Embedded System Design and Modeling

EE382N.23, Fall 2017

Lab #1

Specification

Part (a)+(b) due: September 27, 2017 (11:59pm)
Part (c)+(d) due: October 4, 2017 (11:59pm)

Instructions:

- Please submit your solutions via Canvas. Submissions should include a lab report (single PDF) and a single Zip or Tar archive with the source and supplementary files (code should include a README and has to be compilable and simulatable by running 'make' and 'make test', respectively).
- You are allowed to work in teams of up to three people and you are free to switch partners between labs and the project. Please submit one solution per team.

SUSAN Edge Detector Specification Model

The purpose of this lab is to convert the SUSAN (*smallest univalue segment assimilating nucleus*) edge detection reference code into a clean SpecC specification model that conforms to the structure, rules and specification modeling guidelines discussed in class. A C reference implementation of the core SUSAN algorithm is part of the automotive subset in the MiBench benchmark suite (http://www.eecs.umich.edu/mibench/). A slightly modified version of the SUSAN code that we will use as a starting point for our design, is available at

```
/home/projects/courses/fall_17/ee382n-23/SUSAN.tar.gz
```

Install the SUSAN edge detector example as follows:

```
% mkdir lab1
% cd lab1
% gtar xvzf /home/projects/.../ee382n-23/SUSAN.tar.gz
% cd SUSAN
```

Now you can compile and run the example using the provided `Makefile`:

```
% make
% make test
```

The latter command runs the example on a "input_small.pgm" sample input and validates the generated "output_edge.pgm" file against an expected "golden.pgm" reference output.

You are free to perform the conversion process into a SpecC specification in one step. However, good software engineering principles highly recommend breaking the process into as many small steps as possible. That way the model can be compiled and simulated after each change, to continuously validate that it is still syntactically and functionally correct:

(a) First, we need to become familiar with the reference code and prepare it for conversion:

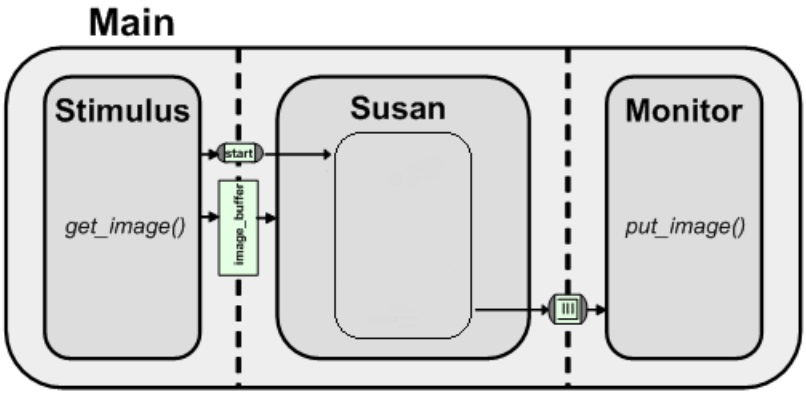
1. Browse the source, analyze the source code structure and draw a high-level block diagram of the function hierarchy and their communication dependencies (critical variables). Submit the block diagram of the software architecture of the reference code as part of your lab report. Note: in addition to the edge detection, the original source code also supports image corner detection and smoothing. You can ignore those functions that are not used in the edge detection code. Run the program with -e option:

```
%./susan input_small.pgm out.pgm -e
```

2. Clean up the source code to make it static and synthesizable. Modify the sources for a fixed input image sensor size of 76×95 pixels. Simplify the code as much as possible, remove any unnecessary communication/dependencies, and convert all dynamic memory allocation into appropriate static data structures (i.e. remove all *malloc* calls, which are not implementable in hardware and not supported on many embedded processors and operating systems). At the end, put each function called inside the main() function in a separate header/source file (<*function_name*>*.h/.c*) in order to make the example easier to manage. Report on the code changes that you performed.
3. In preparation for conversion to SpecC, you should make sure that parameters passed between functions (which will become communication between behaviors) are not of pointer type. Explicitly pass arrays and convert pointers into arrays into array indices instead.

