EECS 222: Embedded System Modeling
Spring 2017

Assignment 8 and Final Report

Posted: June 5, 2017
Due: June 14, 2017 at 6pm

Topic: Throughput optimization of the Canny Edge Decoder

1. Setup:

This assignment is the final chapter in the modeling of our application example, the Canny Edge Detector, as a system-level specification model suitable for SoC implementation. Here, we will optimize the pipelined DUT model obtained in the previous assignment so that the pipeline stages are better balanced and therefore the throughput of the design is improved.

Again, we will use the same setup as for the previous assignments. Start by creating a new working directory with a link to the video files.

```
mkdir hw8
cd hw8
ln –s ~eecs222/public/video video
```

As before, you have the choice of using either SpecC or SystemC for your modeling. However, we will rely on the SpecC-based System-on-Chip Environment (SCE) for the estimation of the stage delays. So if your modeling language is SystemC, please follow the workaround indicated below. There is little extra work expected for the workaround.

If you use the `csh` or `tcsh` shell, setup the SpecC environment as follows:

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

Otherwise, if you use the `sh` or `bash` shell, then setup the SpecC environment like this (note the dot command at the beginning!):

```
. /opt/sce/bin/setup.sh
```

As starting point, you can use your own SLDL model which you have created in the previous Assignment 7. Alternatively, you may start from the provided solution files for Assignment 7 which you can copy as follows:

```
cp ~eecs222/public/CannyA7_ref.sc Canny.sc
cp ~eecs222/public/CannyA7_ref.cpp Canny.cpp
```

For SystemC designers, please copy also the SpecC model, as you will need that for the workaround.

You may also want to reuse the **`Makefile`** from the previous assignments:

```
cp ~eecs222/public/MakefileA4SpecC ./
cp ~eecs222/public/MakefileA4SystemC ./
```

Again, depending on whether you design in SpecC or SystemC, rename the corresponding file into the actual **`Makefile`** to be used by **`make`**.

Finally, we will use one more tool named **`ImageDiff`** in this assignment which you can copy as well:

```
cp ~eecs222/public/ImageDiff ./
```

We will use this tool for comparing the generated images instead of the previously used Linux **`diff`** tool, as outlined in the instructions below.

2. Throughput optimization of the Canny Edge Decoder model

Step 1: Improve the test bench to include the logging of frame throughput

As discussed in Lecture 18, we are mostly interested in observing the performance of our model by means of its throughput, i.e. the frames per second (FPS) coming out of the processing pipeline. To measure this, we will extend the timing logs produced by the test bench.

Specifically, we let the Monitor block measure and report the frame throughput upon receiving a new frame. A sample output fragment shall look like this:

```
[...]
  2823125: Monitor received frame  1 with  2823125 us delay.
  2823125:    2.823 seconds after previous frame,  0.354 FPS.
  2823125: Stimulus sent frame 12.
  4183125: Monitor received frame  2 with  4183125 us delay.
  4183125:    1.360 seconds after previous frame,  0.735 FPS.
[...]
```

The frame throughput is observed by measuring the arrival time of two consecutive frames and calculating the difference of the two time stamps. Converted to seconds, the reciprocal value is the desired FPS result.

Adjust your model to print the extra log line for each received frame. Keep a copy of the model at this stage, say **`CannyA8_step1`**, so that you can compare the observed timing with the following improvements.

Step 2: Estimate the timing delays for allocated PEs in each pipeline stage

In the previous Assignment 7, we used the timing estimates obtained by SCE when using a single ARM_926EJS processor (slide 11 of Lecture 15). We also assumed that we can improve the default clock frequency of 100 MHz by a factor of 10 to 1.0 GHz. Specifically, we back-annotated the following delays for the seven stages in the DUT pipeline:

1. **Gaussian_Smooth**	**0 ms**
2. **BlurX**	**41680/20/8 ms**
3. **BlurY**	**43740/20/8 ms**
4. **Derivative_X_Y**	**5340/20 ms**
5. **Magnitude_X_Y**	**5340/20 ms**
6. **Non_Max_Supp**	**27200/20 ms**
7. **Apply_Hysteresis**	**7650/20 ms**

We will now refine our model with more realistic assumptions. That is, we will assign suitable processing elements (PEs) to each pipeline stage. Specifically, we will aim at a system architecture of a 4-core ARM9-based multi-core processor assisted by two Application Specific Integrated Circuit (ASIC) units as hardware accelerators. To connect our SoC platform to the outside world, we will also use two I/O units with Direct Memory Access (DMA) capability (which we model at high abstraction level as virtual hardware components, since I/O details are beyond the scope of this EECS 222 course).

