Due date: Thursday September 20 11:59:59pm EST.

Getting started: To create your initial Lab 1 repository, please visit the repository creation page at https://6004.mit.edu/web/fall18/user/labs/lab1. Once your Lab 1 repository is created, you can clone it into your VM by running:

```bash
git clone git@github.mit.edu:6004-fall18/labs-lab1-{YourMITUsername}.git lab1
```

Turning in the lab: To turn in this lab, commit and push the changes you made to your git repository. After pushing, check the course website (https://6004.mit.edu, Builds tab) to verify that your submission was correctly pushed and passes all the tests. If you finish the lab in time but forget to push, you will incur the standard penalties for late submissions.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting within 6 days of the lab’s due date. See the course website for lab hours.

The purpose of this lab is to use logic gates to create these combinational circuits, so this lab has restrictions on what subset of Bluespec you can use in your designs. Although Bluespec has `and (&)`, `or (|)`, `xor (^)`, and `not (~)` operators and higher-level constructs (such as `if` statements, `for` loops, `+`, `-`, and `==`), you should not use them in this lab. Using any of these will cause a compiler error. If you read the error, it should tell you in all caps what symbol you used.

For this lab, unless otherwise stated, you are only allowed to use constants, the base functions shown in Figure 1 below (which are defined in Common.bsv), and any functions you define yourself using these functions.

To pass the lab you must complete and PASS all of the exercises.

```ml
function Bit#(1) and1(Bit#(1) a, Bit#(1) b);
function Bit#(1) or1(Bit#(1) a, Bit#(1) b);
function Bit#(1) xor1(Bit#(1) a, Bit#(1) b);
function Bit#(1) not1(Bit#(1) a);
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
```

Figure 1: The set of allowed base functions for this lab.

In this lab you will design a variety of combinational digital circuits.

Building and Testing Your Circuits

You will build your code with `make`. If you just type `make` it will build all of the exercises. You can instead pass in a target so that only one of the exercises will be built like

```bash
make <target>
```

This will then create a program `<target>` that you can run which simulates the circuit. It will run through a set of test cases printing out each input and whether or not it fails to match the expected output. If it passes all the tests it will print out PASSED at the end.

- To build all the targets, run `make all`
- To build and test everything, run `make test`
1 Bit Scan Reverse

A useful circuit is \texttt{bit\_scan\_reverse}. This determines the index of the first non-zero bit scanned from the largest index. Incidentally, this is the same as taking the log\(_2\) of the input. \texttt{bit\_scan\_reverse}(0) is undefined, so we will not check your output in this case. For example,

\begin{verbatim}
bit_scan_reverse(4'b1000) = 3
bit_scan_reverse(4'b0110) = 2
bit_scan_reverse(4'b0001) = 0
\end{verbatim}

\texttt{function Bit#(2) bit\_scan\_reverse(Bit#(4) a);}

\begin{verbatim}
Bit#(2) ret = 0;
// your code
return ret;
endfunction
\end{verbatim}

Figure 2: Skeleton code for \texttt{bit\_scan\_reverse}.

Exercise 1 (10 points): In BitScanReverse.bsv, implement the \texttt{bit\_scan\_reverse} function for a 4-bit input using only the basic gates provided in Common.bsv. Figure 2 shows the skeleton code for this function. \textit{Hint:} If you are confused on how to construct this function out of logic gates, we recommend writing out a truth table for the Bit-scan Reverse function and trying to deduce how the 4-bit input value relates to the 2-bit output. Then, try drawing the circuit with logic gates before writing any Bluespec code.

You can build and test your circuit by running:

\texttt{make bit\_scan\_reverse\_test}

\texttt{./bit\_scan\_reverse\_test}

2 Power of 2

The \texttt{is\_power\_of\_2} circuit determines whether its input is a power of 2. A binary number is a power of 2 if it has exactly a single 1.

Exercise 2 (10 points): In PowerOf2.bsv, implement \texttt{is\_power\_of\_2} for a 4-bit input.

