Embedded System Design and Modeling

EE382V, Fall 2008

Homework #2

System Design Flow and System-Level Design Tools

Assigned: September 25, 2008
Due: October 9, 2008

Instructions:

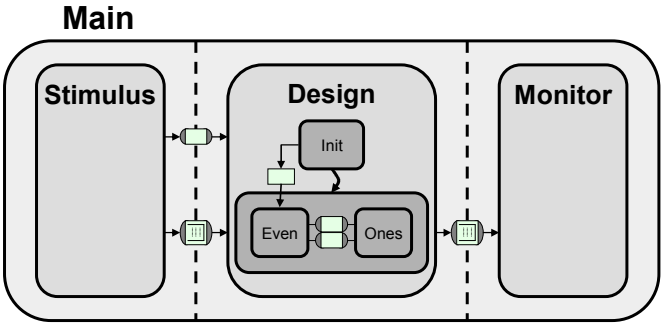
- Please submit your solutions via Blackboard. Submissions should include a single PDF with the writeup and an archive for any supplementary files.
- You may discuss the problems with your classmates but make sure to submit your own independent and individual solutions.
- Some questions might not have a clearly correct or wrong answer. In such cases, grading is based on your arguments and reasoning for arriving at a solution.

Problem 2.1: System Design and Modeling Flow

For this problem, we will further upgrade the parity checker example developed in Homework 1, Problem 1.8 to a proper specification model (conforming to the structure, rules and guidelines discussed in class) and then manually refine it down to computation and communication models:

(a) Starting from the code you developed in Problem 1.8, modify the parity checker into a parity generator/encoder. At its input, the parity generator accepts a stream of 7-bit words (represented as `char` bytes where the MSB is not used) over a `c_queue` channel. At the output, a `c_queue` channel produces the stream of parity-encoded words (MSB is the parity bit). Furthermore, add an initialization behavior before the actual `Parity` generator that waits for an external start message and sets an internal flag to select odd or even parity encoding depending on a mode contained in the start message.

Finally, enclose this design into a typical testbench setup. Modify the top level of the example (`Main` behavior) to describe a proper structure consisting of `Stimulus`, `Design` and `Monitor` behaviors:

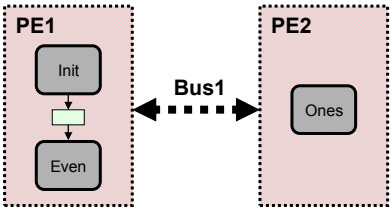


The `Stimulus` should read the mode and the stream of input words from a file and feed them into the design over the input queue. The `Monitor` should receive the encoded words on the output queue and write them to an output file. Make sure that the testbench cleanly terminates the simulation (via an `exit(0)` system call) when the end of the streams has been reached (e.g. in the `Monitor` after a fixed number of words have been received). We will reuse this same testbench as we go through the design process. At

every step you can then use the `diff` command to compare the generated output file with a file of known-good/golden values.

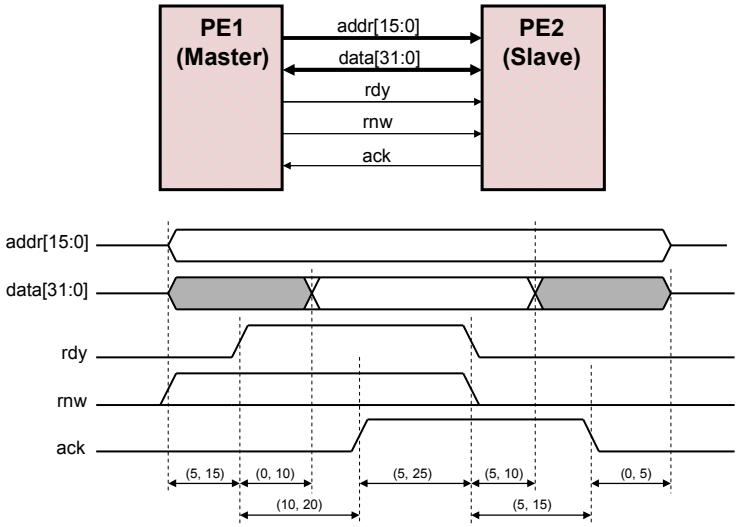
Briefly describe if and how this specification model employs the concepts of and follows the guidelines for granularity, encapsulation, hierarchy, concurrency, and communication.

