Liberty Exercise Set 1

1. In this first exercise, let’s just get comfortable with the basic Liberty Structural Specification (LSS) language syntax. To do this, let’s hook up some logic gate modules to perform some logical function (this is not the granularity at which normal LSE simulations are done, but its a good warm up!).

First recall that to include module definitions from other files we use the import or include statement. In this example, we will make use of the source module, the sink module, and some other modules found in logic_gate.lss.

(a) Create a new LSS file which imports corelib and includes logic_gate.lss. Create two instances of the source module that will be used to generate true and false constants. The source module has a user point called create_data which defines what data the data source will create. A utility function source::create_constant is defined, by default, to help parameterize a source module when it should produce the same constant value each cycle. This function takes the constant value as a string argument and returns a user function that can be plugged into the create_data user point. Parameterize one of the instantiated data sources to produce the value TRUE and the other to produce the value FALSE.

(b) Now let’s build and run the configuration. You will need to run the following two commands to build and link the simulator:

```
$ ls-build exercise1.lss
$ ls-link exercise1.lss
```

Finally, to run the simulator, use the following command:

```
$ ./Xsim
```

The output you should see is:

```
LSE: Simulator has no more timesteps at time 1/0
CLP: Error -1 returned from LSE_sim_engine
Finish time: 1/0
```

The simulator tells you at what cycle the simulation finished, and also it tells you that the simulation ended because there were no more events. The LSE framework identifies situations when the simulation reaches steady-state. The framework can identify when each module would keep producing the same outputs cycle after cycle, and when this occurs, it terminates simulation.

(c) logic_gate.lss defines modules named and_gate, or_gate, xor_gate, and not_gate. Each module (except the not_gate) has an input port named a and an input named b. The module not_gate has a single input port named in. They each have an output port named out. The modules calculate the obvious logical functions. Instantiate some logic gates and connect the true and false data sources defined earlier to the inputs of various gates. The name of the output port on the source is out. Note that the multiport behavior of the logic gates is a slicing behavior. That is to say, inputs connected to port instance 0 produce an output on port instance 0. The source will
produce the same value on all instances of its multiport output. Connect the output of any logic gate not connected to another logic gate to the in port of an instance of the sink module. The sink module can sink as many values as necessary during a single cycle using multiports.

Compile your new configuration to make you haven’t made any syntax errors. Try using both explicit and implicit multiport numbering on the data sources and sinks.

(d) Add collectors to the in port of the sink(s) and to any other port whose values you wish to observe during the simulation. Recall that to print out the cycle number you can use the LSE_time_print_args(LSE_time_now) as the format string passed to printf. Compile, link, and run your simulator and verify that the output is what you’d expect it to be. Ensure that you understand why the simulation ends when it does.

(e) Add instances of the module delay to make the logic simulation run for more than one cycle. The delay instance module has one input port name in and one output port named out. Each instance of the input multiport corresponds to the same instance of the output multiport and represents one storage/delay element.

2. In this exercise we will design a state machine using the delay module and the logic gates we saw in the previous exercise. We will then replace the logic gates with a parameterizable module that encapsulates all the desired behavior into a user function.

For the rest of this question, we will be designing a 2-bit counter. We will refer to the low order bit of the counter as bit₀ and the high order bit as bit₁. The transitions transitions for the counter are as follows:

\[
\begin{align*}
\text{bit}_0' & = \text{bit}_0 \\
\text{bit}_1' & = \text{bit}_0 \oplus \text{bit}_1
\end{align*}
\]

where primes indicate the value the state will have next cycle.

(a) Implement the two bit counter using the delay module and the logic gates used in the last exercise. Add a collector to the output of the delay instance to ensure that it is getting updated correctly. Recall that the delay module has a user point called initial_state which allows you to set the delay module’s initial state. Also remember to use the convenience function delay::init with a string argument to help fill in the user function.

(b) i. Did your design use a control function?
   ii. If so, why did you need the control function?
   iii. Did your design use one or two delay instances?
   iv. Could your design be changed to only use one delay?

