Specification and Modeling of a Canny Edge Detector for System-on-Chip Design

Final Report of EECS222 EMBEDDED SYS MODLNG

Jieneng Yang
Department of Electrical Engineering and Computer Science
The Henry Samuil School of Engineering, University of California, Irvine Irvine, U.S.A
jienengy@uci.edu;
06/10/2017

Abstract— **This project conducts a case study System-level modeling and design to model a special application on System-on-Chip. The overall project goal is to design a suitable embedded system model of this application and describe it in a System-Level Description Language (SLDL). This embedded specification model will then not only be simulated for functional and timing validation, but also be refined for synthesis and implementation as an embedded System-on-Chip (SoC) suitable for use in a digital camera. This article explains the details throughout this case study of Canny Edge Detection, including the basic of Canny application, creating a stimulatable model in SLDL, creating structural hierarchy, pipelining, parallelization, performance estimation and optimization. As the outcomes of each steps of modeling recorded, the general procedure of System-level modeling and design and developing new methodologies of System-on-Chip is also presented. A well-defined methodology like the one presented in this article will help product planning divisions to quickly develop new products or to derive completely new business models.**

I. INTRODUCTION (*Heading 1*)

A. System-level Modeling and Design

System design is the process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development. There is some overlap with the disciplines of systems analysis, systems architecture and systems engineering.

A system on a chip or system on chip (SoC or SOC) is an integrated circuit (also known as an "IC" or "chip") that integrates all components of a computer or other electronic systems. It may contain digital, analog, mixed-signal, and often radio-frequency functions—all on a single substrate. SoCs are very common in the mobile computing market because of their low power-consumption. A typical application is in the area of embedded systems.

System level design and verification is an electronic design methodology, focused on higher abstraction level concerns. To raise the level of abstraction in design, System-Level Description is required.

B. Essential concepts and coverage in SpecC SLDL

System-level design methodology has been developed to improve the productivity and shorten the time to market of designing complex embedded system. With proper methodology and tools, the designer can perform early design space exploration, high-level synthesis and software refinements. The initial model, called a specification model, is usually described in System Level Design Languages (e.g. SystemC and SpecC). The model is required to contain explicit structure, com- munication, and parallelism.

The SpecC language is defined as extension of the ANSI-C programming language. This document describes the syntax and semantics of the SpecC constructs that were added to the ANSI-C language.

The SpecC language is a formal notation intended for the specification and design of digital embedded systems, including hardware and software portions. Built on top of the ANSI-C programming language, the SpecC language supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing.

II. CASE STUDY USING THE CANNY EDGE DETECTOR APPLICATION

A. Obtaining and studying the Canny application

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986.
In this project, it is assumed that the input image processing is to be performed in real-time in a digital camera by a SoC that is going to be designed and developed.

After copying the source file and a suitable input image from the dedicated instructor account for this course, the source code file canny.src is converted into normal C code. However, there are some systex errors in the .src file that need to be modified. For example, the canny.src file is actually 3 library files combined together, and some code should be commented. As the source code is successfully converted into runnable C code file, the C code can be complied and run by typing the command below.

```
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 looks as Fig 1.

The Canny application can be basically described as 5 steps, as is shown in Fig. The first one is to apply Gaussian filter to smooth the image to remove the noise. Then find the intensity gradients of the image. The third part is to apply non-maximum suppression to get rid of spurious response to edge detection. Then apply double threshold to determine potential edges. The last step is tracking edge by hysteresis to finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

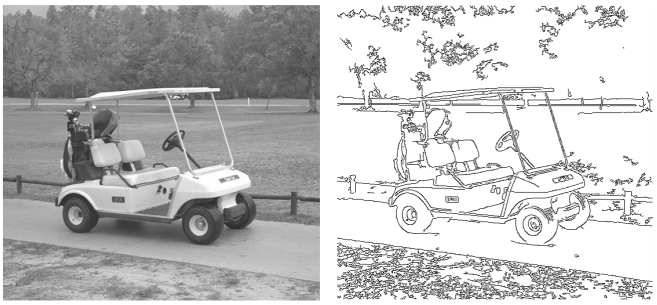

Figure 1 Canny Application Result

B. Creating a simulatable model in SpecC/SystemC SLDL

The main propose of the step is to convert the reference code of Canny to an initial SpecC model which can be simulated for validation.

After fixing bugs in non_max_supp function, code is cleaned-up so that there are no compilation warnings. The warnings include non-used variables, converting different types of variables, etc.

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. Therefore, all NULL should be replaced by 0.

The nest step is to fix the application parameters for synthesis, 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.
```

Those parameters can be defined in the #define part.

