ECE 4750 Computer Architecture, Fall 2015
Lab 4: Ring Network

School of Electrical and Computer Engineering
Cornell University

revision: 2015-11-08-12-36

In this lab you will implement an eight-node bidirectional ring network with elastic-buffer flow-control and round-robin arbitration and explore two different routing algorithms: a greedy routing scheme for the baseline design and an adaptive routing scheme for the alternative design. You will eventually compose your network with the processor designs you developed in Lab 2. All packets in the network will be a single flit and each flit will be composed of a single phit. You will implement a bubble flow-control scheme for deadlock avoidance in the ring network in both directions. To evaluate the performance of the network you will be driving the simulator with different traffic patterns and analyzing the resulting latency-bandwidth plots. **As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with:

- basic network design;
- single-cycle controllers with adaptive control logic;
- microarchitectural techniques for implementing a ring network;
- abstraction levels including functional- and register-transfer-level modeling;
- design principles including modularity, hierarchy, encapsulation, and regularity;
- design patterns including message interfaces and control/datapath split;

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. To get started, you should access the ECE computing resources and you have should have used the `ece4750-lab-admin` script to create or join a GitHub group. If you have not do so already, source the setup script and clone your lab group's remote repository from GitHub:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:cornell-ece4750/lab-groupXX
```

where `XX` is your group number. **You should never fork your lab group's remote repository! If you need to work in isolation then use a branch within your lab group's remote repository.** If you have already cloned your lab group's remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% py.test ../lab4_net --verbose
```

All of the tests for the provided functional-level model should pass, while the tests for the baseline and alternative network designs should fail. For this lab you will be working in the `lab4_net` subproject which includes the following files:

- `RingNetFL.py` – Functional-level 8-port network implemented as a global crossbar
- `RingNetFL_test.py` – Unit tests for the functional-level network
- `RouterBasePRTL.py` – PyMTL RTL router with greedy routing
- `RouterBaseVRTL.v` – Verilog RTL router with greedy routing
- `RouterBaseVRTL.py` – PyMTL wrapper around Verilog RTL
- `RouterBaseRTL.py` – Wrapper to choose which RTL language
- `RouterBaseRTL_test.py` – Unit tests for router with greedy routing
- `RouterAltPRTL.py` – PyMTL RTL router with adaptive routing
- `RouterAltVRTL.v` – Verilog RTL router with adaptive routing
- `RouterAltVRTL.py` – PyMTL wrapper around Verilog RTL
- `RouterAltRTL.py` – Wrapper to choose which RTL language
- `RouterAltRTL_test.py` – Unit tests for router with adaptive routing
- `RingNetBasePRTL.py` – PyMTL RTL eight-port ring network with greedy routing
- `RingNetBaseVRTL.v` – Verilog RTL eight-port ring network with greedy routing
- `RingNetBaseVRTL.py` – PyMTL wrapper around Verilog RTL
- `RingNetBaseRTL.py` – Wrapper to choose which RTL language
- `RingNetBaseRTL_test.py` – Unit tests for ring network with greedy routing
- `RingNetAltPRTL.py` – PyMTL RTL eight-port ring network with adaptive routing
- `RingNetAltVRTL.v` – Verilog RTL eight-port ring network with adaptive routing
- `RingNetAltVRTL.py` – PyMTL wrapper around Verilog RTL
- `RingNetAltRTL.py` – Wrapper to choose which RTL language
- `RingNetAltRTL_test.py` – Unit tests for ring network with adaptive routing
- `__init__.py` – Package setup

1. Introduction

Monolithic integration using a standard CMOS process provides a tremendous cost incentive for including more and more components on a single die. On-chip interconnection networks play an important role in connecting these components and in providing communication between the various sub-systems. The performance of the network depends on many design choices, including the topology of the network, the routing algorithm, and the flow-control algorithm. In this lab, you will design an eight-node bidirectional ring network with elastic-buffer flow-control and round-robin arbitration. You will explore two different routing algorithms: (1) a greedy routing scheme that *deterministically* routes packets to destinations, and (2) an adaptive routing scheme that *non-deterministically* routes packets to avoid network congestion. Both networks will use only single-flit/phit packets.

The message format for network messages are shown in Figure 1. The number of routers and the payload bitwidth together determine the network packet size. We have provided you with `NetMsg.py` in the `pclib/ifcs` library for PyMTL or `vc/net-msgs.v` for Verilog, which provides useful macros and pack/unpack modules for network packets. Figure 1 shows how each network packet contains the packet's destination/source, an opaque field for meta information, and the payload.

