UCI University of California, Irvine

Nitin Chauhan
31001417

Specification and Modeling of a Canny Edge Detector for Embedded Systems Design

Abstract:

In this project we created a performant embedded systems model for Image processing in a System level description language, the IEEE SystemC that automatically detects edges in a digital camera. For our application, we used a purely sequential, multi-stage C code for Canny Edge Algorithm that detects a wide range of edges in grey-scale images. We removed costly dynamic memory allocation and migrated it to C++ compliant code. Later we took each sequential stage of Canny algorithm as a separate module in System C model and following a top down design approach introduced pipelining, parallelization and compiler optimizations to achieve a much better simulated throughput (frames per second) for our real time recording drone camera.

MECPS UCI University of California, Irvine

Embedded Systems Modeling and Design
ECPS 203, Course Code: 16905
12/07/2017

(949) 880-5167
chauhan1@uci.edu
www.mecps.uci.edu

Introduction

Embedded Systems Modelling and design concepts

Top down design approach

Embedded system design faces tremendous increase in design complexity. To tackle this, we need to move up in the levels of abstraction where we hide the unnecessary details to make the system easier to understand. At a higher level of abstraction this is facilitated by decrease in the number of components to work with. However, as we move up we are increasing uncertainty (decreasing accuracy), so moving down is also important.

So, the most appropriate design approach is a top down design approach where we start from top and build system downwards and increasing the implementation details at every level.

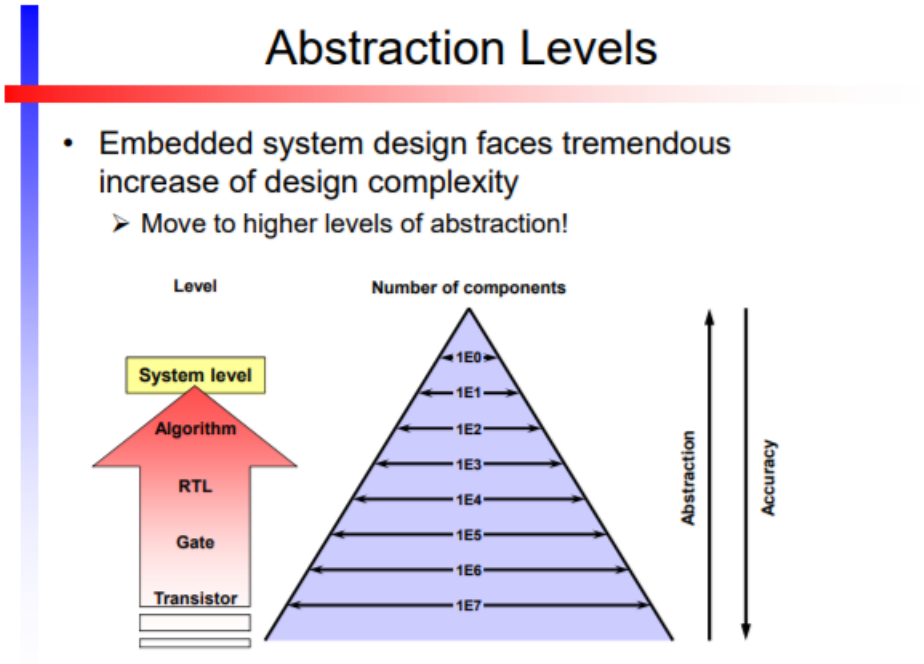

Figure 1 (c) 2017 R. Doemer

A typical Top down design approach for embedded systems consists of following stages:

1. Product features – here we list down all the features desired from the product.
2. Specification model – here we come out with the actual numbers we want like final clock frequency, power requirements etc.
3. Architecture model – here we decide how many and of what kind chips and memory we want.
4. Communication model – here we layout the channels or buses for communication to happen between various modules on a chip.
5. Implementation model – here we are at the register transfer level (RTL) working on digital electronics components like flip flops, adders and multipliers.
6. Manufacturing – Finally the model is sent for manufacturing.

