Embedded System Design and Modeling

EE382V, Fall 2011

Lab #1

Specification

Part (a) due: September 26, 2011 (11:59pm)
Part (b) due: October 3, 2011 (11:59pm)
Part (c) due: October 10, 2011 (11:59pm)

Instructions:

- Please submit your solutions via Blackboard. 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.

Digital Camera Specification Model

The purpose of this lab is to convert the JPEG Encoder reference code into a clean SpecC specification model of the digital camera design that conforms to the structure, rules and specification modeling guidelines discussed in class. A C reference implementation of the core JPEG encoder, which we will use as a starting point for our design, is available at

```
/home/projects/courses/fall_11/ee382v_17190/jpegencoder.tar.gz
```

Install the JPEG encoder example as follows:

```
mkdir lab1
cd lab
gtar xvzf /home/projects/…/ee382v_17190/jpegencoder.tar.gz
cd jpegencoder
```

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

```
make
make test
```

The latter command runs the example on a "ccd.bmp" sample input and validates the generated "test.jpg" file against an expected "golden.jpg" 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:

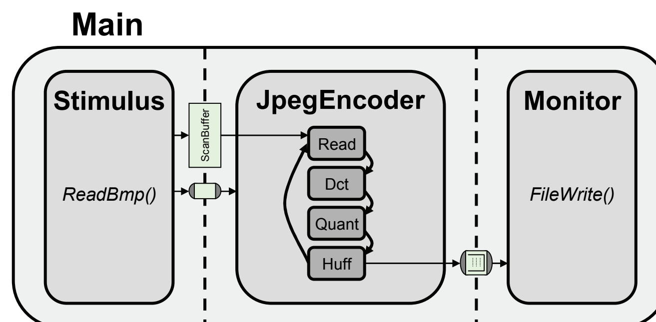
(a) First, we need to become familiar with the reference code and prepare it for conversion:

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.
2. Next, clean up the source code to make it static and synthesizable. Modify the sources for a fixed input image sensor size of 116×96 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). Report on the code changes that you performed.

(b) Next, we transform the simplified, static code into an initial SpecC model with proper behavioral and structural hierarchy. Gradually convert *<name>.c/.h* C files into *<name>.sc/.sir* SpecC modules, where each module gets translated into one or more SpecC behaviors, which can then be hierarchically imported and composed into an overall design:

1. Convert *read.c*, *dct.c*, *quantize.c*, *zigzag.c* and *huffencode.c* into corresponding *.sc* files. 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). Convert the externally accessible function listed in the *.h* file into the behavior's `main` method and 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.
2. Convert *preshift*, *chendct* and *bound* methods in *dct.sc* into separate behaviors and transform the *Dct* behavior into a sequential composition of these subbehaviors. Connect child behaviors to communicate through variables mapped onto their ports.
3. Introduce a new behavior and file *huff.sc* that implements the sequential composition of (imported) *Zigzag* and *Huffencode* child behaviors. Connect behavior ports to appropriate external ports or local variables throughout the hierarchy.
4. Convert *ReadBmp_aux.c* and *file.c* into *Stimulus* and *Monitor* behaviors for the testbench, respectively. The *Stimulus* behavior reads the input file into a shared *ScanBuffer* port (`ReadBmp`) and then sends a start signal over a *c_handshake* channel. The *Monitor* reads bytes from a *c_queue* interface and writes them into an output file (`FileWrite`) continuously, one byte at a time until the end-of-file marker is reached.
5. Convert *jpegencoder.c* into a *jpegencoder.sc* file and behavior that first waits for a start signal via a *c_handshake* interface and then executes *ReadBlock*, *Dct*, *Quantize* and *Huff* child behaviors sequentially in a loop. Let child behaviors communicate through variables mapped onto their ports and introduce external ports and mappings as necessary.
6. Introduce a top-level *digicam.sc* file that contains the *Main* behavior implementing a typical testbench running concurrent *Stimulus*, *JpegEncoder* and *Monitor* subbehaviors:



The *Stimulus* is connected to the *JpegEncoder* through a shared *ScanBuffer* variable representing the CCD sensor array. In addition, a *c_handshake* channel represents the

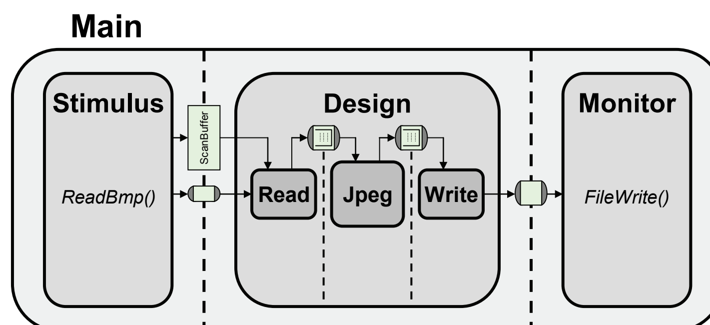
signal that the camera shutter has been triggered and that encoding of the CCD sensor picture should be started. At the other end, the *Monitor* receives a stream of encoded bytes from the Huffman encoder (*Huffencode*) through a *c_queue* representing the file I/O interface.