We have provided you with a functional-level model of a eight-port network, which essentially just passes network messages through a eight-port crossbar from the source to the destination of each network message. While this might not seem useful, the functional-level model will enable us to

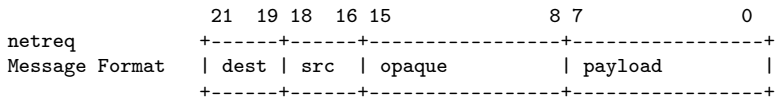

Figure 1: Network Message Format – Network messages are sent from different test sources to the network and from the network to different test sinks. An example message is shown for `payload_nbits` = 8, `srcdest_nbits` = 3, and `opaque_nbits` = 8.

develop many of our test cases before attempting to use these tests with the baseline and alternative designs. Figure 2 shows a block diagram of the FL model.

2. Baseline Design

The baseline design is an eight-node bidirectional ring network with elastic-buffer flow-control, greedy routing, round-robin arbitration, and bubble flow-control for deadlock avoidance. If we need to send a packet half-way around the ring such that there are two minimal paths, then always choose the clockwise direction. Figure 3 shows a block diagram of the network. Note that the ring network has eight routers and each router can use channels to send messages in both directions – east and west – as well as to its own terminal. All packets in the network are a single flit and each flit is composed of a single phit.

Your design should implement *elastic-buffer flow-control*: a simple flow-control algorithm that exploits the implicit elastic buffer present in the channels of the network to reduce the amount of storage required to design a network-on-chip, thereby simplifying router microarchitecture and reducing router area and power consumption. Elastic-buffer flow-control works by re-using the channels between routers as buffers to hold messages. In this lab, we pipeline the channels by using two-entry queues, making elastic-buffer flow-control easy to understand. For example, consider a single router and the channels connecting the router in the east and west directions. When a packet is injected into the network from the input terminal port, the router will decide which way to send the packet. Suppose the packet is routed east; the router will then set the network message and raise the output valid signal on the eastward channel. If the downstream router is not ready to receive the packet from the channel, the packet will be held in the channel queue or input queue of the downstream router. If the downstream router is ready to receive the packet, the val-rdy transaction will take place, and the packet will be accepted by the next node. Notice that queueing creates the elasticity in the network.

Figure 4 and Figure 5 shows the method-based interface for the simple two-element queues that we have provided for you to model pipelined channels. The `NormalQueues` (PyMTL) or `vc_Queue` (Verilog) module has a method-based interface that cleanly expresses enqueue- and dequeue- message operations. Both operations use a val-rdy handshake to complete the transactions. In this lab, we will use normal queues (i.e., in Verilog, `p_type` of `` `VC_QUEUE_NORMAL ``; in PyMTL, `NormalQueue module`) for both router input queues and channel queues. The remaining parameters, `dtype` (PyMTL) or `p_msg_nbits` (Verilog), and `dtype` (PyMTL) or `p_num_msgs` (Verilog), configure the bitwidth of the message and the number of messages the queue can hold, respectively. Internally, the queue implementation tracks elements using head and tail pointers. To `enqueue` a message, set `enq_msg` to the message to be enqueued and complete the val-rdy handshake. To dequeue a message, read `deq_msg` and complete the val-rdy handshake. The queue also provides an output port with the number of free entries in the queue. Use the corresponding unit test to further understand the queue operations. You can find the definition of PyMTL queues in `pclib` here:

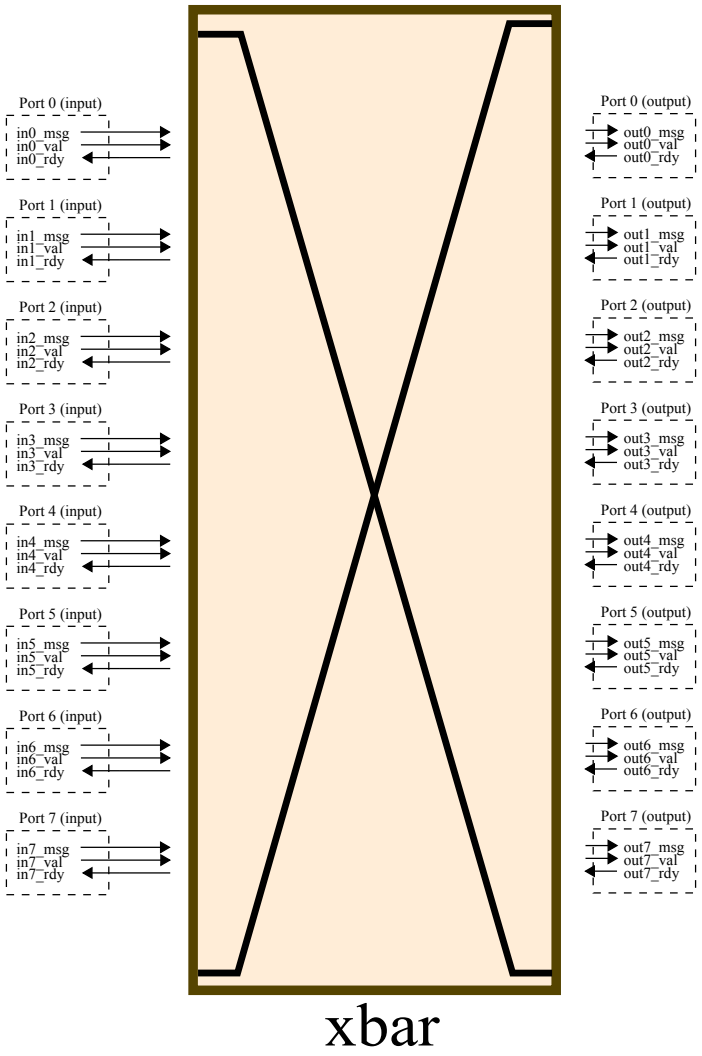

Figure 2: FL Model of the eight-port network – Functional-level model of a eight-port network, which passes network messages through a eight-port crossbar from the source to the destination of each network message.

- `https://github.com/cornell-brg/pymtl/blob/ece4750/pclib/rtl/queues.py`

The equivalent Verilog component is located in `vc/queues.v`.

For this lab, you are required to use the following mappings for the three bidirectional ports. These mappings are arbitary but help keep all of the designs in the class consistent:

- Port 0: connects to the west-side port
- Port 1: connects to the input/output terminal
- Port 2: connects to the east-side port

The datapath for the single-cycle router is shown in Figure 6. Three input queues, corresponding to the three ports, should each have four entries. The messages are dequeued and passed into a 3×3 global crossbar. Make sure the global crossbar has the bitwidth of a network message. To imple-

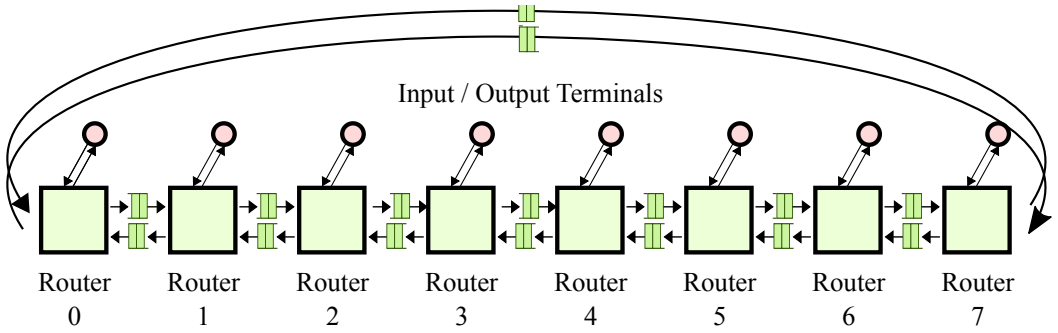

Figure 3: Eight-Node Ring Network Block Diagram – An eight-node ring network is composed of eight routers (boxes) and eight terminals (circles), connected by bidirectional channels (arrows). Note that channels are modeled as two-entry queues.

```
1
2  #-------------------------------------------------------------------------
3  # NormalQueue
4  #-------------------------------------------------------------------------
5
6  class NormalQueue( Model ):
7
8    def __init__( s, num_entries, dtype ):
9
10     s.enq             = InValRdyBundle ( dtype )
11     s.deq             = OutValRdyBundle( dtype )
12     s.num_free_entries = OutPort( get_nbits(num_entries) )
```

Figure 4: PyMTL Queue Interface – PyMTL queues in `pymtl/pclib/rtl/queues.py` use a method-based interface. A val-rdy protocol is used to enqueue messages into and dequeue messages from the queue.

ment the global crossbar, you can use the modules available in `pclib/rtl/Crossbar.py` (PyMTL) or `vc/crossbars.v` (Verilog). You can take a look at the PyMTL crossbars here:

- `https://github.com/cornell-brg/pymtl/blob/ece4750/pclib/rtl/Crossbar.py`

The control for the single-cycle router is fairly complex. In Figure 6, blue signals indicate control signals. Figure 7 illustrates our overall sturcture of the control module for the router. Note that we are not implementing the control unit structurally and you are not required to implement the control module structurally either. You can implement the control unit as a flat module. In Figure 7, the `dest` fields are bitsliced from the messages dequeued from the input queues in Figure 6. The `dest` fields should be inputs to your deterministic greedy routing algorithm for route computation. You can implement the logic for the greedy routing algorithm in a separate module or directly inside your control unit, whichever makes your design cleaner. The greedy routing computation calculates the number of hops it takes to reach the destination in the east and west directions. Then it picks the direction with the least number of hops. When the number of hops are equal in either direction, arbitrarily pick one way, e.g. always east. Route computation generates a one-hot request bit-vector for the output ports, where each bit in the bit-vector represents an output port. In Figure 7, these bit-vectors are labeled: `in_reqs0`, `in_reqs1`, and `in_reqs2`. All of the requests for the same output port are gathered into bit-vectors labeled: `out_reqs0`, `out_reqs1`, and `out_reqs2`. A round-robin arbiter decides which request is granted the output port, and grant signals are propagated back to the router

```
//------------------------------------------------------------------------
// Queue
//------------------------------------------------------------------------

module vc_Queue
#(
  parameter p_type      = `VC_QUEUE_NORMAL,
  parameter p_msg_nbits = 1,
  parameter p_num_msgs  = 2,

  // parameters not meant to be set outside this module
  parameter c_addr_nbits = $clog2(p_num_msgs)
)(
  input                    clk,
  input                    reset,

  input                    enq_val,
  output                   enq_rdy,
  input  [p_msg_nbits-1:0] enq_msg,

  output                   deq_val,
  input                    deq_rdy,
  output [p_msg_nbits-1:0] deq_msg,

  output [c_addr_nbits:0]  num_free_entries
);
```

Figure 5: Verilog Queue Interface – Verilog queues in `vc/queues.v` use a method-based interface. A val-rdy protocol is used to enqueue messages into and dequeue messages from the queue.

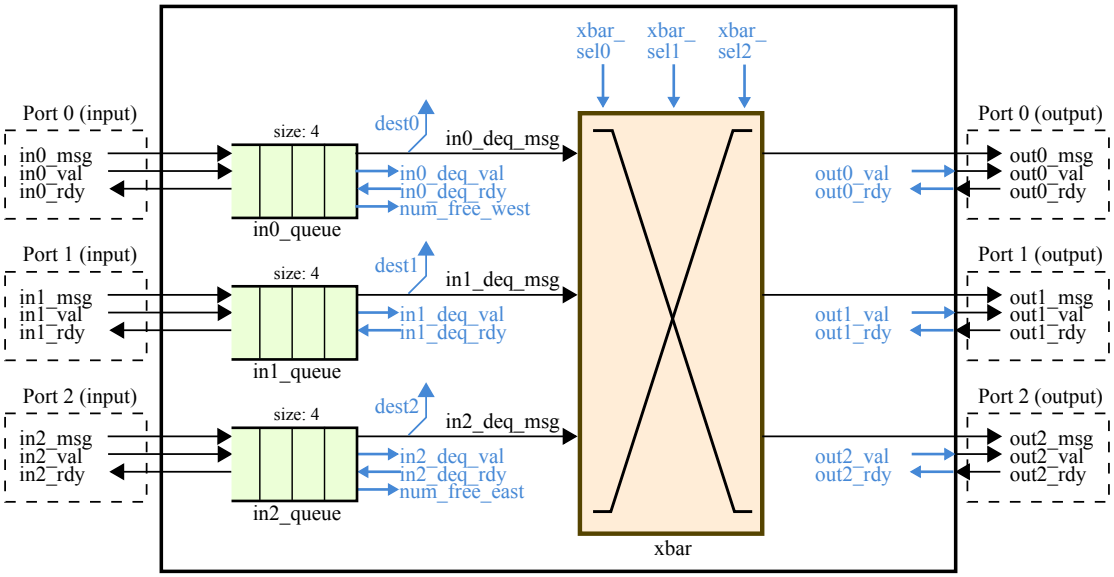

Figure 6: Baseline Design Router Datapath

input control to be used in val-rdy logic. Arbiters are provided for you in `pclib/rtl/arbiters.py` (PyMTL) or `vc/arbiters.v` (Verilog). You can take a look at the PyMTL arbiters here:

- `https://github.com/cornell-brg/pymtl/blob/ece4750/pclib/rtl/arbiters.py`

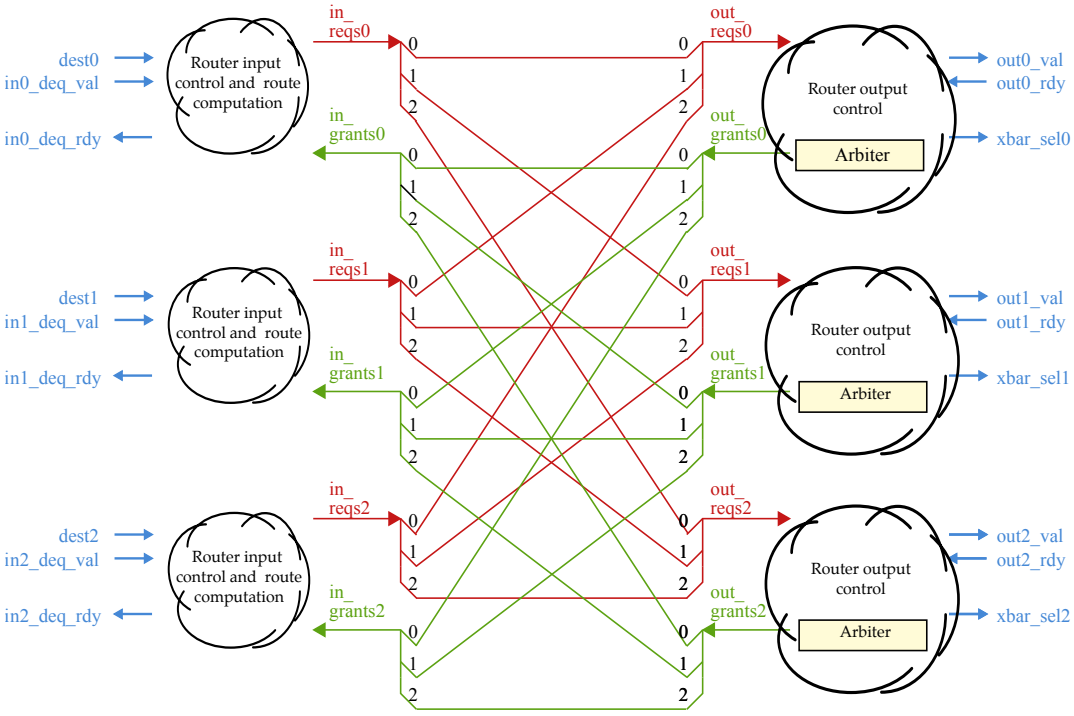

Figure 7: Incremental Baseline Design – Router control unit with no deadlock avoidance

Based on the arbiter's decision, the control module sets the select bits for the global crossbar in the datapath to determine which messages are routed to each output port; for example, the `xbar_sel0` signal determines whether `in0_deq_msg`, `in1_deq_msg`, or `in2_deq_msg` is passed through to `out0_msg`. All val-rdy signals are propagated to the correct input and output ports.

We have provided `RingNetBasePRTL.py` (PyMTL) and `RingNetBaseVRTL.v` (Verilog) to compose many instances of the router you design into a ring network. Composing the routers otherwise is a tedious task. Make sure you understand how this top level network composes the routers and channel buffers and feel free to change the file if your baseline implementation is different than what we have suggested.

We strongly encourage you to take an incremental design approach using the following steps:

- Implement a router with elastic-buffer, only routes network messages in clockwise, and without deadlock avoidance.
- Ring with elastic-buffer flow-control, only routes in clockwise, and no deadlock avoidance.
- Implement a router with elastic-buffer, only routes network messages in clockwise, and with deadlock avoidance.
- Ring with elastic-buffer flow-control, only routes in clockwise, and with deadlock avoidance.
- Add counter-clockwise routing to the router.
- Ring with elastic-buffer flow-control, can route in both directions, and bubble flow-control for deadlock avoidance.

The next subsection explains bubble flow-control in more detail.

2.1. Deadlock Avoidance: Bubble Flow-Control

The ring network described does not guarantee deadlock avoidance. You will implement a ring with bubble flow-control for deadlock avoidance as the final step in your baseline design. Before beginning work on bubble flow-control, design the simplest possible test case that puts your network into deadlock and observe the deadlock in your line tracing. Then, implement bubble flow-control and test the network for deadlock avoidance. You will not need to modify the top-level ring network as long as you maintain the interface of the router design.

Deadlock occurs when a group of packets are unable to make progress toward their destinations because they are waiting on one another for resources. Once a network is deadlocked, it will remain in this state indefinitely. Unless there is a deadlock-avoidance mechanism in place, ring networks are easily deadlocked due to the cyclic dependency introduced by the wrap-around channel.

