Bluespec Model of a Network Linecard

6.884 Laboratory 3
March 4, 2005 - Version 20050304

In the previous lab assignments you were working with a Verilog hardware description. In the third lab assignment you will use Bluespec System Verilog to design a simple network *linecard*. A linecard processes incoming network packets and determines for each input packet the desired output port. There are usually several linecards in a real router and since each linecard might want to send packets to the same output port, there must be some *arbitration*. The linecards arbitrate for access to the crossbar, and when a linecard wins arbitration it is then able to send the packet across the crossbar to the desired output port. Figure 1 illustrates how a linecard fits into an abstract router. The primary objective of this lab is to introduce you to the Bluespec language syntax and semantics, as well as to give you some broader intuition on how to approach hardware design in Bluespec. The lab assignment is due via CVS at the start of class on Friday, March 11.

This lab assignment has three parts and each part involves building a different type of linecard. Although we will work with several linecard *implementations*, all of the linecards in this lab will have the same *interface*. Figure 2 shows the Bluespec linecard interface we will be using. For this lab, packets contain two fields: a 16 bit address and a 64 bit payload for a total fixed packet length of 80 bits. In Part A of the lab, you will work through a simple example linecard which uses source routing. In Part B, you will implement your own linecard which uses a lookup table for routing. Finally in Part C, you will implement your own lookup table which implements a longest prefix match algorithm with a circular pipeline.

Figure 3 illustrates the 6.884 ASIC toolflow we will be using for the third lab. You should already be familiar with the simulation, synthesis, and place+route toolflow from the previous lab assignments. In this lab we will be writing our model in Bluespec and then using the *Bluespec compiler* to generate Verilog source code. Our test infrastructure will be similar in spirit to the assembly toolchain used in the previous labs. A *packet generator* uses packet description files to create Verilog memory dumps of streams of input packets, linecard lookup tables, and reference output packet streams. We can then uses these tests to evaluate our linecard model. Although this lab assignment will focus only on simulation, you should still feel free to experiment with pushing your design all the way through synthesis and place+route. Be aware that Design Compiler can take quite a while to synthesize the larger lookup tables used in the later part of this assignment.

For this lab you will almost certainly need to consult the Bluespec documentation to clarify some of the topics which were only touched on briefly in lecture. The following list identifies what documentation is available in the course locker (`/mit/6.884/doc`).

- `bluespec-user-guide.pdf` - General bluespec user guide
- `bluespec-style-guide.pdf` - Examples, idioms, and patterns
- `bluespec-reference-guide.pdf` - Detailed language reference
- `bluespec-known-issues.pdf` - Some known bugs and problems with Bluespec
- `bluespec-timing-closure.pdf` - Information on how to improve performance
- `bluespec-examples` - Directory with several examples from Bluespec Inc.

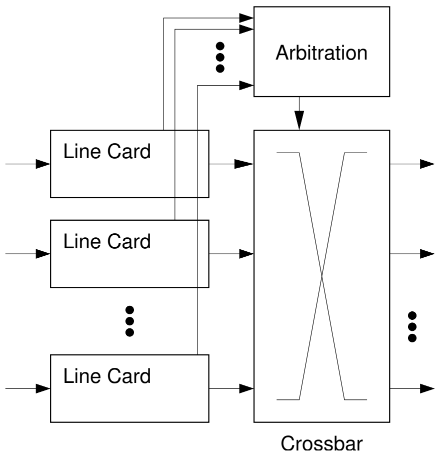

Figure 1: Abstract router

```
interface ILineCard;

  method Action putPacket( Packet packet );

  method ActionValue#(Packet) getPacket0();
  method ActionValue#(Packet) getPacket1();
  method ActionValue#(Packet) getPacket2();
  method ActionValue#(Packet) getPacket3();

  method Action loadLookupTable(
                   Bit#(12) entryIndex,
                   Bit#(12) entryData );

endinterface
```

