ECE 4750 Computer Architecture, Fall 2013

Lab 4: Ring Network

School of Electrical and Computer Engineering
Cornell University

revision: 2013-11-02-21-35

In this lab you will implement an 8-node bidirectional ring network with elastic-buffer flow control, round robin arbitration and explore two different routing algorithms: greedy routing for the baseline design and an adaptive routing scheme for the alternative design. All packets used in the network will be a single flit and each flit will be composed of a single phit. You will use implement a bubble-flow control scheme for deadlock prevention in the ring network in either 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 includes an interesting and challenging control logic implementation compared to the previous lab assignments. This lab is designed to give you experience with:

- a single cycle-router datapath which includes input queues, crossbar switch and round-robin arbiter networks
- implement an elastic-buffer flow control algorithm based on the familiar val-rdy handshaking protocol
- effective design patterns including control/datapath split for interconnection networks and message interfaces

This handout assumes that you have read and understand the course tutorials. To get started, you should access a course development machine and perform the following steps:

```
% cd ${HOME}
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-lab4-net
% cd ${HOME}/ece4750/ece4750-lab3-mem/build
% ../configure
% make
% make check
```

For this lab you will be working in the `plab4-net` subproject which includes the following files:

- `plab4-net-RingNetSimple.v` – 8-port simple network implemented as a global crossbar
- `plab4-net-RingNetSimple.t.v` – unit tests for the simple network
- `plab4-net-RouterBase.v` – empty shell for baseline router
- `plab4-net-RouterBase.t.v` – test harness for baseline router
- `plab4-net-RouterInputCtrl.v` – empty shell for input control
- `plab4-net-RouterInputCtrl.t.v` – test harness for input control
- `plab4-net-RouterInputTerminalCtrl.v` – empty shell for input terminal control

- `plab4-net-RouterInputTerminalCtrl.t.v` – test harness for input terminal control
- `plab4-net-RouterOutputCtrl.v` – empty shell for output control
- `plab4-net-RouterOutputCtrl.t.v` – test harness for output control
- `plab4-net-RingNetBase.v` – 8-port baseline ring network implementation
- `plab4-net-RingNetBase.t.v` – unit tests for the baseline network
- `plab4-net-RingNetAlt.v` – 8-port alternative ring network implementation
- `plab4-net-RingNetAlt.t.v` – unit tests for the alternative network
- `plab4-net-test-harness.v` – test harness for the network shared among all implementations
- `plab4-net-input-gen.py` – Python script to generate patterns for testing
- `plab4-net-plot-gen.py` – Python script to generate plots for the evaluation
- `plab4-net-sim-simple.v` – simulator for the simple network
- `plab4-net-sim-baseline.v` – simulator for the baseline network
- `plab4-net-sim-alt.v` – simulator for the alternative network
- `plab4-net-sim-harness.v` – simulator harness for the network shared among all implementations
- `plab4-net.mk` – makefile fragment to build the project

You can use the `ece4750-lab-admin` script and the `git` command to add members to your lab group, use your own group repository to collaborate with your group members, and ultimately prepare your submission for uploading to CMS. **Remember to use the ece4750-lab-admin script to create the submission tarball. All the tests, simulation experiments and VCD dump generation must be run in the build directory!**

1. Introduction

Modern semiconductor chips have an increasing number of on chip subsystems such as processor cores, caches and specialized hardware. On-chip interconnection networks play an important role in connecting these components and providing a means for communication between various subsystems. In class you have learnt about basics of networks and have explored various network topologies, routing algorithms and discussed a few flow control-algorithms.

In this lab, you will get an opportunity to design an 8-node bidirectional ring network which uses an elastic-buffer flow control and round-robin arbitration. You will explore the benefits of two different routing algorithms as part of the baseline design and alternative design. The network you implement considers only single-flit/phit packets.

In class you learnt about different flow-control schemes such as on-off flow control and channel buffer flow control for networks. In this lab, we introduce to a simple flow control algorithm called as elastic-buffer flow control which exploits the elastic buffer or storage present on the pipelined channels. With an increasing transistor budget, wires which are available in abundance render the buffers and storage elements expensive. Elastic-Buffer (EB) flow-control algorithm is an example of a flow-control algorithm which reduces the amount of storage required to design a network-on-chip infrastructure. It simplifies the router microarchitecture and reduces the area and power in routers.