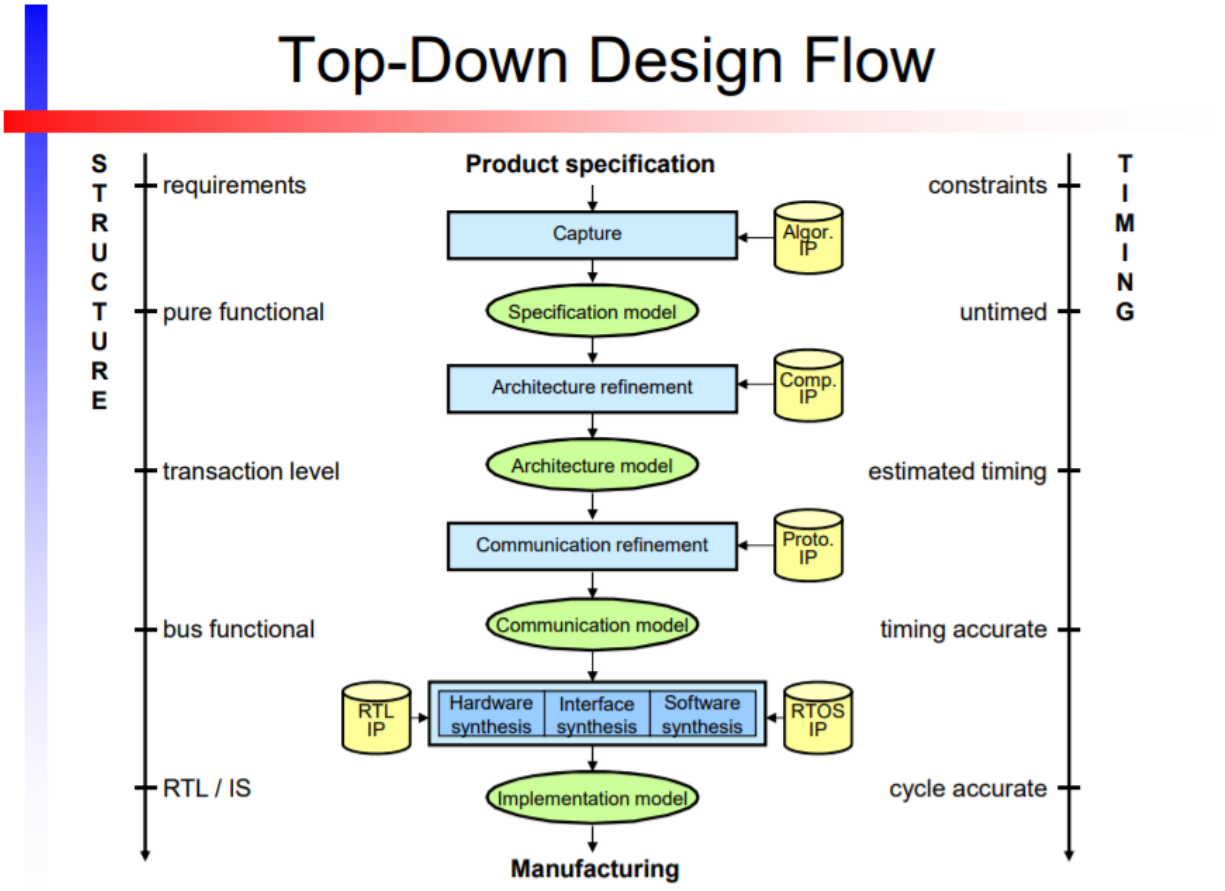

Figure 2 (c) 2017 R. Doemer

Separation of Concerns

Another fundamental principle in Embedded systems modeling and design is the separation of concerns, which is to address separate issues independently otherwise the complexity of designing the system increases. The idea here is to separate Computation from Communication and there should be no overlap. So, one system is either crunching numbers and doing calculations or doing communication. Traditional models that have Computation and Communication on the same system cannot be a plug and play when protocols for either change. Automatic replacement is impossible in such a design. Following the separation of concerns design concept in System level modelling the computation is encapsulated in modules and communication is encapsulated in channels. In System C we have what is called communication protocol in lining.

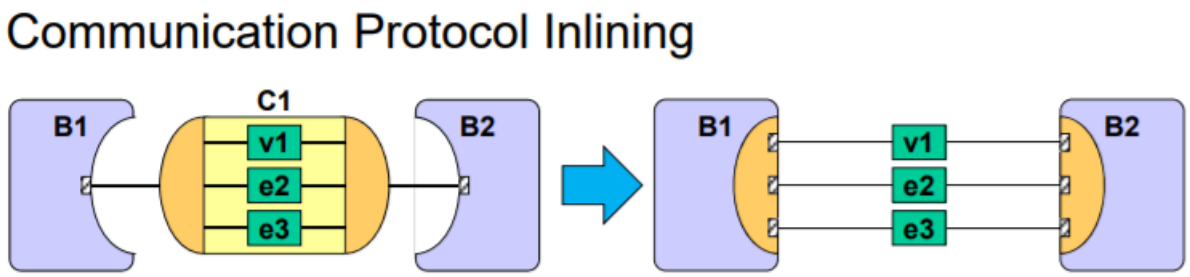

Figure 3 (c) 2017 R. Doemer

Structural hierarchy: Top level behavior model

Here top behavior (spec C) is the biggest module in System C and child behavior are instances of other modules within the top module. So, there is a hierarchy where inner modules are instances within the outer module classes.

Our implementation of structural hierarchy consists of:

1. Test bench – Typically called Main or Top.
2. Design under test – The chip we want to design is Design under test (DUT), which is put into test bench.
3. Stimulus – serves like a system that sends input/test data (test picture/frame)
4. Monitor – serves like a system that receives the output (Canny edge image)

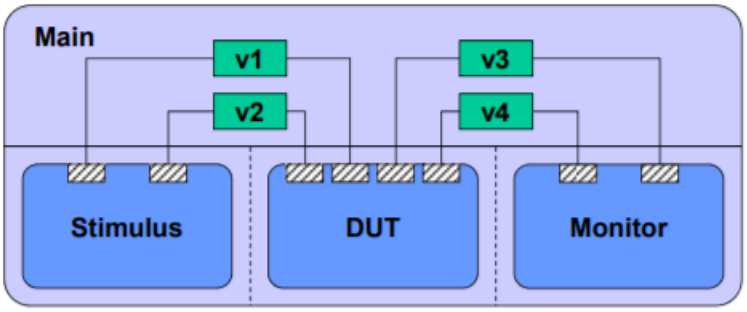

Figure 4 (c) 2017 R. Doemer

Important thing to note is that in pipelined execution sequential and parallel execution happens simultaneously. We just need to make sure that every stage is run equal number of times.

The IEEE SystemC language

System C is a system-level description language (SLDL) and now an IEEE and de-facto standard. System C has its foundation in C++, so the software requirements are fully covered. SystemC is a superset of C++ (class library for system modeling) which leverages a large set of existing programs in C++ that are SLDL programs themselves. With System C a library within C++ keywords specific to System C were added like sc_module, sc_fifo etc. Our reason for choosing SystemC for this project was the possibility of pipelining and parallelization that SystemC has on offer.

SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. SystemC is often associated with electronic system-level (ESL) design, and with transaction-level modeling (TLM).

Following components of SystemC were used in the project:

Modules
SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
Modules are the basic building blocks of a SystemC design hierarchy. A SystemC model usually consists of several modules which communicate via ports. The modules can be thought of as a building block of SystemC.

Ports
Ports allow communication from inside a module to the outside (usually to other modules) via channels.

Signals
SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.

Processes
Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers. Processes are the main computation elements. They are concurrent.

Channels
Channels are the communication elements of SystemC. They can be either simple wires or complex communication mechanisms like FIFOs or bus channels.
Elementary channels:
signal: the equivalent of a wire
fifo

Events
Events allow synchronization between processes and must be defined during initialization.

Structure of the Canny edge detection algorithm

The sequential Canny edge algorithm consists of following stages in the same order

1. Gaussian_smooth - Apply Gaussian filter to smooth the image to remove the noise.
2. Derrivative_x_y - Compute the first derivative of the image in both the x any y directions.
3. Magnitude_x_y - Compute the magnitude of the gradient. This is the square root of the sum of the squared derivative values.
4. Non_max_supp - This routine applies non-maximal suppression to the magnitude of the gradient image.
5. Apply_hysteresis - This routine finds edges that are above some high threshold or are connected to a high pixel by a path of pixels greater than a low threshold.