We will use SCE for the allocation of the PEs and corresponding estimation of the stage delays. If you use SystemC for your modeling, use the provided Assignment 7 solution **CannyA7_ref.sc** as workaround in this step.

In SCE, import your model and add it to a new project "A8" as model "A8step2". Compile and simulate the model in SCE. Then run the SCE profiler to obtain the execution counts for each block in the model. Next, allocate the following PEs:

4 **ARM_926EJS** at 1.0 GHz, named **ARM9x10core1** … **ARM9x10core4**
2 **HW_Standard** at 500 MHz, named **ASIC1** and **ASIC2**
2 **HW_Virtual** at 1.0 GHz, named **DMA1** and **DMA2**

Note that the **ARM_926EJS** processor in the SCE database is limited to maximal 500 MHz, but we will assume a more modern 4-core CPU (modeled as 4 PEs) with higher clock frequency. To work around the database limit, allocate a **ARM_926EJS** PE with default configuration (100 MHz), then click **Tables** in the yellow PE allocation window, and adjust the overall profiling **Factor** from **1.0** to **0.1** in order to imitate a 10 times faster execution of the operations. Click **OK** to confirm this configuration and use **Copy** to create the other 3 cores with the same parameters.

For both the standard and virtual hardware PEs, you can directly adjust their clock frequency in the dialog box to the desired values, as these are within the limits specified in the SCE database.

Next, map the blocks in the Canny platform to the allocated PEs, as follows:

`DataIn`	**`DMA1`**
`DUT`	**`ARM9x10core1`**
`BlurX`	**`ASIC1`**
`BlurY`	**`ASIC2`**
`Magnitude_X_Y`	**`ARM9x10core2`**
`Non_Max_Supp`	**`ARM9x10core3`**
`Apply_Hysteresis`	**`ARM9x10core4`**
`DataOut`	**`DMA2`**

With this more realistic allocation and mapping, use the SCE profiler to estimate and evaluate the performance (**`Validation->Evaluate`**, then **`Validation->Show Estimates`**). To view the estimation results in a bar chart, select **`gaussian_smooth`**, **`sliceX8`**, **`sliceY8`**, **`derivative_x_y`**, **`magnitude_x_y`**, **`non_max_supp`** and **`apply_hysteresis`** to represent the pipeline stages and generate the computation profile graph (**`Graph->Computation`**). For your final report (see below), take a screenshot of this bar chart. Note the badly balanced load for the pipeline stages!

Further, double-click on the highest bar to get a pie chart of the type of the operations performed in the corresponding block. Then double-click on the ALU operations and note the ratio of integer vs. floating-point operations. Take note of the relation. Also take a screenshot of this pie chart for your report, so that you can easily explain the following steps taken below.

To reflect the new stage delays in your own model, look up the updated numerical estimation results in the SCE window. Here, when you select the parent of a given block, you can view its estimated delay in the tab corresponding to the selected PE for this block. Be sure to save the model and current project in SCE, as you will need it again in the next step.

We expect the estimated timing for the ARM9x10 cores to remain the same values as before (as in Assignment 7), but the stage delays for the individual blur slice blocks will be different since these are now mapped to hardware PEs. Take note of these values and back-annotate them into the source code of your model.

Keep a copy of your model with the updated timing at this point and name it **`CannyA8_step2`**. Then compile and simulate it. Does the timing shown in the simulation log differ from the log of **`CannyA8_step1`**? Why is that the case? Discuss this question in your final report.

Step 3: Replace floating-point arithmetic with fixed-point calculations (NMS only)

In order to improve the throughput of our pipeline in the DUT, we need to optimize the stage(s) with the longest stage delay. As the profiling results in SCE show (in the pie chart), floating-point calculations can be a significant bottleneck in the overall performance. Sometimes those can be replaced by faster and cheaper fixed-point arithmetic with acceptable loss in accuracy. One example of this situation is the `Non_Max_Supp` block in the Canny algorithm where we can easily apply this optimization. (Generally, this technique can be applied also to other components, but we will limit our efforts to only the `Non_Max_Supp` block in this assignment.)

Find the `non_max_supp` function in the source code of your model. Identify those variables and statements which use floating-point (i.e. `float` type) operations. There are only 4 variables defined with floating-point type. Change their type to integer (`int`).

Next, we need to adjust all calculations that involve these variables. In particular, we need to add appropriate shift-operations so that the integer variables can represent fixed-point values within appropriate ranges. Since the details of such arithmetic transformations are beyond the scope of this course, we provide specific instructions here.

