ECE 4750 Computer Architecture, Fall 2016
Lab 4: Ring Network

School of Electrical and Computer Engineering
Cornell University

revision: 2016-11-04-09-16

In this lab you will implement two topologies for a four-terminal interconnection network: the baseline design is a multiple-writer, multiple-reader bus network and the alternative design is a ring network. All packets in the network will be a single flit and each flit will be composed of a single phit. You will need to carefully consider what routing algorithm to use in the ring interconnection network to trade-off zero-load latency and ideal terminal throughput while still avoiding deadlock. You are required to implement the baseline design and alternative designs, verify the designs using an effective testing strategy, and perform an evaluation comparing the two implementations. **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;
- combinational controllers;
- microarchitectural techniques for implementing bus and ring networks;
- 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;
- agile design methodologies including incremental development and test-driven development.

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
```

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:

• `NetFL.py`	– FL network
• `BusNetDpathPRTL.py`	– PyMTL bus network's datapath
• `BusNetCtrlPRTL.py`	– PyMTL bus network's control unit
• `BusNetPRTL.py`	– PyMTL bus network
• `BusNetDpathVRTL.v`	– Verilog bus network's datapath
• `BusNetCtrlVRTL.v`	– Verilog bus network's control unit
• `BusNetVRTL.v`	– Verilog bus network
• `BusNetRTL.py`	– Wrapper to choose which RTL language
• `RouterDpathPRTL.py`	– PyMTL router's datapath
• `RouterCtrlPRTL.py`	– PyMTL router's control unit
• `RouterPRTL.py`	– PyMTL router
• `RouterDpathVRTL.v`	– Verilog router's datapath
• `RouterCtrlVRTL.v`	– Verilog router's control unit
• `RouterVRTL.v`	– Verllog router
• `RouterRTL.py`	– Wrapper to choose which RTL language
• `RingNetPRTL.py`	– PyMTL ring network
• `RingNetVRTL.v`	– Verilog ring network
• `RingNetRTL.py`	– Wrapper to choose with RTL language
• `net-sim`	– Network simulator for evaluation
• `__init__.py`	– Package setup
• `test/NetFL_test.py`	– FL network tests
• `test/BusNetRTL_test.py`	– Bus unit tests
• `test/RouterRTL_test.py`	– Router unit tests