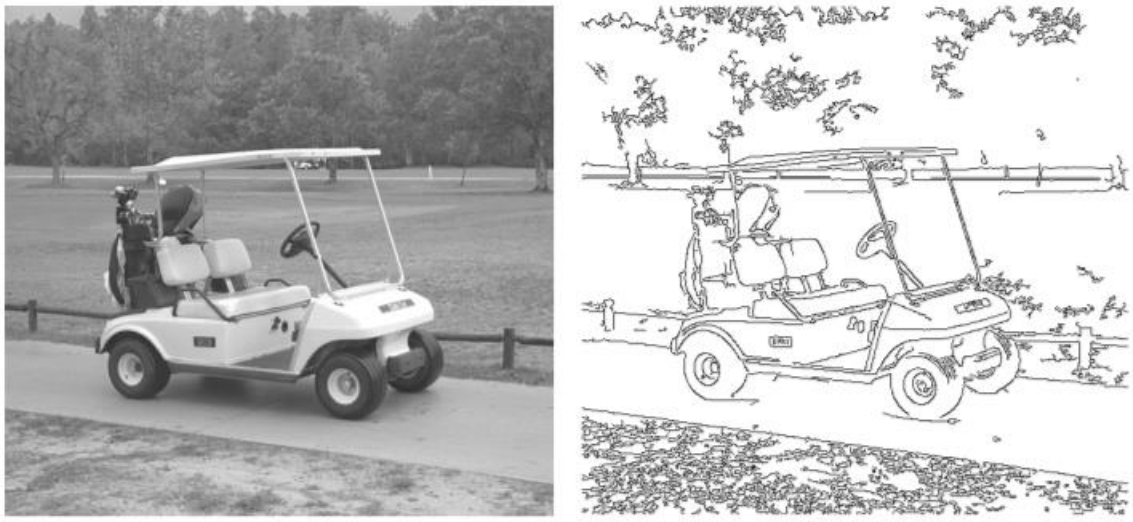

Figure 5 (c) 2017 R. Doemer

Case Study of a Canny Edge Detector for Real-time Video

Setup, introduction to canny application example

For our application we started with downloading the Canny edge detection algorithm source code, which was a purely sequential C code and started with analyzing all the functions in the canny program for which we created a function call tree like a sequence diagram in UML. The purpose of doing this as an initial step was to familiarize ourselves with the canny algorithm and C programming which is purely sequential (One of the challenges in using Canny algorithm in real time video processing that we tackle later in the project.)

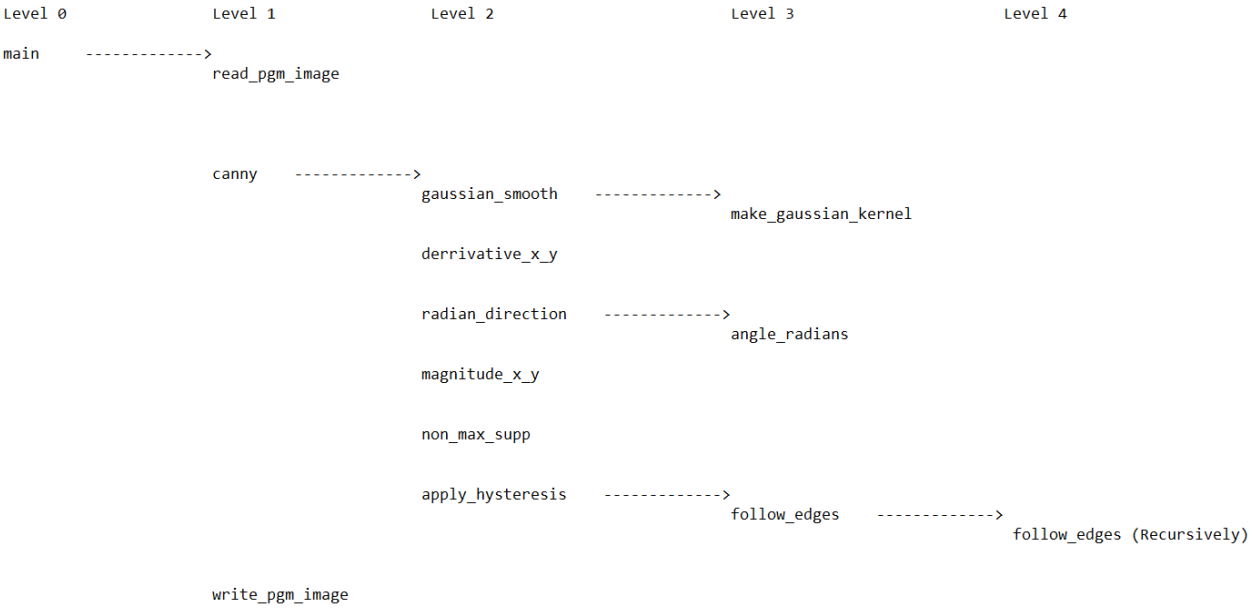

Figure 6

Create a clean C++ model with static memory allocation

Next, we did some refactoring of the source code like removing compiler warnings, removing unused variables, removing classic off-by-one bug in loop iterations in non_max_supp method. We also fixed the user-adjustable configuration parameters for embedded system design like instead of reading parameters sigma, thigh, tlow as command line arguments we hardcoded their value. Also since the input image (golfcart.pgm) has a fixed resolution we hardcoded the row and column fields too.

Dynamic memory allocation (i.e. use of functions malloc, calloc, and free is clearly not feasible in a hardware implementation, because the desired SoC cannot instantiate a new memory chip at runtime! Thus, we used static arrays with fixed sizes at compile time and removed all dynamic memory allocation from the source code. Further we are going to be working on a set of drone images which have a fixed rows and columns size that we know of beforehand and does not need any code that checks it at runtime and allocates memory dynamically.