For the networks designed in this lab all pipelined channels are modeled as two-element normal queues which forward a flit from one router to another by reusing the val-rdy handshaking protocol. Before we jump to the discussion of EB flow-control, we discuss the implementation of normal queues defined in `vc-queues.v`. The interface to multiple-entry normal queues is as given below:

```
//------------------------------------------------------------------------
// 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
);
```

The `vc_Queue` module has a message based-interface which uses the val-rdy handshaking protocol. We specify the type of the queue using `p_type` parameter. In this lab, we will use normal queues for both router input queues and the channel queues. The remaining parameters `p_msg_nbits` and `p_num_msgs` configure the bitwidth of the message and the number of messages the queue can hold respectively.

The implementation internally tracks the elements being enqueued and dequeued by using head and tail pointers. To `enqueue` a message at a location pointed by the head pointer, `enq_msg` is set to the value to be enqueued and a corresponding `enq_val` signal is set. To dequeue an entry from the queue the `deq_rdy` signal communicates to the queue that a requester is popping an entry out of the queue. Additionally, the queue also provides an output port which provides the number of free entries in the queue. Take a look at the corresponding unit test to further understand the operations on a normal queue.

2. Baseline Design

Figure 1 shows the 8-node bidirectional ring network you will be implementing for the baseline and alternative designs. The ring has 8 nodes as shown and has two sets of ring networks one in each of the east and west directions.

The incremental design process for this lab will start with a router design using only EB flow control, but no deadlock-avoidance. The router will use a deterministic greedy-routing algorithm scheme for the baseline design. You should thoroughly unit test the individual control units for input and output as well as the router as a whole. You will then test a ring network composed of these routers. After constructing a situation which causes deadlock, you will implement bubble flow control to eliminate deadlock. Note that bubble flow control does not replace EB flow control and it is an additional

mechanism to prevent deadlocks Finally, the ring network with these updated routers will be tested. In summary we suggest you follow the following incremental design strategy:

- Implement individual control units
- Router with only EB flow-control
- Ring with only EB flow-control
- Router with EB flow-control and bubble-flow control
- Ring with EB flow-control and bubble-flow control

The network information such as the number of routers, number of messages and payload bitwidth is used to determine the network packet size. Similar the `vc-mem-msgs.v`, we have provided you `vc-net-msgs.v` which provides macros and pack/unpack modules regarding network packets. Take a look at this model and the corresponding unit test to understand the network packet. Each network packet has the route-information (dest-src fields), an opaque field to put meta information about the packet, and lastly the payload or the actual message to be communicated.

The datapath for the single-cycle router is as shown in Figure 2. The router you will implement has three ports each with its own set of inputs and outputs. For this lab, you should the following definitions for the ports to ensure that unit tests and simulation harness work as intended. Note, the definitions for the ports are arbitary but we choose the below:

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

Each router input queue should have 4 entries. The enqueue and dequeue operations use the val-rdy handshaking protocol you are familiar with. To implement the crossbar switch we have provided `vc-crossbars.v` which provide crossbars of various number of inputs. You need to instantiate a crossbar with the bitwidth of the message field. As usual, the blue line indicate the control and status signals between the control and datapath units.

2.1. Elastic-Buffer flow control

Elastic Buffer (EB) flow control uses the val-rdy handshaking protocol to forward packets between channels and routers. Each channel in the elastic-buffer flow control is reused as buffers when forwarding packets between routers to a destination. The storage available in form of the pipeline channels form the elastic buffers. You will model each pipelined channel as a two-entry normal queue in the ring network you will be implementing. Consider a single router and the channels connecting the router in east and west directions. When a packet is injected into the network from the input terminal port or is being forwarded in either of the west or east directions, the ready signal from the channel connecting to the downstream router indicates available space to the packet being forwarded from the current router. The output valid signal on any of the output ports (in east, west or output terminal ports) indicates that the there is a valid packet for the downstream. When both the output valid and ready signals are high a packet gets forwarded from the upstream router to the downstream. The congestion in the channels or the lack of available space downstream gets communicated by the output ready signals (in each of east, west or output terminal ports) and the packets are not dropped but get held up in either the channel queues or input queues in the routers.

For the baseline design, you will be implementing a deterministic greedy routing algorithm. As an incremental step you are required to first implement the router which has only the EB flow control. You will then implement a ring network with the these routers.

Figure 3 shows the control unit for the router with only the EB flow-control. Note that for this lab, the control unit you will implement differs significantly compared to all the previous labs. We take apply the principles of modular design and break the control logic into the following sub-models: `RouterInputCtrl` and `RouterOutputCtrl`. For each port present in the router we instantiate the `RouterInputCtrl` and `RouterOutputCtrl` models as shown and connect them structurally.