You can build and test your circuit by running:

\texttt{make is\_power\_of\_2\_test}

\texttt{./is\_power\_of\_2\_test}

3 \(\log_2\) of Powers of 2

We now want to combine the previous two circuits. We want a circuit \texttt{log\_of\_power\_of\_2} that will output 0 if the input is not a power of 2, and \(\log_2\) of the input if it is a power of 2.

Exercise 3 (10 points): In LogOfPowerOf2.bsv, implement \texttt{log\_of\_power\_of\_2} for a 4-bit input using your previous two circuits as well as any of the basic gates.

You can build and test your circuit by running:

\texttt{make log\_of\_power\_of\_2\_test}

\texttt{./log\_of\_power\_of\_2\_test}

Discussion Question 1 (10 points): How many gates of each kind do each of the three circuits above use? Include all the gates used by each circuit. Could you build any of these circuits with fewer gates?
4 Equality Testing

We often need to know whether two numbers are the same. Write a function `equal` that returns 1 if two 8-bit numbers are the same, and 0 otherwise.

**Exercise 4 (10 points):** Implement the `equal` function in Equal.bsv using only the basic gates.

You can build and test your circuit by running:

```
make equal_test
./equal_test
```

5 Vector Equality Testing

We are often interested in comparing groups on values for equality. For example, given two vectors of four 8-bit (1-byte) values, we would like to produce a circuit that returns a 4-bit result, where the \(i^{th}\) bit is 1 if the \(i^{th}\) elements of the vectors are equal, and 0 otherwise. Design a `vector_equal` circuit that performs this function. This circuit should take two 32-bit (4-byte) inputs, with each byte representing a different element of the 4-element vector. For example,

\[
\begin{align*}
\text{vector_equal}(0xaabbccdd, 0xaal1ccdd) &= 0b1011 \\
\text{vector_equal}(0xaabbccdd, 0x0011ccdd) &= 0b0011 \\
\text{vector_equal}(0xaabbcc00, 0x0011ccdd) &= 0b0010
\end{align*}
\]

**Exercise 5 (10 points):** Implement the `vector_equal` function using your `equal` circuit as well as the basic gates in VectorEqual.bsv.

You can build and test your circuit by running:

```
make vector_equal_test
./vector_equal_test
```

6 Seven-Segment Decoder

A seven-segment display shows numbers by lighting up a subset of the 7 LED segments. Each segment is traditionally lettered from \(a\) to \(g\) as shown in Figure 3, and each segment has its own input signal for enabling and disabling the segment. Figure 4 shows how the numbers should be displayed on the seven-segment display.

You are in charge of designing a digital logic controller to take in a binary number between 0 and 9 inclusive, and output the 7-bit control signals to drive a seven-segment display. The output control signal takes the form `abcdefgh` where \(a\) is the most significant bit of the output (bit 6) and \(g\) is the least significant bit of the output (bit 0). When the input signal is outside the range of legal inputs, you should make the display show an \(E\) indicating an error.

```
```
function Bit#(7) seven_segment_decoder(Bit#(4) input_binary_number);
    Bit#(7) ret = 7'b1001111; // value for "E"
    return ret;
endfunction

Exercise 6 (10 points): Implement the seven_segment_decoder function in SevenSegment.bsv. For this exercise only, in addition to the basic gates you can also use case statements in your Bluespec implementation.

You can build and test your circuit by running:

make seven_segment_test
./seven_segment_test

7 Population Count

Next, you are in charge of implementing a population count circuit, which counts the number of 1’s in the input. Your population count circuit should take in a 4-bit input and return an output representing the number of 1’s in the input.

Discussion Question 2 (10 points): Why is the return type 3 bits wide?

Exercise 7 (10 points): Implement the population_count function in PopulationCount.bsv using only the basic gates.

You can build and test your circuit by running:

make population_count_test
./population_count_test

8 Comparator

As your final task, you are in charge of implementing a 4-bit comparator. Write a function is_geq that takes in two 4-bit binary unsigned numbers a and b and returns 1 if a is greater than or equal to b, and 0 otherwise.
Exercise 8 (10 points): Implement the `is_geq` function in IsEq.bsv using only the basic gates.

You can build and test your circuit by running:

```
make is_geq_test
./is_geq_test
```