EECS 222A
System-on-Chip Description and Modeling
Spring 2012

Assignment 2

Posted:	April 20, 2012
Due:	April 27, 2012 at 12pm (noon)
Topic:	Introduction to Application Example

1. Setup:

We will use the same Linux account and the same remote server as for Assignment 1.

For this and the following assignments, however, we will also use tools with GU (graphical user interface). For this to work, you will need some X client software, for example Xming for the Windows platform. Please refer to the course web page at https://eee.uci.edu/12s/18422/resources.html for hints on how to install the X server and how to establish the X communication.

We will also use a new version of the SpecC tools for this assignment (in fact, the most recent alpha version). To use this version, setup your Linux environment as follows:

```
source /opt/sce/bin/setup.csh
```

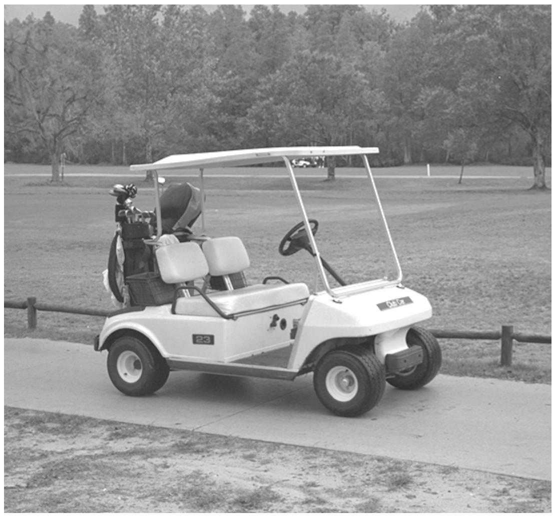
Finally, we will use the same **turnin** command for the submission of deliverables as in the previous assignment. So, please create a new directory named **hw2** (next to your **hw1** directory) and work in there:

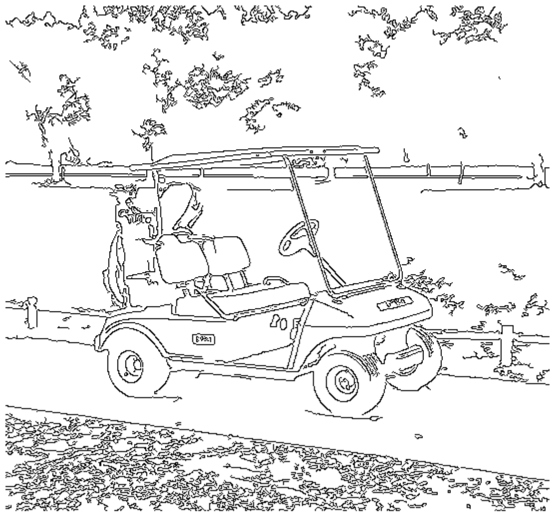
```
mkdir hw/hw2
cd hw/hw2
```

2. Application Example

For the System-on-Chip modeling project in this course, we will use an image processing application, namely a *Canny Edge Detector* algorithm. The project goal is to design a model of this application and describe it in the SpecC system-level description language so that the created design model can be used for implementation as a System-on-Chip (SoC) suitable for use in a digital camera.

The Canny Edge Detector algorithm takes an input image, e.g. a digital photo, and calculates an output image that shows only the edges of the objects in the photo, as illustrated in the figure below. We will assume that this image processing is to be performed in real-time in a digital camera by a SoC that we design and develop.





Please refer to the following sources for more information on the application:

http://en.wikipedia.org/wiki/Canny_edge_detector

http://marathon.csee.usf.edu/edge/edge_detection.html

3. Tools

In the following sections, we'll provide some hints on helpful Linux tools you can use for this and the following assignments.

3.1 Image Tools:

There are many command-line tools available in Linux that allow you to manipulate images. Most start with **`pnm`**, **`pbm`**, or **`pgm`**, depending on the file format they process. To enumerate them, you can type their prefix into your shell followed by a **`TAB`**. Documentation is available via corresponding **`man`** pages.

In addition, graphical tools are available as well (if you have an X server running). These may be more convenient for occasional use.

To view images, you can use **`eog`** which supports most image types (including our pgm format).

To manipulate images (e.g. if you want to use your own photos as test case for our application), you can use **`gimp`**, a very powerful image editor.

3.2 Source Code Editor:

To convert the C reference code into an executable SpecC model, you may use any text editor of your choice and use the SpecC compiler via the command line interface as in Assignment 1. However, we offer as an alternative an extended version of *Eclipse*, an open source IDE, which includes specific support for SpecC projects.

Currently, supported features include:

(a) SpecC syntax highlighting: SpecC keywords are colored in the editor to increase readability of the code.

(b) Automatic compiling on save: When the SpecC source file is saved (e.g. by pressing Ctrl-S), the code is automatically compiled, and any error messages are displayed in case of unsuccessful compiling.

If you choose to use Eclipse, please follow the following steps:

1. Setup environment: **`source /opt/sce/bin/setup.csh`** (same as above)

2. Start Eclipse: **`eclipse`**

The tool takes some time to start up and asks you for your workspace path.

3. Create a new C++ project:

Select on menu: **`File -> New -> (expand)C/C++ -> C++ Project`**, and click **`Next`**. In the next window, types in a project name, and for project type choose **`Makefile Project -> Empty Project`**, and then click **`Finish`**.

