CS220
Introduction to Computer Organisation
Lab 1

Abhinav Agarwal*

1 Introduction

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

The left shift (`<<`) and right shift (`>>`) operations are employed in computer programs to manipulate bits and to simplify multiplication and division in some special cases. Shifting is considered a simple operation because shift has a relatively small and fast implementation in hardware; shift can typically be implemented in a single processor cycle, while multiplication and division take multiple cycles.

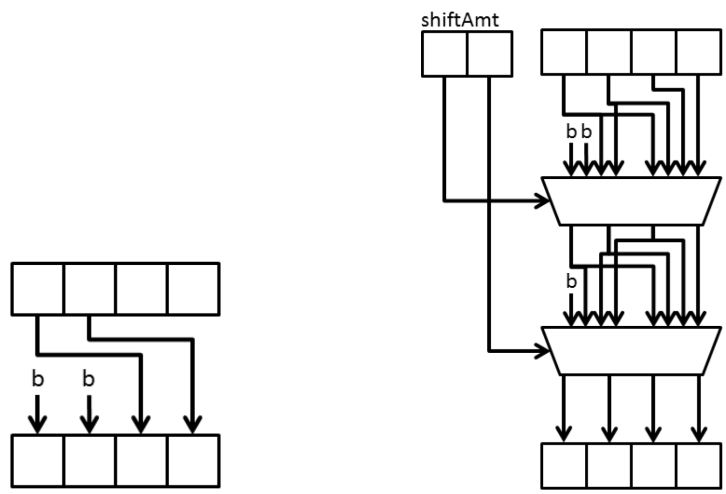
Shifts are inexpensive in hardware because their functional implementation involves wiring, rather than transistors. For example, a shift by a constant value is implemented with wires, as shown in Figure 1a. Variable Shifting is more complicated, but still efficient.

The barrel shifter (shown in Figure 1b) is more general than the wired shifter, but requires more hardware. The shift amount (`shiftAmt`) determines how far the input is shifted.

The microarchitecture of the barrel shifter is a logarithmic circuit for variable length shifting. The key observation in this microarchitecture is that any shift can be done by applying a sequence of smaller shifts. For example, a left shift by three can be decomposed into a shift by one and a shift by two. At each level of the shifter, the shifter conditionally shifts the data, using a multiplexer to select the data for the next stage.

In this lab we will implement the right shift operation, for which there are two possible meanings, logical and arithmetic which differ in preservation of the two's complement sign bit.

*Adapted by Amey Karkare for CS220



(a) Wired Shifting by a Constant Value (b) Variable Shifting by a Register Value

Figure 1: Different Types of Shifters

1.1 Deliverables

The lab test-bench (provided) includes tests for the aforementioned multiplexer and the combinational barrel-shifter, both of which you will implement yourselves.

2 Building a Barrel Shifter in Bluespec

2.1 Multiplexers

The first step in constructing or barrel shifter is to build a basic multiplexer from gates. Let's first examine Multiplexer.bsv.

```
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
```

This begins a definition of a new function called multiplexer1. 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.

This function 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 dificult, multi-dimensional problem, tools are limited in the kinds of optimizations that they can do. As we shall see later with the high-level barrel 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 be able to compile the module.

Exercise 1: Using the and, or, and not gates, re-implement the function `multiplexer1` in Multiplexer.bsv. (The required functions, called `and1`, `or1` and `not1`, respectively, are provided)

2.2 Static Elaboration

For typical multiprocessors, we need multiplexers that are larger than a single bit (for e.g., 32 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. Static elaboration can be used to express extremely flexible 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]`. 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 write:

```
function Bit#(32) multiplexer32(Bit#(1) sel, Bit#(32) a, Bit#(32) b);
  Bit#(32) aggregate;
  for(Integer i = 0; i < 32; i = i + 1)
    aggregate[i] = multiplexer1(sel, a[i], b[i]);
  return aggregate;
endfunction
```

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

```
aggregate[0] = multiplexer1(sel, a[0], b[0]);
aggregate[1] = multiplexer1(sel, a[1], b[1]);
aggregate[2] = multiplexer1(sel, a[2], b[2]);
...
aggregate[31] = multiplexer1(sel, a[31], b[31]);
```

Exercise 2: Complete the implementation of the function `multiplexer32` in Multiplexer.bsv using for-loops.