`RouterInputCtrl` module takes the destination of the packet present at the tail which is ready to be dequeued and handles the dequeue val-rdy signals. Based on the destination the model computes the route and computes the input requests to the arbiters present in the `RouterOutputCtrl` modules. You can implement the logic which computes the route in a separate module in `plab4-net-GreedyRouteCompute.v` which implements the greedy-routing algorithm. The greedy route computation calculates the number of hops it takes to go to the destination using east or west direction. Then it picks the direction with the least number of hops. Note that because we have an even number of nodes in the ring network, it takes the same number of hops either direction to go across the network. For the baseline design, you should implement a deterministic routing algorithm. When the number of hops are equal either direction, you can just pick one way, e.g. always east. The input control also obtains the grant signals which is used to calculate the input dequeue ready signals passed to the input queues present in the datapath. Note that if you do handle route computation in a separate module, be sure to unit test this module.

`RouterOutputCtrl` module accepts all the incoming requests and handles the output val-rdy signals used to communicate with the outgoing channels. The module instantiates a round-robin arbiter with enable signal. The arbiters are provided in `vc-arbiters.v`. The arbiter takes the incoming requests as an input using `reqs` port and grants one of these requestors the access to use the resource. It uses the `grants` output port, which uses a one-hot encoding, to notify the requestor that it has the right to use the resource. While assigning the grants, it keeps track of the last user of the resource and updates the priority accordingly. The `RouterOutputCtrl` communicates the grant signals to the `RouterInputCtrl` models as shown in the Figure 3. It is also responsible to make sure to grant the resource if the channel queues are ready to accept a new message and set the correct crossbar select bits.

We have provided you with unit tests for each of the control modules, but feel free to modify these if your implementation slightly varies with our reference implementation. Once you implement the router as specified, you should unit test the router. We have provided you `plab4-net-RingNetBase.v` which composes the routers you designed. Composing the routers in Verilog 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 suggest. As always, talk about any differences in the datapath or control logic in the lab report.

We have provided some tests which would help you verify the functionality of the ring in test harness file (`plab4-net-test-harness.v`). You could extend the tests by adding more test cases. Note that some of the tests could fail due to deadlock at this stage. As part of the Testing Strategy, you should come up with the shortest possible test case that will put your network into a deadlock. Once you are confident that the ring network with EB flow-control is working you can proceed to implement the bubble-flow control.

2.2. Deadlock Prevention: Bubble-Flow Control

The ring network with only EB flow control does not guarantee deadlock prevention. You need to implement a ring with bubble-flow control for deadlock prevention as the final submission baseline design. You do not need to modify the ring network source as long as you maintain the interface. The

main task here is to test the network and determine the case in which deadlock occurs. Implement the bubble-flow control in the routers and test the network for deadlock avoidance.

Deadlock occurs when a group of packets are unable to make progress toward their destinations because they are waiting on one another to release 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 susceptible to deadlock because of the cyclic dependency introduced by the wrap-around channel.

Your first task before you add logic to implement deadlock avoidance is to create a scenario which results in a deadlock condition in your ring network. One method of avoiding the deadlocks you encountered in the previous step is to add another flow control technique called bubble flow control.

Bubble flow control is a simple deadlock-avoidance scheme that we will be using for this lab. A bubble in this context is a free packet buffer. In a unidirectional ring, the guaranteed existence of a bubble is sufficient to avoid deadlock. We can guarantee the existence of a bubble by restricting the conditions under which a packet may be forwarded from the buffer of an input terminal. Consider a packet which we intend to inject into any of the ring directions, east or west. A packet may be forwarded from the input terminal port only if the number of free entries present in the input buffer holding packets being routed in the intended direction is greater than one. For example, if the input terminal intends to inject a packet to a router in the east direction it can only do so if the number of free entries in the input queue present at the west-side port is greater than one. Similar conditions are determined when routing packets in the west direction. Please convince yourself that imposing this restriction avoids deadlock. Note that bubble flow control introduces a minimum input buffer size of two packet buffers.

The datapath for the router implementing bubble-flow control and EB flow-control is as shown in Figure 2. Figure 4 shows the control unit implementation for the router with bubble-flow control for deadlock avoidance. Most of the control logic units remain the same with the replacement of `RouterInputTerminalCtrl` module for the input control logic of the input terminal port. This model additionally takes the number of free entries in the input queues present at west and east side ports to implement the bubble-condition as described in the paragraph above.

