**Lab 1**

You will build up a right shifter from gate primitives. First, you will build a simple 1-bit multiplexer. Next, you will write a simple polymorphic multiplexer using for loops. Using the gate-level multiplexer function, you will then construct a combinational right-shifter. Finally, we will add a simple gate-level modification to the right shifter to support the arithmetic right shift operation. This allows us to use the right shifter for both logical and arithmetic operations.

**Files Provided:**

RightShifter.bsv, RightShifterTypes.bsv, Gates.bsv, and RightShifterTest.bsv.

### Multiplexers

The first step in constructing our right shifter is to build a basic multiplexer from gates. Let's examine the code found in the provided RightShifter.bsv file.

These lines import definitions contained in the labs\_common's constituent .bsv files, as well as the interface for the combinational right shifter whose implementation you will need to expand.

function Bit#(1) multiplexer1HighLevel(Bit#(1) sel, Bit#(1) a, Bit#(1) b);

This begins a definition of a new function called multiplexer1HighLevel. This multiplexer function takes several arguments which will be used in defining the behavior of the multiplexer. This multiplexer operates on single bit values, the concrete type Bit#(1). Later we will learn how to implement polymorphic functions, which can handle arguments of any width.

We call this multiplexer function "HighLevel" because it uses C-like constructs in its definition. Simple code, such as the multiplexer can be defined at the high level without implementation penalty. However, because hardware compilation is a difficult, multi-dimensional problem, tools are limited in the kinds of optimizations that they can do. As we shall see later with the high-level right shifter, high-level constructs can sometimes result in inefficient hardware. As a result, even complicated designs like processors may be implemented at the gate-level (or even transistor-level) to achieve maximum performance.

return (sel == 0)?a:b;

endfunction

The return statement, which constitutes the entire function, takes two input and selects between them using sel. The endfunction keyword completes the definition of our multiplexer function. You should now be able to compile and simulate the module. After simulating verify the output is correct.

**Part 1:** Using the and, or, and not gates provided by the labs\_common module, re-implement multiplexer1 in RightShifter.bsv. How many gates are needed?

### Static Elaboration

The data path width of the SMIPS processor is 32 bits, so we will need multiplexers that are larger than a single bit. However, writing the code to manually instantiate 32 single-bit multiplexers to form a 32-bit multiplexer would be tedious. Fortunately, Bluespec provides constructs for powerful static elaboration which we can use to make writing the code easier. Static elaboration refers to the process by which the Bluespec compiler evaluates expressions at compile time, using the results to generate the hardware rather than generating hardware to evaluate the expressions dynamically. Static elaboration can be used to express extremely exible designs in only a few lines of code.

In Bluespec we can use bracket notation ([]) to index individual bits in a wider Bit type, for example bitVector[1](http://asim.csail.mit.edu/redmine/projects/mit6s078/wiki/Lab_1#fn1). We can use a for-loop to copy many lines of code which have the same form. For example, to aggregate the multiplexer1 function to form a larger multiplexer, we could:

1 function Bit#(8) multiplexer8(Bit#(1) sel, Bit#(8) a, Bit#(8) b);  
2 Bit#(8) aggregate = 0;  
3 **for** (Integer i = 0; i < 8; i = i+1)  
4 begin  
5 aggregate[i] = multiplexer1(sel, a[i], b[i]);  
6 end  
7 **return** aggregate;  
8 endfunction

The Bluespec compiler, during its static elaboration phase, will replace this for-loop with its fully unrolled version.

1 aggregate[0] = multiplexer1(sel, a[0], b[0]);  
2 aggregate[1] = multiplexer1(sel, a[1], b[1]);  
3 aggregate[2] = multiplexer1(sel, a[2], b[2]);  
4...  
5 aggregate[7] = multiplexer1(sel, a[7], b[7]);

**Part 2:** complete the implementation of multiplexer32 module in MuxBasedrightShifter.bsv using for-loops.

### Polymorphism and Higher-order Constructors

So far, we have implemented two versions of the multiplexer function, but it is easy to imagine needing an n-bit multiplexer. It would be nice if we did not have to completely re-implement the multiplexer whenever we want to use a different width. Using the for-loops introduced in the previous section, our multiplexer code is already somewhat parametric because we use a constant size and the same type throughout. We can do better by changing the size and type to use type parameters rather than concrete types and changing our code to use those type parameters. Our new multiplexer code looks like:

1 **typedef** 8 N;  
 2 function Bit#(N) multiplexerN(Bit#(1) sel, Bit#(N) a, Bit#(N) b);  
 3 Bit#(N) aggregate = 0;  
 4 **for** (Integer i = 0; i < valueof(N); i = i+1)  
 5 begin  
 6 aggregate[i] = multiplexer1(sel, a[i], b[i]);  
 7 end  
 8 **return** aggregate;  
 9 endfunction

The typedef gives us the ability to change the size of our multiplexer at will. The valueOf function introduces a small subtlety in our code: N is not an Integer but a numeric type and must be converted to an Integer before being used in an expression. Even though it is improved, our implementation is still missing some flexibility. All instantiations of the multiplexer must have the same type, and we still have to produce new code each time we want a new multiplexer. However, in Bluespec, we can further parameterize the module to allow different instantiations to have instantiation-specific parameters. This sort of module is polymorphic, the implementation of the hardware changes automatically based on compile time configuration. Polymorphism is the essence of design-space exploration in Bluespec.

Our multiplexer functions are already almost polymorphic, but we have only used it with concrete types. Polymorphic modules either use type variables or take functions or values as compile time arguments, which are used during static elaboration to construct the correct module. The simplest polymorphic module, mkReg is already present in our code:

1 Reg#(Int) anInt <- mkReg(0);  
2 Reg#(Bool) aBool <- mkReg(True);

Here, two different types of registers are created from the same module, mkReg. The values 0 and True are being provided as an initializer to the register, permitting us to instantiate a register with any well-typed initial value. The question, then, is how to write a multiplexer module accommodating this level of parameterization. Given our previous code, this is simple:

1// Not needed  
2// typedef 8 N;  
3function Bit#(n) multiplexerN(Bit#(1) sel, Bit#(n) a, Bit#(n) b);  
4....

The variable n represents the width of the multiplexer, replacing the concrete value N. As usual, Bluespec variables are lower case and definitions are upper case. Otherwise, n and N function in the same way.

**Part 3**: Complete the definition of the multiplexerN function.

### Building a Right Shifter

We will now use the multiplexers that we implemented in the previous section to build a logical right shifter. As you know, a logical right shifter shifts bits to the right and pads the most significant bits with 0s. For example, if we have a word consisting of 0xABCD, a right right shift by 4 bits would result in 0x0ABC.

Although there are many ways to build this shifter, we want you to use a logarithmic number of multiplexers. At each stage, we will shift over twice as many bits as the previous stage, based on the control value.

Notice that the right shifter is declared a module, as opposed to a function. This is a subtle, but important distinction. In Bluespec, functions are inlined by the compiler automatically, while modules must be explicitly instantiated using the '<-' notation. If we made the right shifter a function, using it for arithmetic and logical shift would instantiate two shifters. One purpose of this lab was to build shift algorithms that share as much logic as possible, so we make the shifter a module.

**Part 4:** Complete an implementation of the 32-bit logical right shifter. Use exactly five multiplexerNs. The method should return the shifted value.

**Part 5:** In the logical right shifter, we always shift in 0 to the high-order bits. The arithmetic right shift preserves sign, so we need to examine the two's complement sign (high-order) bit and the shift mode to fill in the correct bits. ***Using for-loops and a multiplexer***, add this functionality in your right shifter from part 4. Use no more than one additional multiplexer over the five you used in part 4.

**Part 6:** Test your design using TestRightShifter.bsv. The two test provided are very simple. Add your own tests to the code. Your tests will also be used to check your peers’ work.

## Discussion Questions

1. How many gates does your one-bit multiplexer use? The 32-bit multiplexer? Write down a formula for an N-bit multiplexer.
2. We wrote a polymorphic function implementing an N-bit multiplexer. Explain how to write a polymorphic version of the left shifter.
3. One purpose of this lab was to demonstrate a microarchitectural optimization. How many gates did we save by combining the logical and arithmetic right shifts?
4. Our right shifter handles right shifts only. However, with a small extension, it can handle left shifts as well. Draw a microarchitecture for this kind of combined shifter. How much hardware do we save?

## Submitting Your Solution

You should hand in two files: RightShifter.bsv and TestRightShifter.bsv. Your tests should output their results in exactly the same for as the tests in TestRightShifter.bsv. You should not change Gates.bsv or RightShifterTypes.bsv. Your tests will be used to test solutions to other students in the class.

All components of your solution should be included in a single tar file of a single directory named Lab1\_\_<last name student1>\_<last name student2>. That directory should include (i) a README file containing your names, (ii) your answers to the questions posed in the Parts and the Discussion Questions in a doc or a .pdf file, named Lab1\_<last name student1>\_<last name student2>.doc or .pdf, and (iii) your RightShifter.bsv and TestRightShifter.bsv.