Homework 1
Design, MoC, Languages, SpecC

Assigned:	**Fri**	**09/20/19**	
Due:	**Tue**	**10/01/19,**	**12pm**

Instructions:

- Please submit your solutions via Canvas. Submissions should include a single PDF with the write-up and an archive for any supplementary files.
- You may discuss the problems and the concepts with your classmates. This fosters group learning and improves the class' progress as a whole. However, 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 1.1: System Design (15 points)

During design space exploration as part of the system design process, the target system architecture and its key architectural parameters are decided on. These design decisions have a major influence on the final design quality metrics such as (i) performance, (ii) power, (iii) cost, and (iv) time-to-market:

(a) Briefly discuss how the following target platform styles rate in relation to each other in terms of the metrics listed above:
- A pure software solution on a general-purpose processor
- A general-purpose processor assisted by a custom hardware accelerator/co-processor
- A general-purpose processor and a specialized processor (DSP)

(b) Try to sketch a potential simple strategy for exploring the design space for a given application under a given set of constraints/requirements.

Problem 1.2: Non-determinism (10 points)

(a) What is nondeterminism?

(b) How might nondeterminism arise? (give one example not discussed in class)

(c) What are the advantages and disadvantages of having nondeterminism in a language or model, i.e. in what circumstances might it be positive/desired or negative/undesired?

Problem 1.3: Models of Computation and Languages (10 points)

In class, we learned about the different Models of Computation (MoCs): KPN, SDF, FSM(D), HCFSM, PSM.

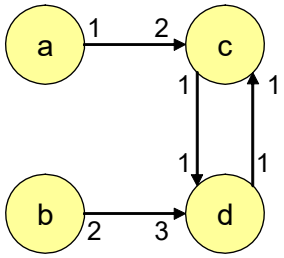
(a) What is the relationship between MoCs and languages?

(b) Compare SDF and PSM in terms of expressiveness (how much can be captured) and analyzability. Give a use case when you would use each MoC respectively.

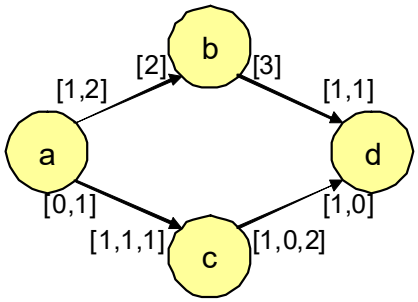
Problem 1.4: Synchronous Dataflow (SDF) (12 points)

For each of the following (C)SDF graphs, indicate whether or not the graph is consistent and has a valid periodic schedule. Show the balance equations, the repetition vector, and give a minimal static schedule if one exists.

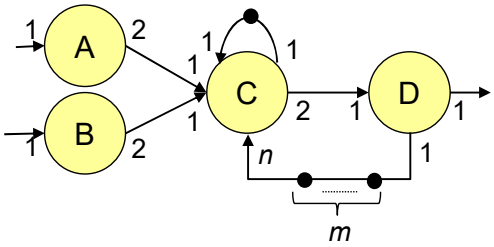
(a)



(b)

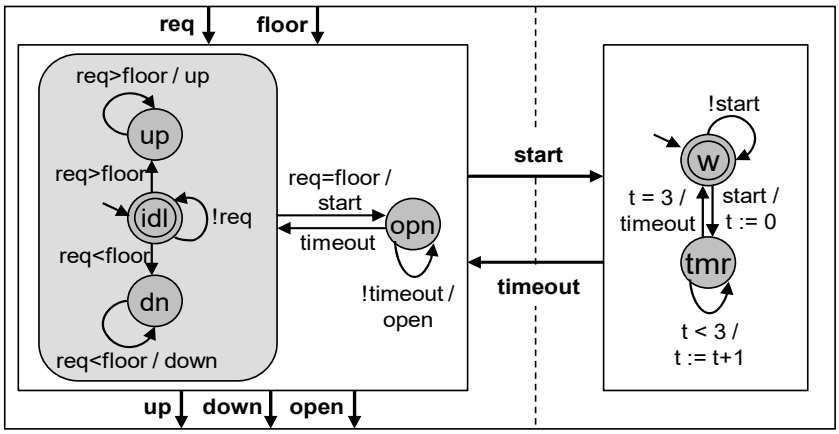


(c) For what integer values of m and n is this graph consistent and schedulable? What is the repetition vector and a minimal static schedule when n is fixed to its minimal value and $m=2n$?



Problem 1.5: State Machines (8 points)

Convert the following HCFSMD model of an elevator controller consisting of two concurrent, communicating state machines into an equivalent single, flat Mealy FSM *without* data (i.e. an FSM, *not* an FSMD). The HCFSMD has two external inputs (requested and current floor), three external outputs (up/down motor control and door open signals), and two internal signals (start timer and timer timeout). Unless specified otherwise, default value for all signals is absent/0.