Once you implement the bubble-flow control try to run the ring tests which create the deadlock condition and your unit tests must pass with deadlock avoidance.

In your report discuss why this is a good baseline design. Describe how you implemented the provided datapath and control units. Describe the input and output logic structures and how you implemented it. Describe the route computation calculations and any other changes you made to the design provided.

3. Alternative Design

The goal of the alternative design is to implement an adaptive routing algorithm which senses the congestion in the network and routes the packets to balance the load on channels. The specification for the alternative design remains the same for the router and the ring network with the only change being in the route computation. The alternative design must implement an adaptive routing scheme which adapts to the conditions in the network and aim to improve the performance. You can sense the congestion on the channels by observing the output ready signals and change the route computation compared to the baseline greedy-routing algorithm. Additionally, you can also sense the congestion by looking at the number of free entries in the channel queues in either direction.

Note that the most immediate channel queue is still non-global congestion information. You could also experiment with more global congestion information by looking at channel queues farther away. However, in an on-chip setting, we might not be able to get this congestion information combinationally in a single cycle. So you must pass this congestion information through registers to simulate the latency in wires. You do not need to have this latency for the most immediate channel queues, but you do need to have an additional cycle of latency for each extra hop you are making to get the channel congestion.

Once you have congestion information, you can implement a weighted routing scheme which factors in the congestion and misrouted hop count to decide the route for an outgoing packet.

In your report you must clearly explain why and how you implemented the adaptive routing scheme. Under what conditions would you expect the router to perform than the baseline design. What are the trade offs involved? Include a modified datapath or control units in your report.

4. Testing Strategy

For this lab, you will be using a unit test framework which uses test sources and sinks to test the router and the ring network. Specifically you are required to extend the provided tests for the ring network with the following:

- tests which induce a deadlock in the east and west direction ring networks
- tests which trigger adaptive non-minimal routes

For both of these tests, try to come up with shortest possible test case that has the requested behavior. Some of the test case we have provided you might have the behavior, such as deadlocking, but design a test case that is the shortest.

To test the routers, as you can see in `plab4-net-RouterBase.t.v`, we use the `init_net_msg` task, which lets you define four fields of a network message: the source, destination, opaque and payload fields. In addition, we also specify which input port the message is coming from and which output port is it expected to go. There are three test sources and sinks corresponding to the three ports of a router: east, west and terminal. The following is what a test case for the routers looks like:

```
//------------------------------------------------------------------------
// basic test
//------------------------------------------------------------------------

‘VC_TEST_CASE_BEGIN( 1, "basic test" )
begin
  init_rand_delays( 0, 0 );

  //            in    out   src   dest  opq    payload
  init_net_msg( 2’h0, 2’h2, 3’h1, 3’h3, 8’h00, 8’hce );
  init_net_msg( 2’h2, 2’h0, 3’h7, 3’h0, 8’h05, 8’hfe );
  init_net_msg( 2’h1, 2’h1, 3’h2, 3’h2, 8’h30, 8’h09 );
  init_net_msg( 2’h2, 2’h0, 3’h4, 3’h1, 8’h10, 8’hfe );
  init_net_msg( 2’h0, 2’h2, 3’h1, 3’h4, 8’h15, 8’h9f );
  init_net_msg( 2’h2, 2’h1, 3’h3, 3’h2, 8’h32, 8’hdf );
  init_net_msg( 2’h0, 2’h2, 3’h1, 3’h3, 8’h23, 8’hfe );
  init_net_msg( 2’h0, 2’h1, 3’h1, 3’h2, 8’h31, 8’hb0 );
  init_net_msg( 2’h1, 2’h0, 3’h2, 3’h1, 8’h70, 8’h89 );
```

```
  run_test;
end
`VC_TEST_CASE_END
```

