not after!* Since you can submit as many times as you want (newer submissions will overwrite older ones), it is highly recommended to submit early and even incomplete work, in order to avoid missing the hard deadline.

Late submissions will not be considered!

To double-check that your submitted files have been received, you can run the **`~ecps203/bin/listfiles.py`** script.

For any technical questions, please use the course message board.

--
Rainer Doemer (EH3217, x4-9007, doemer@uci.edu)