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 be viewed). However, this functionality serves no purpose in our embedded system model where we are only interested in the final edge image. Thus, we safely removed both functions (and the included dynamic memory allocation) from the C++ source code of our model.

Further, in make_gaussian_kernel an array kernel is filled with parameters. The size of this array (variable window_size) 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. We safely replaced window_size with the constant WINSIZE=21.

From single image to video stream processing

To create an embedded system model for real time video stream processing using the canny algorithm we created a video recording from our drone flight camera. Next, we extracted individual video frames from the movie file using ffmpeg. To keep later timing calculations simple, we extracted 30 frames from the stream, representing 1 second of realtime video at 30 frames per second (FPS). Then converted color frames to grey-scale images in PGM format. Lastly, we changed the canny source code in C++ to process these 30 sample video frames instead of just a single frame earlier.

The frames from the drone video had a high definition resolution of **2704x1520** which naturally leads to higher memory usage of our application because we use local array variables to store the pixels in rows and columns. Specifically, the local array variables holding the image data grows large. Note that many of those variables are local variables which get memory allocated on the stack. At the same time, the stack size of a process in the Linux environment is typically limited. If so, your application will "crash" with a core segmentation fault error or similar memory related error. To get rid of this we increased the stacksize to 128MB on the Linux server.

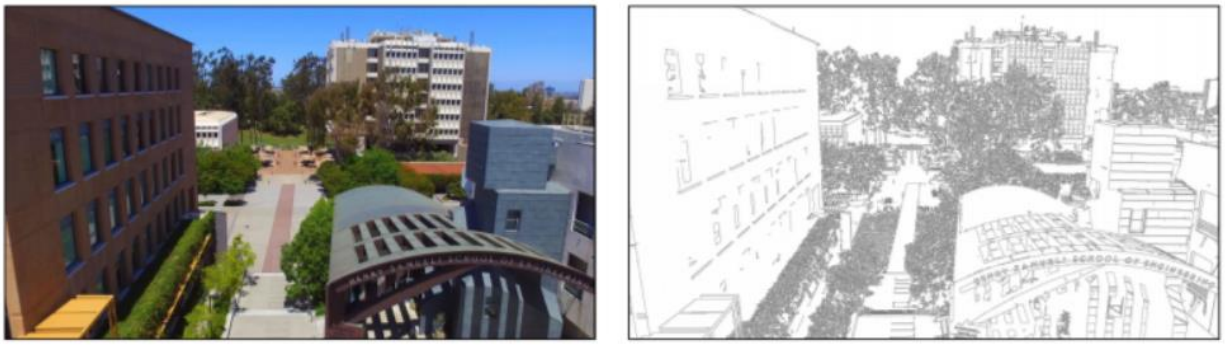

Figure 7 (c) 2017 R. Doemer

Test bench model of the Canny Edge Decoder in SystemC

The purpose of this step was to introduce a proper test bench and overall structural hierarchy into our application model. We introduced the top-level module **Top**. This consisted of three modules, namely **Stimulus**, **Platform**, and **Monitor**. The Platform module, in turn, contained a dedicated input unit **DataIn**, an output unit **DataOut**, and the actual design under test **DUT** (Canny). DataIn and DataOut are of importance only when we drill down to the RTL modelling stage of the communication bus. For now, it's just redundant. The purpose of creating these modules like Stimulus, Monitor, DUT was to separate the testing parts from the features of the modelled SoC i.e. DUT

```
Top top
|------ Monitor monitor
|------ Platform platform
|       |------ DUT canny
|       |------ DataIn din
|       |------ DataOut dout
|       |------ sc_fifo<IMAGE> q1
|       \------ sc_fifo<IMAGE> q2
|------ Stimulus stimulus
|------ sc_fifo<IMAGE> q1
\------ sc_fifo<IMAGE> q2
```

Figure 8 (c) 2017 R. Doemer

For communication, we instantiated fifo-type channels from the SystemC standard library. Specifically, used the regular first-in-first-out primitive channel sc_fifo<IMAGE> where template parameter IMAGE is the type of the data you need to communicate. Since IMAGE is an array and C++ does not provide an operator for array assignment, we wrapped the array into a proper class with overloaded operators.

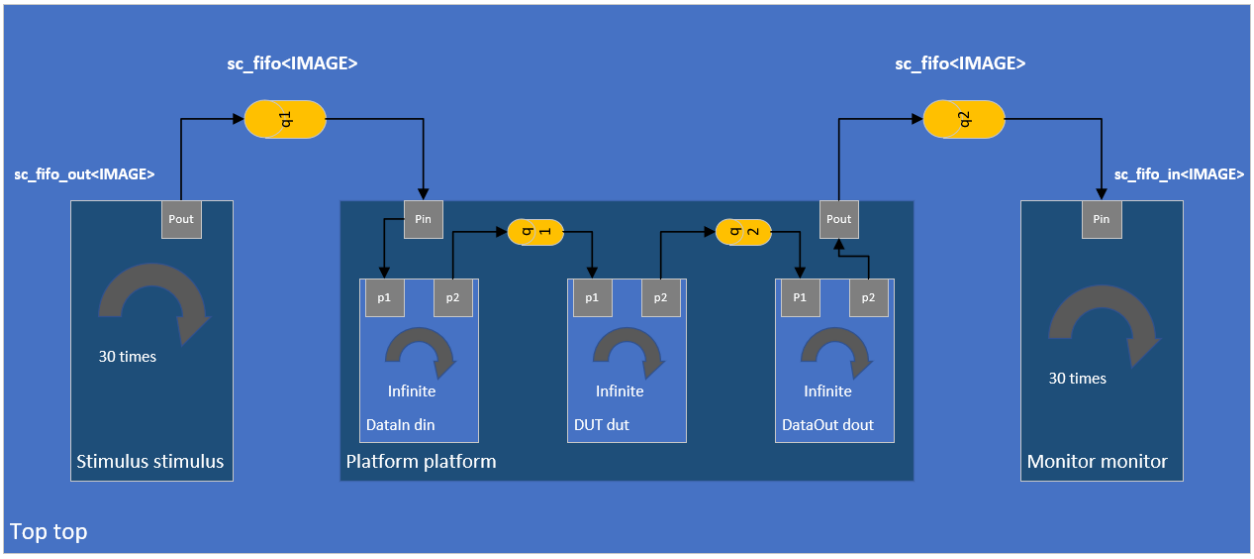

Figure 9

For the above described top-level structural hierarchy, a total of four channel instances were needed, two at the test bench level (Top module), and two within the Platform module. Specifically, the Top module instantiated Stimulus, Platform and Monitor modules in parallel. The Stimulus module read the input image from the file system and passed it into the Platform via the first queue channel. Correspondingly, the Monitor received via the second channel the generated edge image from the Platform and wrote it out into an output file.