Bubble flow-control is a simple deadlock-avoidance mechanism for ring networks. Bubbles are free packet buffers. In a unidirectional ring, the guaranteed existence of a bubble at each node is sufficient to avoid deadlock. We can provide this guarantee by restricting the conditions under which a packet may be forwarded from the buffer of an input terminal. Consider a packet we wish to route in the east direction. The packet may be forwarded only if a bubble exists in the current router's west input queue – i.e., we are checking that there is a bubble in the ring network in the same direction that we wish to route the new packet. Similarly, a packet we intend to route in the west direction may be forwarded only if a bubble exists in the current router's east input queue. A bubble exists in an input queue if the number of free entries is greater than one. Please convince yourself that imposing this restriction avoids deadlock. Note that bubble flow-control requires a minimum input buffer size of two packets. Most of the control logic remain the same. The input terminal port control logic is modified with extra inputs for the number of free entries in the input queues to the west and east in order to implement the bubble guarantee.

3. Alternative Design

For the alternative design, you will implement an adaptive routing algorithm that senses the congestion in the network and routes packets to balance load on the channels. The goal is to improve the performance of the network. The only modification from the baseline design is in the route computation. You will need to create a new module similar to `RouterBasePRTL.py` (PyMTL) or `RouterBaseVRTL.v` (Verilog) that uses your adaptive routing algorithm instead of the greedy routing algorithm. Your router with adaptive routing algorithm should be named as `RouterAltPRTL.py` (PyMTL) or `RouterAltVRTL.v` (Verilog).

You can sense congestion in the channels by observing output ready signals on the channels. You could also sense congestion by looking at the number of free entries in the channel queues in either direction. Note that using information from neighboring channel queues provides only non-global congestion information. You could experiment with more global congestion information by looking at channel queues further away. However, you may not be able to retrieve this congestion information combinationally in a single cycle. You should pass this congestion information through registers to simulate the latency in wires – an additional cycle of latency for each extra hop you make to retrieve congestion information.

Once you have retrieved congestion information, you can implement a weighted routing scheme that factors in the congestion to decide a better route for an outgoing packet.

4. Testing Strategy

As in the previous lab, the tests for this lab may be challenging since you will need to carefully craft directed tests that exercise all paths in your datapath and control. In contrast with the previous lab, testing your network relies heavily on unit testing sub-components of the design as well as on testing the entire design. We provide you with one very basic test in each of the following sub-units:

- The entire router with greedy routing in `RouterBaseRTL_test.py`
- The ring network in `RingNetBaseRTL_test.py`

Note that if you choose to design the router in a different structure, you will need to provide unit tests for your own modules. You will need to add many more test cases until you are sure each subcomponent works. Note that rigorous unit-testing of the router sub-components and of the router itself can help to avoid hard-to-find bugs at the top-level design.

You will write most of your tests for the top-level ring network. You can begin writing tests right away using the functional-level model provided in `RingNetFL_test.py`, which contains a global crossbar. You do not need to have a working router in order to write these tests. Once these tests are working on the functional-level model, you can move on to testing the top-level baseline and alternative ring network designs.

The following commands illustrate how to run all of tests for the entire project, how to run just the tests for this lab, and how to run just the basic test we provide on the various designs.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% py.test ..
% py.test ../lab4_net
% py.test ../lab4_net/RingNetFL_test.py
% py.test ../lab4_net/RingNetBaseRTL_test.py
% py.test ../lab4_net/RouterBaseRTL_test.py
```

You will add your directed and random tests for the ring network to `RingNetFL_test.py`. Since this harness is shared across the functional-level model, the baseline design, and the alternative design you can write your tests once and reuse them to test all three models. You will be adding more test cases. Do not just make the given test case larger.

Figure 8 illustrates how we will be writing tests for a single router using various helper tasks which are defined in the same test file. The `mk_net_msg` function will create a network request message and write the messages into the corresponding test sources and sinks. For each test case we define a Python function that returns a list of request-response message pairs. Note that there are three test sources and three test sinks in the testing infrastructure. This specific example shows one network request sent from a router (with ID #2) to itself. Port 1 corresponds to the router's terminal. The network message is injected from the terminal into the router, processed in the router, and then passed back to the terminal. Once you create a new test case, you need to add it to the test case table, as shown on lines 26–29 in Figure 8. A test case table has four columns. The first column is the name of tests, the second one is the function that generates source/sink messages, the third one is the source delay, and the fourth one is the sink delay.

Figure 9 illustrates how we will be writing tests for the ring network. Note that there are eight test sources and sinks and the ring network itself in the test harness. We specify a different `mk_net_msg` function that takes the source and destination router IDs, an opaque field for meta-information, and the payload of the network message. In this specific example, a single source sends one packet to each node. Similarly, you also need to add your new test cases to the test case table.