In the final step, all dynamic memory allocation from the algorithm should be removed. The one remaining malloc() and the corresponding free() call should be replaced with the use of an array with fixed size. 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.

The two functions radian_direction and angle_radians in the original Canny implementation can be removed as it is no use.

C. Creating structural hierarchy with test bench

The next part of the modeling is to create basic structure and communication for the model.

1) Step 1: Convert the application to process a stream of video frames

A stream of video frames will be processed in the system. Therefore, adjustment of the model source code should be conducted so that it processes 20 images in a loop. The input images are named “video/EngPlaza001.pgm” and so on, with increasing numbering. After processing the image, the model will output the generated edge image as “EngPlaza001_edges.pgm”, and so on.

2) Step 2: Add a test bench and platform structure to your SLDL model

The purpose of this part is to introduce a proper test bench and overall structural hierarchy into the application model. The top-level behavior Main (SpecC) will consist of three blocks, namely Stimulus, Platform, and Monitor. The Platform behavior/module, in turn, should contain a dedicated input unit DataIn, an output unit DataOut, and the actual design under test DUT. For communication, queue-type channels from the respective standard channel library is introdeuced. For SpecC modeling, typed-queue channels (of size 2) is used to send and receive the image data between the behaviors. As data type for the queue channels, the type is defined by the following:

```
typedef unsigned char img[SIZE]; //
image data type
```

For the above described top-level structural hierarchy, a total of four channel instances will be needed, two at the test bench level (Main behavior or Top module), and two within the Platform behavior. Overall, the model is structured as the following instance tree shows.

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors*).

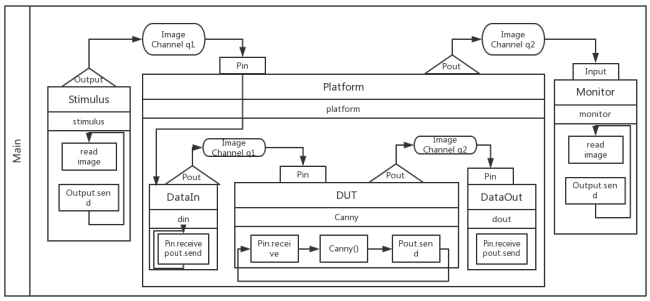

Figure 2 Chart of model top-level structure

Specifically, the Main behavior or Top module should instantiate Stimulus, Platform and Monitor in parallel. The Stimulus block should read the input image from the file system and pass it into the Platform via the first queue channel. Correspondingly, the Monitor should receive via the second channel the generated edge image from the Platform and write it out into the output file. In the Platform, the DataIn block should, in an endless loop, receive an input image and pass it unmodified to the DUT. Similar, the DataOut block should, also in an endless loop, receive an input image from the DUT and pass it on. These two instances will be needed later during model refinement. They will allow our test bench to remain unmodified even when later in the design flow the communication to the DUT is implemented via detailed bus protocols. Finally, the DUT block should contain the entire Canny algorithm source code. Its main thread will receive an image via the input port, call the canny() function to process it, and then send out the edge image via the output port. Since our target SoC will never stop working (unless its power is turned off), this processing will run in an endless loop, similar as the infinite loops in the DataIn and DataOut blocks.

After constructing the structural hierarchy of the whole model, the model should be refined with structural hierarchy in the DUT, and steps are as followed.

3) Step 1: Create an additional level of hierarchy in the DUT, including creation of an additional level of hierarchy in the Gaussian Smooth block.

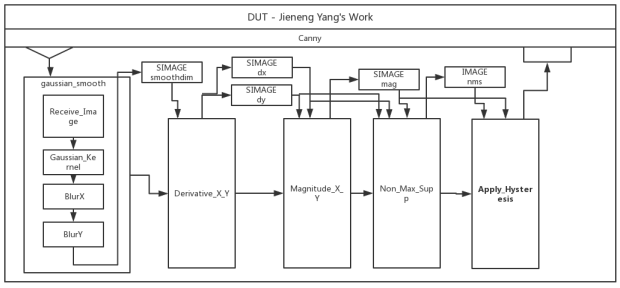

Figure 3 Chart of Refined DUT Structure

4) Step 2: Profile the application components

For an initial performance estimation of our Canny Edge Detector model, it is critical to identify the computational complexity of its main components. In other words, we want to find out which components can become a bottleneck in the implementation. To this end, we will profile our design model in this step. For the SpecC model, the profiler integrated into the System-on-Chip Environment (SCE) is utilized.