In the Platform module, the DataIn module, in an endless loop, received an input image and passed it unmodified to the DUT. Similar, the DataOut module, also in an endless loop, receive an input image from the DUT and passed it on. These two instances will be needed later during model refinement. They allow our test bench to remain unmodified even when later in the design flow the communication to the DUT will be implemented via detailed bus protocols.

Finally, the DUT module contains the entire Canny algorithm source code. Its main thread receives an image via the input port, calls the canny() function to process it, and then sends out the edge image via the output port. Since our target embedded system 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 modules.

Structural refinement of DUT module and algorithm profiling

At this stage we refined the previous model with a suitable structural hierarchy inside the design-under-test (DUT) module. We will later use this model for initial performance profiling to identify the functions with the highest computational complexity.

The original canny function consists of a sequence of function calls to five functions, namely **gaussian_smooth**, **derivative_x_y**, **magnitude_x_y**, **non_max_supp**, and **apply_hysteresis**. While in the previous model all these are local methods in the DUT, we now encapsulated them into separate modules by themselves.

```
Platform platform
|------ DataIn din
|------ DUT canny
|       |------ Gaussian_Smooth gaussian_smooth
|       |------ Derivative_X_Y derivative_x_y
|       |------ Magnitude_X_Y magnitude_x_y
|       |------ Non_Max_Supp non_max_supp
|       \------ Apply_Hysteresis apply_hysteresis
\------ DataOut dout
```

Figure 10 (c) 2017 R. Doemer

```
DUT canny
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       |------ Gaussian_Kernel gauss
|       |------ BlurX blurX
|       \------ BlurY blurY
|------ Derivative_X_Y derivative_x_y
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

Figure 11 (c) 2017 R. Doemer

The Gaussian Smooth function further consists of several tasks that we wrapped into three separate child modules, namely **Gaussian_Kernel**, **BlurX**, and **BlurY**. The module instance tree of the new DUT should then look like shown above right.

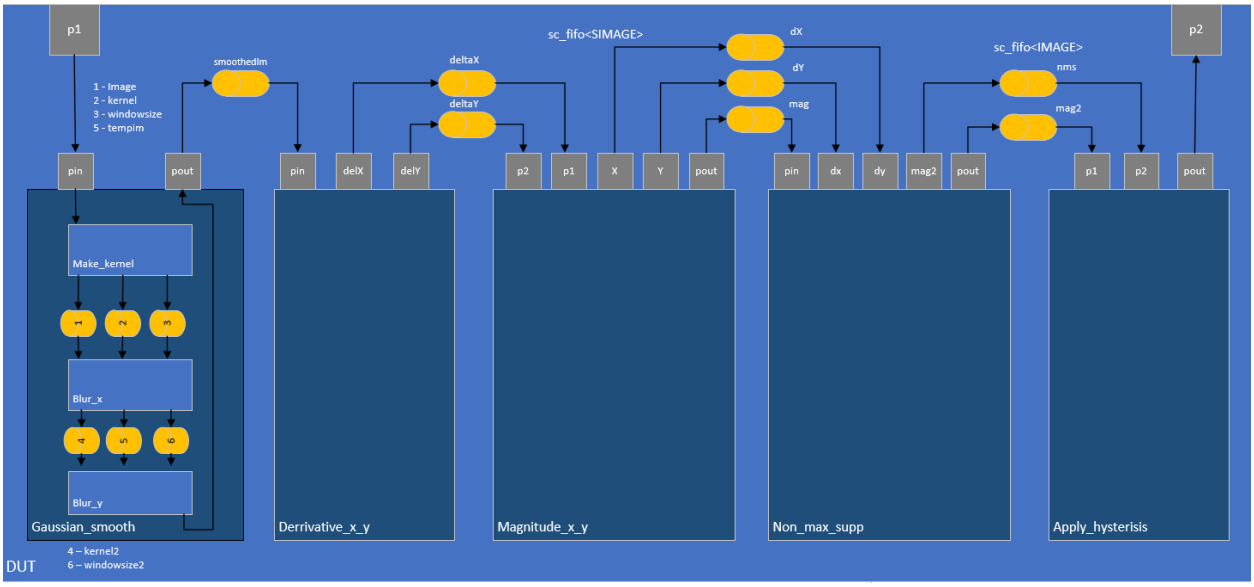

Figure 12

For an initial performance analysis of our Canny Edge Detector model, it is critical to identify the computational complexity of its main functions. In other words, we want to find out which functions can become a bottleneck in the implementation. To this end, we profiled our design model in this step. For the SystemC model, we used the profiling tools provided by the GNU community, namely **gprof**. For the computational complexity we are interested in, we focused on the "flat profile" in the profiling report and got the following numbers:

```
Gaussian_Smooth                          38.48%
|------ Gaussian_Kernel          00.00%
|------ BlurX                    18.23%
\------ BlurY                    20.25%
Derivative_X_Y                           05.81%
Magnitude_X_Y                            16.33%
Non_Max_Supp                             23.68%
Apply_Hysteresis                         15.70%

Total                                   100.00%
```

Figure 13

Performance measurement of the Canny Edge Detector on RPi board

While our SystemC model can be simulated on the Linux servers and provide relative timing measurements, we needed to also obtain some absolute measurements on the target platform to estimate the expected real-time performance there. Thus, we took the application C++ model and measured its run-time on the Raspberry Pi prototyping board. These measurements would then serve as an absolute reference point for performance estimation of the actual embedded system we envision for the final implementation.

We used the clock_t timer within the <time.h> library of C++ to check the average time it takes to process one video frame across various modules within DUT (Canny). We got the following numbers on running canny.