Figure 2: Interface for a linecard

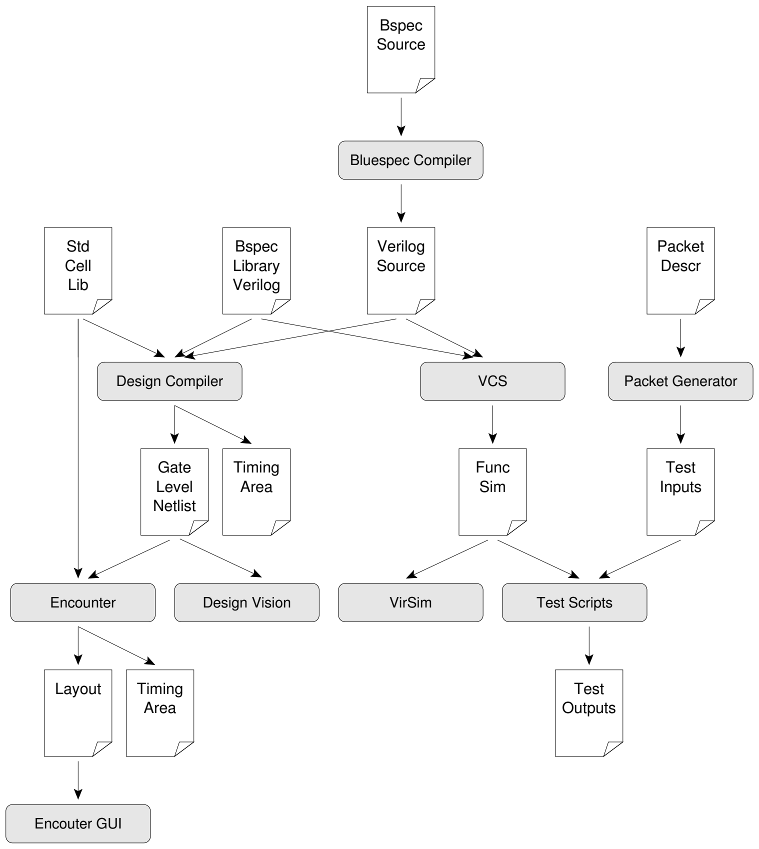

Figure 3: 6.884 Toolflow for Lab 3

Setup

For the third lab assignment you will be creating several Bluespec models of network linecards. You will need to create a new project called `linecard` in your toplevel CVS directory. You can use the following commands to checkout your toplevel CVS directory, untar the provided harness for the lab, and add the new project to CVS.

```
% cvs checkout 2005-spring/<username>
% cd 2005-spring/<username>
% tar -xzvf /mit/6.884/lab-harnesses/linecard-harness.tgz
% find linecard | xargs cvs add
```

Let's spend a moment taking a closer look at what is provided in the lab harness. You will notice that the lab harness is very similar to the one used in the previous lab assignments. As before, the `Makefile.in` and `configure.pl` scripts found in the project's root directory can be used to generate Makefiles for different designs.

The `tests` directory contains tests for each of the three parts, and the `config` directory contains makefile fragments and synthesis/place+route scripts. Each part has its own makefile fragment. If you plan on pushing your design through synthesis and place+route, you must ensure that the `bluespec_extra_synth_srcs` variable in your makefile fragments lists any Bluespec primitives and any Verilog generated by the Bluespec compiler (excluding the file containing the toplevel Bluespec module).

The `src` directory contains Bluespec and Verilog source code. You will notice that we have provided you with quite a bit of code. The following list briefly outlines what is provided; we will go into more detail about these files when they are required for a given part of the lab.

- `ILineCard.bsv` - Interface for linecards
- `ILookupTable.bsv` - Interface for lookup tables used in linecards (used in Parts B & C)
- `ILpmCompletionBuf.bsv` - Interface for a completion buffer (used in Part C)
- `ILpmRam.bsv` - Interface for a synchronous RAM (used in Part C)
- `PacketTypes.bsv` - Types which describe a packet
- `LpmTypes.bsv` - Types useful for implementing longest prefix match (used in Part C)
- `mkLineCardSimple.bsv` - Very simple linecard module (used in Part A)
- `mkLookupTableScycle.bsv` - Lookup table with single-cycle latency (used in Part B)
- `mkLookupTableMcycle.bsv` - Lookup table with multi-cycle latency (used in Part B)
- `mkLpmCompletionBuf.bsv` - Implementation of completion buffer (used in Part C)
- `mkLpmRam.bsv` - Implementation of synchronous RAM (used in Part C)
- `LineCardTH_nofiles.v` - Verilog test harness where packets are explicitly specified
- `LineCardTH.v` - Verilog test harness where packets are specified in files

You will notice some important file naming conventions which you should follow. Bluespec System Verilog files use the extension `.bsv` while standard Verilog files use the extension `.v`. It is more common to use `CamelCaseStyleNaming` as opposed to `underscore_style_naming` when working in Bluespec System Verilog. Interfaces begin with a capital `I` and modules use the prefix `mk`. It is usually cleaner to keep interfaces and modules in separate files. Do not confuse the `mk` prefix used for Bluespec modules with the `.mk` extension used for makefile fragments. It is often useful to group type definitions into a single file, and these files usually have a `Types` suffix. We use the `TH` suffix to denote files which are only involved in the test harness.

The Bluespec compiler is very resource intensive and runs particularly slowly on the machines in the 38-301 computer lab. We recommend that you make use of the more powerful Athena/Linux machines located in the public Athena clusters. More specifically there are IBM-S50 Athena/Linux machines in 38-370 (4 machines) and 37-318 (12 machines) which should be adequate for this assignment. As an example, compiling the design in Part A takes almost 5 minutes on the machines in 38-301 and takes only 16 seconds on the IBM-S50 machines.

Part A: Linecard with Source Routing

Source routing means that all routing decisions are made in advance by the host which is injecting the packet into the network. The host prepends routing information to the packet, and then the routers in the network simply use this information to quickly steer the packet to its destination. In the lab harness, we have provided you with a Bluespec module (called `mkLineCardSimple`) for a very basic linecard which assumes that the incoming packets are source routed. In this section, we will examine the `mkLineCardSimple` module and go through the steps involved in using the Bluespec compiler, creating a simulator executable, and testing the implementation.