(b) Assume a partitioning where `Init` and `Even` behaviors are mapped to *PE1*, the `Ones` behavior is mapped to *PE2*, and everything is statically scheduled:



Manually refine the specification model from (a) into a computation model where the `Design` reflects this partitioning. Insert execution delays of 30/50 time units per word in `Even`/`Ones`. Briefly describe the transformation steps you applied.

(c) Assuming that *Bus1* connecting *PE1* (master) and *PE2* (slave) uses a modified double-handshake protocol according to the following timing diagram. In this protocol, the master signals the type of transaction (read or write) to the slave through an additional *rnw* (read, not write) control wire:



Implement a protocol channel for this bus (with master and slave interfaces and corresponding `masterRead/Write` and `slaveServeRead/Write` transactions). Assume worst case delays. Implement both a pin-accurate and a transaction-level model of the bus. Manually refine the computation model from (b) down to a pin-accurate and transaction-level communication model of the system using and instantiating these bus channels (inlined into the PEs or as-is between PEs, respectively). Briefly describe the transformation steps you applied. Validate that both bus models produce the same simulated delays and try to measure and quantify the speed difference between them.

Simulate all models to validate their correctness. Report on lines of code for each model. Turn in the source code for all models and all input and output test files.

Problem 2.2: System-On-Chip Environment (SCE)

The goal of this problem is to make you familiar with the System-On-Chip Environment (SCE) by going through the tutorial that demonstrates SCE on the GSM Vocoder design example introduced in class. 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/ee382Vf08/docs/SCE_Tutorial_Errata.pdf

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/ee382Vf08/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 run the provided setup script (depending on your $SHELL):

```
source /home/projects/courses/…/sce-20080601/bin/setup.{c}sh
```

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

```
mkdir demo
cd demo
setup_demo
ls
acroread SCE_Tutorial/sce-tutorial.pdf &
   (or point your web browser to SCE_Tutorial/html/)
acroread $SPECC/doc/SCE_Tutorial_Errata.pdf &
sce
```

Read and go through the tutorial up to and including Section 3 (you are free to venture into HW and SW synthesis steps but those have not been tested and are not required at this point; use at your own risk but if you do, we'd be happy to hear about any successes/failures). Make sure to simulate all the generated models and generate both a TLM and a PAM in the final Communication Synthesis step.

Report on model complexities (File→Statistics), simulation times and simulated encoding delays (maximum/worst-case delay over all frames) after each design step. Hint: To have simulation times reported after each simulation run, go to Project→Settings→Simulator and prepend `/usr/bin/time` in front of the Simulation Command.

Problem 2.3: Exploration and Refinement

In this problem, we will take the parity encoder/generator example from Problem 2.1 and run it through the SCE exploration and refinement process:

(a) Go into your working directory for the new parity example from Problem 2.1(a), launch `sce` and import the complete specification model (`tb.sc`) into SCE. Create a new project, add the parity specification model to it and rename it (e.g. to `spec.sir`). Simulate and profile the model. Which behavior is the most computation intensive (and why)? Which behavior is the most communication-intensive (and why)?

(b) Set the `Design` behavior as the top-level in SCE, allocate the same *Motorola_DSP56600* processor (using default parameters) and *HW_Standard* custom hardware unit that were used in the Vocoder tutorial, and explore the following system platforms:

i. Pure software solution with everything running on the DSP.

ii. Software on the DSP with `Ones` mapped to the HW unit.

iii. HW/SW solution with the internal mode flag memory-mapped into HW (to allow mapping of variables make sure to enable Synthesis→Show Variables).

Evaluate and report the design quality metrics for each platform. Refine all platforms down to their communication models and simulate all models to validate their correctness. Document, compare and contrast all design decisions that were made and are necessary in each case. Submit the SpecC/PSM charts of the system `Design` (Right-Click→Chart, show the structural hierarchy by enabling View→Connectivity, single level of hierarchy only) for all platforms at the architecture/computation and the communication levels.

(c) Browse the hierarchy (View→Chart) and source (View→Source) of the generated models. For (b)i, take a look at the model of the DSP 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? For (b)ii, compare the generated models to the manually refined ones you developed in Problem 2.1(b) and 2.1(c). Other than the processor model, what differences can you make out? For (b)iii, look at the code in the leaf behaviors. What has changed (and why)? Can you find the instance of the mode flag variable in the architecture model (where did it end up)?