7. Remove the *.h* files and compile all *.sc* sources into *.sir* files and check for compile errors. Finally, compile the top-level *digicam.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) Lastly, we finalize the SpecC specification model to obtain a clean and parallel/pipelined specification that can be use for design space exploration and MPSoC synthesis:

1. During the exploration 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 encoder. Feel free to insert additional delays between start signals of consecutive images to model timing of user start button presses. Make the start time of each image available to the *Monitor* and print the total delay required for encoding of each single picture (from sending the start signal to receiving the last byte of the image). Simulate the model to check timing info is printed correctly (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 digital camera. The testbench (*Stimulus* and *Monitor*) will not be synthesized (and hence can contain non-synthesizable functionality – e.g. 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 (CCD Control and Flash Interface). These I/O behaviors can then communicate with outside behaviors, i.e. the unrefined testbench. During backend synthesis they will eventually be replaced with pre-designed hardware blocks that implement the real I/O with the CCD sensor and the Flash memory.



Move the *ReadBlock* behavior outside of the *JpegEncoder*, move the waiting for the start signal into *ReadBlock* and modify *ReadBlock* to independently loop over all 180 blocks in a picture and send them over its outgoing queue after the start signal has been received. Introduce a *WriteBlock* behavior (*write.sc*) that continuously reads bytes 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 *ReadBlock*, *JpegEncoder* and *WriteBlock* instances communicating via *c_queue* channels, where the input queue should have space for 1 block of data and the output queue should be 512 bytes in size.

3. Parallelize the *JpegEncoder* into a KPN model with continuously running parallel processes. Remove the *ReadBlock* instance (as discussed above) and change the top-level *JpegEncoder* execution into a single `par` statement in which the three remaining child behaviors communicate via *c_typed_queue* channels of size 1 data blocks. 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
   ```

   Finally, modify *Dct*, *Quantize* and *Huff* to work on continuous streams of input and output data over *c_int64_queue* channels. Change the sequential sub-composition inside *Dct* and *Huff* behaviors into an `fsm` that runs child behaviors sequentially in an endless loop. Introduce an additional level of hierarchy in *quantize.sc* as a behavior *Quant* that runs *Quantize* in an endlessly looping `fsm`. Replace the top-level *Quantize* instance in *JpegEncoder* with *Quant*.

Your final hierarchy should look like the following SIR tree:

```
% sir_tree -blt digicam.sir
B i o   behavior Main
B i c   |------ Design design
B i c   |       |------ JpegEncoder jpeg
B i f   |       |       |------ Dct dct
B i l   |       |       |       |------ Bound bound
B i l   |       |       |       |------ ChenDct chendct
B i l   |       |       |       \------ Preshift preshift
B i f   |       |       |------ Huff huff
B i l   |       |       |       |------ Huffencode huffencode
B i l   |       |       |       \------ Zigzag zigzag
B i f   |       |       |------ Quant quant
B i l   |       |       |       \------ Quantize quantize
C i l   |       |       |------ c_int64_queue dctout
C i l   |       |       \------ c_int64_queue quantizeout
B i l   |       |------ ReadBlock read
B i l   |       |------ WriteBlock write
C i l   |       |------ c_queue dataout
C i l   |       \------ c_int64_queue dctin
B i l   |------ Monitor monitor
B i l   |------ Stimulus stimulus
C i l   |------ c_double_handshake data
C i l   \------ c_handshake start
```

Congratulations on successful conversion! 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. Also discuss if and how this model could be improved to better support exploration and implementation. Is there additional parallelism that could be exposed or could the application be better modeled using a MoC other than a KPN, e.g. in SDF form? Furthermore, can you formally prove that the design can run in bounded memory? What influence does the queue size have on the implementability/schedulability of the model, i.e. what effect (e.g. on the available concurrency) would it have to increase the top-level queue depths to more than 1 data block each?