Embedded System Design and Modeling

EE382V, Spring 2014

Lab #2

Exploration

Part (a) due: March 17, 2014 (11:59pm)
Part (b) due: March 24, 2014 (11:59pm)
Part (c)+(d) due: March 31, 2014 (11:59pm)
Extra credit: Any time before the last class day

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.

SUSAN Edge Detector Design Space Exploration

The purpose of this lab is to perform design space exploration and to bring the SUSAN edge detector example down to an (optimal) implementation using the System-On-Chip Environment (SCE). As discussed in class, a system specification consists of a functional model plus associated architectural and other implementation constraints. We will be using the specification model that you developed in Lab #1 for the former, while the latter will be largely dictated by SCE's capabilities, restrictions and provided databases (as well as bugs in the tool).

Note that this assignment is by nature open-ended. There is no single solution, and part of the evaluation is who can come up with the best design. The instructions below are just meant to provide a general framework and some initial directions/tips. We have not tested all the possibilities ourselves, but want to explore what a good design for this example looks like. You will likely run into bugs in the tools, so please bring up any issues in class such that everybody can learn from them.

SCE is installed next to the SpecC tools on the ECE LRC Linux servers. Instructions for accessing and setting up SCE and the tutorial are posted on the class website:

http://www.ece.utexas.edu/~gerstl/ee382v_s14/docs/SCE_setup.pdf

Again, once logged in (e.g. remotely via `ssh -X` and make sure to have an X11 server running locally), you need to setup the environment:

```
module load sce
```

SCE comes with an extensive tutorial and it is highly recommended to go through the first part of the tutorial that demonstrates SCE's system exploration and synthesis capabilites on a GSM Vocoder design example. The tutorial instructions are available as part of the SCE installation (see below) and online at:

http://www.cecs.uci.edu/~cad/publications/tech-reports/2003/TR-03-41.tutorial.pdf

Note, however, that the tutorial is based on an older version of SCE. As such, some steps have changed and communication design steps have been expanded. A list of errata with all modified and added tutorial steps necessary for the current SCE version is available on the class website:

http://www.ece.utexas.edu/~gerstl/ee382v_s14/docs/SCE_Tutorial_Errata.pdf

To run the tutorial, setup a local working directory for the tutorial demo, launch the SCE GUI and follow the steps of the tutorial document:

```
mkdir demo
cd demo
setup_demo
```
(open `SCE_Tutorial/sce-tutorial.pdf` or browse `SCE_Tutorial/html/`)
(open `SCE_Tutorial_Errata.pdf`, see above)
```
sce &
```

Go through the tutorial up to and including Section 3.

We are now ready to load the edge detector specification model into SCE and start the analysis, exploration and refinement process. We start from the SpecC code that you developed as a result of Lab #1. A reference solution can be found under:
`/home/projects/courses/spring_14/ee382v_17303/susan_spec.tar.gz`

(a) Profile, analyze and estimate the edge detector specification model:

1. Open SCE in the susan edge detector directory and create a new project "susan.sce" (`Project→New`, `Project→SaveAs`…). Adjust the simulator and compiler options as needed. Set the simulation command to
   ```
   /usr/bin/time ./%e && diff –s out.pgm goldgen.pgm
   ```
   Set the compiler verbosity level to 3 and the warning level to 2.
2. Import the *susan_edge_detector.sc* specification model into SCE and add it to the project. Rename the model in the project window to *SusanSpec*.
3. Compile and simulate the model to validate its correctness.
4. Browse the graphical hierarchy chart. Expose all levels of hierarchy and submit a printout of the chart of the complete specification model (`Window→Print`… to file *SusanSpec.ps*).
5. Profile (`Validation→Profile`) the model and generate the bar graph for the raw Computation profile of all behaviors in the *Susan* part of the design. Submit a printout of the computation graph (`Window→Print`… to file *SusanRawProfile.ps*).
6. Allocate a single PE of *ARM7_TDMI* type and a single PE of *HW_Standard* type, using default parameters (100MHz clock frequency). Reanalyze (`Validation→Analyze`) the design and generate the bar graph for the HW/SW Computation profile of all behaviors in the *Susan* part of the design. Submit a printout of the computation graph (`Window→Print`… to file *SusanSpecProfile.ps*).