(b) Next, transform the simplified, static code into an initial SpecC model with proper behavioral and structural hierarchy:

1. Gradually convert <*name*>*.c/.h* C files into <*name*>*.sc/.sir* SpecC modules, where each module gets translated into one or more SpecC behaviors. Introduce a single behavior of appropriate name in each file. Let the behavior encapsulate all local variables and functions (i.e. files must not have any variables or functions outside of behaviors). Replace parameters with equivalent behavior ports for external communication. Ensure that behaviors are free of side effects, i.e. that they only communicate with other behaviors through their ports and do not access any global variables outside of their body. Note: it is recommended to hierarchically encapsulate *SetupBrightnessLut* and *SusanEdges* into a separate parent *DetectEdges* behavior (and *detect_edges.sc* file).
2. Convert *get_image.c* and *put_image.c* into *Stimulus* and *Monitor* behaviors for the testbench, respectively. The *Stimulus* behavior reads the input file into a shared *ImageBuffer* port (`get_image`) and then sends a start signal over a *c_handshake* channel. The *Monitor* reads bytes of manipulated image data from a *c_queue* interface and writes them into an output file (`put_image`). Create a top-level Susan behavior that communicates with the testbench and executes child behaviors sequentially in a loop. Finally, introduce a main *susan_edge_detector.sc* file that contains the *Main* behavior implementing a typical testbench setup running concurrent *Stimulus*, *Susan* and *Monitor* sub-behaviors. Your whole setup should look something like this:

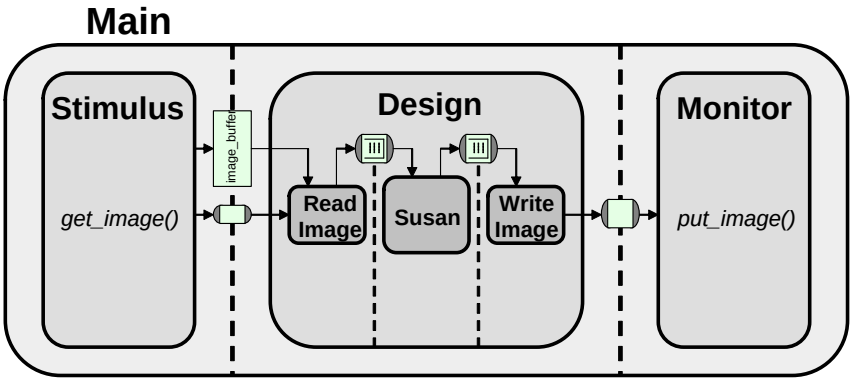


3. Remove the *.h* files and compile all *.sc* sources into *.sir* files and check for compile errors. Finally, compile the top-level *susan_edge_detector.sc* source into an executable and simulate the design. Validate the generated output against the known good data to ensure the design is working correctly. Note: it is highly recommended to update the *Makefile* in order to automate the compilation process using the `make` utility.

(c) In the last parts of this lab, you are to parallelize the SpecC specification model to obtain a clean and parallel/pipelined specification that can be used for design space exploration and synthesis:

1. During the implementation process we are interested in printing the simulated time it took for encoding a single image. To achieve this, insert timing checks into the testbench. Update the *Stimulus* behavior to wait for 1000 time units before sending the first start signal to the *Susan*. Furthermore, modify the *Stimulus* to send consecutive images (you can just send the same image over and over for a fixed number of times) with some fixed delay between images. Print the total delay required for processing of each single picture (from sending the start signal to receiving the last byte of the image) in the *Monitor*. Simulate the model to check timing info is printed correctly (since our specification model is untimed, delays should be zero at this point).

2. For synthesis, we need to develop an accurate model of the actual I/O structure for the susan edge detector. The testbench (*Stimulus* and *Monitor*) will not be synthesized (and hence can contain non-synthesizable functionality, such as file accesses). As a result, we can also not refine the communication to the testbench (that means regardless of the refinement process, testbench communication will always use abstract communication channels or variables but never any bus). To more accurately reflect the I/O structure of the real system, we want to create another set of parallel behaviors representing I/O blocks that will be synthesized into hardware I/O components, such as network or disk controllers that interact with other peripherals. During synthesis they will eventually be replaced with pre-designed hardware blocks that implement the real I/O.