Figure 4 shows the design for the `mkLineCardSimple` module which implements the `ILineCard` interface shown in Figure 2. Figure 5 shows the corresponding Bluespec source code found in `mkLineCardSimple.bsv`. At the top of the file we declare a package. Usually each of your Bluespec files contains a single package which has the same name as the filename. We use several `import` commands to make various interfaces and modules available for use in this package. See the Bluespec

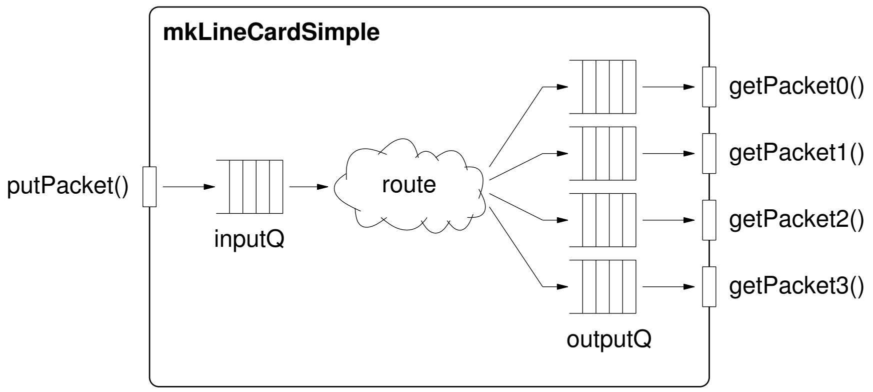

Figure 4: Design for `mkLineCardSimple` Bluespec module

Reference Guide for a list of the interfaces and modules available in the standard Bluespec library. The design for `mkLineCardSimple` includes a single input packet queue and four output packet queues (one for each output port). Note that the `mkFIFO` module implements a two element FIFO - if you want to set the size of the FIFO yourself you should use the `mkSizedFIFO` module (see the Bluespec Reference Guide for more information).

A single `route` rule moves packets from the input queue to the output queues. The rule pops a packet off the input queue and then uses the bottom two bits of the packet's address to choose the correct output queue. Whenever writing a rule in Bluespec it is important to remember how implicit conditions will influence that rule's firing. The `route` rule has implicit conditions from the `inputQ.first()` and `inputQ.deq()` method calls (i.e. the `route` rule will not fire if the input queue is empty) as well as implicit conditions from the four `outputQ.enq()` method calls (i.e. the `route` rule will not fire if *any* of the output queues are full).

The `mkLineCardSimple` module includes methods to insert and extract packets. Since this simple linecard does not use a lookup table the `loadLookupTable()` is implemented with a `noAction` which does nothing. We cannot just leave the `loadLookupTable()` method empty, since the method needs to return something of type `Action`.

We will now go through the steps required to build a simulator for this module. The following commands create a build directory and use the Bluespec compiler to generate Verilog from the Bluespec source code for `mkLineCardSimple.bsv`.

```
% pwd
.../2005-spring/<username>/linecard
% mkdir build
% cd build
% cp ../src/ILineCard.bsv ../src/PacketTypes.bsv ../src/mkLineCardSimple.bsv .
% bsc -u -verilog -g mkLineCardSimple mkLineCardSimple.bsv
```

Unfortunately, the Bluespec compiler does not currently support compiling in a directory which is different than the source directory. So we must first copy the source into the build directory. *Never edit the Bluespec files directly in the build directory - always edit them in the source directory and then copy them into the build directory.* Later we will use the makefiles to take care of this for us, but remember that we only want generated product in the build directory. After copying the source files, we run the Bluespec compiler (called `bsc`). The `-u` flag indicates that the compiler should manage dependency tracking - this means that the compiler will determine when it needs to recompile a module and when it is okay to use a previously compiled module. The `-verilog` flag indicates that we want to generate Verilog from the Bluespec code and the `-g` argument indicates which module the `bsc` should compile. Take a look at the generated Verilog in `mkLineCardSimple.v`. Notice that the Bluespec compiler has implemented the methods using `RDY` and `EN` signals and that the `route` rule has turned into combinational logic. The FIFOs are implemented with Bluespec Verilog primitives. Find the `WILL_FIRE_RL_route` signal in the generated Verilog. This signal will be high during clock cycles when the `route` rule fires and low during clock cycles when the `route` rule does not fire. We will use the `RDY`, `EN`, and `WILL_FIRE_RL` signals to help debug our Bluespec designs.

```
package mkLineCardSimple;
import PacketTypes::*;
import ILineCard::*;
import FIFO::*;

module mkLineCardSimple( ILineCard );

  //-- State -------------------------------------------------

  FIFO#(Packet) inputQ   <- mkFIFO(); // Input packet FIFO
  FIFO#(Packet) outputQ0 <- mkFIFO(); // Output packet FIFO 0
  FIFO#(Packet) outputQ1 <- mkFIFO(); // Output packet FIFO 1
  FIFO#(Packet) outputQ2 <- mkFIFO(); // Output packet FIFO 2
  FIFO#(Packet) outputQ3 <- mkFIFO(); // Output packet FIFO 3

  //-- Rules -------------------------------------------------

  rule route;

    Packet packet = inputQ.first();
    inputQ.deq();

    case ( packet.m_addr[1:0] )
      2'd0 : outputQ0.enq(packet);
      2'd1 : outputQ1.enq(packet);
      2'd2 : outputQ2.enq(packet);
      2'd3 : outputQ3.enq(packet);
    endcase

  endrule

  //-- Methods -----------------------------------------------

  method Action putPacket( Packet packet );
    inputQ.enq(packet);
  endmethod

  method ActionValue#(Packet) getPacket0();
    outputQ0.deq();
    return outputQ0.first();
  endmethod

  // ... other getPacket() implementations are analagous to getPacket0() ...

  method Action loadLookupTable( Bit#(12) entryIndex, Bit#(12) entryData );
    noAction;
  endmethod

endmodule
endpackage
```