At this point in the design process, the relative computational load of the components in the DUT is wanted. Thus, assuming the total DUT load is 100%, what needs to find out is how much load each of the DUT components contributes. The relative load of the DUT components as a percentage value in the following complexity comparison table is shown.

Table 1 Complexity Comparison

Behavior	Sub-behavior	Complexity / %
Guassin_Smooth		83.42
	-Gaussian_Kernel	~0
	-BlurX	40.19
	-BlurY	39.26
D_x_y		3.13
Mag_xy		2.66
NMS		6.61
Apply+Hysteresis		4.14
	Sum	100

D. Pipelining and Parallelization

In order to observe the performance of the application in the simulator, we need to insert statements to monitor the simulation time in the test bench (Step 1) and then instrument the model with estimated delays in the DUT (Step 2).

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

It is recommended to prefix each log line with the current simulation time as this significantly simplifies understanding and any needed debugging. Also shown above is the choice of micro-seconds (noted as us) as the time unit which fits well for our application.

2) Step 2: Back-annotate the estimated timing in the DUT components

For consistency and easier discussion of this assignment, we will choose here the timing estimates obtained by SCE for the ARM_926EJS processor assuming we can improve those by 10x for a 1.0 GHz clock frequency. Specifically, the following total delays for the DUT components are assumed as below.

Table 2 DUT Delay Time

Behavior	Delay time /*ms*
Receive_Image	0
Gaussian_Kernel	0
BlurX	61480
BlurY	43740
Derivative_X_Y	5340
Magnitude_X_Y	5340
Non_Max_Supp	27200
Apply_Hysteresis	7650

These delays are back-annotated into the model by inserting suitable wait-for-time statements at the beginning of the main method of each DUT component. Wait for function is used in this time.

The following log illustrates the screen output.

```
    0: Stimulus sent frame 1.
    0: Stimulus sent frame 2.
    0: Stimulus sent frame 3.
    0: Stimulus sent frame 4.
    0: Stimulus sent frame 5.
    0: Stimulus sent frame 6.
    6547500: Monitor received frame 1 with
6547500 us delay.
    6547500: Stimulus sent frame 7.
    13095000: Monitor received frame 1
with 13095000 us delay.
    13095000: Stimulus sent frame 8.
    19642500: Monitor received frame 2
with 19642500 us delay.
    19642500: Stimulus sent frame 9.
    [...]
    91665000: Monitor received frame 13
with 39285000 us delay.
    91665000: Stimulus sent frame 20.
    98212500: Monitor received frame 14
with 39285000 us delay.
    104760000: Monitor received frame 15
with 39285000 us delay.
    111307500: Monitor received frame 16
with 39285000 us delay.
    117855000: Monitor received frame 17
with 39285000 us delay.
    124402500: Monitor received frame 18
with 39285000 us delay.
    130950000: Monitor received frame 19
with 39285000 us delay.
    130950000: Monitor exits simulation.
```

3) Step3 Pipeline the DUT into stages for each component

Pipelining is used as the overall technique to improve the throughput of the DUT. Pipelining can be applied by simply replacing the endless loop in the Canny behavior with an infinite pipeline in a pipe construct. Then, to allow for the necessary buffering of the data between the pipeline stages, piped qualifiers should be added to the port-mapped variables between the stages. As a result of this step, the model should contain 5 pipeline stages and, because of this, execute significantly faster than before.

4) Step 4: Integrate the Gaussian Smooth components into the pipeline stages

To further improve the performance of the design, we will the first pipeline stage, namely the Gaussian Smooth block, is decomposed. And two additional pipeline stages for the BlurX and BlurY blocks are created. In other words, the BlurX and BlurY blocks from the Gaussian_Smooth parent are moved one level up into the DUT. Properly arrangement of the port connectivity and any needed buffering between the new pipeline stages is conducted.

The expected DUT block is shown below.