(b) The next step is to go through the computation (architecture and scheduling) exploration and refinement process. The goal is to find an optimal realization on a system architecture consisting of up to three ARM processors or hardware accelerators:

1. Allocate two custom hardware PEs of *HW_Virtual* type and name them *INPUT* and *OUTPUT*. Those two custom hardware blocks are placeholders for the peripherals implementing the I/O with the external world. As such, map the *ReadImage* and *WriteImage* behaviors onto the *INPUT* and *OUTPUT* PEs, respectively.
2. Enable the channel view (`Synthesis→Show Channels`) and map the *in_image* and *out_image* input and output queues at the *Design* level into the *INPUT* and *OUTPUT* PEs,

respectively. This is necessary because we want to have the two I/O blocks implement the dedicated input and output buffers associated with the queues. In general, mapping a complex channel into a PE means that the channel will result in a specific implementation being synthesized as part of that PE. Unmapped complex channels, on the other hand, will simply be resolved into their basic elements without any guarantees about a particular buffer size, for example.

3. Allocate between at least one PE of *ARM7_TDMI* type and as many additional PEs of *ARM7_TDMI* or *HW_Standard* type (with default parameters, i.e. 100MHz clock frequency) as you want/need, and explore possible mapping options of *Susan* behaviors onto PEs. Make sure to reprofile and reanalyze the design (`Validation→Evaluate`) every time you change the allocation or mapping. Perform architecture refinement for every feasible design alternative.

4. If your partitioned model has shared variables between parallel behaviors mapped to different processors (e.g. in the dataparallel parts), those variables need to be mapped into additional shared memory components. (Unlike for variables between sequential behaviors, which SCE will, by default, automatically refine into a distributed realization, there is no other implementation option for concurrently accessed variables.) You can use additional or existing *HW_Standard* components as shared memories. Select `Synthesis→Show Variables` and map any such shared variables into a chosen memory (HW) component.

5. The SCE database currently does not include any multi-core processors. As such, all behaviors mapped to an ARM processors need to be statically or dynamically serialized. (By contrast, HW units can run multiple blocks/behaviors truly in parallel, i.e. don't need to be scheduled.) Explore various feasible scheduling strategies for each allocated ARM processor in each architecture alternative. You can choose between static and round-robin or priority-based dynamic scheduling (with various task priority assignments) of parallel behaviors mapped to the same PE. Note that you should not schedule (i.e. select *None* under dynamic scheduling) ARM processors with only one mapped behavior. Do not schedule any of the I/O hardware units (*INPUT* or *OUTPUT*). Perform scheduling refinement for every feasible design alternative.

6. As part of your exploration process, you should consider modifying the original specification model to better match a particular architecture mapping. While ideal tools should support arbitrary application/architecture mappings, there are many optimizations current tools can't or won't do. For example, based on an analysis of your specification MoC from Lab #1 in relation to the concurrency available in a selected architecture (see also Homework #2), you may want to sequentialize some of the specification code to explicitly realize a semi-static schedule that matches your chosen mapping. Among other tuning, this can be used to optimize memory requirements, which are an issue on the more restricted SWARM targets used in the final synthesis step (see part (d) below).

7. Compile and simulate all generated scheduled architecture models. Record the simulated encoding times for each alternative and plot the design space as points in an encoding time vs. cost graph. Assume that an ARM PE and a hardware accelerator PE have a cost of 100 and 150, respectively. What is the best design?

(c) Identify at least three promising candidate architectures. We can then go into the communication design (network exploration and communication synthesis) process to synthesize the best designs down to a TLM and PAM realization:

1. Open network allocation to define the overall network topology. Busses for each ARM processor in the system should already be pre-allocated and ARM processors should be pre-connected as masters on their respective busses. Connect the *INPUT* and *OUTPUT* hardware PEs as slaves on the same bus as the ARM processor running their direct communication partners (i.e. *SusanEdges* and *EdgeDraw* behaviors, respectively). Note that processors are always masters on their bus, where an additional "slave0" connectivity is reserved by the ARM processor itself and should never be used.
2. Hardware accelerators can play the roles of masters or slaves on any bus. Connect any hardware accelerators as masters, slaves or masters & slaves to achieve required connectivity. As an alternative to communication over the *AHB* busses, you can freely allocate *DblHndShkBus* instances for separate, dedicated connection between any custom hardware blocks (including the I/O blocks *INPUT* or *OUTPUT*).
3. If there is more than one ARM (and hence more than one bus) in the system, allocate transducer CEs of *T_Custom* type to bridge and connect busses as necessary (where transducers are slaves on each bus they connect to). Note that transducers by default only have one port, but any number of additional ports can be created by right-clicking on the transducer name in the Connectivity tab and selecting `Add port`….
4. Perform network refinement and explore different custom packet sizes. By default, each packet going through a transducer can only hold 1 byte. An increased packet size can reduce communication overhead if larger blocks of data are transferred over any transducers in the design. What is the optimal packet size (and why)?
5. Assign the link parameters for each channel on each bus. You can freely choose the interrupt/synchronization scheme. However, due to the mux-based architecture of the ARM/AMBA AHB bus being used, addresses need to be assigned to match the slave connectivity. Specifically, channels served by a particular "slave*N*" have to be assigned a bus address in the range between `0x`*`N`*`0000000-0x`*`N`*`fffffff` (otherwise, you will see a deadlock in the PAM simulation). Note that right-clicking into the link parameter dialog and selecting `Autofill addresses`… should automatically assign proper default addresses.
6. Perform communication refinement to generate both a transaction-level and pin-accurate model of each design. Compile and simulate each model to record the final encoding delays. How much percent communication overhead does each design have?
7. Browse the hierarchy (View→Chart) and source (View→Source) of one of the generated models. Specifically, take a look at the model of an ARM processor that SCE inserts. Can you identify the model of the interrupt controller, the modeling of processor suspension and interrupt handling in the processor core, and the OS model?

(d) In the final step, we will synthesize the actual software binaries for all ARM target processor in our selected candidate designs. To validate final software execution, we will then run the binaries on an instruction-set simulation (ISS) based virtual platform model of each design:

1. We will use a uCOS-II real-time operating system (RTOS) for each ARM in the system. uCOS only supports priority scheduling. As such, make sure that all ARM processors in

your candidate designs either use priority-based dynamic scheduling or do not use an OS at all (i.e. have *None* selected). In the former case, make sure that all tasks have unique priorities assigned (required by uCOS).

2. Select `Synthesis→C Code Generation` to perform backend software synthesis for each ARM processor in your design. In the dialog, select the ARM processor you want to synthesize and use the default parameters for cross-compiler and target OS. Generate an output model with ISS reintegration each. You can choose between a fully cycle-accurate SWARM ISS or a fast functional OVP ISS with only rough timing. Generate simulations with both types of ISSs and record the differences in simulation times and simulated encoding delays. Repeat C code generation until all ARM processors in the design have been replaced with their reintegrated ISS models.

3. Before we can simulate the software code, we need to cross-compile the generated source code into a final target binary. Change into each subdirectory with generated code and compile the executable:

   ```
   cd ARMx
   make
   ```

   (Before compiling the code, we need to make sure that the necessary libraries are linked against while also increasing the uCOS thread stack size for our example. Modify the `Makefile` and add *–lm* to USR_LFLAGS and *–DSTACK_SIZE_DEFAULT=40960* to USR_CFLAGS.)

4. Compile and simulate the design. The code will now run the real binary in an instruction-accurate simulator for the ARM processor(s). You might see some IRQ messages flying by as the ISS is running, but in the end the simulation should stop after some time when the pictures are encoded. Congratulations, we achieved a full-system co-simulation of the actual target software binary together with its surrounding hardware for the complete SoC! Plot the final cost vs. performance for each alternative in a design space graph similar to (b)-7. Compare the final encoding times to the previous PAM simulation result, how much differences are there?

Submit a lab report that documents all your steps and includes a discussion and analysis of your results and observations. Record and document the changes and trends in model complexities (`File→Statistics`), simulation runtimes and simulated encoding delays between different steps and models in the design process. Assuming that a SWARM simulation provides cycle-accurate results, show the tradeoffs in simulation speed vs. accuracy of different design models. What conclusions can you draw?

Extra credit: Can you improve the timing accuracy of the TLM or PAM simulations? Hint: SCE inserts `waitfor()` statements into behavior code to model estimated execution timing based on initial source-level profiling data (step (a)). By their nature, those early profiling results are coarse and inaccurate. You can significantly improve estimated timing by adjusting back-annotated `waitfor()` delays, e.g. based on measurements taken on the cycle-accurate ISS.

Finally, what is the optimal architecture, and why is it better than others? Explain and discuss the differences in performance you see between different designs.