`plab4-net-test-harness.v` file contains test cases for the ring network. At the top level test harness which tests the whole network, there are 8 test sources and sinks, the ring network itself. The `init_net_msg` task lets you define four fields of a network message: the source, destination, opaque and payload fields. Unlike the test for the router, this task does not need to know input and output ports because we always use the terminal ports of the corresponding sources and destination, so the task uses those fields to make sure the correct test source sends this packet and the correct test sink receives it. The following is a snippet of a test case using this task:

```
//------------------------------------------------------------------------
// single source
//------------------------------------------------------------------------

`VC_TEST_CASE_BEGIN( 1, "single source" )
begin
  init_rand_delays( 0, 0 );

  //            src   dest  opq    payload
  init_net_msg( 3'h0, 3'h0, 8'h00, 8'hce );
  init_net_msg( 3'h0, 3'h1, 8'h01, 8'hff );
  init_net_msg( 3'h0, 3'h2, 8'h02, 8'h80 );
  init_net_msg( 3'h0, 3'h3, 8'h03, 8'hc0 );
  init_net_msg( 3'h0, 3'h4, 8'h04, 8'h55 );
  init_net_msg( 3'h0, 3'h5, 8'h05, 8'h96 );
  init_net_msg( 3'h0, 3'h6, 8'h06, 8'h32 );
  init_net_msg( 3'h0, 3'h7, 8'h07, 8'h2e );

  run_test;
end
`VC_TEST_CASE_END
```

We additionally have provided test messages for the following patterns:

- nearest neighbor pattern
- hotspot traffic pattern
- uniform random traffic pattern
- tornado traffic pattern

As outlined in the course lab assignment assessment rubric, the testing strategy section in your lab report should briefly summarize the approach used in testing the baseline design before focusing in detail on the tests you used to verify your alternative design. Describe why your tests create a deadlock and how does it trigger adaptive non-minimal routes for the alternative design.

5. Evaluation

To evaluate the baseline and alternative design you have implemented, you should use the provided simulator which sweeps across injection rates until the network is saturated. Example usage of the simulator is as below:

```
% cd ${HOME}/ece4750/ece4750-lab4-net/build
% make
% ./plab4-net-sim-base +help
% ./plab4-net-sim-simple +input=urandom +stats
% ./plab4-net-sim-base   +input=urandom +stats
% ./plab4-net-sim-alt    +input=urandom +stats
```

The simulator supports a variety of patterns described below. For each one, we can succinctly describe the pattern using Verilog syntax. Note that `random` is a 32-bit random integer.

- `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`
- `reverse` – `dest = {src[0], src[1], src[2]}`
- `rotation` – `dest = {src[0], src[2], src[1]}`

The simulator will inject packets at an injection rate and measure the latency to go through. It will sweep the injection rate and report the average latency for each injection rate. As the simulator sees that the network is approaching saturation, it will decrease the steps to increase the injection rate to have more data points. The `+stats` option displays the zero load latency and the injection rate which saturates the network, which is also the bandwidth. We also provide you with a plotting script that will plot the latency bandwidth curve for you. We plot three designs, the baseline and alternative, as well as the simple network, which is a global crossbar. The global crossbar is close to be an ideal network, and it should be thought of a functional level modelling of a real network such as the one you are implementing. In most cases, you will see that the simple network performs better than your networks, and this is because of it being closer to an ideal network.

To run the simulation on all three networks and on all patterns, and to generate the plots as png files, you can use the following command:

```
% make eval-plab4-net -j16
```

This is similar to the evaluation of the previous labs except for the `-j16` flag. Because the injection rate sweeps and the required warmup periods for each injection rate, the simulations take longer than previous labs. The `-j` flag is a built-in feature to `make` utility where it parallelizes the work by the amount you specify. Because `amdpool` machines are 8-core machines with multi-threading (concepts we will soon learn in class), we can take advantage of this by parallelizing the simulations 16-way.

In your lab report discuss the performance of each design for the given traffic patterns. Does the simulation results correspond to what you expect to see? Does the baseline design perform better than the alternative design? Which design would you pick for a generalized network traffic pattern? Why does the "simple" network almost always perform better? Make sure that the plots make sense to you and reason out what you observe.

6. Extensions

We will be accepting extensions for this lab. As with the previous lab, you should not attempt the extensions after you have fully finished the lab and the report. The following are some suggestions you could implement, but you are free to come up with your own idea to implement:

- **Routing algorithms** – You can implement other routing algorithms. For example, you could try uniform random or weighted random routing and see if the network indeed performs as you predicted using the methods you have learned in the lecture.
- **Flow-control mechanisms** – You could try implementing on-off or credit-based flow control scheme instead of elastic-buffer flow control and see how your network performs.
- **Deadlock avoidance** – You could implement an alternative deadlock avoidance scheme. One way to do that is to use virtual channels. You can do some read-up on what virtual channels are and how to implement them. Another idea is to implement bubble flow control only on one single router. Only one bubble in the whole network is needed to prevent deadlocking, but our current scheme where each router implements bubble flow control is conservative. You can experiment and see how the new deadlock avoidance performs.
- **Network size** – You could change the number of routers in the design, and even do a sweep of different sizes. Note that this might be a little more involved than you might initially think. It is not possible to parameterize everything in Verilog, and some of the test harnesses might have hard-coded some bitwidths or number of ports. So you would need to figure out a way to have a different number of routers. You could evaluate how do these routing algorithms perform when the network is larger or smaller.