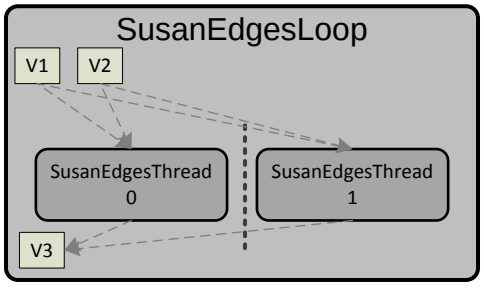
Introduce a *ReadImage* behavior (*read_image.sc*), move the waiting for the start signal into *ReadImage* and send the input image over its outgoing queue after the start signal has been received. Introduce a *WriteImage* behavior (*write_image.sc*) that continuously reads output images from a queue and forwards them into an outgoing double-handshake channel. Introduce an additional level of hierarchy as a *Design* behavior (*design.sc*) that sits between *Monitor* and *Stimulus* and is a parallel composition of *ReadImage*, *Susan* and *WriteImage* instances communicating via *c_queue* channels, where both the input and output queues should have space for 1 image.

3. Finally, it is time to parallelize the *Susan* model. Convert the top-level *Susan* model into a KPN that maximally exposes available task-level parallelism. Use appropriate behavioral and structural composition. Your final Susan hierarchy should only consist of behaviors that are either leaf behaviors with code, or a pure *par, seq* or *fsm* composition. In the process, modify behaviors as necessary, e.g. turn them into non-terminating Kahn processes by introducing an additional level of behavior hierarchy that runs a corresponding child behavior in an endlessly looping `fsm`. Use *c_typed_queue* channels of appropriate type for communication. An example and tutorial for use of typed queues can be found at:

   ```
   $SPECC/examples/sync/c_bit64_queue.sc
   $SPECC/examples/sync/typed_queue.sc
   ```

   Can your model run in bounded memory, i.e. is there an upper limit to the queue sizes required to run your model? How do queue sizes influence the possible execution schedules and parallelism available in the model? Justify the choice of queue sizes in your code. Is your model deadlock-free and deterministic?

4. In addition to any top-level task parallelism, the SUSAN algorithm exhibits ample data parallelism across the outer loops within in each leaf behavior. As in many image processing applications, this means that multiple instances of a behavior can, for example, operate in parallel and independently on different parts of a larger image. Modify at least one of the leaf behaviors to model and expose the available data parallelism. Hint on modeling such data parallelism in SpecC: Decompose the outer loops inside each behavior into multiple parallel instances of an appropriate subbehavior, then pass the entire input data plus some indication of the range of data to work on (such as a thread ID or index) to each subbehavior. You are allowed to select an arbitrary number of threads. The following sketches an example of decomposing a loop into two data-parallel threads (this is not a full solution):



(d) Finally, we want to perform additional validation and analysis of the resulting model:

1. We can execute the model in a parallel simulation environment to validate whether your parallelism works safely. As discussed in class, a parallel simulation can not only achieve faster speeds, but will also help to detect concurrency problems. It will run concurrent blocks truly in parallel, i.e. in non-predetermined order, which will potentially expose and debug any non-determinism issues in your code.
   You can create a parallel simulation of your model as follows:

   ```
   % scc <design> <opts> -par
   ```

   This will create an executable that will parallelize the simulation across 4 host CPU cores by default. Compile and execute your model in the parallel SpecC simulator. Report on your results. Measure the runtime of both a sequential and parallel simulation of your

model. (Use the `/usr/bin/time` utility to get reliable measurements. Make sure that your testbench feeds a large enough number of consecutive images into the design such that the simulation time is long enough to get meaningful data – the resolution of host timer measurements on Linux is only 10ms). What differences in simulation speeds do you see? Discuss and explain the speed results.

2. At this point, you should have successfully converted the design into a model ready for synthesis. Make sure your final model compiles, simulates and produces the golden reference output. Include a brief description of the status of your model in the lab report. Make sure to include a documentation of your behavior hierarchy (output of the `sir_tree –blt` command) for each of your models. Also include a graphical representation of your model. You can open your model in the SpecC viewer and include a screenshot (double clicking in the viewer window increases the levels of hierarchy shown, and display of structural connectivity can be enabled from the View menu):

   ```
   % scchart <design>
   ```

   Could your model be improved to better support exploration and implementation, if at all? Is there additional parallelism that could be exposed or could the application be better modeled using a different MoC? Specifically, can the application be modeled in SDF form? If not, why not? If yes, how would the model have to be changed? In general, how does the code differ between a KPN and SDF modeled in SpecC? Could a tool be developed to automatically recognize whether a model is a KPN or SDF? What advantages and disadvantages does it have to model an application as SDF versus KPN?