Embedded Systems Modeling and Design
ECPS 203
Fall 2019

Assignment 6

Posted: November 6, 2019
Due: November 13, 2019 at 6pm

Topic: Structural refinement of the design-under-test module and algorithm profiling

1. Setup:

This assignment continues the modeling of our application example, the Canny Edge Detector, as an embedded system specification model. In this project assignment, we will refine the previous model with a suitable structural hierarchy inside the design-under-test (DUT) module. We will then use this model for initial performance profiling in order to identify the functions with the highest computational complexity.

Again, we will use the same setup as for the previous assignments. Start by creating a new working directory, so that you can properly submit your deliverables in the end.

```
mkdir hw6
cd hw6
```

For functional validation, create again a symbolic link to your or the provided video stream files, as follows:

```
ln -s ~ecps203/public/video video
```

As starting point, you can use your own SystemC model which you have created in the previous Assignment 5. Alternatively, you may start from the provided solution for Assignment 5 which you can copy as follows:

```
cp ~ecps203/public/CannyA5_ref.cpp Canny.cpp
```

You may also want to reuse the simple **`Makefile`** from the previous assignment (which does not need any modification for this assignment):

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

Finally, you probably will need to adjust your stack size in your shell again. 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
```

2. Structural DUT refinement and Canny algorithm profiling

Step 1: Create an additional level of hierarchy in the DUT

The original **`canny`** function consists of a sequence of function calls to five functions, namely **`gaussian_smooth`**, **`derivative_x_y`**, **`magnitude_x_y`**, **`non_max_supp`**, and **`apply_hysteresis`**. While in the previous model all these are local methods in the DUT, we will now encapsulate them into separate modules by themselves.

The expected module instance tree of the **`Platform`** block should then look like this:

```
Platform platform
|------ DataIn din
|------ DUT canny
|       |------ Gaussian_Smooth gaussian_smooth
|       |------ Derivative_X_Y derivative_x_y
|       |------ Magnitude_X_Y magnitude_x_y
|       |------ Non_Max_Supp non_max_supp
|       \------ Apply_Hysteresis apply_hysteresis
\------ DataOut dout
```

The **`Canny`** module should be a concurrent composition of its children where each child module will have its own **`SC_THREAD`**. For communication, the encapsulated modules should be connected by ports mapped to connecting channels. As discussed in Lecture 12, the new channels which will be of either **`sc_fifo<IMAGE>`** type, or of **`sc_fifo<SIMAGE>`** type where **`SIMAGE`** is based on **`short int`** pixel representation. All channel instances should have a buffer size of 1 element.

Since FIFO channels use uni-directional communication, be sure to use suitable ports with directions **`sc_fifo_in`** or **`sc_fifo_out`**. Also, since some intermediate images in the Canny algorithm are generated by one function and then used by multiple others, you may need to duplicate some channel instances.

After this level of hierarchy has been added, you should compile and simulate your model to ensure functional correctness.

Step 2: Create an additional level of hierarchy in the Gaussian Smooth module

The Gaussian Smooth function consists of several tasks that we will wrap into four separate child modules, namely **`Receive_Image`**, **`Gaussian_Kernel`**, **`BlurX`**, and **`BlurY`**. The module instance tree of the new **`DUT`** should then look like this:

```
DUT canny
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       |------ Gaussian_Kernel gauss
|       |------ BlurX blurX
|       \------ BlurY blurY
|------ Derivative_X_Y derivative_x_y
```

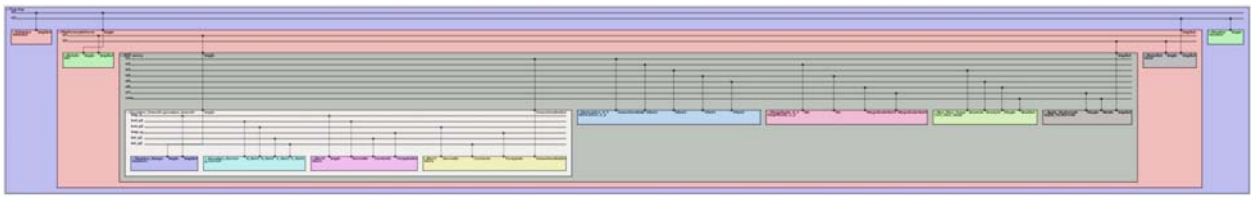
```
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

As in the previous step, create this level of hierarchy as concurrent modules with appropriate interconnections consisting of port-mapped **`sc_fifo`** channels.

Complete this step by validating that your refined model still compiles, simulates, and generates the correct output images. Ensure also that your code still compiles cleanly without any errors or warnings.

Hint1: Visualize your model

As in the previous Assignment 5, you can double-check your model's structure by using the RISC **`visual`** tool. Your model should look similar to the following figure.



Step 3: Profile the Canny algorithm

For an initial performance analysis of our Canny Edge Detector model, it is critical to identify the computational complexity of its main functions. In other words, we want to find out which functions can become a bottleneck in the implementation. To this end, we will profile our design model in this step.

For the SystemC model, we will use the profiling tools provided by the GNU community, namely **`gprof`**. In order to use the GNU profiler, you need to instrument your model (prepare it for profiling) by supplying the **`-pg`** option to the GNU compiler **`g++`**. This will result in extra instrumentation inserted into the executable which performs timing measurements.

After compilation, run your executable file once, just as you would for regular simulation. This will produce a file **`gmon.out`** with profiling statistics that you can then analyze with the following command:

```
gprof Canny
```

This command generates a detailed profiling report in textual format where you can inspect the function call tree and other results. For the computational complexity we are interested in, see the "flat profile" in the report.

At this point in our design process, we are only interested in the relative computational load of the functions in the DUT. We intentionally want to ignore all computation performed by the functions in the test bench. Thus, assuming the total DUT workload is 100%, we want to find out how much load each of the DUT functions contributes.

For this assignment, obtain the relative workload of the DUT functions and fill the results as percentage values into the following complexity comparison table:

```
Gaussian_Smooth                              ...%
|------ Gaussian_Kernel    ...%
|------ BlurX              ...%
\------ BlurY              ...%
Derivative_X_Y                               ...%
Magnitude_X_Y                                ...%
Non_Max_Supp                                 ...%
Apply_Hysteresis                             ...%
                                             ____
                                             100%
```

Submit the filled table in your text file **Canny.txt** with a brief explanation of how you obtained these results.

3. Submission:

For this assignment, submit the following deliverables:

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

As before, the text file should briefly mention whether or not your efforts were successful and what (if any) problems you encountered. In addition, include the profiling comparison table and a brief explanation.

To submit these files, change into the parent directory of your **hw6** directory and run the **~ecps203/bin/turnin.sh** script.

As before, be sure to submit on time. 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)