```
Receive, 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
                       =======
TOTAL:                 6540 ms
                       =======
```

Figure 14 (c) 2017 R. Doemer

Important to note is that the timing numbers obtained in this step on raspberry pi are a lot different from the ones we get on the Linux servers. This was because of the

1. Difference in manufacturers (Intel Xeon server vs ARM chip on RPi).
2. Different no. of cores and clock frequency.
3. Multiple users and time slicing on the server vs single user on RPi.
4. X86 instruction set on server vs RISC CPU on RPi.

To achieve a throughput of 30 frames per second for our application we need our program to process 1 video frame of the drone footage per every **0.033** seconds but on the RPi prototyping board it currently takes **6.54** seconds per frame which is around **198 times slower** than our desired throughput. The main problem with the step 4 C++ code that we used here is that it is purely sequential. We see in future steps how this can be improved in SystemC.

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

At this stage we continued the modeling of our application example, the Canny Edge Detector, towards an embedded system implementation. This time we refined the previous SystemC model with back-annotated timing and further pipelined and parallelized the components in the design-under-test (DUT) module. Over the course of this stage, our design model got 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.

Step 1 - Instrument the model with logging of simulation time and frame delay. To observe the performance of the application in the simulator, we had 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). To measure the time, it takes to process each frame 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, we instantiated a sc_fifo channel with sufficient buffer space for the frame start times.

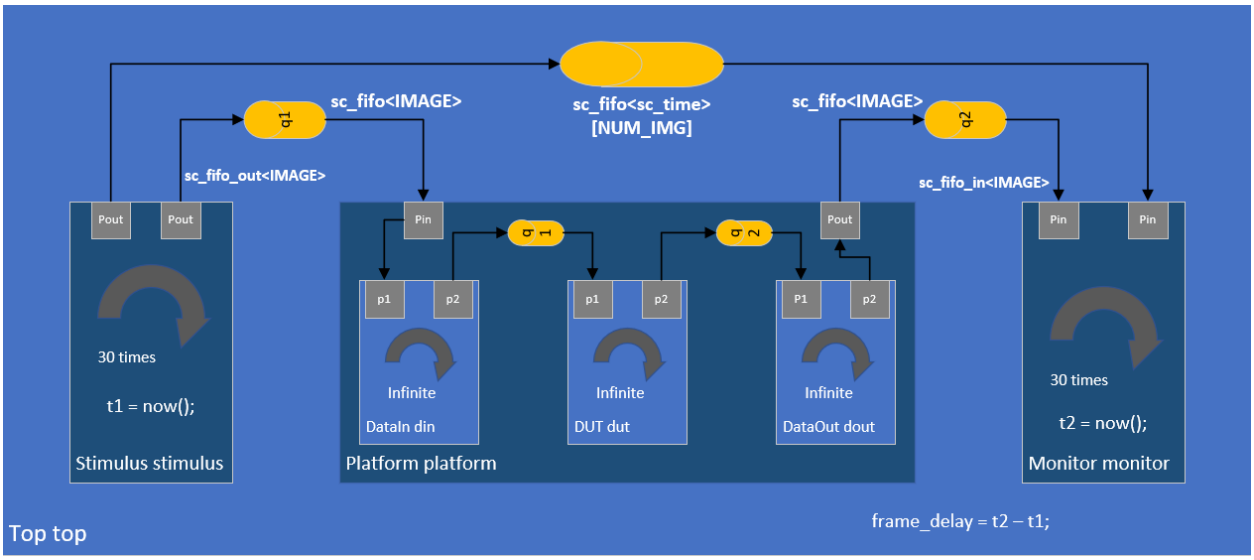

Figure 15

Step 2 - Back-annotated the estimated timing into the main DUT modules. Here we chose the timing measurements from previous stage (7). We back-annotated these delays into our SystemC model by inserting corresponding wait-for-time statements into the main method of each DUT component. For consistency, we placed these wait-for-time statements right after receiving the input data for the function and before its computation code.

Step 3 - Pipelined the DUT into a sequence of 7 stages. Here, we need to make sure that all the data in the DUT pipeline is passed only from one stage to its immediate next stage, and we use buffer sizes of 1 everywhere. Since our DUT components are already communicating via sc_fifo channels, there was not much to do for this step, the model was already pipelined. Because of this step, our DUT model contains a total of 7 pipeline stages, 3 of which are wrapped inside the Gaussian Smooth module. Before parallelizing blur our bottleneck was the blurY module as depicted below:

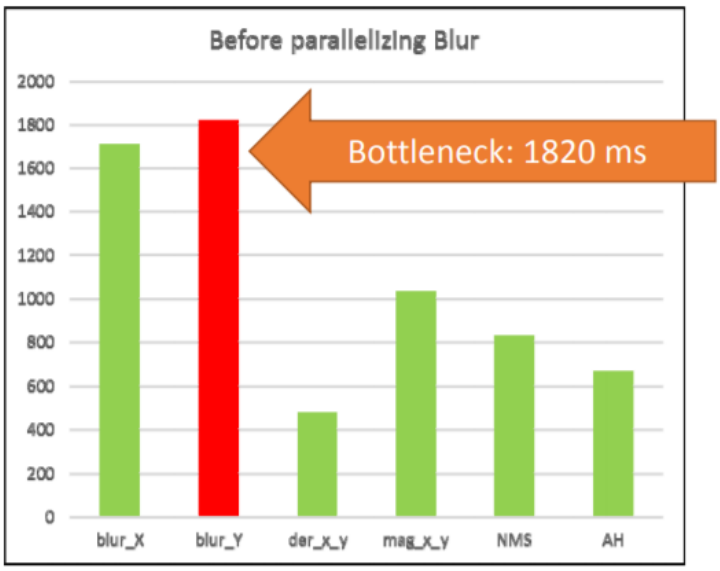

Figure 16 (c) Zhongqi Cheng