Figure 5: Bluespec System Verilog source code for the `mkLineCardSimple` module

Although we have generated Verilog for the `mkLineCardSimple` module, we still need a test harness to drive the module. We have provided two Verilog test harnesses which wrap around modules that implement the `ILineCard` interface. The `LineCardTH_nofiles.v` test harness is analogous to `mips_simple_test.v` used in the previous labs since the harness explicitly writes the test packets in the verilog code. Take a look at the code and find where four test packets are written into the test packet buffer. The `LineCardTH.v` test harness is analogous to `mips_test_harness.v` since it loads an input packet stream from an external file. Let's use VCS to compile `LineCardTH_nofiles.v` and `mkLineCardSimple.v` into a functional simulator.

```
% pwd
.../2005-spring/<username>/linecard/build
% vcs +v2k -I +define+LINECARD_IMPL="mkLineCardSimple" \
      -y ${BLUESPECDIR}/Verilog +libext+.v \
      ../src/LineCardTH_nofiles.v mkLineCardSimple.v
% ./simv
```

The `+v2k` flag tells VCS to use Verilog-2001 and the `-I` flag tells VCS to compile a simulator with support for VCD plus. The `+define+` flag is used to tell the test harness which implementation of the `ILineCard` interface we would like to test. By using a macro definition we avoid requiring a different test harness for each linecard implementation. The `-y` lets VCS know the location of the source code corresponding to the built-in Bluespec Verilog modules. Finally, we list the Verilog source we wish to compile. After running VCS we will have a simulator with the default name `simv`. Running the simulator produces some debug output which shows four packets being inserted into the linecard and four packets exiting the linecard. Verify that the packets are going to the correct output ports. The test harness also outputs the total number of packets in the test, how many cycles the test took, and the *linecard rate* in packets per cycle. We will use the linecard rate as our performance metric for this lab. Ideally, we would like to sustain a linecard rate of 1 packet per cycle, but this may not always be possible due to architectural bottlenecks in our design. It is important to also note that the total cycle count includes some start up overhead and also includes the cycles when the packets are draining out of the linecard so for short tests these fixed overheads will be significant. That is why for just four packets the `mkLineCardSimple` module achieves 0.667 packets per cycle - to get a more representative performance number we would want to use a test with many more packets.

Let's take a look at the operation of the linecard using the VirSim waveform viewer (`vcs -RPP`). Figure 6 shows several signals in the design. The `EN_putPacket`, `RDY_putPacket`, and `putPacket_packet` signals can be used to observe what packets are going into the `putPacket()` method of the module. Similarly we can use the `EN` and `RDY` signals corresponding to the `getPacket()` methods to observe what packets are leaving the module. You can see the A, B, C, and D packets enter the linecard and then come out of the linecard at the appropriate output port.

At the bottom of the trace we see a `D_OUT` signal for the input queue and the four output queues. This is a very useful way to observe what data is coming out of which queues. Finally, at the very bottom of the trace you can see the `WILL_FIRE_RL_route` signal; this signal is high when the `route` rule is firing - notice on the trace that when this signal is high the packets are being routed from the input queue to the output queues. Using `EN`, `RDY`, `D_OUT`, and `WILL_FIRE_RL` signals will be critical when you start debugging your Bluespec designs.

Now we are going to use the `LineCardTH.v` test harness so that we can load input packet streams from files. Use the following commands to build a new functional simulator.

```
% pwd
.../2005-spring/<username>/linecard/build
% vcs +v2k -I +define+LINECARD_IMPL="mkLineCardSimple" \
      -y ${BLUESPECDIR}/Verilog +libext+.v \
      ../src/LineCardTH.v mkLineCardSimple.v
```

You must provide an input packet stream to the simulator using the `+file-in` command line option. The following commands use the `packet-generator.pl` script to create the appropriate Verilog memory dump file and then runs the simulator.

```
% pwd
.../2005-spring/<username>/linecard/build
% packet-generator.pl ../tests/simple_explicit_nodelay.pgen
% ./simv +file-in=simple_explicit_nodelay.pgen.in-vmh
```

In addition to just running the test we also want to verify that it is correct. The packet generator can produce a reference output file which contains the correct output packet streams. We can then use the `packet-verify.pl` script to compare the reference output to the test output. The following commands run the test and then verify that the results are correct.

```
% pwd
.../2005-spring/<username>/linecard/build
% packet-generator.pl ../tests/simple_explicit_nodelay.pgen
% ./simv +file-in=simple_explicit_nodelay.pgen.in-vmh \
         +file-out=simple_explicit_nodelay.pgen.res
% packet-verify.pl simple_explicit_nodelay.pgen.res
  [ PASSED ] simple_explicit_nodelay.pgen.res
```

The `packet-verify.pl` script takes into account that the ordering between output ports is not important, but that the order of packets from the same output port must be correct. The lab harness includes several tests in the `tests` directory. The infrastructure has support for specifying a delay with each packet - this means that each packet can be delayed a certain number of cycles before being inserted into the linecard. The tests with the `delayed` suffix include delays, while those with the `nodelay` suffix do not.

