In this exercise you will build a demultiplexer module by hierarchically composing the **tee** module and the **gate** module. The **tee** fans out a single signal into multiple signals. The **gate** module has data inputs and control inputs and a user function which determines, based on the control inputs, whether the data inputs should pass to the outputs.

The **tee** has a single input port named **in** and a single output port named **out**. The **tee** module supports multiple connections on both its input port and its output port. The ratio of output port connections to input port connections must be an integer. We'll use *ratio* to refer to this ratio. The data sent to an instance of the **tee** on port instance *i* will appear on output port instances **out[ratio * i]**, **out[ratio * i + 1]**, ..., **out[ratio * i + ratio - 1]**. Recall, that each connection in LSE actually carries three signals: data, enable, and acknowledge. The **tee** module will fan out both the data and enable signals. The reverse flowing acknowledge signal on a given **in** port instance is the logical AND of the acknowledge signals received on the relevant **out** port instances. That is to say, a sender will be acknowledged if all of the receivers acknowledge and will be negatively acknowledged if any of the receivers sends a negative acknowledgement.

The **gate** module has two input ports, **in** and **control** and a single output port, **out**. The **gate** defines a user function called **gate_control** that will be invoked for each input connection (remember, all the ports are **multiports**). The user function is provided the port index number, port status, data, and dynID for the connection as well as the data and port status for all the connections to the **control** port. The function must return 1 if the data is to be passed to the output or 0 otherwise. The function can return -1 if it cannot yet make a decision. For reference, the signature of the user point is:

```c
parameter gate_control :
    userpoint(<<<int porti, LSE_signal_t status, LSE_dynid_t id,
        LSE_port_type(in) *data, LSE_signal_t *cstatus,
        LSE_dynid_t *cid, LSE_port_type(control) **cdata,
        int cwidth>>> => <<<int>>>);
```

The control flow semantics of the **gate** are fairly simple. With the default parameterization, data and enable pass from **in** to **out** if the user function for the particular port instance returns 1. Otherwise the output is set to **LSE_signal_nothing | LSE_signal_disabled**. Similarly, the acknowledge signal flows from **out** to **in** if the user function returns 1. Otherwise, if the value of the acknowledge signal is controlled by the parameter **gate_style**. If this parameter is set to **gate::blocker** then the input is negatively acknowledged (**LSE_signal_nack**). If the parameter is set to **gate::filter**, then the input is acknowledged (**LSE_signal_ack**). The **control** port is always acknowledged on all port instances.

With this information, you should now be ready to start the exercise.

1. Create a module called **demux**. Give the module two input ports: **in** and **control**. Give the module a single output port named **out**. The types of input and output should be polymorphic
but should be equal. The type of control should be int. The data on the control port will determine which instance of the out port the data on in will go to. For this part of the exercise, we will disallow multiple input connections to the in and control port. Add checks to your module to ensure this is the case.

Connect the in port to the input of a tee module. Instantiate a gate module and make one connection between the tee's output port and the gate's input port for each connection to the demux's out port. Connect the control input to the control input of the gate. Connect the out port of the gate to the out port of the module multiple times ensuring the correct number of connections.

Fill in the gate_control user function on the gate such that it passes the data on port index i if the value on the port control[0] equals i. Make sure that your user function handles cases where cstatus has unknown data and enable signals. Your user function will be reinvoked after the data and enable is known. Your user function can return -1 to indicate that it cannot yet make a decision.

Finally decide whether the instance of the gate module should be configured as a gate::filter or gate::blocker to create the correct demultiplexing control behavior.

Build a configuration demux_test.lss to test your demux module. Run the configuration and make sure your demux works.

2. Now add multiport behavior to your demux module. You need to handle multiple connections to the control and in ports. Make sure you verify that the correct number of connections are made to all the ports (your demux component should report an error if this is not the case).

The behavior should be that of an array of demultiplexors. Control signal i should route input signal i to $\lfloor \frac{m}{n} \rfloor \times i + \text{valueof(control[i])}$, where n is the total number of connections on the input port and m the total number of output port connections. Make a test file to test this configuration as well; ensure that the test file actually makes multiple connections.

Please also answer the following questions about your multi-ported demux module:

- How many control port connections are needed if there are i in port connections?
- What mathematical relation must hold between the number of input port connections, n, and the output port connects m? For example, in the delayn example seen in class $m = n$ is the correct relation.