(c) Recall that the standard Liberty module library contains a module called combiner. This module has a parameter called inputs and another called outputs. Setting these parameters causes the module to have ports with the given names. For example, the following code:

```cpp
using corelib;

instance state_machine : combiner;
state_machine.inputs[0] = "in1";
state_machine.inputs[1] = "in2";
state_machine.outputs[0] = "out1";
state_machine.outputs[1] = "out2";
```
would cause the instance state_machine to have four ports: in1, in2, out1, and out2. The combiner also has a user point named combine which provides an argument named porti which informs you which multiport slice needs to be processed along with three parameters per port. Those variables are named port-name_status, port-name_id, and port-name_data. For output ports, the ID and status variables are pointers so that the variables are effectively passed by reference. The data parameters are all passed as pointers for efficiency (prevents copying of large data items) and to allow pass by reference for output ports.

The code you fill into the combine user point must set the status (to LSE_signal_something or LSE_signal_nothing), ID, and data of all the output ports. Continuing the example above, we could unconditionally pass in1 and in2 to out1 and out2 respectively with the following code:

```c
state_machine.combine = <<<
  *out1_status = in1_status;
  *out1_id = in1_id;
  *out1_data = *in1_data;

  *out2_status = in2_status;
  *out2_id = in2_id;
  *out2_data = *in2_data;
>>>;
```

For this exercise, replace the logic gates you used to implement the two bit counter with one or more combiner modules to implement the state transitions. Remember, the code inside the <<< and >>> is C.

Hint: The combiner and delay are both polymorphic modules. Since you are dealing with booleans, you need to give a hint to the type inference engine that your connections have type boolean. Try using ->[boolean] instead of -> for at least one connection on any given path.

3. In this exercise you will implement the round-robin arbiter you saw earlier. Recall that in a round-robin arbiter, the requester closest to the most preferred requester wins the arbitration. Closest is decided by a positive distance which wraps around. The index of the most preferred requester increments from cycle to cycle.

You will need to use the arbiter module. This module performs pairwise arbitration. That is to say, the user point comparison_func is presented with two inputs, and it must decide which of the two inputs would win arbitration if they were the only requesting signals. The function should return 0, if the first input should win over the second, it should return 1, if the second should win over the first, and it should return -1 if it cannot yet be determined. The arguments passed to the user point are: status1, id1, data1, port1.status2, id2, data2, and port2. The status fields are the port statuses of the inputs. The ID and data fields represent the data of the arbitrating requests. And finally, the port fields are the index numbers of the requesting entries (for position based arbitration).

Recall that you can add state to a module using runtime_var’s. Don’t forget to use the ($) notation when accessing a runtime variable inside a user point. Further, recall that there is a user point on each module named end_of_timestep which you can use to update state.

Hint: The following code snippet is very useful for doing modulus arithmetic which would otherwise involve negative numbers. To find the positive distance (with modulus wrap around) from a to b use the following code:

```c
dist = (b + WIDTH - a) % WIDTH;
```
(a) Implement a round robin arbiter. To test, feed the arbiter with two data signals generated from two data sources. The arbiter should feed a sink from which you can monitor what data won arbitration during each cycle of execution.

(b) How many cycles did the simulation run? Should it have gone longer? If so, why did it not continue?

The last question was a bit of a trick question. So, it warrants a bit of explanation. The simulator reported that it ran out of events. Which means the simulation framework believes that the system has reached steady-state. But how is this possible? In the next cycle the arbiter would have picked the other input to win arbitration. Since the state that was added changed at the end of this cycle, someone needs to inform the simulation framework that even with the same inputs, this instance can produce different outputs. This way, the framework knows that steady-state has not been reached. You inform the scheduler by explicitly asking it to invoke this instance at a future time. Insert the following line of code in your end_of_timestep user point:

```c
LSE_sim_keep_alive(LSE_time_one_cycle);
```

This call notifies the scheduler to keep the simulation alive for one cycle. Rebuild your configuration and rerun to ensure that the output does toggle between the two inputs.