We have provided you with makefiles which help automate building and testing your Bluespec designs. The makefiles include `simv` and `run-tests` targets just as they did in the previous lab assignments. Try the following commands after deleting the build directory you have been using so far.

```
% pwd
.../2005-spring/<username>/linecard
% mkdir build
% cd build
% ../configure.pl ../config/mkLineCardSimpl.mk
% make simv
% make run-tests
```

You are now done with Part A of this lab assignment. In the next part you will be implementing a linecard which uses a lookup table to do the routing instead of relying on source routing. Delete your build directory and start with a fresh build directory for the next part.

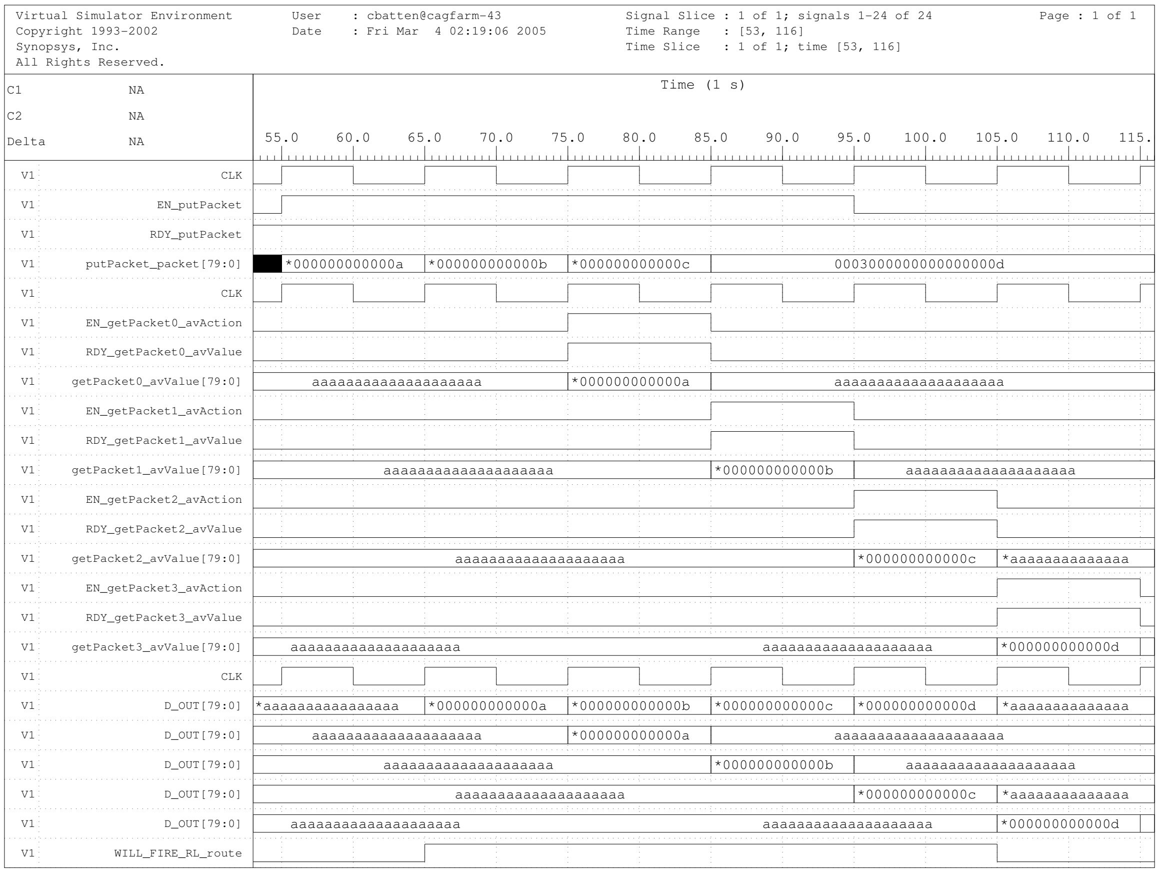

Figure 6: Waveform trace for `mkLineCardSimple` module using `LineCardTH_nofiles.v`

Part B: Linecard with Basic Lookup

In Part B of this lab assignment you will be writing your own linecard. We want to build a linecard which is suitable for networks where routing information is distributed amongst the various routers. Instead of using source routing, the linecard in Part B will use a lookup table to map packet addresses to output ports. Your linecard must implement the linecard interface shown in Figure 2. Because it will have the same interface, the test harness is exactly the same in Part B as it was in Part A.

