# Lab 1: Combinational Circuits

SNU Computer Architecture Short Course
January 9, 2011

## 1 Introduction

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 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 1(a). Variable shifting is more complicated, but still efficient. Figure 1(b) shows the microarchitecture of the barrel shifter, a logarithmic circuit for variable length shifting. The key observation in this microarcitecture 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 series of labs 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. Figure 2 shows the high-level code describing an ALU that we saw in lecture. In this code, both right shift implementations are implemented with independent operators. As a result, downstream tools will implement two pieces of shift hardware. However, just as the adder we designed in lecture could be reused, with minor modification, for subtraction, so too can the logical right shift hardware be reused to do arithmetic right shift. To obtain efficient hardware, we must describe the low-level, shared implementation.

### 1.1 Lab Organization

The lab begins by describing how to use the Bluespec Workstation to compile and simulate your design. Bluespec code for a trivial barrel shifter implemented using the >> primitive function is provided as a reference.

You will build up a barrel shifter from gate primitives. First, you will build a simple 1-bit multiplexer. Next, you will write a simple polymorphic multiplexer using for loops.

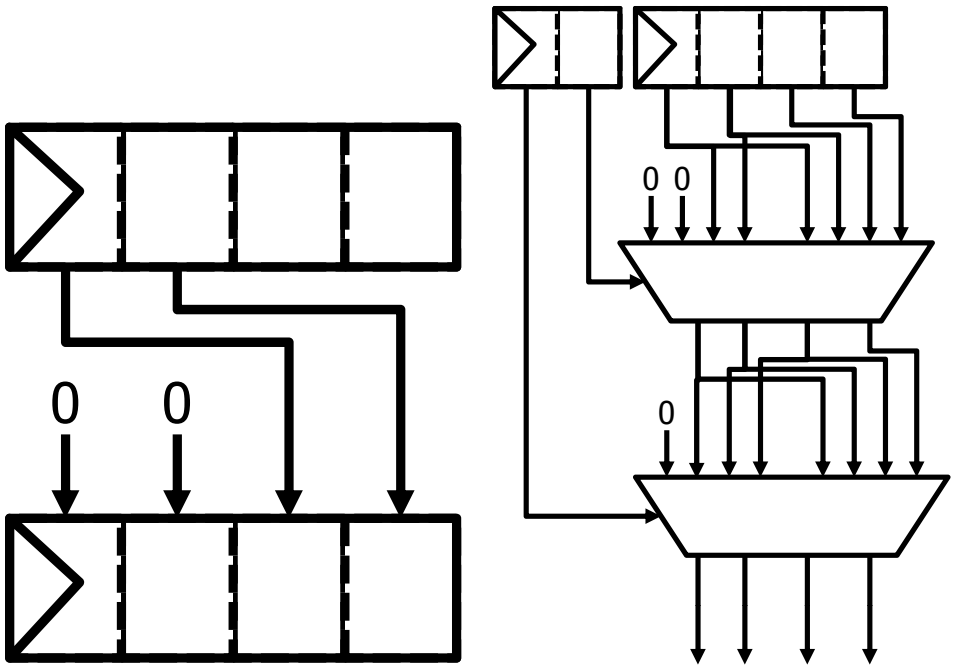
Using the gate-level multiplexer function, you will then construct a combinational barrel-shifter. Finally, we will add a simple gate-level modification to the barrel shifter to support the arithmetic right shift operation. This allows us to use the barrel shifter for both logical and arithmetic operations.

## 2 Getting Started

All of the SNU Computer Architecture Short Course laboratory assignments should be completed on the class virtual machine. We will be using Virtualbox 3.2, which can be downloaded at `http://www.virtualbox.org`. More recent versions of Virtualbox are also acceptable. All the tools and commands described in this and subsequent labs can be run directly from command line in the virtual machine.

We will be using subversion for laboratory handouts. To get the harness for this lab, execute:

```
% svn co https://asim.csail.mit.edu/svn/snuw12/lab1
```




(a) Constant Shift (b) Barrel Shift

Figure 1: Two different four-bit shift implementations. The barrel shifter is more general, but requires more hardware. The two registers on the left of the barrel shifter determine how far to shift.

```
Data res = case(di.func)
    Add   : (src1 + src2);
    Sub   : (src1 - src2);
    And   : (src1 & src2);
    Or    : (src1 | src2);
    Xor   : (src1 ^ src2);
    Nor   : ~(src1 | src2);
    Slt   : slt(src1, src2);
    Sltu  : sltu(src1, src2);
    LShift: (src1 << src2[4:0]);
    RShift: (src1 >> src2[4:0]);
    Sra   : signedShiftRight(src1, src2[4:0]);
endcase;
```

Figure 2: Bluespec code for ALU. Note that each shift function will infer separate hardware.