Acknowledgments

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

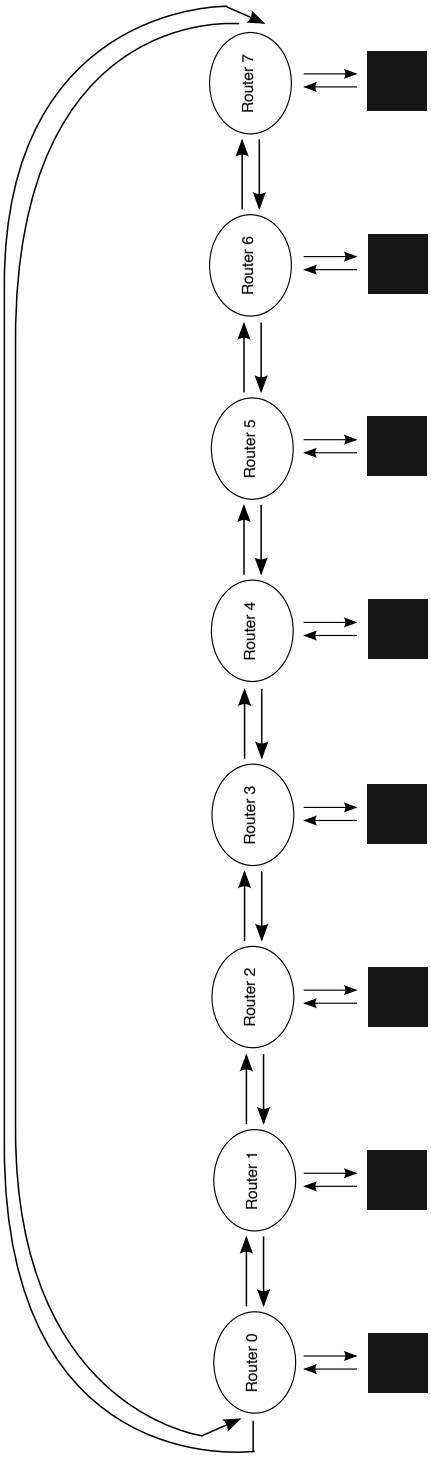

Figure 1: Ring Network Overview