Figure 7 shows the design for the `mkLineCardLookup` module you are to write. You will be making use of lookup tables through a `ILookupTable` interface. To use the lookup table, call `lookupRequest()` to start a lookup request, and then call `lookupResponse()` to retrieve the results. Note that the `ILookupTable` interface makes no guarantees about the number of cycles it takes to actually complete a request. The interface does, however, guarantee that responses are returned in the same order in which the corresponding requests were made. For Part B of the lab we have provided you with two basic lookup tables which implement the `ILookupTable` interface. The `mkLookupTableScycle` module is a single-cycle lookup table, and you should be able to achieve greater than 90% linecard rate with this module. A fixed latency lookup is rather unrealistic. In IPv4 there is an address space of 2^{32}, meaning that a flat lookup table would require a Gigabyte of storage. To help reduce the storage requirements, designers turn to sparse table descriptions. Although such descriptions are much smaller, they no longer have a fixed latency for a complete lookup. The `mkLookupTableMcycle` module is a multi-cycle lookup table which randomly delays responses by 0 to 4 cycles to emulate the behavior of lookups in a sparse table. This means that the rate of responses out of the lookup table is less than the rate at which you can make requests. This will be an architectural bottleneck and will make it difficult to achieve the full linecard rate.

For this part of the lab, you need to write `mkLineCardLookup.bsv` which contains the implementation for `mkLineCardLookup`. You should begin by using the `mkLookupTableScycle` lookup table.

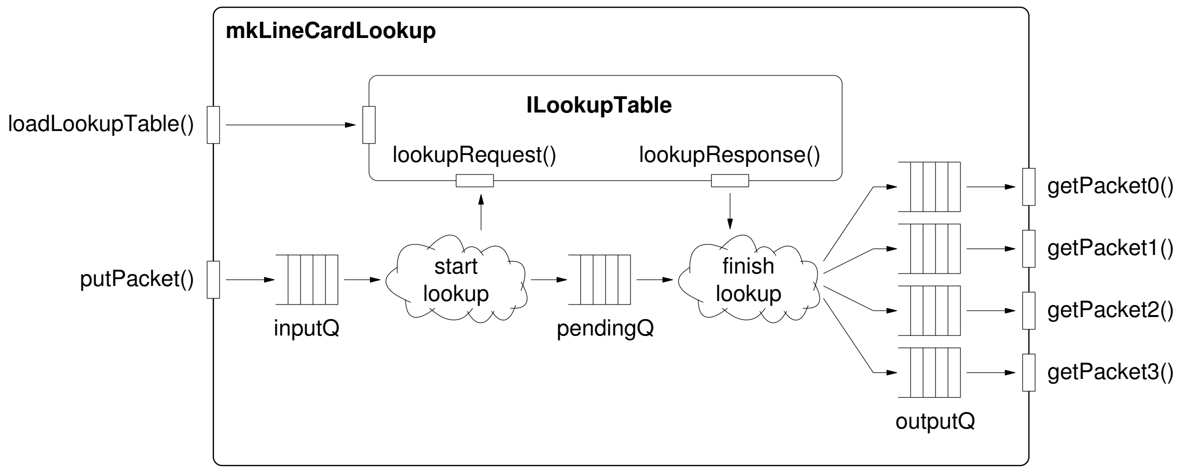

Figure 7: Design for `mkLineCardLookup` module

You should use the `mkLineCardLookup.mk` makefile fragment for building your design, and use `lookup_explicit_delayed.pgen` and `lookup_explicit_nodelay.pgen` to help test your design. You should be able to achieve a linecard rate of more than 90% when using `lookup_explicit_nodelay.pgen`. Once you have your design working, change the lookup table to `mkLookupTableMcycle` and observe the change in linecard rate. Although the throughput of the lookup table is limiting the overall throughput of your linecard, you can still increase performance by increasing the size of some of your FIFOs. If you are using the default `mkFIFO` for your `pendingQ` try using a `mkSizedFIFO` instead. Set the size of your `pendingQ` to better match the lookup table latency. A larger `pendingQ` means you can overlap more requests in the lookup table. Of course, larger queues mean more area. You should be able to achieve a linecard rate greater than 43% when using `lookup_explicit_nodelay.pgen` and `mkLookupTableMcycle`.

The following list shows which files you will be using in this part of the lab. You should check `mkLineCardLookup.bsv` into CVS along with any other changes you make.

- `ILineCard.bsv` - Interface for linecards
- `ILookupTable.bsv` - Interface for lookup tables used in linecards
- `PacketTypes.bsv` - Types which describe a packet
- `mkLineCardLookup.bsv` - *You write this ...*
- `mkLookupTableScycle.bsv` - Lookup table with a single-cycle latency
- `mkLookupTableMcycle.bsv` - Lookup table with a multi-cycle latency
- `LineCardTH_nofiles.v` - Verilog test harness where packets are explicitly specified
- `LineCardTH.v` - Verilog test harness where packets are specified in files

In this part of the assignment you implemented your own linecard and you used interfaces to try different lookup table implementations without changing your linecard code. In Part C, you will implement your own lookup table which performs a longest prefix match for the lookup. Because this new lookup table will also implement the `ILookupTable` interface you should not need to change your linecard code when you start using your new lookup table.

Part C: Linecard with Longest Prefix Match Lookup

In the previous section you designed a linecard which was able to handle responses from a lookup table which took a variable number of cycles. We supplied the lookup table for you. Now you will be building a realistic lookup table based around a single RAM. We will base this design off of the longest prefix match algorithm which is commonly used in IPv4 network routers. IPv4 uses a 32 bit address, but in this lab we will only be using 16 bit addresses.

