Embedded Systems Modeling and Design
ECPS 203
Fall 2018

Assignment 8

Posted: November 21, 2018
Due: November 28, 2018 at 6pm

Topic: Pipelined and parallel design-under-test module of the Canny Edge Decoder

1. Setup:

This assignment continues the modeling of our application example, the Canny Edge Detector, towards an embedded system implementation. This time we will refine the previous SystemC model with back-annotated timing and further pipeline and parallelize the components in the design-under-test (DUT) module. Over the course of the steps outlined below, our design model will be refined from an untimed model into one with estimated delays where the simulation allows us to observe the improved performance due to pipelining and parallelization.

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 ~ecps203/public/video video
```

As starting point, you can use your own SystemC model which you have created in the earlier Assignment 6 (not the C++ model from Assignment 7). Alternatively, you may start from the provided solution for Assignment 6 which you can copy as follows:

```
cp ~ecps203/public/CannyA6_ref.cpp Canny.cpp
```

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

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

Finally, you will need to adjust your stack size again. If you use the **`csh`** or **`tcsh`** shell, use:

```
limit stacksize 128 megabytes
```

On the other hand, if you use the **`sh`** or **`bash`** shell, use this:

```
ulimit -s 128000
```

2. Pipelining and Parallelization of the Canny Model

In order to observe the performance of the Canny application in the simulator, we need to insert statements into the test bench for monitoring the simulated time (Step 1 and Step 3) and also instrument the model in the DUT with the estimated delays from Assignment 7 (Step 2). Then we observe the effects of pipelining (Step 4) and parallelization (Step 5).

Step 1: Instrument the model with logging of simulation time and frame delay

As starting point, let's measure the time it takes to process each frame. To do that, we let the Stimulus module note the start time of processing each frame and communicate that to the Monitor which, in turn, can then compute and display the delay incurred while processing each frame.

For the needed communication from Stimulus to Monitor, instantiate a **`sc_fifo`** channel with sufficient buffer space for the frame start times. The channel of type **`sc_fifo<sc_time>`** should pass timestamps from the Stimulus to the Monitor module. In the Stimulus, take the current simulated time right after sending out the frame image, print it to the screen for observation, and also send it to the Monitor through the new channel. In the Monitor, take the difference between the current simulated time and the time the frame was sent, and display this delay on the screen for each frame.

The following log shows the desired screen output:

```
      0 s: Stimulus sent frame  1.
      0 s: Stimulus sent frame  2.
      0 s: Stimulus sent frame  3.
      0 s: Stimulus sent frame  4.
      0 s: Stimulus sent frame  5.
      0 s: Stimulus sent frame  6.
      0 s: Monitor received frame  1 with      0 s delay.
      0 s: Stimulus sent frame  7.
      0 s: Monitor received frame  2 with      0 s delay.
  [...]
      0 s: Stimulus sent frame 30.
      0 s: Monitor received frame 23 with      0 s delay.
      0 s: Monitor received frame 24 with      0 s delay.
      0 s: Monitor received frame 25 with      0 s delay.
      0 s: Monitor received frame 26 with      0 s delay.
      0 s: Monitor received frame 27 with      0 s delay.
      0 s: Monitor received frame 28 with      0 s delay.
      0 s: Monitor received frame 29 with      0 s delay.
      0 s: Monitor received frame 30 with      0 s delay.
      0 s: Monitor exits simulation.
```

As shown above, it is recommended to prefix each log line with the current simulation time as this significantly simplifies understanding and any needed debugging. (If you prefer, you may use the official SystemC **`SC_REPORT_INFO("type","msg")`** facilities with enabled timing.)

Due to the untimed model (all times are zero at this step), it is not visible yet, but a good choice of time unit here is milli-seconds (noted as **`ms`**) which fits well for our application.

You want to keep a copy of your model at this stage, say **`CannyA8_step1.cpp`**, so that you can compare the observed timing between the different models in this assignment at the end.

Step 2: Back-annotate the estimated timing into the main DUT modules

As result of the previous Assignment 7, we have obtained timing measurements for the main modules in the DUT. These measurements can now serve as good estimates for the SystemC model and serve the purpose of observing the effects of model transformations we apply in this and the following assignment.

For consistency and easier discussion of this assignment, we will assume that the timing measurements in Assignment 7 resulted in the following delays for the DUT components:

```
Receive_Image            0 ms
Make_Kernel              0 ms
BlurX                 1710 ms
BlurY                 1820 ms
Derivative_X_Y         480 ms
Magnitude_X_Y         1030 ms
Non_Max_Supp           830 ms
Apply_Hysteresis       670 ms
```

Back-annotate these delays into your SystemC model by inserting corresponding wait-for-time statements into the main method of each DUT component. For consistency, these wait-for-time statements should be placed right after receiving the input data for the function and before its computation code.

After inserting these wait-for-time statements, recompile and simulate your model. Observe the simulated time and the frame delays reported by the log. (Hint: The total simulated time after the 30 frames are processed should be 59320ms.)

Again, you want to keep a copy of your model at this stage, say **`CannyA8_step2`**, so that you can compare this observed timing with the improvements in the following refinement steps.

Step 3: Improve the test bench to also log the frame throughput

As discussed in the lectures, the frame delay measured in Step 1 is helpful, but 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 video processing pipeline. To measure this, we will extend the timing log produced by the test bench in Step 1.

Specifically, we let the Monitor module measure and report the frame throughput upon receiving a new frame. The extended log should look like this at the end:

```
[...]
55680 ms: Monitor received frame 28 with 15860 ms delay.
55680 ms: 1.820 seconds after previous frame, 0.549 FPS.
57500 ms: Monitor received frame 29 with 15860 ms delay.
57500 ms: 1.820 seconds after previous frame, 0.549 FPS.
59320 ms: Monitor received frame 30 with 15860 ms delay.
```

```
59320 ms: 1.820 seconds after previous frame, 0.549 FPS.
59320 ms: Monitor exits simulation.
```

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

Adjust your model to print the extra log line for each received frame. Again, keep a copy of the model at this stage, say **`CannyA8_step3`**, so that you can compare the observed timing with the following improvements.

Step 4: Pipeline the DUT into a sequence of 7 stages

As discussed in the lectures, we use pipelining as the overall technique to improve the performance of the DUT.

Since our DUT components are already communicating via **`sc_fifo`** channels, there is not much to change for this step. The model from Assignment 6 is already pipelined. However, there may be data being passed from one pipeline stage to a stage later than the next one. Without sufficient buffering, this will lead to performance problems. Thus, make sure that all data in the DUT pipeline is passed only from one stage to its immediate next stage, and you use buffer sizes of 1 everywhere. You may need to pass some data explicitly through a stage to get this "clean" pipeline structure.

As a result of this step, your DUT model should contain a total of 7 pipeline stages, 2 of which may be wrapped inside the Gaussian Smooth module. Be sure to simulate the model and observe the timing, as well as validate the produced images for correctness.

Hint: The throughput time shown in the log should match the longest stage delay in the video pipeline. In our example, the expected throughput delay of 1.82 seconds matches the back-annotated timing of the **`Blur_Y`** module because that is the slowest pipeline stage in the model at this time.

Again, please keep a copy of your model at this stage and name it **`CannyA8_step4`**.

Step 5: Slice the **`BlurX`** and **`BlurY`** modules into parallel threads

Finally we will remedy the identified performance bottleneck in the **`BlurX`** and **`BlurY`** modules by use of parallelization. As discussed in the lectures, both blocks can be optimized by parallelizing the operations in the rows and columns, respectively. While we could technically operate on every single row or column in parallel (as a real graphics processing unit (GPU) would do it), we will limit our efforts to 4 parallel slices for this assignment.

Specifically, extend the existing **`BlurX`** and **`BlurY`** modules by using 4 parallel **`SC_THREAD`**s which each operate on a one-quarter slice of the image. For example, the 1st thread will process the rows from **`(ROWS/4)*0`** through **`(ROWS/4)*1-1`** and the 2nd thread will process the rows from **`(ROWS/4)*1`** through **`(ROWS/4)*2-1`**, and so on. Be sure to adjust the back-annotated timing delays for the expected speedup of 4x.

For synchronizing the operation of the 4 parallel threads, let a main thread read the input image, then send a **`start`** event to each of the 4 worker threads, and finally wait for a **`done`** event from each of the worker threads, before the main thread sends the produced image to the output.

As a result of this assignment, your final model **`CannyA8_step5`** should execute significantly faster (in simulated time!) than in the previous step.

Note the timing of each model and report it in your text file submission. Specifically, we are interested in the total simulated time and the longest delay for processing a frame for each of the 5 steps of model refinement. Thus, report the observed timings in the following table:

Model	Frame Delay	Throughput	Total simulated time
CannyA8_step1	... ms	... FPS	... ms
CannyA8_step2	... ms	... FPS	... ms
CannyA8_step3	... ms	... FPS	... ms
CannyA8_step4	... ms	... FPS	... ms
CannyA8_step5	... ms	... FPS	... ms

3. Submission:

For this assignment, submit the following deliverables:

`Canny.cpp` (the final model **`CannyA8_step5`** from above)
`Canny.txt` (the filled table with the timings observed)

As before, the text file should also briefly mention whether or not your efforts were successful and what (if any) problems you encountered.

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