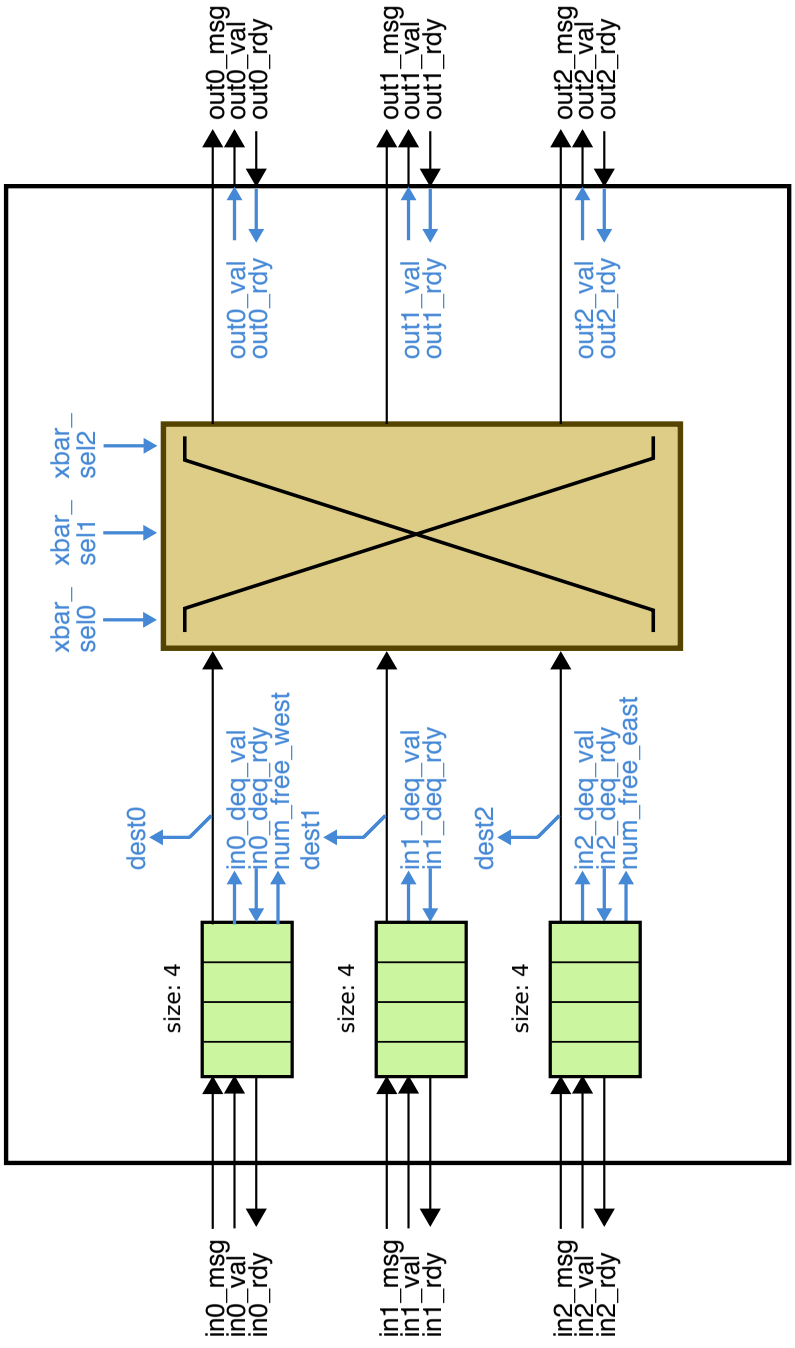

Figure 2: Baseline Router Datapath

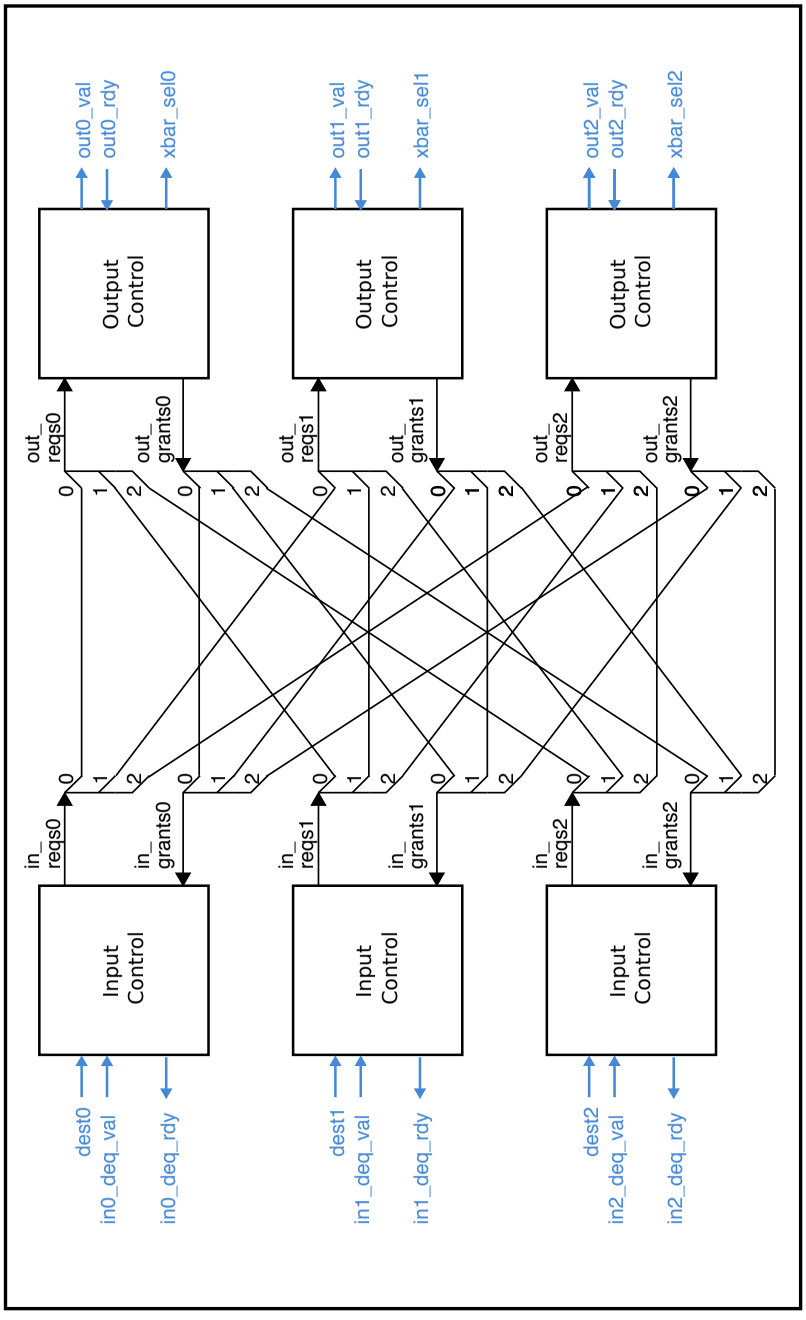

Figure 3: Incremental design: Router