The longest prefix match algorithm works as follows. When given a 16-bit address to lookup, the algorithm will take the 8 most significant bits. This will be zero extended to 11-bits and used as the effective address for a lookup into the RAM. The RAM will return a either a *pointer* or a *leaf*. If the RAM returns a pointer, you will add as an offset the next 4 most significant bits. You will then lookup this new pointer in the RAM. If the RAM again returns a pointer you will add the

```
function LuResponse doLookup( LuRequest req )
{
  Address addr1 = zeroExtend(req[15:8]);
  Result res1   = ram_lookup(addr1);

  if ( isLeaf(res1) )
    return(leafvalue(res1));

  // First request was a pointer
  Address addr2 = pointervalue(res1) + req[7:4];
  Result res2   = ram_lookup(addr2);

  if ( isLeaf(res2) )
    return(leafvalue(res2));

  // Second request was a pointer
  Address addr3 = pointervalue(res2) + req[3:0];
  Result res3   = ram_lookup(addr3);

  // This has to be a leaf node
  return(leafvalue(res3));
}
```

Figure 8: Pseudo-code for longest prefix match algorithm

last 4 bits to the new pointer and lookup this new value. If at any point, the RAM returns a leaf, you can simply return the corresponding value (this is the output port corresponding to the lookup request). Figure 8 shows pseudo-code for the longest prefix match algorithm.

We provide you a synchronous RAM (`ILpmRam` and `mkLpmRam`) which has a four cycle latency and is fully pipelined. We also provide you some types which you might find useful in your design (`LpmTypes`). A naive implementation might handle lookup requests one a time; do the first RAM read, then check to see if it is a pointer or a leaf, then do the second RAM read if necessary, and so on. Once the proper response is found, the naive implementation would return the response and move onto the next request. Unfortunately, this approach would be very slow due to the four cycle latency of the RAM. It would take a minimum of four cycles to service a lookup and in the worse case it could take 12 cycles. Note that in this naive implementation the RAM request port is idle much of the time.

Figure 9 shows a circular pipeline implementation which will better saturate the RAM request bandwidth by processing multiple lookups at the same time. The design works as follows. When `lookupRequest()` is called, a *LuRequest* is pushed onto the input queue. The `enter` rule then pops *LuRequests* off the input queue and obtains a *CbufToken* from the completion buffer. The completion buffer's job is to keep track of the order in which lookup requests were made. Remember, that the `ILookupTable` interface requires that responses be returned in FIFO order - so the *CbufToken* is basically an index into a FIFO. This enables you to write the FIFO out of order but only drain things out of the FIFO in order. We provide you with a completion buffer (`ILpmCompletionBuf`

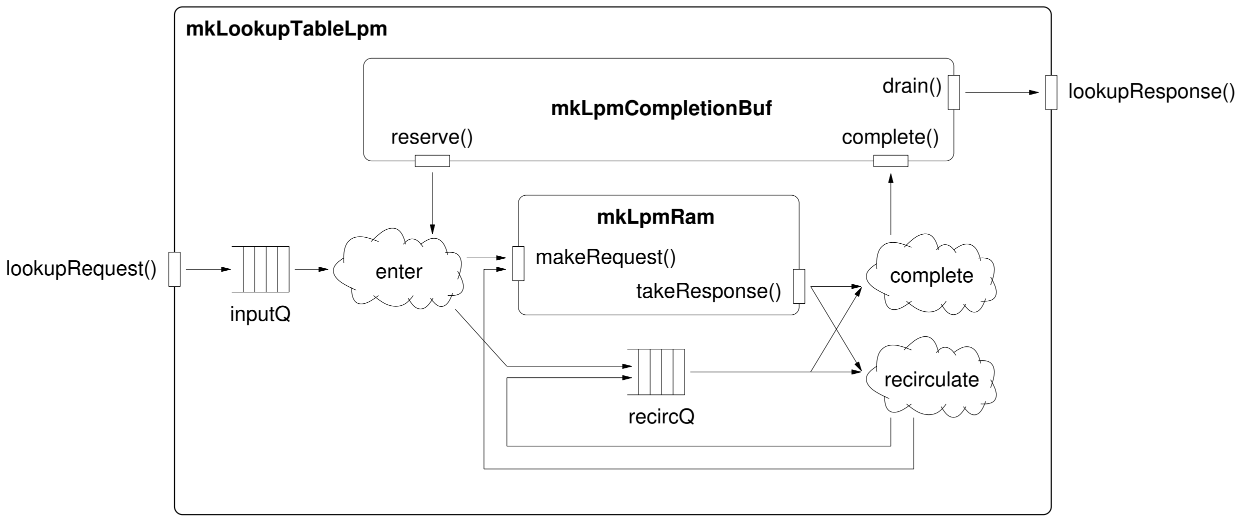

Figure 9: Design for `mkLineCardLpm` module (`loadLookupTable()` method not shown)

and `mkLpmCompletionBuf`) which you can use. After the `enter` rule has obtained a *CbufToken*, the rule creates a *PartialLookup* composed of the *CbufToken* and part of the *LuRquest*. The `enter` rule then makes a RAM read request and pushes the *PartialLookup* onto the recirculation queue.