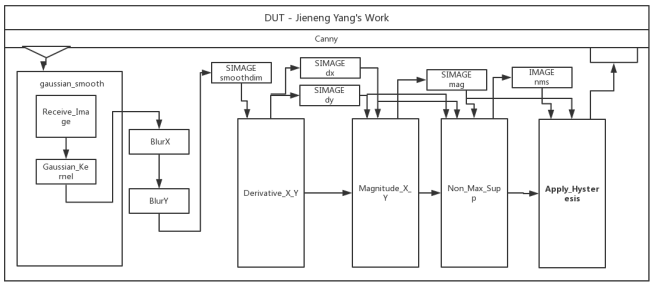

Figure 4 DUT with Pipeline Stages

5) Step 5: Slice the BlurX and BlurY blocks into parallel components

The identified performance bottleneck in the BlurX and BlurY components is remedied by parallelization. Both blocks are straightforward to optimize by parallelizing the operations in the rows and columns, respectively. Specifically, the existing BlurX and BlurY blocks are converted into BlurX_Slice and BlurY_Slice components that only operate on a one-eighth slice of the image. For example, the first BlurX_Slice instance sliceX1 will process the rows from `(ROWS/8)*0` through `(ROWS/8)*1-1` and sliceX2 will process the rows from `(ROWS/8)*1` through `(ROWS/8)*2-1` and so on. Then, instantiate 8 parallel instances of these slice processors in replacements of the previous BlurX and BlurY blocks. In the end, the expected instance tree of the DUT is shown below.

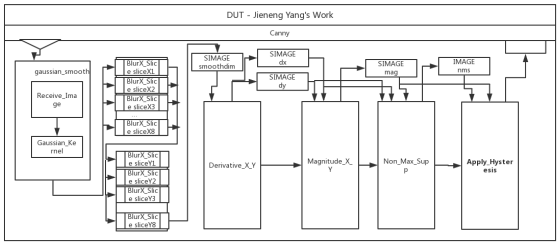

Figure 5 DUT with Parallel Pipeline

The results of computation of each step is shown in Table 3.

Table 3 Canny Edge Detector, Performance Estimation

	Frame Delay /*us*	Total Time /*us*
CannyA7_step1	0	0
CannyA7_step2	39285000	130950000
CannyA7_step3	38821500	98615500
CannyA7_step4	24057000	52767500
CannyA7_step5	14960000	28663125

.

E. Performance Estimation and Throughput Optimization

In this part, throughput optimization of the Canny Edge Decoder model is conducted.

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

To measure FPS, the timing logs produced by the test bench is extended. Specifically, the Monitor block measure and report the frame throughput upon receiving a new frame. 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.

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

The model is refined with more realistic assumptions. Suitable processing elements (PEs) are assigned 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 the SoC platform to the outside world, two I/O units with Direct Memory Access (DMA) capability are utilized. SCE is used for the allocation of the PEs and corresponding estimation of the stage delays. After compiling and simulating the model, the SCE profiler can 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.