• `test/RingNetRTL_test.py`	– Ring unit tests
• `test/__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 detailed network microarchitecture. In this lab, you will implement and evaluate two network microarchitectures each with its own topology and routing algorithm: (1) a four-terminal multiple-writer, multiple-reader bus network that only allows a single transaction to use the bus at a time, and (2) a four-terminal ring network that includes four radix-three routers interconnected by eight unidirectional channels. All networks use single flit, single phit packets with a paramterized payload width. This will enable the same network design to be instantiated multiple times with each instantiation using different packet sizes.

We have provided you with a functional-level model of a four-terminal network, which essentially just passes network messages through a magic four-port crossbar. The functional-level model will enable us to develop many of our test cases before attempting to use these tests with the baseline and alternative designs.

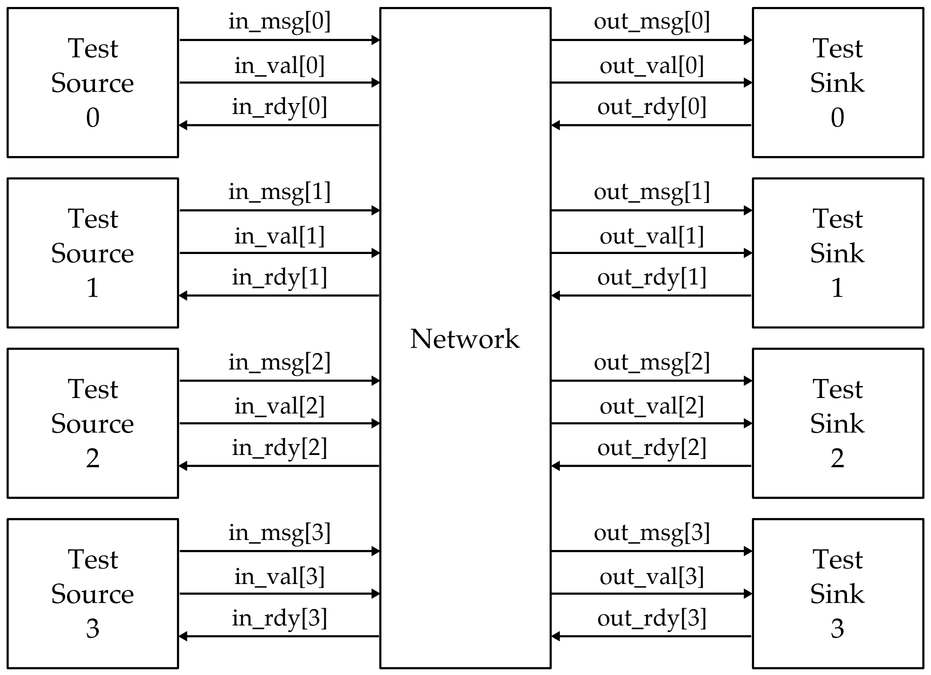

Figure 1: Network – The network is integrated with four test sources and four test sinks for testing.

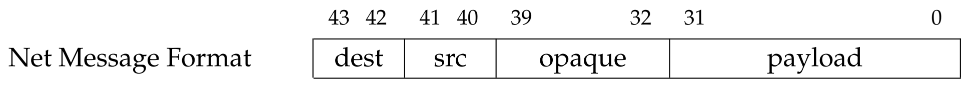

Figure 2: Network Message Format – Network messages are sent from a network input terminal to a network output terminal. Example message is shown with a payload of 32 bits.

Figure 1 shows a block diagram illustrating how the functional-level, baseline, and alternative designs are integrated with test sources and test sinks for testing. Each test source contains a list of network packets which will be sent from an input terminal through the network to the appropriate test sink based on the desired destination terminal. Network evaluation will use a more sophisticated harness with support for infinite source queues and accurately handling warm-up, measure, and drain simulation phases. We make extensive use of the latency insensitive val/rdy micro-protocol in the network interface. Note that the network interface uses *port arrays* to facilitate more productive testing and composition. There are two kinds of val/rdy channels:

- `in[i]` : from input terminal *i* to network
- `out[i]` : from network to output terminal *i*

The message format for network messages is shown in Figure 2. Corresponding PyMTL `BitStructs` are defined in `pclib` here:

- `https://github.com/cornell-brg/pymtl/blob/ece4750/pclib/ifcs/NetMsg.py`

Similar Verilog `structs` are defined in `vc/net-msgs.v` included within the lab release. The source and destination fields are two bits since we are implementing a four-terminal network. The opaque field can be used for implementation defined behavior. You should always ensure the opaque field is correctly preserved from when the packet is injected into the network to when it is ejected from the

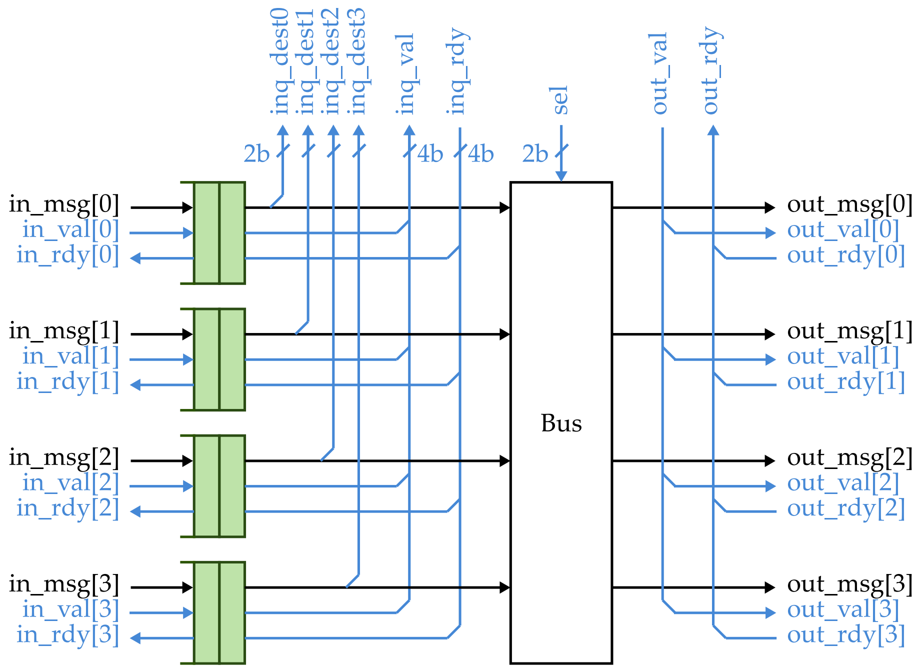

Figure 3: Baseline Datapath – Four-terminal multiple-writer, multiple-reader bus network.

network. While the example in Figure 2 has a 32-bit payload, the payload is configurable. This causes a slight challenge for the Verilog interface since Verilator does not currently allow parameterizing a module by the type of a struct. So we need to be a bit clever to enable using the same network for different size payloads. Our solution for the Verilog networks is for each val/rdy interface to include four ports (i.e., `val`, `rdy`, `msg_hdr`, `msg_payload`), instead of the traditional three ports (i.e., `val`, `rdy`, `msg`). The `msg_hdr` input is a Verilog `struct` while the `msg_payload` input is a standard logic signal with parameterized bitwidth.

2. Baseline Design

The baseline design is a four-terminal multiple-writer, multiple-reader bus network with round-robin arbitration. As with the earlier labs, we will be decomposing the baseline design into two separate modules: the datapath which has paths for moving data through various arithmetic blocks, muxes, and registers; and the control unit which is in charge of managing the movement of data through the datapath. The control unit will be neither pipelined nor an FSM but will instead be a simple combinational controller (i.e., status signals go into the control unit and the corresponding control signals come out within the same cycle).

The datapath for the baseline design is shown in Figure 3. The blue signals represent the control/status signals for communicating between the datapath and the control unit. Your datapath module should instantiate four input queues and and a bus module.

The PyMTL `NormalQueue` model is defined in `pclib` here:

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