```
#-------------------------------------------------------------------------
# Test case: example messages
#-------------------------------------------------------------------------

def example_msgs():

  src_msgs  = [ [] for x in xrange( 3 ) ]
  sink_msgs = [ [] for x in xrange( 3 ) ]

  # mk_net_msg
  def mk_net_msg( src_, sink, src, dest, opaque, payload ):
    msg = mk_msg( src, dest, opaque, payload )
    src_msgs [src_].append( msg )
    sink_msgs[sink].append( msg )

  #           in   out
  #           port port src  dest opaque payload
  mk_net_msg( 0x1, 0x1, 0x2, 0x2, 0x00,  0xce )

  return [ src_msgs, sink_msgs ]

#-------------------------------------------------------------------------
# Test Case Table
#-------------------------------------------------------------------------

test_case_table = mk_test_case_table([
  (                    "msgs               src_delay sink_delay"),
  [ "example",          example_msgs(),    0,        0          ],
])
```

Figure 8: Directed Test Example for Router – Simple directed test for a router that uses the `mk_net_msg task` to send one network request from a router (with ID #2) to itself. Port 1 corresponds to the router's terminal.

Some suggestions for what you might want to test in the ring network are listed below. Each of these would probably be a separate test case.

- One packet from some node A to itself
- One packet from node A to node B
- A packet from node A to node B and a packet from node B to node A
- A single source sending one packet to each node
- Each node sending one packet to a single destination
- One packet from each node to its neighbor
- Testing all or some of the above using random source and sink delays

You can then create directed test cases that capture longer strings of traffic patterns using `for` loops with many iterations. Here are some suggestions:

- `nearest neighbor`: prolonged traffic to nearest neighbor
- `hotspot`: prolonged traffic to a single node
- `tornado`: prolonged traffic half-way around the ring

You are required to **create at least one dedicated test case to force deadlock** on your network without bubble flow-control. This test should be as simple and as short as possible while still creating deadlock. You are also required to **create at least one dedicated test case to trigger adaptive non-minimal routing** on your alternative design.

```
#-------------------------------------------------------------------------
# Test case: single source
#-------------------------------------------------------------------------

def single_src_msgs():

  src_msgs  = [ [] for x in xrange(8) ]
  sink_msgs = [ [] for x in xrange(8) ]

  def mk_net_msg( src, dest, opaque, payload ):
    msg = mk_msg( src, dest, opaque, payload )
    src_msgs [src ].append( msg )
    sink_msgs[dest].append( msg )

  #           src dest opaque payload
  mk_net_msg( 0,  0,   0x00,  0xce )
  mk_net_msg( 0,  1,   0x01,  0xff )
  mk_net_msg( 0,  2,   0x02,  0x80 )
  mk_net_msg( 0,  3,   0x03,  0xc0 )
  mk_net_msg( 0,  4,   0x04,  0x55 )
  mk_net_msg( 0,  5,   0x05,  0x96 )
  mk_net_msg( 0,  6,   0x06,  0x32 )
  mk_net_msg( 0,  7,   0x07,  0x2e )

#-------------------------------------------------------------------------
# Test Case Table
#-------------------------------------------------------------------------

test_case_table = mk_test_case_table([
  (                     "msgs                 src_delay sink_delay"),
  [ "single_src",        single_src_msgs(),   0,        0          ],
])
```

Figure 9: Directed Test Example for Ring Network – Simple directed test for the ring network that uses a different `mk_net_msg` task than in Figure 8. In this example, a single source sends one packet to each node.