Next, the blocks in the Canny platform to the allocated PEs is mapped as:

```
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. The estimation results in a bar chart, of gaussian_smooth, sliceX8, sliceY8, derivative_x_y, magnitude_x_y, non_max_supp and apply_hysteresis is shown below, representing the pipeline stages and generating the computation profile.
The ALU operations and note the ratio of integer vs. floating-point operations are also shown in Fig.

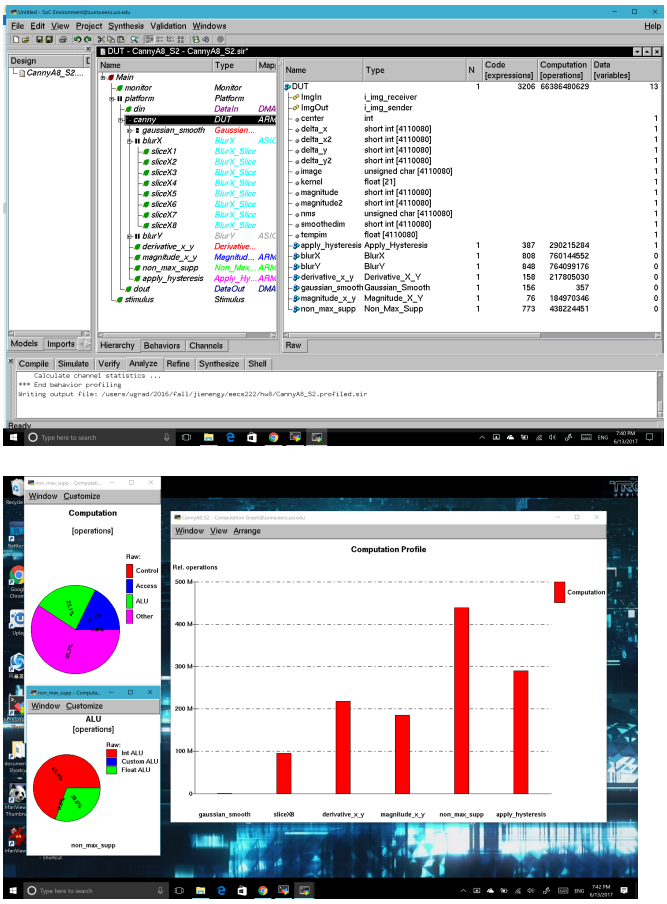

Figure 6 Step 2 Results

The stage delays for the individual blur slice blocks will be different since these are now mapped to hardware PEs. In this case, the wait for of BlurX and BlurY is assigned as 111.9 ms and 102.7 ms respectively. Then compile and simulate it. The timing shown in the simulation log is similar to the log of CannyA8_step1, shown in Table.

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

In order to improve the throughput of our pipeline in the DUT, optimize the stage(s) with the longest stage delay need to be optimized. As the profiling results in SCE show, 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.
```

From Fig. we can see the int operation has taken up all operation in ALU of nms.

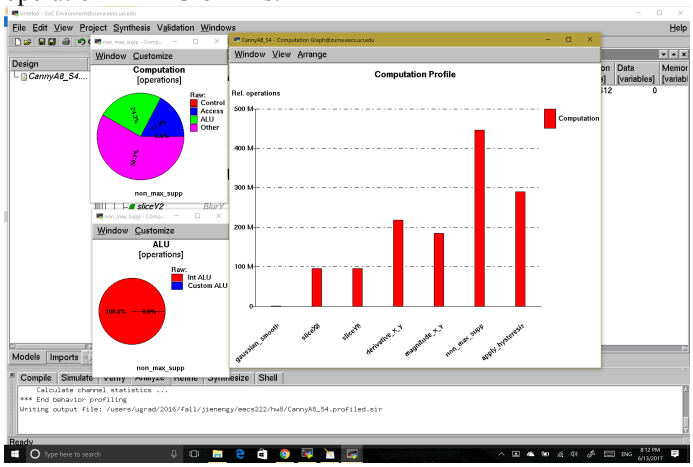

Figure 7 Step 3 Results

4) 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, the SCE profiling and evaluation tools are used again. Then the improved performance is evaluated.

Then the obtained NMS stage delay is back-annotated into the source code of the model. Compile and simulate this final model CannyA8. The timing shown in the simulation log is shown below. It is better than for any of previous models and it supports real-time display.

```
       0: Stimulus sent frame  1.
          0: Stimulus sent frame  2.
          0: Stimulus sent frame  3.
          0: Stimulus sent frame  4.
          0: Stimulus sent frame  5.
          0: Stimulus sent frame  6.
          0: Stimulus sent frame  7.
     111900: Stimulus sent frame  8.
     223800: Stimulus sent frame  9.
     490800: Stimulus sent frame 10.
     757800: Stimulus sent frame 11.
    1670900:  Monitor  received  frame   1
with  1670900 us delay.
    1670900:    1.670900   useconds   after
previous frame, 0.598480 FPS
  1670900: Stimulus sent frame 12.
   2201500:  Monitor  received  frame   2
with  2201500 us delay.
    2201500:    0.530600   useconds   after
previous frame, 1.884659 FPS
  2201500: Stimulus sent frame 13.
   2732100:  Monitor  received  frame   3
with  2732100 us delay.
    2732100:    0.530600   useconds   after
previous frame, 1.884659 FPS
  2732100: Stimulus sent frame 14.
   3262700:  Monitor  received  frame   4
with  3262700 us delay.
    3262700:    0.530600   useconds   after
previous frame, 1.884659 FPS
  3262700: Stimulus sent frame 15.
   3793300:  Monitor  received  frame   5
with  3793300 us delay.
    3793300:    0.530600   useconds   after
previous frame, 1.884659 FPS
  3793300: Stimulus sent frame 16.
  4323900: Monitor received frame  6
    5915700:    0.530600   useconds   after
previous frame, 1.884659 FPS
  5915700: Stimulus sent frame 20.
   6446300:  Monitor  received  frame  10
with  5955500 us delay.
    6446300:    0.530600   useconds   after
previous frame, 1.884659 FPS
  6446300: Stimulus sent frame  1.
```