```
#-------------------------------------------------------------------------
# NormalQueue
#-------------------------------------------------------------------------

class NormalQueue( Model ):

  def __init__( s, num_entries, dtype ):

    s.enq              = InValRdyBundle ( dtype )
    s.deq              = OutValRdyBundle( dtype )
    s.num_free_entries = OutPort( get_nbits(num_entries) )
```

Figure 4: PyMTL Queue Interface

```
//------------------------------------------------------------------------
// 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

The Verilog `vc_Queue` is defined in `vc/queues.v` included within the lab release. Figure 4 and Figure 5 show the interfaces for the normal queues that you will be using in this lab assignment. The `NormalQueue` (PyMTL) or `vc_Queue` (Verilog) module have a simple interface that cleanly expresses enqueue- and dequeue-message operations. Both operations use the same latency insensitive val/rdy interface we have seen elsewhere in the lab assignments. In this lab we will only be using normal queues, so Verilog implementations should ensure that `p_type` is always set to `‘VC_QUEUE_NORMAL`. The `dtype` (PyMTL) and `p_msg_nbits` (Verilog) parameters configure the bitwidth of the message, and the `num_entries` (PyMTL) and `p_num_msgs` (Verilog) parameters configure the number of messages the queue can hold. 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 which may be useful.

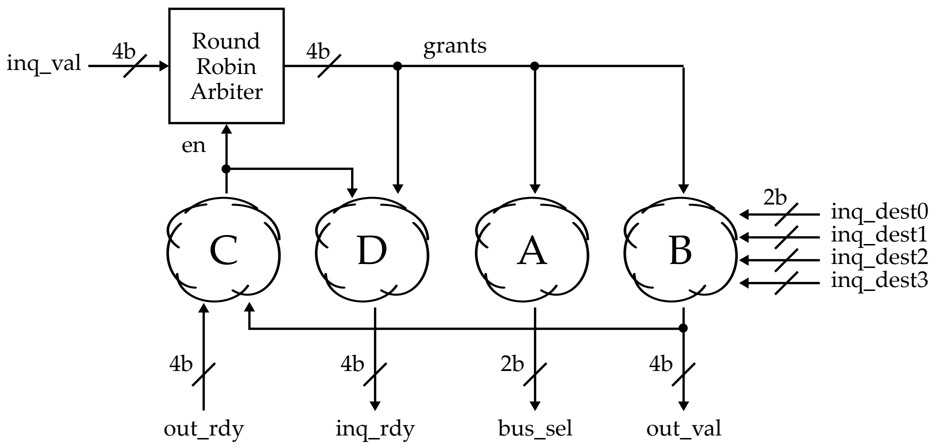

Figure 6: Baseline Control Unit – Combinational controller for four-terminal multiple-writer, multiple-reader bus network.

The PyMTL `Bus` is defined in `pclib` here:

- `https://github.com/cornell-brg/pymtl/blob/master/pclib/rtl/Bus.py`

The corresponding Verilog `vc_Bus` is defined in `vc/buses.v` included within the lab release. The bus essentially uses the `sel` signal to choose which bus input is broadcast to all of the bus outputs.

Figure 6 sketches the bus control unit. The control unit should be a combinational module which first uses round-robin arbitration to choose one of the packets in the four input queues and then sends this queue across the bus and out the appropriate output terminal. Note that while we recommend explicitly instantiating a round-robin arbiter module, the remainder of the control logic (i.e., the "clouds" in Figure 6) can simply be flat logic. The PyMTL `RoundRobinArbEn` is defined in `pclib` here:

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

The corresponding Verilog `vc_RoundRobinArbEn` is defined in `vc/arbiters.v` included within the lab release. Figure 7 and Figure 8 show the interfaces for these round-robin arbiters. The combinational arbiter takes as input a vector of request signals (`reqs`) and produces a vector of grant signals (`grants`). If a "requester" would like to arbitrate for a resource (e.g., the bus), then the requester sets its request signal high. The arbiter looks across all requesters that are making a request and chooses one using round-robin arbitration (i.e., after wining arbitration that requester has the lowest priority in the next cycle). The arbiter sets the grant signal high that corresponds to the winning requester. This means the vector of grant signals is either all zeros or one-hot (i.e., only one bit can be set since only one requester can use the resource at a time). The round-robin arbiters also have an enable signal (`en`). If the `en` signal is low, then the round-robin arbiter does *not* update its internal priority state (i.e., the arbiter does not make the winning requester have lowest priority, instead the priority remains unchanged).

Figure 6 shows four "clouds" of combinational control logic with each cloud being responsible for setting one of four key signals. Cloud A simply encodes the one-hot grant signal as a two-bit binary number. Note that if all four grants are zero, then we can use any value for the bus select (i.e., we aren't actually sending a packet out one of the output terminals so it doesn't matter what packet goes across the bus). Cloud B should choose one of the four destination fields from the input queues

```python
#-------------------------------------------------------------------------
# RoundRobinEn
#-------------------------------------------------------------------------

class RoundRobinArbiterEn( Model ):

  def __init__( s, nreqs ):

    s.en     = InPort ( 1 )     # 1 = update priorities
    s.reqs   = InPort ( nreqs ) # 1 = making a req, 0 = no req
    s.grants = OutPort( nreqs ) # (one-hot) 1 is req won grant
```

Figure 7: PyMTL Round-Robin Arbiter Interface

```verilog
//------------------------------------------------------------------------
// vc_RoundRobinArbEn
//------------------------------------------------------------------------

module vc_RoundRobinArbEn
#(
  parameter p_num_reqs = 2
)(
  input  logic                  clk,
  input  logic                  reset,
  input  logic                  en,      // 1 = update priorities
  input  logic [p_num_reqs-1:0] reqs,    // 1 = making a req, 0 = no req
  output logic [p_num_reqs-1:0] grants   // (one-hot) 1 is req won grant
);
```