Important thing to note after this step is that we did not see an actual speed improvement of 7x which is because all pipelining stages are not equal, and the total simulated time is not max stage delay multiplied by 30 frames because pipelines have a filling and flushing stage which have an addon time. Further the frame delays increase with each frame because it's in a FIFO channel and it increases because other frames are stuck in the queue after starting the timer on their start time.

Step 4 - Sliced the BlurX and BlurY modules into parallel threads. These methods blurX and blurY iterate through the pixels in X and Y direction in a single for loop which can be divided among multiple threads with different starting points of the loop. Both blocks are straightforward to optimize by parallelizing the operations in the rows and columns, respectively. Note we choose to parallelize BlurX and BlurY among 4 threads because these are the bottlenecks and the prototyping board Raspberry Pi Model 3B has quad core processor so parallelization to reduce their timing would be a straightforward option. For this we extended the existing BlurX and BlurY modules by using 4 parallel SC_THREADs 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. Also, we adjusted the back-annotated timing delays for the expected speedup of 4x. Did the same for BlurY as well. The model finally looks like the figure below:

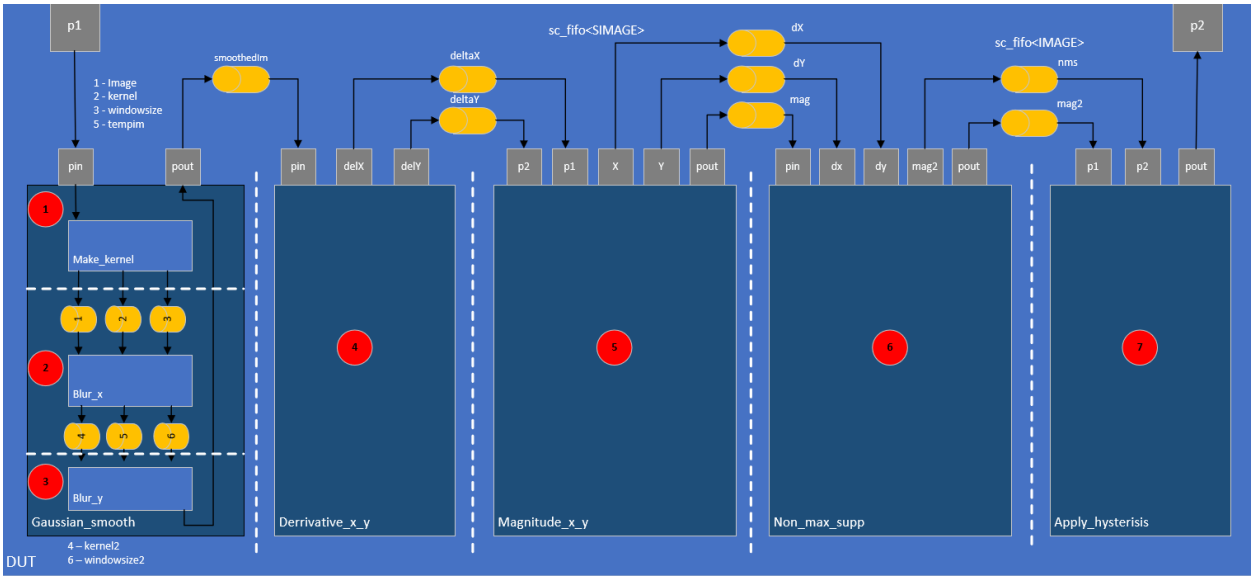

Figure 17

Receive, Make_Kernel	0 ms	0 ms
BlurX	1710 ms	427 ms
BlurY	1820 ms	455 ms
Derivative_X_Y	480 ms	480 ms
Magnitude_X_Y	1030 ms	1030 ms
Non_Max_Supp	830 ms	830 ms
Apply_Hysteresis	670 ms	670 ms
TOTAL:	6540 ms	3892 ms

Figure 18 (c) Rainer Doemer

Results as shown above and below show that pipelining and parallelization did do the trick for us.

	Pipe stages	Max Stage Delay	Latency	Throughput	Pretend time	Simulator run time
Step 1	1	-	-	-	0 ms	3 min
Step 2	1	6.54 sec	6.54 sec	0.152 FPS	196200 ms	3 min
Step 3	7	1.82 sec	12.74 sec	0.55 FPS	57500 ms	3 min
Step 4	7	1.03 sec	7.21 sec	0.97 FPS	33762 ms	3 min

An important thing to note from the Table is that the simulator run time remains to be the same and does not improve on the server because for Reference Simulator IEEE SystemC specifies cooperative multi-threading i.e. A single thread is active at any time (even if multiple cores are available). The simulator is sequential, so it cannot really execute the four BlurX/Y slices in parallel, nor the pipelined stages. Also, the speed of the CPU on the Linux sever is fixed so no matter how you change the wait statement, the simulator speed will never be affected.

Throughput optimization of the Canny Edge Decoder

This is the final chapter in the modeling of our application example, the Canny Edge Detector, as an embedded system model in SystemC suitable for SoC implementation. We applied further optimizations to reduce the execution time of the pipeline stages. 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.

Step 1 - Improve the test bench to include the logging of frame throughput. To measure this, we extended the timing logs produced by the test bench. Specifically, we let the Monitor module measure and report the frame throughput upon receiving a new frame 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.