.....

```
 10160500:    0.530600   useconds   after
previous frame, 1.884659 FPS
 10160500: Stimulus sent frame  8.
 10691100:  Monitor  received  frame  18
with  5836600 us delay.
 10691100:    0.530600   useconds   after
previous frame, 1.884659 FPS
 10691100: Stimulus sent frame  9.
 11221700:  Monitor  received  frame  19
with  5836600 us delay.
 11221700:    0.530600   useconds   after
previous frame, 1.884659 FPS
 11221700: Stimulus sent frame 10.
 11752300:  Monitor  received  frame  20
with  5836600 us delay.
 11752300:    0.530600   useconds   after
previous frame, 1.884659 FPS
 11752300: Monitor exits simulation.
```

The image quality of the output image is compared with the reference. Some results are shown as below.

```
[jienengy@zuma hw8]$ ./ImageDiff
EngPlaza020_edges.pgm
video/EngPlaza020_edges.pgm diff20.pgm
0 mismatching pixels (0.000%) identified
in diff20.pgm.
[jienengy@zuma hw8]$ ./ImageDiff
EngPlaza001_edges.pgm
video/EngPlaza001_edges.pgm diff1.pgm
0 mismatching pixels (0.000%) identified
in diff1.pgm.
[jienengy@zuma hw8]$ ./ImageDiff
EngPlaza002_edges.pgm
video/EngPlaza002_edges.pgm diff2.pgm
0 mismatching pixels (0.000%) identified
in diff2.pgm.
……
```

The rates of mismatch are 0 to every image, meaning that fixed-point arithmetic is acceptable for this application.

The difference of results in step1,2 and 3 is shown in Table 4.

The similarity of the results of 1 and 2 shows that the new stage delay does not greatly improve the performance of the application. It is probably because the new stage is not greatly faster than the previous one. The HW_Standard just shortens little dealy of the Blur behavior. Also, in the pineline-7 structure, the Blur with 8 parallel sub-behaviors does not take up too much in the overall computation.

Table 4 The Timing of Application

Step	Time / us	Delay	FPS
1	28663125	14960000	0.735294
2	28340300	14960000	0.735294
3	11752300	5836600	1.884659

After back-annotating the new stage delay of non_max_supp, the performance of the application is hugely improved. The ARM9x10core greatly, improves the performace of nms. Since nms has taken up a lot in the final computation, the overall performace can be improved.

III. Summary and Conclusion

A. Lessons Learned

In this course, I have learnt the basic of design and model of embedded system on system level, including using SLDL to description a system, modeling it and optimizing it with different methods. The main application of the system discussed is Canny Edge Detection. By going through all assignment, I have been taught how to design and model an image processing system based on real hardware, and to optimize it step by step. During the process, I have practiced thinking as a hardware designer as well as a programmer since the overall model is constructed by program.

B. Future work

The detail of a special image process application based on system level is discussed in the article. Other similar process in image, audio or sensors data can be implemented in similar method. After going through the whole procedure, any component in embedded system, or the big cyber-physical-social systems can be designed and modeled.

References

[1] Gajski DD, Zhu J, Dömer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and design methodology. Kluwer Academic Publishers, Basel

[2] Zeng J, Yang L T, Ma J. A System-Level Modeling and Design for Cyber-Physical-Social Systems[M]. ACM, 2016.

[3] Canny Application: http://marathon.csee.usf.edu/edge/edge_detection.html

[4] SpecC Language Reference Manual, Version 2.0, at http://www.cecs.uci.edu/~doemer/publications/SpecC_LRM_20.pdf

[5] System-on-Chip Environment (SCE) at http://www.cecs.uci.edu/~cad/sce.html

[6] Adamov, Alexander. "Electronic System Level Models for Functional Verification of System-on-Chip". CAD Systems in Microelectronics.

[7] Brian Bailey; Grant Martin (2010). ESL Models and Their Application: Electronic System Level Design and Verification in Practice. Springer