The `complete` and `recirculate` rules are mutually exclusive (you should use explicit conditions to ensure this). The `complete` rule only fires when the RAM returns a leaf, and the `recirculate` rule only fires when the RAM returns a pointer. The `recirculate` rule gets the data coming back from the RAM and pops the next *PartialLookup* off the recirculation queue. The rule then uses the RAM data to calculate a new effective address and to make a new request to the RAM. The last thing the `recirculate` rule does is push an updated *PartialLookup* onto the recirculation queue. The `complete` rule also gets the data coming back from the RAM and pops the next *PartialLookup* off the recirculation queue. The difference is that this rule uses the *CbufTag* stored in the *PartialLookup* to write the response into the appropriate slot of the completion buffer. The `lookupResponse()` method then eventually retrieves responses in order from the completion buffer.

For Part C you need to write `mkLookupTableLpm.bsv`. You also need to write `mkLineCardLookupLpm.bsv`, which should be almost identical to `mkLineCardLookup.bsv` from Part B. The following list shows which files you will be using in this part of the lab. You should check `mkLineCardLookup.bsv` and `mkLookupTableLpm.bsv` into CVS along with any other changes you make. You should use the `mkLineCardLookupLpm.mk` makefile fragment to build your design.

- `ILineCard.bsv` - Interface for linecards
- `ILookupTable.bsv` - Interface for lookup tables used in linecards
- `ILpmCompletionBuf.bsv` - Interface for a completion buffer
- `ILpmRam.bsv` - Interface for a synchronous RAM
- `PacketTypes.bsv` - Types which describe a packet
- `LpmTypes.bsv` - Types useful for implementing longest prefix match

- `mkLookupTableLpm.bsv` - *You write this ...*
- `mkLineCardLpm.bsv` - *You write this ...*
- `mkLpmCompletionBuf.bsv` - Implementation of completion buffer
- `mkLpmRam.bsv` - Implementation of synchronous RAM
- `LineCardTH_nofiles.v` - Verilog test harness where packets are explicitly specified
- `LineCardTH.v` - Verilog test harness where packets are specified in files

Writing the `loadTable()` method depends on how the test harness is implemented, so please use something similar to the code shown below to implement this method.

```
method Action loadTable( Bit#(12) entryIndex, Bit#(12) entryData );

  RamResponse resp;
  if ( entryData[11] == 0 )
    resp = tagged Leaf(entryData[1:0]);
  else
    resp = tagged Pointer(entryData[10:0]);

  RamRequest req = entryIndex[10:0];
  ram.write( req, resp );

endmethod
```

There are some very important questions you should ask yourself when implementing your design. How many lookups can be active in your design at once? How large should your completion buffer be to accommodate this? What happens if the recirculation queue is full? Will this cause reduced performance or deadlock? How can you prevent such an occurrence?

Once your design is functional, you should see if you can achieve a linecard rate greater than 90% on the `lpm_explicit_fullthruput.pgen` test input. This test has no delays and the lookup table only has leaves, so your design should be able to saturate the lookup table request bandwidth.

Deliverables

For this lab assignment you are to complete Parts A, B, and C. There is nothing to be turned in for Part A. You should check into CVS any Bluespec source code as well as any new tests you wrote for Parts B and C. At a bare minimum you will need to add the following three files.

- `mkLineCardLookup.bsv` - From Part B
- `mkLineCardLpm.bsv` - From Part C
- `mkLookupTableLpm.bsv` - From Part C

It would be useful if you use a CVS tag once you have committed all of your changes. This will make it easier for you to go back and reexamine your design. To create a CVS tag for the final project use the following commands.

1. Make sure you add all the appropriate files and commit your changes

```
% pwd
.../2005-spring/<username>/linecard
% cvs update
% cvs commit
```

2. Create a new junk directory somewhere

```
% cd
% mkdir junk
```

3. Checkout your project in the junk directory

```
% cd junk
% cvs checkout 2005-spring/<username>/linecard
```

4. Verify that your project builds correctly

```
% cd 2005-spring/<username>/linecard
% mkdir build-partB
% cd build-partB
% ../configure.pl ../config/mkLineCardLookup.mk
% make run-tests
% cd ..
% mkdir build-partC
% cd build-partC
% ../configure.pl ../config/mkLineCardLookupLpm.mk
% make run-tests
```

5. Add the CVS tag

```
% cd ..
% pwd
.../junk/2005-spring/<username>/linecard
% cvs tag lab3-final .
```

6. Delete the junk directory

   ```
   % cd
   % rm -rf junk
   ```

I will be doing the following to test your design, so please try building your project in a clean working directory to make sure these steps work.

```
% mkdir temp
% cd temp
% cvs checkout -r lab3-final 2005-spring/<username>/linecard
% cd 2005-spring/<username>/linecard
% mkdir build-partB
% cd build-partB
% ../configure.pl ../config/mkLineCardLookup.mk
% make run-tests
% cd ..
% mkdir build-partC
% cd build-partC
% ../configure.pl ../config/mkLineCardLookupLpm.mk
% make run-tests
```