Figure 8: PyMTL Round-Robin Arbiter Interface

based on the one-hot grant signal and then set the corresponding output valid bit to one. Note that if all four grants are zero, then all four output valid bits should be zero. Cloud C should combine the output valid bits and the output ready bits to determine if a packet is being sent to an output terminal this cycle and if so, is the corresponding output terminal ready. Only if both conditions are true should we enable the round-robin arbiter and update the internal priority state. Finally, cloud D should set the input queue ready bits based on the grant signal, but only if the enable signal is one (i.e., we only dequeue a packet from an input queue if we know the packet will be able to go across the bus and out the desired output terminal). Note that this is just a sketch of the control logic to get students started; students are free to implement the control logic in any way they want.

We strongly encourage you to take an incremental design approach. For example, you could first implement the datapath structurally and then hard-code the control unit to always route input terminal 0 to output terminal 0 and you could ignore back-pressure (i.e., assume the output interfaces are always ready). After testing this initial step, you could add round-robin arbitration but assuming all input terminals always want to go to output terminal 0 again ignoring back-pressure. Or after the initial step, you could choose instead to always route input terminal *i* to output terminal *i* without back-pressure. Then you could incrementally add support for back-pressure or add support for routing based on the destination field. Hopefully, it is clear that there are many incremental design strategies. The important point is that you sit down and plan out an incremental-design strategy, gradually add complexity to the design, and incrementally test every step of the way. We strongly discourage implementing the entire control unit in a single step.

3. Alternative Design

For the alternative design you will be implementing a four-terminal ring network with four routers and eight unidirectional channels. Figure 9 illustrates the composition of routers and channels you should use to form the ring. Notice that each router has three inputs and three outputs, and that each channel is modeled using a two-element normal queue. This means these channels are using *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.

Before implementing the ring, you will want to spend time designing and testing the ring router in isolation. Each ring router has three input val/rdy interfaces and three output val/rdy interfaces. You are required to use the following mappings for the three interfaces, and Figure 9 also shows the corresponding interface numbers you should use.

- Interface 0: connects to the previous west-side router
- Interface 1: connects to the input/output terminal
- Interface 2: connects to the next east-side router

The router should use a control/datapath split. The datapath should look very similar to the bus datapath shown in Figure 3 except that the router will only have three input/output interfaces, and the router should use a crossbar instead of a bus. The PyMTL `Crossbar` is defined in `pclib` here:

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

The corresponding Verilog `vc_Crossbar3` is defined in `vc/crossbars.v` included within the lab release. The crossbar models three multiple-writer, single-reader buses (one bus per output port) to enable all inputs to send packets to all outputs as long as every input is going to a different output. The control unit will be more complex compared to the bus control unit shown in Figure 6. The control unit will need three round-robin arbiters, one per output port. The control unit will need to look at the destination field from each of the three input queues to decide how to set the requester bits going to each of the three round-robin arbiters. The control unit will also need to carefully control the `en` signal of each round-robin arbiter so that the priority is only updated if the corresponding output port is ready.

As always we recommend an incremental design approach. You can use a similar strategy as used for the baseline design, except you may want to start by ignoring the routing algorithm. Simply use the destination field in each packet to decide the corresponding output port. Once this incremental router is completely functional you could add the routing logic. You will need to spend some time thinking very carefully about what routing algorithm to use. A greedy routing algorithm can create