control unit without deadlock prevention

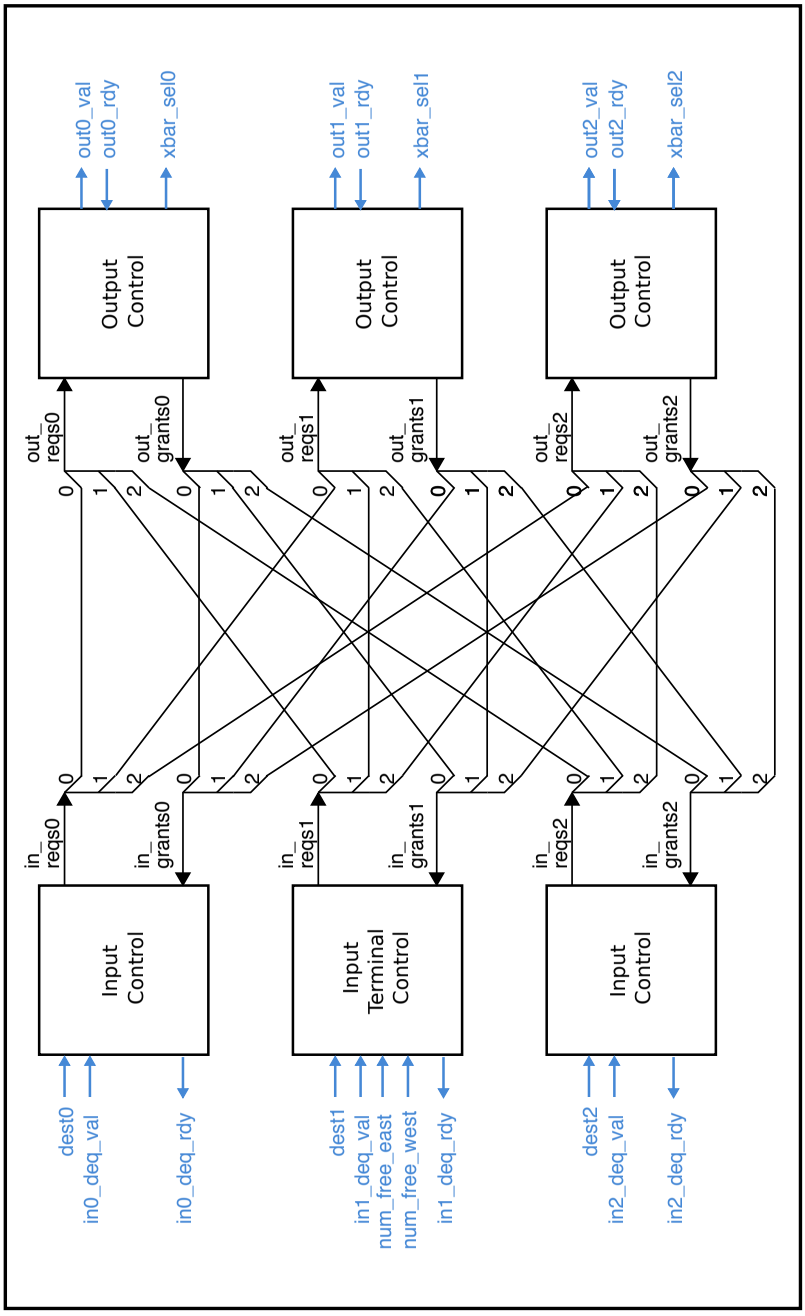

Figure 4: Baseline design: Router control unit with bubble flow control