```
Cycle     Source Ports            Router Ports                Sink Ports
         0      1      2         0      1      2            0      1      2
    0:       |      |       >>> (.     |.     |.     ) >>> .     |.     |.
    1:       |00:2>2|       >>> (.     |.     |.     ) >>> .     |      |.
    2:       |      |       >>> (.     |00:2>2|.     ) >>> .     |00:2>2|.
```

Figure 10: Line Trace for Basic Router Test – The line trace shows one network request sent from a router (with ID #2) to itself. Port 1 corresponds to the router's terminal. The network message is injected from the terminal into the router, processed in the router, and then passed back to the terminal.

Once you have finished writing your directed tests you should move on to writing random tests for your ring network. You can use the tests in the `RingNetFL_test.py`. We only gave you a very basic traffic pattern. You may wish to generate other traffic patterns (e.g., tornado or uniform random), and you can also draw inspiration from the traffic patterns used in the evaluation. Each of these would probably be a separate test pattern, or potentially multiple test patterns with different random parameters and random delays.

You will almost certainly want to use line tracing to visualize the execution of transactions on your baseline and alternative designs. We have provided some line tracing code for you in the test harnesses which you can use to trace the network messages at the source/sink and router interfaces. Figure 10 illustrates a line trace for the basic router test in Figure 8 executing on the baseline router

design with extra annotations to indicate what the columns mean. Each network message has the format: *(lowest bits of the opaque field) : (source) > (dest)*. The opaque field has no functionality but is useful to differentiate messages from each other. The first three columns represent the three sources, and they show when a message is injected into the router. The middle three columns show what is happening in the router. The last three columns represent the three sinks, and they show both when and on which port a network message is passed out of the router. Similar line tracing is available for the ring network designs.

5. Evaluation

Once you have verified the functionality of the baseline and alternative designs, you should then use the provided simulator to evaluate your designs. You can run the simulator to see the performance of each network implementation as follows:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% ../lab4_net/net-sim --impl base --pattern urandom --injection-rate 10
```

Use `--impl` option to choose your baseline or alternative design. The simulator supports a variety of patterns. You can use `--pattern` option to choose one. For each one, we succinctly describe the pattern using Verilog syntax.

- `urandom` -- `dest = random % 8`
- `partition2` -- `dest = (random & 3'b011) | (src & 3'b100)`
- `partition4` -- `dest = (random & 3'b001) | (src & 3'b110)`
- `tornado` -- `dest = (src + 3) % 8`
- `neighbor` -- `dest = (src + 1) % 8`
- `complement` -- `dest = ~src`

The simulator injects packets and measures the latency. You can specify the injection rate using the `--injection-rate` option. The `--stats` option displays the zero-load latency at the injection rate you specified. You should study the line traces (with the `--trace` option) and possibly the waveforms (with the `--dump-vcd` option) to understand the reason why each design performs as it does on the various patterns.

The simulator also supports a sweep mode which sweeps the injection rate. More specifically, the simulator sweeps the injection rate and reports the average latency at each injection rate. As the network approaches saturation, the simulator increases the injection rate to gather more data points. We assume the network saturates when its average latency is larger than 100 cycles. To use the sweep mode, set the `--sweep` option, for example:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% ../lab4_net/net-sim --impl alt --pattern urandom --sweep
```

The simulator will display a list of injection rates and average latencies. You can use it to plot the latency-bandwidth curve. Here is an example of sweep mode output:

```
 % ../lab4_net/net-sim --impl base --pattern urandom --sweep

Pattern: urandom

Injection rate (%)   | Avg. Latency
5                    | 5
15                   | 5
25                   | 5
35                   | 5
45                   | 6
55                   | 51
56                   | 54
57                   | 83
58                   | 181

Zero-load Latency = 5
```

If you use the sweep mode, simulations take significantly longer than in previous labs because of the injection rate sweeps and the required warmup period for each injection rate.

6. PARCv3 Extensions

Each lab assignment includes additional extensions that would be required to transform the alternative design into a subsystem suitable for use in a full PARCv3 multicore. Students are free to work on these extensions, although they must implement them in a separate `lab3_mem_ext` subdirectory. Do not implement any design extensions in the main lab subdirectory! It is important that any work on design extensions not cause the tests for your baseline and alternative designs to fail. Design extensions will not be used to award extra credit, nor will they be factored into the grading of labs 1–4 in any way. However, design extensions can be factored into the grading for the baseline and alternative design section of the final lab assignment.

For this network to be used in a reasonable PARCv3 multi-core processor, we might need to add some of the following features:

- **Four-node Ring Network** – Try to implement a four-port network that can be used in the final lab to connect four processors, four-banked L1 cache, and memory. You can also make your network parametrizable so that you can configure it to have any number of nodes (routers).
- **Bus Network** – Try to implement a four-port bus. The ring network in this lab has multiple cycles of latency, which is a little bit too high to connect L1 cache in a quad-core processor in lab 5. You can try creating a simple single-cycle bus which would be more realistic than the ring network in terms of the lab 5 composition.

Acknowledgments

This lab was created by Moyang Wang, Christopher Torng, Berkin Ilbeyi, Shreesha Srinath, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.