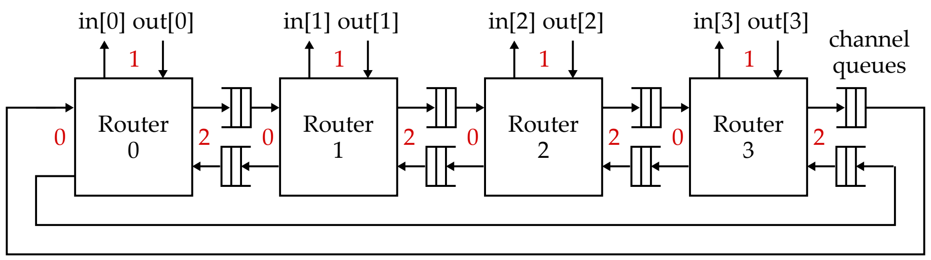

Figure 9: Alternative Ring Network – Every arrow represents an independent val/rdy interface. Red numbers indicate the router port numbers for that interface.

deadlock if all input terminals can potentially route in the same direction when sending packets to the terminal on the opposite side of the network. We recommend using the odd/even routing algorithm: odd input terminals use the clockwise channels when the destination is equidistant from the source, but even input terminals use the counter-clockwise channels when the destination is equidistant from the source. In other situations the algorithm behaves the same as greedy routing. Note that this routing algorithm only prevents deadlock on a four-node ring and will not prevent deadlock in rings with more nodes. Students are free to experiment with other routing algorithms (e.g., oblivious non-deterministic algorithms, adaptive algorithms) although they will need to implement a different deadlock avoidance scheme (e.g., more queues, bubble-flow control).

Once the router is thoroughly unit tested, you should compose four routers and eight channel queues to create the full ring network shown in Figure 9 and perform integration testing.

4. Testing Strategy

The tests for this lab may be challenging since you will need to carefully craft directed tests that exercise all paths through the network. As with the previous labs, you will want to initially write tests using the functional-level model. Once these tests are working on the functional-level model, you can move on to testing the baseline and alternative designs. One key difference from previous labs is the need for rigorous unit testing of a sub-module before integrating the full design. In the alternative design, you must spend significant time unit testing the router before moving on to testing the complete ring network.

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 tests we provide on the various designs.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% py.test ..
% py.test ../lab4_net
% py.test ../lab4_net/NetFL_test.py
% py.test ../lab4_net/BusNetRTL_test.py
% py.test ../lab4_net/RouterRTL_test.py
% py.test ../lab4_net/RingNetRTL_test.py
```

You will add your directed and random tests for the full network to `NetFL_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. You will be adding more tests cases to `RouterRTL_test.py` to unit test the router in isolation.

Figure 10 illustrates how we will be writing tests for a full network. The `mk_net_msgs` helper function takes a list of four-tuples, where each tuple includes the source and destination terminal numbers, an opaque field for meta-information, and the payload of the network message. In this specific example, a single input terminal sends one packet to each of the four output terminals. As with other labs, you also need to add your new test cases to the test case table. Some suggestions for what you might want to test in the full 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

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

def single_src_msgs():
  return mk_net_msgs( 4,
    #   src dest opaque payload
    [ ( 0,  0,   0x00,  0xce ),
      ( 0,  1,   0x01,  0xff ),
      ( 0,  2,   0x02,  0x80 ),
      ( 0,  3,   0x03,  0xc0 ), ]
  )

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

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

Figure 10: Directed Test Example for Full Network – Simple directed test for full network that uses a `mk_net_msgs` helper function to create a stream of packets from input terminal 0 to each of the four output terminals.

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

You can then create directed test cases that capture longer sequences 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
- `opposite`: prolonged traffic half-way around the ring

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 for the ring network. Figure 11 illustrates a line trace for the basic test in Figure 10 executing on the bus network design with extra annotations to indicate what the columns mean. Each network message has the format: *(opaque field) : (source) > (dest)*. The opaque field has no functionality but is useful to differentiate messages from each other. The first four columns represent the four test sources (i.e., the network input terminals), and they show when a message is injected into the network. The middle eight columns show what is happening in the bus network; they shown when a packet enters the bus and when it exits. The last three columns represent the four test sinks (i.e., the network output terminals) and they show when a message is ejected from the network. The line trace clearly shows four packets entering the network on cycles 3–6, and then each packet taking one cycle to exit the bus and reach the corresponding output terminal.

Figure 12 illustrates a line trace for the basic test in Figure 10 executing on the ring network design. Each router has three columns corresponding to the router's three input ports. Packet 0 enters the network on cycle 3, goes through router 0, and exits the network on cycle 4. Packet 1 enters the network on cycle 4 but since this packet is destined for output terminal 1 it must go clockwise to