Problem 1.6: SpecC Compiler and Simulator Preparation (0 points)

The goal of this problem is to make you familiar with the SpecC environment and basic tools, in particular the SpecC compiler and simulator (`scc`), using some simple examples included with the SpecC tool set.

The SpecC environment is installed on the COE Linux servers. Instructions for accessing and setting up the tools are posted on the SpecC project Wiki at:

https://github.com/neu-ece-esl/specc/wiki

In short, once logged in (e.g. remotely via `ssh or VLAB`), you need to run the provided setup script (depending on your $SHELL):

```
source /ECEnet/Apps1/sce/latest/bin/setup.{c}sh
```

Next, copy the examples found in `$SPECC/examples/simple/` into a working directory for this problem:

```
mkdir hw1_6
cd hw1_6
cp $SPECC/examples/simple/* .
ls
```

You can then use the provided `Makefile` to compile and simulate all examples:

```
make all
make test
```

Inspect the sources of all examples and the included `Makefile` to understand the use of `scc` for the compilation and simulation process, experiment with the `scc` command-line usage and start modifying the examples to experiment with different features of the SpecC language.

Information about all tools and `scc` is available via their man pages:

```
% man scc
% man sir_xxx
```

You can manually inspect, compile and execute an example on the command line as follows:

```
% less HelloWorld.sc
% scc HelloWorld -vv
% ./HelloWorld
```

Also practice working with the SpecC Internal Representation (SIR) and associated tools. You can compile an example into its (binary) SIR representation as follows:

```
% scc Adder -sc2sir -vv
```

You can then use the various sir_xxx tools to inspect and manipulate your design:

```
% man sir_list
% sir_list -t Adder.sir
% man sir_tree
% sir_tree -bt Adder.sir FA
```

Finally, a SIR file can be compiled into a simulation executable as follows:

```
% scc Adder -sir2out -vv
% ./Adder
```

This part of the assignment does not require submitting anything. It is just the preparation for the actual assignment following below.

Problem 1.7: SpecC Compiler and Simulator (35 points)

For this assignment, you are asked to develop, simulate and debug a simple Producer-Consumer example in SpecC. The program should contain two parallel behaviors *S* and *R* that communicate the string "Hello world" from sender to receiver. The communication should be character by character (one byte at a time), and both behaviors should print the characters as they are sent and received to the screen. After the entire message is transmitted, both behaviors should end and the simulation should cleanly terminate. You program output should look like this:

```
Receiver starting...
Sender starting...
Sending 'H'
Received 'H'
Sending 'e'
Received 'e'
Sending 'l'
Received 'l'
Sending 'l'
Received 'l'
Sending 'o'
Received 'o'
Sending ' '
Received ' '
Sending 'w'
Received 'w'
```

```
Sending 'o'
Received 'o'
Sending 'r'
Received 'r'
Sending 'l'
Received 'l'
Sending 'd'
Received 'd'
```

(a) Create a new directory hw1_7. Write a program pd_mix.sc that realizes all communication via shared variables and events (do not use channels at this point). Create a Makefile for compilation and execution. Compile and simulate the code to verify its correctness, and include a log of your program output in your report.

(b) Create a copy of pd_mix.sc and name it pd_custCh.sc Modify the example into a proper SpecC model that cleanly separates computation from communication. Create a new channel that encapsulates basic communication primitives and replace all shared variables and events between *S* and *R* to exclusively use one or more instances of your custom channel. Compile and simulate the modified code to verify its correctness.

(c) SpecC programs can be debugged using the standard GNU Linux debugger (`gdb`). A nice graphical frontend for `gdb` is available as:

```
% ddd <design>
```

This will bring up the `ddd` graphical debugger (use either VLAB or local X server to see the output), which allows you to debug you program directly at the SpecC source code level (if you prefer debugging at the level of the intermediate C++ code generated by the SpecC compiler, you can pass the `-sl` command line option to `scc` – this instructs the compiler to not create the necessary debug annotations that relate assembly and C++ to SpecC code). For example, in the `gdb` prompt of the debugger's command window, you can set breakpoints in SpecC behaviors and then start execution:

```
(gdb) b R::main
(gdb) run
```

Alternatively, you can use the debugger's graphical user interface (GUI) to open any SpecC source file (File→Open Source…), set breakpoints and then hit the Run toolbar button (Program→Run). From there, you can single step through the code, inspect variables, etc.

Modify your code from (b) to remove all 'Ack' related functionality from your custom channel, compile the code, and observe its behavior in the debugger. Explain the behavior of the modified code. Is the 'Ack' necessary? Why or why not?

In preparation of submission clean your working directory (do not submit binary files). Create a tar.gz archive of your assignment code. Ensure that your archive after extraction compiles without warnings (it should compile both pd_mix and pd_custCh).