EECS 222: Embedded System Modeling
Winter 2020

Assignment 5

Posted: February 4, 2020
Due: February 12, 2020 at 6pm

Topic: Structural test bench model of the Canny Edge Decoder

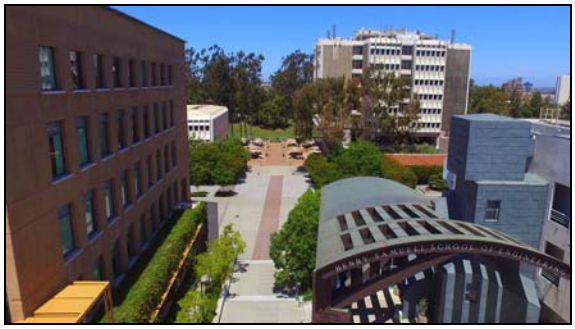
1. Setup:

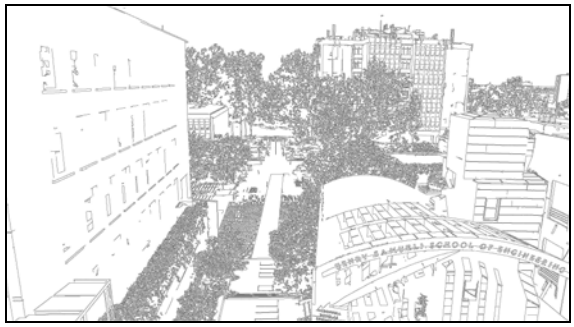
This assignment is the next step in modeling our application example, the Canny Edge Detector, as a proper system-level specification model which we can then use to design our SoC target implementation. In this assignment, we will create a model with a suitable top-level structural hierarchy including a test bench. We will also convert the application from single image processing to real-time video handling. More specifically, we will process a sequence of images extracted from a stream of video frames.

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
```

Instead of the previous golf cart image, we will from now on use a video captured by a drone hovering over the Engineering Plaza at UCI. Then again, we will convert the original video frames into edge images using the Canny algorithm.





The video is available in a shared directory on the server. To save disk space, do not copy the data into your account, but create a symbolic link to it, as follows:

```
ln –s ~eecs222/public/video video
```

The **video** directory contains images extracted from the video stream, as well as the actual movie file. The movie file is called **EngPlaza.mov**. Note that we do not have a working video player available on the EECS Linux servers (for copyright and network bandwidth reasons). However, you can easily download the movie file to your local laptop and play it there.

From the movie file, we have extracted 20 images as single frames and stored them as separate files in the **video** directory. For viewing purposes, you may use the JPEG and bitmap formats (e.g. **EngPlaza001.jpg** and **EngPlaza001.bmp**), but for our application we will use again the PGM greyscale format (i.e. **EngPlaza001.pgm**).

As in the previous assignment, you have the choice of using either SpecC or SystemC for your modeling. Both languages are equally capable of describing (and viewing) the intended top-level structural hierarchy in this assignment. Also, both simulation environments are equally capable to simulate your model in order to validate its functional correctness.

As starting point, you can use your own SLDL 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 ~eecs222/public/cannyA4_ref.sc canny.sc
cp ~eecs222/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 ~eecs222/public/MakefileA5SpecC ./
cp ~eecs222/public/MakefileA5SystemC ./
```

Depending on whether you choose SpecC or SystemC for your modeling, rename the corresponding file into the actual **Makefile** to be used by **make**.

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: Convert the application to process a stream of video frames

Instead of the previous golf cart image (input file "**golfcart.pgm**" and output file "**golfcart.pgm_s_0.60_l_0.30_h_0.80.pgm**"), we will now process the stream of video frames.

Adjust the model source code so that it processes 20 images in a loop. The input images are named "**`video/EngPlaza001.pgm`**" and so on, with increasing numbering. After processing the image, your model should output the generated edge image as "**`EngPlaza001_edges.pgm`**", and so on.