```
cyc  src0   src1   src2   src3        in0    out0    in1    out1    in2    out2    in3    out3        sink0  sink1  sink2  sink3
  2:       |      |      |       >>> (      |.     )(      |.     )(      |.     )(      |.     ) >>> .     |.     |.     |.
  3: 00:0>0|      |      |       >>> (00:0>0|      )(      |      )(      |      )(      |      ) >>>       |      |      |
  4: 01:0>1|      |      |       >>> (01:0>1|00:0>0)(      |      )(      |      )(      |      ) >>> 00:0>0|      |      |
  5: 02:0>2|      |      |       >>> (02:0>2|.     )(      |01:0>1)(      |      )(      |      ) >>> .     |01:0>1|      |
  6: 03:0>3|      |      |       >>> (03:0>3|.     )(      |.     )(      |02:0>2)(      |      ) >>> .     |.     |02:0>2|
  7:       |      |      |       >>> (      |.     )(      |.     )(      |.     )(      |03:0>3) >>> .     |.     |.     |03:0>3
```

Figure 11: Line Trace for Bus Network – The line trace shows input terminal 0 (i.e., `src0`) sending four packets to each of the four output terminals (i.e., `sink0`, `sink1`, etc).

```
                      ----- router 0 -----  ----- router 1 -----  ----- router 2 -----  ----- router 3 -----
                      CW     inject CCW     CW     inject CCW     CW     inject CCW     CW     inject CCW
cyc  src0             in0    in1    in2     in0    in1    in2     in0    in1    in2     in0    in1    in2         sink0  sink1  sink2  sink3
  2:       | ... >>> (      |      |      )(      |      |      )(      |      |      )(      |      |      ) >>> .     |.     |.     |.
  3: 00:0>0| ... >>> (      |00:0>0|      )(      |      |      )(      |      |      )(      |      |      ) >>>       |      |      |
  4: 01:0>1| ... >>> (      |01:0>1|      )(      |      |      )(      |      |      )(      |      |      ) >>> 00:0>0|      |      |
  5: 02:0>2| ... >>> (      |02:0>2|      )(      |      |      )(      |      |      )(      |      |      ) >>> .     |      |      |
  6: 03:0>3| ... >>> (      |03:0>3|      )(01:0>1|      |      )(      |      |      )(      |      |      ) >>> .     |      |      |
  7:       | ... >>> (      |      |      )(      |      |      )(      |      |      )(      |      |02:0>2) >>> .     |01:0>1|      |
  8:       | ... >>> (      |      |      )(      |      |      )(      |      |      )(      |      |03:0>3) >>> .     |.     |      |
  9:       | ... >>> (      |      |      )(      |      |      )(      |      |02:0>2)(      |      |      ) >>> .     |.     |      |03:0>3
 10:       | ... >>> (      |      |      )(      |      |      )(      |      |      )(      |      |      ) >>> .     |.     |02:0>2|.
```

Figure 12: Line Trace for Ring Network – The line trace shows input terminal 0 (i.e., `src0`) sending four packets to each of the four output terminals (i.e., `sink0`, `sink1`, etc).

router 1. Packet 1 enters router 1 on cycle 6. Notice how it took one cycle for packet 1 to go through router 0 (router latency of one cycle) and another cycle to go over the channel connecting router 0 to router 1 (channel latency of one cycle). Packet 2 is destined for output terminal 2 and so there are two equidistant paths. This ring network uses an odd/even routing algorithm, so packet 2 must go counter-clockwise around the ring. Packet 2 enters router 3 on cycle 7, enters router 2 on cycle 9, and exits the network on cycle 10.

Figure 13 illustrates how we will be writing tests for a single router. The `mk_router_msgs` helper function takes a list of six-tuples, where each tuple includes everything needed to create a packet and to specify which input port the packet should enter the router and which output port the packet should exit the router. Note that `tsrc` and `tsink` are the router ports for the network packet, not the network terminal ports. The test case shown in Figure 13 assumes it will be used with router 0. It generates four packets. Each packet has its source set to input network terminal 0, and each packet is destined for a different output network terminal. `tsrc` is 1 since this test assumes it will be used with router 0 so all four packets would be injected into the network at this router and thus would be entering the router through router port 1. `tsink` specifies which router port the network packet is expected to exit the router. Think critically about why the `tsink` column is set in this way.

Figure 14 illustrates a line trace for the basic test in Figure 13 executing on the ring router. There are three columns for the three test sources and three columns for the three test sinks. In addition, the three input ports for router are also shown. All four packets are enter the router on router port 1 and exit either through router port 0 or router port 1.

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:

```
#-------------------------------------------------------------------------
# Test case: one packet
#-------------------------------------------------------------------------

def basic_router0_msgs():
  return mk_router_msgs( 4,
    #   tsrc tsink src  dest opaque payload
    [ ( 1,   1,    0,   0,   0x00,  0xfe ), # inject -> eject
      ( 1,   2,    0,   1,   0x01,  0xde ), # route CW
      ( 1,   0,    0,   2,   0x01,  0xde ), # route CCW
      ( 1,   0,    0,   3,   0x02,  0xce ), # route CCW
    ]
  )

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

test_case_table = mk_test_case_table([
  (                       "msgs                 routerid src_delay sink_delay"),
  [ "basic_router0_msgs", basic_router0_msgs(), 0,       0,        0          ],
])
```

Figure 13: Directed Test Example for Ring Router – Simple directed test for ring router that uses a `mk_router_msgs` helper function to create a stream of packets from input terminal 0 to each of the four output terminals. This test assumes it will be applied to router 0.

```
                            from  input  from
                            CW    term   CCW
cyc  src0  src1   src2      in0   in1    in2       sink0 sink1 sink2
  2:       |      |      > (      |      |     ) > .     |.     |.
  3:       |00:0>0|      > (      |00:0>0|     ) >       |      |
  4:       |01:0>1|      > (      |01:0>1|     ) >       |00:0>0|
  5:       |01:0>2|      > (      |01:0>2|     ) >       |.     |01:0>1
  6:       |02:0>3|      > (      |02:0>3|     ) > 01:0>2|.     |.
  7:       |      |      > (      |      |     ) > 02:0>3|.     |.
```

Figure 14: Line Trace for Ring Router – The test assumes this router is router 0. The line trace shows the input terminal sending injecting fourt packets destinated for all four network output terminals.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab4_net/net-sim --impl bus  --pattern urandom --injection-rate 10
% ../lab4_net/net-sim --impl ring --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 % 4`
- `partition2` : `dest = (random & 2’b01) | (src & 2’b10)`
- `opposite` : `dest = (src + 2) % 4`
- `neighbor` : `dest = (src + 1) % 4`
- `complement` : `dest = ~src`

The simulator injects packets and measures the latency. You can specify the injection rate using the `--injection-rate` option (value given to `--injection-rate` is a percentage). 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. 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/sim/build
% ../lab4_net/net-sim --impl bus  --pattern urandom --sweep
% ../lab4_net/net-sim --impl ring --pattern urandom --sweep
```

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

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

Pattern: urandom

Injection rate (%)    | Avg. Latency
1                     | 1.0
5                     | 1.1
10                    | 1.3
15                    | 1.6
20                    | 2.9
22                    | 3.4
23                    | 4.9
24                    | 7.2
25                    | 52.3
26                    | 177.1

Zero-load latency = 1.0
```

These results suggest the bus saturates around 25%, meaning that each network input terminal can only sustain injecting one packet on average every four cycles. The zero load latency is one cycle as expected. 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.

Acknowledgments

This lab was created by Shunning Jiang, Ian Thompson, 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.