Check the correctness of the code by running the multiplexer testbench:

```
$ make mul
$ ./simMul
```

2.3 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 giving a name (`N`) to the size of the multiplexer using `typedef`. Our new multiplexer code looks like:

```
typedef 32 N;
function Bit#(N) multiplexerN(Bit#(1) sel, Bit#(N) a, Bit#(N) b);
  Bit#(N) aggregate;
  for(Integer i = 0; i < valueOf(N); i = i + 1)
    aggregate[i] = multiplexer1(sel, a[i], b[i]);
  return aggregate;
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.

The truly polymorphic multiplexer will be as follows:

```
//typedef 32 N; // Not needed
function Bit#(n) multiplexer_n(Bit#(1) sel, Bit#(n) a, Bit#(n) b);
```

The variable `n` represents the width of the multiplexer, replacing the concrete value `N` (=32). In Bluespec *type variables* (`n`) start with a lower case whereas concrete types (`N`) start with an upper case.

Exercise 3: Complete the definition of the function `multiplexer_n`.

Exercise 4: Re-implement `multiplexer32` to only use the polymorphic function `multiplexer_n`.

Make sure that the re-implementation allows the test benches to test your new implementation without modification.

2.4 Building a Barrel Shifter

We will now use the multiplexers that we implemented in the previous section to build a logical barrel shifter. To build this shifter we need 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, as shown in Figure 1b.

The implementation for the barrel-shifters can be found in BarrelShifterRight.bsv. Notice that all the barrel-shifters are declared as modules, as opposed to as functions. 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 barrel shifter a function, using it in multiple locations would instantiate multiple barrel shifters. One purpose of this lab was to build shift algorithms that share as much logic as possible, so we make the barrel shifter a module.

There are three modules inside BarrelShifterRight.bsv, `mkBarrelShifterRight`, `mkBarrelShifterRightLogical`, and `mkBarrelShifterRightArithmetic`. All three modules contain a method named `rightShift`, which implements the right shift. Notice that `rightShift` in `mkBarrelShifterRight` takes three arguments while `rightShift` in the others take only two. The third argument, `shiftValue`, specifies whether 0 or 1 is shifted in. In the logical barrel 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. The `mkBarrelShifterRightLogical` and `mkBarrelShifterRightArithmetic` should instantiate the basic `mkBarrelShifterRight`, and should supply the appropriate arguments to the `shiftRight` in `mkBarrelShifterRight` to implement the logical and arithmetic shifters, respectively.

Exercise 5: Complete an implementation of the 32-bit barrel shifter in `mkBarrelShifterRight`. Use exactly five 32-bit multiplexers in a for-loop.

Exercise 6 Complete an implementation of the 32-bit logical shifter in `mkBarrelShifterRightLogical`, using `mkBarrelShifterRight` that you have implemented already. Check the correctness of the code by running the logical shifter testbench:

```
$ make rl
$ ./simRl
```

Exercise 7: Complete an implementation of the 32-bit arithmetic shifter in `mkBarrelShifterRightArithmetic`.

Check the correctness of the code by running the arithmetic shifter testbench:

```
$ make ra
$ ./simRa
```

3 Exercise Questions

[Repeated here for ease of reference]

Exercise 1: Using the and, or, and not gates, re-implement the function `multiplexer1` in Multiplexer.bsv. (The required functions, called `and1`, `or1` and `not1`, respectively, are provided.)

Exercise 2: Complete the implementation of the function `multiplexer32` in Multiplexer.bsv using for-loops. Check the correctness of the code by running the multiplexer testbench:

```
$ make mul
$ ./simMul
```

Exercise 3: Complete the definition of the function `multiplexer_n`.

Exercise 4: Reimplement `multiplexer32` to only use the polymorphic function `multiplexer_n`. Make sure that the re-implementation allows the test benches to test your new implementation without modification.

Exercise 5: Complete an implementation of the 32-bit barrel shifter in `mkBarrelShifterRight`. Use exactly five 32-bit multiplexers in a for-loop.

Exercise 6 Complete an implementation of the 32-bit logical shifter in `mkBarrelShifterRightLogical`, using `mkBarrelShifterRight` that you have implemented already. Check the correctness of the code by running the logical shifter testbench:

```
$ make rl
$ ./simRl
```

Exercise 7: Complete an implementation of the 32-bit arithmetic shifter in `mkBarrelShifterRightArithmetic`. Check the correctness of the code by running the arithmetic shifter testbench:

```
$ make ra
$ ./simRa
```

4 Discussion Questions

NOTE: Please provide answers in a text file named "discussions.txt".

1. How many gates does your one-bit multiplexer use? The 32-bit multiplexer? Write down a formula for an N-bit multiplexer. (2 Points)