With these new file names in place, you should be able to simulate and check your model with a simple command: **`make test`**

HINT: Note that you will need to change the image size from formerly 320x240 pixels to the new higher resolution of 2704x1520 pixels produced by the drone camera. This higher image resolution naturally leads to higher memory usage of our application. Specifically, the array variables holding the image data (which we introduced in Assignment 4) grow large. Note that many of those variables are local variables which get allocated on the stack. At the same time, the stack size of a process in the Linux environment is typically limited. If so, your application will "crash" with a segmentation fault or similar memory error.

To avoid stack overflow, you can adjust the stack space allocation in your Linux shell. This configuration depends on the shell you are using, which you can identify with the following command:

```
echo $SHELL
```

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

```
limit stacksize 128 megabytes
```

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

```
ulimit -s 128000
```

With this larger stack allocation in place, your application model should run without memory problems. Run the Canny application on the prepared video frames and validate the generated edge images by comparing them against the reference images provided in the **`video`** directory.

Step 2: Add a test bench and platform structure to your SLDL model

The main goal 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 behavior **`Main`** (SpecC) or top-level module **`Top`** (SystemC). This will then consist of three blocks, namely **`Stimulus`**, **`Platform`**, and **`Monitor`**.

The **`Platform`** behavior/module, in turn, should contain a dedicated input unit **`DataIn`**, an output unit **`DataOut`**, and the actual design under test **`DUT`**.

For easy observation of the simulation, the **Stimulus** block should print a message “Stimulus sent frame” and the **Monitor** should print “Monitor received frame” in every iteration. Thus, your simulation should print the following log:

```
Stimulus sent frame  1.
Monitor received frame  1.
Stimulus sent frame  2.
Monitor received frame  2.
Stimulus sent frame  3.
Monitor received frame  3.
...
```

For communication, we will introduce queue-type channels from the respective SLDL channel library.

For SpecC modeling, we will use typed-queue channels (of size 1) to send and receive the image data between the behaviors. For reference, please see the simple example of a typed-queue channel in **~eecs222/public/queue.sc** which we have discussed in Lecture 5. As data type for the queue channels, please define the following:

```
typedef unsigned char img[SIZE]; // image data type
```

For SystemC modeling on the other hand, we will use the standard first-in-first-out channel **sc_fifo<IMAGE>** where **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, copy the **class IMAGE** from this provided file:

```
~eecs222/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 desired top-level structural hierarchy, a total of four channel instances will be needed, two at the test bench level (**`Main`** behavior or **`Top`** module), and two within the **`Platform`** behavior.

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

```
Main / Top
|------ Stimulus stimulus
|------ Platform platform
|       |------ DataIn din
|       |------ DUT canny
|       \------ DataOut dout
\------ Monitor monitor
```

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

In the **`Platform`**, the **`DataIn`** block should, in an endless loop, receive an input image and pass it unmodified to the **`DUT`**. Similar, the **`DataOut`** block 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`** block 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 SoC 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`** blocks.

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`** in SystemC), neither any global variables, nor any wait-for-time statements. For communication, only standard library channels should be used (no plain events or user-defined channels).

HINT: Please see also the Addendum to Assignment 5 posted next to these assignment instructions. The addendum provides instructions to helpful tools for SpecC and SystemC, which can visualize the model structure and connectivity.

HINT for SystemC: As indicated in the setup instructions above, stack size is an issue that requires special attention due to the large image sizes in local variables. For SystemC models in particular, we need to consider not only the root thread (which derives its stack size from the surrounding shell), we must also adjust the SystemC worker threads accordingly. Specifically, for every **`SC_THREAD`** in your model, you need to increase its stack size. This can be accomplished by a statement **`set_stack_size(128*1024*1024);`** which directly follows each **`SC_THREAD()`** statement.

3. Submission:

For this assignment, submit the following deliverables:

`canny.sc` *or* **`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 **`~eecs222/bin/turnin.sh`** script. As before, note that the submission script will ask for both the SystemC and SpecC models, but you need to submit only the one that you have chosen for your modeling.

Finally, 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 **`~eecs222/bin/listfiles.py`** script.

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

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