Lab 2: Barrel Shifters

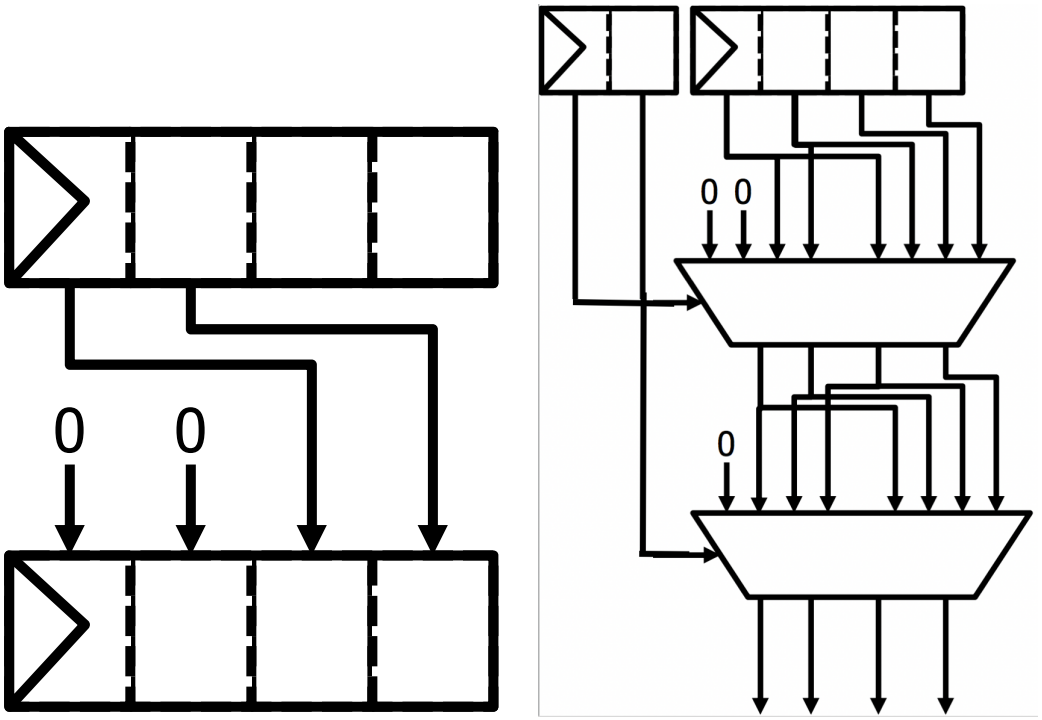
Acknowledgement: This lab is based on the course material provided by Prof. Arvind of MIT.

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 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 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. At each level, 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. In this code, both right shift implementations are implemented with independent operators. We will also enable left shift operation by reversing the input and result of a right shifting barrel shifter.



(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.

2 Lab Assignment

We only need multiplexers for building a barrel shifter. In this assignment, we will build a logic right shifter, an arithmetic right shifter and an logic shifter that can shift left or right based on the control bit. You are not allowed to include any << or >> operators in your code.

We have given the implementations of 32-bit multiplexer in `BarrelShifter.bsv`, when ever a 32-bit multiplexer is needed, please directly call this function.

```
function Bit#(32) multiplexer32(Bit#(1) sel, Bit#(32) a, Bit#(32) b);

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

endfunction
```

2.1 Building a Logic Right Shifter

We will now use the multiplexers 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). The input of the shifter is the number you want to shifter `operand`, and the number of bits you want to shift `shamt`. You can regard `shamt` as an 5-bit unsigned integer. When you do logic right shift, you fill the higher order bits is zero.

Exercise 1

In `BarrelShifter.bsv`, complete the code for logical barrel right shifter, as shown in the following:

```
function Bit#(32) logicalBarrelShifter (Bit#(32) operand, Bit#(5) shamt);
    ...
endfunction
```

You are only allowed to use `multiplexer32` for the implementations and you are not allowed to use << or >>.

Please test the your implementations using

```
$ make logicshifter
$ ./logicshifter
```

If your implementation is correct, `PASSED LOGICAL` will be displayed.

2.2 Building an Arithmetic Right Shifter

In this section, you will build a arithmetic barrel shifter. Different from logic right shifter, which fills the left bits with zero, in arithmetic barrel shifter, we fill the left bits with the sign bit (which is the leftmost bit) of the original number.

Exercise 2

In `BarrelShifter.bsv`, complete the code for arithmetic barrel right shifter, as shown in the following:

```
function Bit#(32) arithmeticBarrelShifter(Bit#(32) operand, Bit#(5) shamt);
    ...
endfunction
```

Please test the your implementations using

```
$ make arithmeticshifter
$ ./arithmeticshifter
```

If your implementation is correct, `PASSED ARITHMETIC` will be displayed.

2.3 Building a Logic Left and Right Shifter

At first, left shift may seem a little complicated – after all we must move bits in opposite directions. At first, it

might seem that this would require a second barrel shifter. However, we get around the introduction of a second

barrel shifter by adding multiplexers to reverse the operands. By reversing the operand bits and doing a logical

right shift, and reversing the result, we can effect a logical left shift. This requires the introduction of two more

multiplexers, but this is somewhat cheaper than introducing a second, full-sized barrel shifter.

Exercise 3

In `BarrelShifter.bsv`, complete the code for logic barrel shifter which can shift left or right based on the control bit `shiftLeft`, as shown in the following:

```
function Bit#(32) logicLeftRightBarrelShifter(Bit#(1) shiftLeft, Bit#(32) operand, Bit#(5) shamt);
    ...
endfunction
```

You MUST use `logicalBarrelShifter` that you implement in Exercise 1, you MUST also use `multiplexer32` in this assignment.

Please test the your implementations using

```
$ make leftshifter
$ ./leftshifter
```

If your implementation is correct, `PASSED LEFT AND RIGHT` will be displayed.