```
1.03 seconds after previous frame, 0.970874 FPS.

1.03 seconds gap b/w two frames is the delay in MAGNITUDE_X_Y from Stage 7.
```

After step 4 of the previous stage (8), where BlurX & blurY modules were parallelized by four threads, the next bottleneck in the pipeline became magnitude_x_y as shown in the image below.

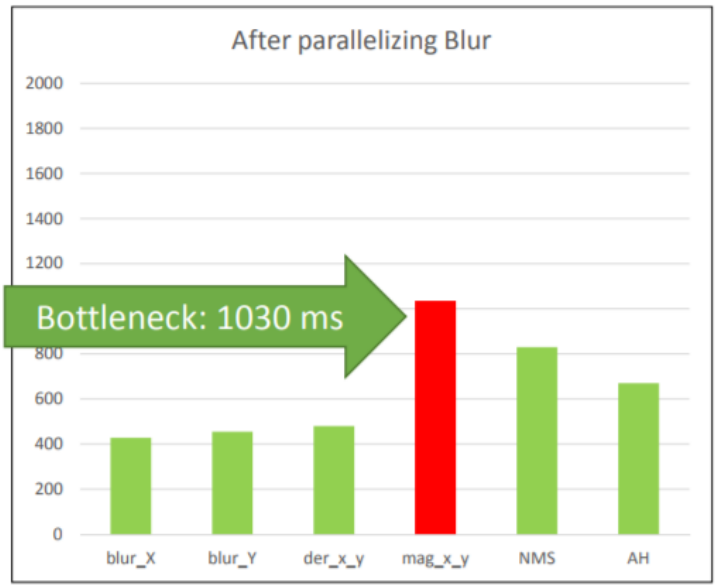

Figure 19 (c) Zhongqi Cheng

Step 2 - Reduce the timing delays of the stages in the pipeline. One easy option is always to enable compiler optimizations. A general-purpose optimization flag for the GNU compiler is –O2. We used **-O3 -mfloat-abi=hard -mfpu=neon-fp-armv8 -mneon-for-64bits.** We ran the compiler again with optimizations enabled, and then measured the timing again. We got the following results:

```
******** Make_gaussian_kernel ******** Avg. run time 0.000023 secs
******** Blur_X ******** Avg. run time is : 0.291107 secs
******** Blur_Y ******** Avg. run time is : 0.473272 secs
******** DERRIVATIVE_X_Y ******** Avg. run time is : 0.151551 secs
******** MAGNITUDE_X_Y ******** Avg. run time is :  0.155745 secs
******** NON_MAX_SUPP ******** Avg. run time is : 0.279303 secs
******** APPLY_HYSTERESIS ******** Avg. run time is : 0.219128 secs
```

Figure 20

Back annotating these delays into our System C model, we get the following throughput

```
0.279 seconds after previous frame 1 , 3.58423 FPS.
```

As the results show the next bottleneck now in the pipeline is Non_max_supp which we tackle next.

Step 3 - Replace floating-point arithmetic with fixed-point calculations (NMS module only). In this, we experiment with fixed-point arithmetic that often can improve execution speed when floating-point operations are slow. In other words, we wanted to replace existing floating-point calculations by faster and cheaper fixed-point arithmetic with an acceptable loss in accuracy.

For this, in the non_max_supp function of the source code of our model we identified those variables and statements which use floating-point (i.e. float type) operations. There were only 4 variables defined with floating-point type. Changed their type to integer (int). Next, we had to adjust all calculations that involve these variables. We had to add appropriate shift-operations so that the integer variables can represent fixed-point values within appropriate ranges. Using fixed point arithmetic instead of floating point (for the bottleneck non_max_supp), we see a decrease in the performance of non_max_supp method on raspberry pi.

```
******** NON_MAX_SUPP ******** Avg. run time is : 0.638427 secs.
```

Back annotating the delay for NON_MAX_SUPP in the SystemC model we see a decrease in throughput as follows:

```
0.638 seconds after previous frame 1 , 1.5674 FPS.
```

Also on using the imagediff tool b/w image produced after step 2 and step 3 produces the following result:

```
6 mismatching pixels (0.000%) identified in diff.pgm.
```

So overall the inaccuracy added after fixed point arithmetic is not too much of a concern but since the Raspberry Pi Model 3B already has a fast FPU unit inside. This 3rd step should be avoided to achieve a better FPS throughput.

	Pipe stages	Max Stage Delay	Latency	Throughput	Pretend time	Simulator run time
Step 1	7	1.03 sec	7.210 sec	0.97 FPS	33762 ms	3 min
Step 2	7	0.279 sec	1.953 sec	3.58 FPS	9087 ms	3 min
Step 3	7	0.638 sec	4.466 sec	1.57 FPS	19857 ms	3 min

Table above shows that we managed a maximum throughput of **3.58** as a part of this project.

Summary and Conclusion

Lessons learnt

For creating a performant embedded system model that takes purely sequential C code as the starting point, features within SystemC as a System Level Description Language (SLDL) provides a lot of design methodologies like Top down design approach, Structural hierarchy – A top level design approach and different kinds of executions like pipelining and parallelization.

Enabling compiler optimizations can further help in optimizing the throughput at a small expense of increase in compile time of the code.

Knowing the hardware of the main SoC that will hold and run the design under test is very important. If its hardware does not have an FPU to support floating point operations its best to move to fixed point arithmetic to boost the throughput at expense of accuracy.

The temperature of the CPU on the prototyping board like raspberry pi can also greatly affect the timing measurements because they generally have a defense mechanism in place that reduces the clock frequency of the CPU when its temperature crosses a threshold value which if not done could fry the board.

Future work

As a part of this project, we could only manage a throughput of 3.58 FPS after applying System C design methodologies and applying various kinds of code optimizations like pipelining, parallelization, compiler optimizations. We could explore more ways to further enhance this throughput to 30 FPS like reducing the output image resolution.

One easy way to increase the throughput to 30 FPS or more would be to slice the resolution of the image we are operating on i.e. 2704 x 1520 to 1/4th their size that could theoretically give a 16x gain in the throughput taking it well beyond the target throughput of 30 FPS.

We could also, (if time and budget allow) design an ASIC hardware or GPU acting as a coprocessor for the BlurX and BlurY modules.

Also, we could do RTL component modelling in SystemC for our canny model. We could have a pin accurate model in hand after that.

We could also simulate the improvements in power consumption of the SoC (creating edges video in real time) that the final model would have made if added to a digital camera on a drone.

Instead of converting to greyscale images using another image tool we could have added a preprocessing step to convert color frames to grey scale frames.

References

- http://marathon.csee.usf.edu/edge/edge_detection.html, Heath, M, Sarkar, S, Sanocki, T, and Bowyer, K
- https://en.wikipedia.org/wiki/Canny_edge_detector
- ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src
- www.doulos.com/knowhow/systemc
- http://www.accellera.org/
- Assignment slides of Professor Rainer Dömer
- Lecture slides of Professor Rainer Dömer
- Slides of the TA - Zhongqi Cheng