# 3 Compiling and Simulating with the Bluespec Workbench

The lab1 project contains Bluespec code we will be using for this lab. The directory layout is shown in figure 3.

`common/BarrelShifterTypes.bsv` defines the Bluespec interface we will be using for our barrel shifter in this series of labs. This file contains a polymorphic interface `Multiplexer`, and the `BarrelShifter` interface, which contains a single method `doShift`.

`common/Gates.bsv` defines the primitive gates that we will be using to build our barrel shifter. And, or, not and xor gates are defined.

```
lab1/
    common/
        BarrelShifterTypes.bsv
        TestDriver.bsv
        Gates.bsv
    barrelshifter/
        BarrelShifter.bsv
        barrelshifter.bspec
```

Figure 3: Lab1 code directory structure.

`common/TestDriver.bsv` contains Bluespec code to drive the barrel shifter module in simulation.

`barrelshifter/BarrelShifter.bsv` defines an initial implementation of a barrel shifter using the '>>' and '>>' primitives.

We can simulate the hardware described by the Bluespec code to get an idea of how it works without having to deal with the complications of real hardware.

- Navigate to the `barrelshifter/` directory and start up Bluespec Workstation.

```
lab1% cd barrelshifter
lab1/barrelshifter% bluespec barrelshifter.bspec&
```

- The Bluespec Workstation GUI will appear. Select `Build->Compile` This compiles the Bluespec source code.
- Select `Build->Link`. This creates the executable `out` which can be used to simulate the hardware described by the top level module `mkTestdriver`. The executable file can then be run like any program. The test driver code provided will exercise your code with a number of different inputs, and compare the result against a reference implementation.
- To simulate the barrel shifter select `Simulate` from the `Build` menu in the Workstation.

If everything worked correctly, you should see `PASSED`.

We can also direct the Bluespec Workstation to generate Verilog code, which can then be used by a whole slew of tools to build the hardware as an ASIC or programed into an FPGA. Under the `Project` menu select `Options`. In the window that popped up go to the `Compile` tab and select the compile to Verilog option. If you compile the project now, it will create the file `fir/bscdir/mkTestDriver.v`. Remember to change this back to `bluesim` when you go to simulate the design.

# 4 Building an Efficient Barrel Shifter in Bluespec

## 4.1 Multiplexers

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

```
import BarrelShifterTypes::*;
import Gates::*;
```

This line import definitions from other Bluespec files. BarrelShifterTypes and Gates are defined in the `common/` directory and contain function and interface definitions.

```
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 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 now be able to compile and simulate the module. After simulating verify the output is correct.

**Exercise:** Using the `and`, `or`, and `not` gates, re-implement `multiplexer1`. How many gates are needed?

## 4.2 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 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

```
function Bit#(8) multiplexer8(Bit#(1) sel, Bit#(8) a, Bit#(8) b);

    Bit#(8) aggregate = 0;
    for (Integer i = 0; i < 8; i = i+1)
    begin
        aggregate[i] = multiplexer1(sel, a[i], b[i]);
    end

    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[7] = multiplexer1(sel, a[7], b[7]);
```

**Exercise:** Write a `mkMultiplexer32` module in BarrelShifter.bsv using for-loops.

## 4.3 Polymorphism and Higher-order Constructors (Optional)

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:

```
typedef 8 N;

function Bit#(N) multiplexerN(Bit#(1) sel, Bit#(N) a, Bit#(N) b);

    Bit#(N) aggregate = 0;
    for (Integer i = 0; i < valueof(N); i = i+1)
    begin
        aggregate[i] = multiplexer1(sel, a[i], b[i]);
    end

    return aggregate;

endmodule
```

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:

```
Reg#(Int) anInt <- mkReg(0);
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 filters, this is simple:

```
// Not needed
// typedef 8 N;

function Bit#(n) multiplexerN(Bit#(1) sel, Bit#(n) a, Bit#(n) b);

    ....
```

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.

**Exercise:** Complete the definition of the `mkMultiplexerN` function. Verify that this function is correct by replacing the original definition of `mkMultiplexer32`:

```
let multiplexer32 = multiplexerN;
```

This redefinition allows the test benches to test your new implementation without modification.

### 4.4 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 1(b).

Notice that barrel-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 barrel shifter a function, using it for arithmetic and logical shift would instantiate two 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.

**Exercise:** Complete an implementation of the 32-bit barrel shifter. Use exactly five 32-bit multiplexers.

**Exercise:** 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. Using for-loops and a multiplexer, implement this functionality in your barrel shifter. Use no more than one additional multiplexer.

## 5 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. (Optional) We wrote a polymorphic function implementing an N-bit multiplexer. Explain how to write a polymorphic version of the barrel 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 barrel 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 barrel shifter. How much hardware do we save?