EECS 222: Embedded System Modeling
Spring 2017

Assignment 4

Posted: May 1, 2017
Due: May 8, 2017 at 12pm (noon)

Topic: SLDL Model of the Canny Edge Decoder

1. Setup:

After we have gained some initial experience with both the SpecC and SystemC system-level description languages in Assignment 2 and Assignment 3, respectively, we will from now on focus on our application example introduced in Assignment 1, namely the *Canny Edge Detector*. Based on the initial study of the algorithm in the beginning, it is now time to create a proper system-level specification model which we can then use to design our SoC target implementation. In this assignment, we will create the initial model specified in a system-level description language (SLDL) which can be simulated for validation. For this, we will need to perform some code cleaning and remove the used dynamic memory allocation.

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 hw4
cd hw4
```

You have the choice of using either SpecC or SystemC for your modeling in this assignment. Both languages are equally capable of describing the clean SLDL model targeted in this assignment. Also, both simulation environments are equally able to simulate your model in order to validate its functional correctness.

If you choose SpecC, then use the SpecC environment installed in this directory:

```
/opt/sce/
```

If you choose SystemC, then use the SystemC library installed in this directory:

```
/opt/pkg/systemc-2.3.1/
```

Please refer to the prior assignments 2 and 3, respectively, for more detailed setup instructions on the language of your choice.

2. Creating a clean SLDL Model for Simulation

As starting point, we will use the single-file ANSI-C source code of the Canny Edge Detector which you have prepared in Assignment 1.

Step 1: Bug fix in **`non_max_supp`** function

Unfortunately, the orginal Canny implementation by Mike Heath contains a bug that we should fix from the beginning. Specifically, the suppression of non-maximum points in the algorithm incorrectly omits a column of pixels at the right and a row of pixels at the bottom of the image. Although it is a classic off-by-one bug in loop iterations, this bug is difficult to find without clear understanding of the internals of the algorithm.

So we will provide the bug fix here. You may copy the patched source code file from this directory on the server:

```
~eecs222/public/canny.c
```

Please compare the provided bug-fix file to your own file from Assignment 1. You will find the bug fix applied in the **`non_max_supp`** function.

Step 2: Clean-up the source code so that there are no compilation warnings

Rename the **`canny.c`** file into an initial SpecC file **`canny.sc`** or SystemC file **`canny.cpp`**, depending on which SLDL you want to use. Then try compiling it with the chosen compiler, while enabling all warnings the compiler has to offer (i.e. use option **`-ww`** with **`scc`** or **`-Wall`** with **`g++`**).

You will need to apply a few patches to the source code so that there are no errors and no warnings during the compilation. This probably will require a few iterations of source code adjustment, but this investment will pay off many times in the end. There is nothing worse than chasing a nasty bug later, when the compiler already points out a bad line in the source code at the beginning!

One note specific to the SpecC compiler **`scc`** is in order, because **`scc`** has some known limitations in its current academic version: In particular, **`scc`** only supports initialization of variables with expressions that are constants at compile time. For example, if you get error #2028 “Expression not constant”, then this is likely a situation like this:

```
char *infilename = NULL;
```

The macro **`NULL`** is not clean in the Linux headers. It should be converted to a plain zero, as follows:

```
char *infilename = 0;
```

You will also notice that **scc** is not as forgiving as **gcc** in terms of type mismatches, and is also more strict in proper declaration of variables and functions before they can be used. A good rule of thumb is that your code should be as clean as possible! This is not only to make the compiler happy, but more so to make your system design flow successful, which will make you happy!

While it is not a requirement (no deliverable) for this assignment, it is highly recommended that you create a suitable **Makefile** for building your application model. This will greatly simplify your compilation and testing iterations.

You are done with this step when your source code compiles fine without errors or warnings and the executable properly creates the output image with the correct edges. Please compare the output image against the one produced in Assignment 1. The only difference should be the pixels at the very right and bottom of the image, which should look better now, due to the bug fix.

Step 3: Fix the application parameters for synthesis

In order to synthesize your model later into an actual hardware chip, you need to decide on the configuration parameters which are flexible in the initial software, but must become fixed constants for the SoC implementation. For example, dynamic memory allocation (i.e. **malloc()**, **calloc()**, and **free()**) is not feasible in a hardware implementation (your SoC cannot instantiate a new memory chip at runtime!). Instead, we will use static arrays with fixed sizes at compile time. Also, command-line parameters, such as the file name, can only be passed to a test bench, not to the actual SoC model.

In your **canny.sc** or **canny.cpp** model, refine the source code such that the following configuration parameters become hard-coded constants:

```
rows = 240
cols = 320
sigma = 0.6
tlow  = 0.3
thigh = 0.8
```

For the file name, you can either leave it as a command-line argument (recommended if you want to process other images later), or hard-code it also, as follows:

```
infilename = "golfcart.pgm"
```

At the same time, we need to remove all dynamic memory allocation from the algorithm. We suggest to start with replacing the **malloc()** and corresponding **free()** calls (and ignore **calloc()** for this step). You will notice that there are only four **malloc()** calls in the entire source code. Three of those are actually

never used, so you can easily remove them. Also, remove all functions from the code that are not used (Hint: our image is a grey-scale image!).

The one remaining **`malloc()`** and the corresponding **`free()`** call should be replaced with the use of an array with fixed size. Double-check your model so that it still simulates correctly after removing the **`malloc()`** and **`free()`** calls. You may also want to create a backup file, before you apply the source code modifications in the next step.

Step 4: Remove or replace all dynamic memory allocation

Last but not least, remove or replace the function calls to **`calloc()`** and corresponding **`free()`** from the source code. Again, we will use arrays with static sizes instead.

Hint 1: In function **`make_gaussion_kernel`**, an array **`kernel`** is filled with parameters. The size of this array (variable **`window_size`**) generally depends on the configuration parameter **`sigma`**. However, since we set **`sigma`** to a constant value in the previous step, **`window_size`** also becomes a fixed value. Here, you can replace **`window_size`** with the constant **`WINSIZE=21`**.

Hint 2: The two functions **`radian_direction`** and **`angle_radians`** in the original Canny implementation are useful to demonstrate the working of the algorithm (the resulting gradient direction image can be output to a file and then viewed). However, this functionality serves no purpose in our SoC model where we are only interested in the final edge image. Thus, you can safely remove both functions (and the included dynamic memory allocation) from the source code of your model.

3. Submission:

For this assignment, submit the following deliverables:

`Canny.sc` *or* **`Canny.cpp`**
`Canny.txt`

The text file should briefly describe whether or not your efforts were successful and what (if any) problems you encountered. We will take this input into account when grading your submission. Please be brief!

To submit your deliverables, change into the parent directory of your **`hw4`** directory and run the **`~eecs222/bin/turnin.sh`** script. Again, this command will locate the current assignment files and allow you to submit them, just 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 Doemer (EH3217, x4-9007, doemer@uci.edu)