Locate the following two lines of code:

```
xperp = -(gx = *gxptr)/((float)m00);
yperp = (gy = *gyptr)/((float)m00);
```

Then, comment those lines out and insert the following replacement statements:

```
gx = *gxptr;
gy = *gyptr;
xperp = -(gx<<16)/m00;
yperp = (gy<<16)/m00;
```

If you are using SystemC for your modeling, apply these source code changes to both your own model as well as the SpecC model for use in SCE.

To ensure functional correctness, compile and simulate your model. However, don't be disappointed if your `make test` fails! Note that the previous `Makefile` compares the generated frames against the reference images and expects exact matches. Our arithmetic transformation, however, is not guaranteed to be exact. It is only an approximation!

In order to determine whether or not fixed-point arithmetic is acceptable for our application, we need to compare the image quality. You can do this by looking at them (e.g. use `eog` to display them on your screen), or better by using the provided **`ImageDiff`** tool. This command-line tool is built from Canny source code functions and compares the individual pixels of two input images (first and second argument) and generates an output image (third argument) which shows the differences. It also reports the number of mismatching pixels found. For example, use **`ImageDiff`** as follows:

```
./ImageDiff Frame.pgm video/Frame.pgm diff.pgm
```

You may want to adjust your **`Makefile`** so that the previously used Linux **`diff`** command is replaced by using the **`ImageDiff`** tool instead.

Discuss in your final report whether or not you find the changes incurred due to the use of fixed-point arithmetic acceptable for our edge detection application.

Again, keep a copy of your model at this point and name it **`CannyA8_step3`**.

Step 4: Estimate and back-annotate the improved stage delay for the NMS block

In order to gauge the timing effects of using fixed-point arithmetic in the **`non_max_supp`** calculations, we will use again the SCE profiling and evaluation tools. Import your new model into SCE and add it to the "A8" project as model "A8step4". Compile, simulate, and profile it as above in Step 2.

Next, allocate the same PEs and perform the same mapping as in Step 2. Then evaluate the improved performance and view the estimation results in the same bar chart showing the new computation profile. Note the better balanced workload for the pipeline stages!

For your final report, take again a screenshot of the bar chart. Also create the pie chart of integer vs. floating-point operations in the NMS block and take a screenshot of it for your report.

Finally, to reflect the new NMS stage delay in your own model, look up the updated numerical estimation results. Select the parent of the NMS block and the **`ARM9x10core3`** tab so that you can view the newly estimated delay. Save the model and current project in SCE. Then back-annotate the obtained NMS stage delay into the source code of your model.

Compile and simulate this final model **`CannyA8`**. The timing shown in the simulation log is expected to be better than for any of your previous models. Discuss this result in your final report. If the throughput is not satisfactory yet for the real-time display of edge images in a video camera, discuss (but don't implement) any additional modifications that could be taken to further improve the performance, and/or any changes in the use of the application that could be acceptable to the end user.

3. Project Report

In lieu of a final exam in this course, we will use a technical report about the project. This project report shall cover all the steps taken in our design of the embedded system model of the Canny Edge Detector. That is, the report shall describe and document all the modeling tasks performed from the starting point of downloading the application source code (Assignment 1) down to creating the final model obtained in this assignment (Assignments 4 through 8), and provide the background and reasoning for taking the performed steps and design decisions.

Your final technical project report shall carry the title *“Specification and Modeling of a Canny Edge Detector for System-on-Chip Design”* and subtitle of the course number and course title. The contents of the report shall describe the overall “story” of our project, all the way from downloading the initial C reference code via describing and simulating the application in SpecC or SystemC SLDL, to our modeling and optimization for top-down SoC design. Finally, the report should conclude with a summary of the lessons learned.

Specifically, the technical report should be structured as follows:

1. Title page
    - Project title, author, date, course number and title
    - Abstract
2. Introduction
    - System-level modeling and design
    - Essential concepts and coverage in SpecC/SystemC SLDL
3. Case study using the Canny Edge Detector application
    - Obtaining and studying the Canny application
    - Creating a simulatable model in SpecC/SystemC SLDL
    - Creating structural hierarchy with test bench
    - Pipelining and parallelization
    - Performance estimation and throughput optimization
4. Summary and Conclusion
    - Lessons Learned
    - Future work
5. References

As a guideline, your project report should contain about 12 pages in length, including the title page, all figures and tables, and appropriate references.

4. Submission:

For this assignment, submit the following deliverables:

> **`Canny.sc`** *or* **`Canny.cpp`**
> **`EECS222_Report.pdf`**

To submit these files, change into the parent directory of your **`hw8`** 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.

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

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

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