Now a project folder with your selected name appears in the project view.

4. Create .sc file(s):

You can either copy an existing file into your project folder or right-click on the project folder, select **`New -> File`** and start with an empty file. Note that the file name should end with a **`.sc`** extension for our case. At this point, you can then edit your SpecC code in Eclipse.

Some more notes on our extended Eclipse:

A. The tool is still under development! While not every feature is complete (and crashes are possible), we are confident that the current version can increase your productivity.

B. We appreciate your feedback! If you spot a bug, please consider posting a problem report to the message board or email it directly to **hanx@uci.edu**, so that we can reproduce and attempt to fix it.

C. If you occasionally encounter a start-up problem, try: **eclipse -clean**

D. Since our extension is based on the original C++ editor, it may give you false warnings of unrecognized SpecC syntax (shown as question marks). To turn this off, select on the menu: **Window -> Preferences ->(expand)C/C++ -> (expand)Editor -> Hovers**, and uncheck **Enable editor problem annotation**.

4. Instructions

The purpose of this assignment is for you to become familiar with the Canny application source code and prepare it for use with the SpecC tool suite.

Step 1: Download the application source code

You can download the Canny application source code from the web site at ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src .

Alternatively, you can copy the same source file and a suitable input image from our course account:

```
cd hw/hw2
cp ~eecs222/EECS222A_S12/canny.src .
cp ~eecs222/EECS222A_S12/golfcart.pgm .
```

We will use this C reference implementation of the Canny algorithm as the starting point for our design model and the golf cart image as test case.

Step 2: Test the given C code

Convert the **canny.src** file into a *single* ANSI-C file **canny.c**. Few adjustments will be necessary, then you can compile and test the application, similar to the following:

```
vi canny.c
gcc canny.c -lm -o canny
./canny golfcart.pgm 0.6 0.3 0.8
eog golfcart.pgm_s_0.60_l_0.30_h_0.80.pgm
```

The generated output image should look similar to the one shown above.

Step 3: Study the structure of the application

Before we go and write a system-level model of the application, we need to study and understand it well. Examine the source code and its execution! You should be able to answer questions like the following:

What are the main functions of the algorithm? Which functions are used for data input and for data output? Where does the actual computation occur? Which of the functions is the one with the most complexity? Can we expect the application to run in real-time as required by our overall goals? How much memory is needed when the algorithm runs? Are there any obvious candidates for hardware acceleration? What should better be performed in software? …

There is nothing to submit for these questions, but be prepared to discuss these issues in class.

Step 4: Create an executable SpecC source file

Please time yourself for this step. At the end, we would like to know how many minutes this step took for you. Thanks!

Copy the **`canny.c`** file into an initial SpecC file **`canny.sc`**. Next, for the SpecC compiler to process this file, there are a few additional adjustments necessary (due to limitations of the current **`scc`** implementation).

-> edit the file **`canny.sc`** (using eclipse or other editor)
-> compile it (e.g. **`scc canny –vv –ww`**)
-> watch for any warnings and errors; if so, fix and repeat...

Note that at this time, we will not yet introduce behaviors or channels into the file (execution will still start from the global **`main`** function). We only want to make the code compliant to the limitations of **`scc`**.

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 case that can be fixed as follows:

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

should be converted to

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

or simply

```
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 referenced/used. A good rule of thumb is that your code should be as clean as possible to make **`scc`** happy.

You are done with this step when your code compiles fine without errors or warnings. Please note the time when you are done.

Step 5: Fix parameters for synthesis

Please time yourself for this step. At the end, we would like to know how many minutes this step took for you. Thanks!

In order to synthesize our model later into an actual chip, we need to decide on certain parameters, which are flexible in the initial software, to become fixed constants for the SoC implementation. For example, dynamic memory allocation (i.e. **`malloc()`**and **`free()`**) is not feasible to be implemented. Instead, we need to use static arrays with fixed size 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.

In your **`canny.sc`**, refine the source code such that the following 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), or hard-code it as follows:

```
infilename = “golfcart.pgm”
```

At the same time, we need to remove all dynamic memory allocation from the algorithm. To simplify this assignment, however, we will ignore all **`calloc()`** calls, and only remove **`malloc()`** and the corresponding **`free()`** calls. You will notice that there are only four **`malloc()`** calls in the entire code. Three of those are actually never used, so you can simply remove those functions.

The remaining **`malloc()`** and the corresponding **`free()`** call should be removed and replaced with the use of a static array.

That’s all the needed changes in the code for this assignment (please note the time it took!), but we will continue with the project in the next one where we will introduce a proper hierarchical structure for system synthesis.

3. Submission:

For this assignment, submit the following deliverables:

```
Canny.sc
Canny.txt
```

The text file should briefly mention whether or not your efforts were successful and what (if any) problems you encountered. Be brief!

To submit the deliverables, change into the parent directory of your **`hw2`** directory and enter **`turnin`**. As in the previous assignment, the **`turnin`** command will locate the files listed above and allow you to submit them.

Remember that you can use the **`turnin`** tool to submit 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 deadline.

Late submissions cannot be considered!

In addition to these deliverables, we would like to ask you to complete a short survey on your experience with the extended **`eclipse`**. Please check the course message board for this survey.

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