Due date: Thursday October 3 11:59:59pm EST.

Points: This lab is worth 10 points (out of 200 points in 6.004).

Getting started: To create your initial Lab 3 repository, please visit the repository creation page at `https://6004.mit.edu/web/fall19/user/labs/lab3`. Once your Lab 3 repository is created, you can clone it into Athena by running:

```
git clone git@github.mit.edu:6004-fall19/labs-lab3-{YourMITUsername}.git lab3
```

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`, Labs/Didit 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 by Wednesday October 9. See the course website for lab hours.

1 Introduction

Combinational logic is a basic building block of digital systems. It can be used to build many computation structures, such as adders, multiplexers, decoders, and the Arithmetic-Logic Unit (ALU) at the heart of every CPU. Combinational logic takes Boolean inputs and produces Boolean outputs, where each output is a function of the current inputs only. In other words, combinational logic has no state or memory.

In this lab, you will build various combinational circuits using Minispec. Minispec is a hardware description language based on Bluespec SystemVerilog (BSV or just Bluespec). Minispec is specialized to design, simulate, and implement digital *hardware* systems, rather than software. A separate language is needed because software programming languages are designed to be translated to sequences of assembly instructions, rather than to a parallel circuit built with gates and wires.

To ensure that you have a good grasp of Minispec's basic syntax and that you describe circuits using gates directly, this lab has restrictions on the subset of Minispec that you can use in your designs. Although Minispec has `and (&)`, `or (|)`, `xor (^)`, and `not (~)` operators and higher-level constructs (such as `if` statements, `for` loops, `+`, `-`, and `==`), you are **not** allowed to use them in this lab. Using any of these will cause a compiler error. The error should tell you what forbidden symbol you used in ALL CAPS.

For this lab, unless otherwise stated, you are only allowed to use **constants**, **the base functions shown below** (which are defined in `Common.ms`), and **any functions you define yourself using these functions**. Finally, you **should NOT** modify **`Common.ms`** since it will be overwritten by the auto-grading script.

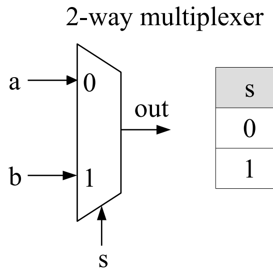
To pass the lab you must complete and PASS all of the exercises.

```
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);
```

2 Getting Started with Minispec

We recommend that you partially complete the Minispec combinational logic tutorial before jumping into the exercises. We have created this tutorial to quickly introduce all the concepts and syntax you will need to design combinational circuits in 6.004. The tutorial is structured as an interactive Jupyter notebook.

a	b	a and b	a or b	a xor b	not(a)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



s	out
0	a
1	b

Figure 1: Truth tables for the allowed base functions.

The tutorial covers both this and the next lab. For most of the exercises in this lab, **Sections 1 through 4 of the tutorial cover everything you need to know**. Exercise 6 and the discussion questions at the end of this lab use concepts described in Sections 6 and 7 of the tutorial, but you can read those later.

Introductory Example: 2-way Multiplexer

A multiplexer, also called a *mux*, is a combinational logic circuit designed to choose a single output signal from multiple input signals based on a select signal. Let's first take a look at a multiplexer implementation using basic logic gates. We saw in lecture that a 2-way multiplexer can be built using 1 Inverter, 2 AND gates, and 1 OR gate as shown in Figure 2. In Boolean algebra representation, $out = a\bar{s} + bs$ such that an output is `b` if a select signal `s` is 1 and `a` if `s` is 0. In Minispec, a 2-way multiplexer that takes two 1-bit inputs and produces a single 1-bit output can be implemented as follows using the given base functions.

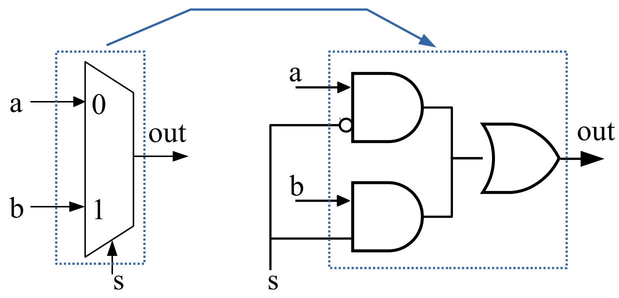


Figure 2: 2-way multiplexer implementation using basic gates.

```
// A 2-way multiplexer in Minispec that takes two 1-bit inputs
// and produces a single 1-bit output
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    return or1(and1(a, not1(sel)), and1(b, sel));
endfunction
```

Alternatively, we can also implement a multiplexer in Minispec using a conditional expression, as shown below. Note that for this lab, you are not allowed to use conditional expressions. (Section 6 of the Minispec combinational logic tutorial describes conditional expressions.)

```
// A 2-way multiplexer in Minispec that takes two 1-bit inputs
// and produces a single 1-bit output
function Bit#(1) multiplexer1(Bit#(1) sel, Bit#(1) a, Bit#(1) b);
    return (sel == 0)? a: b;
endfunction

# Python equivalent code
def multiplexer1(sel, a, b)
    return a if sel == 0 else b
```

You will often have to design functions that take inputs whose sizes are more than 1-bit wide. For these cases, it is very useful to compose these larger functions out of simpler functions with smaller inputs. A simple example of this is building a 2-bit multiplexer using two 1-bit multiplexers, as shown below.

```
// A 2-way multiplexer which takes two 2-bit inputs
// and produces a single 2-bit output
function Bit#(2) multiplexer2(Bit#(1) sel, Bit#(2) a, Bit#(2) b);
    return {multiplexer1(sel,a[1],b[1]), multiplexer1(sel,a[0],b[0])};
endfunction
```

First of all, `function Bit#(2)` tells us that this function returns a 2-bit output. The input arguments in `multiplexer2(Bit#(1) sel, Bit#(2) a, Bit#(2) b)` tell us that the function takes a 1-bit value, `sel`, and two 2-bit inputs, `a` and `b`. Note that bit concatenation is used in the above example to return a 2-bit output signal by concatenating two 1-bit multiplexers using `{ , }`. As a result, a 2-bit multiplexer takes two 2-bit inputs `a` and `b` and produces a 2-bit output based on a 1-bit selection value `sel`. For more details on bit concatentaion, selection, and working with multi-bit values, see Section 4 of the Minispec combinational logic tutorial.

Building and Testing Your Circuits

During grading your code will be built with `make`. You can also use `make` to test any of your implementations before submitting for grading. 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

```
make {target}
```

This will then create a program `{target}` that you can run which simulates the circuit. You can type `make` with a space afterward and then double-tap the Tab key to see a list of targets for make. Running the created program 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`

Finally, you may want to test a particular function that the staff-provided tests do not cover. For example, you may have built your function out of smaller functions; if the whole function is not working properly, you'd want to test the smaller functions first. You can use the `ms eval` command for this purpose. `ms eval` takes two arguments: the file where the function is, and the call to the function in quotes. For example, to test `bit_scan_reverse` from the next exercise with argument `4'b0101`, run:

```
ms eval CombDigitalSystems.ms "bit_scan_reverse(4'b0101)"
```

3 Bit Scan Reverse

`bit_scan_reverse` determines the index of the first non-zero bit scanned from the largest index. This is equivalent to taking the $\log_2$ of the input and rounding it to the next smaller integer (i.e. $\lfloor \log_2(\texttt{Input}) \rfloor$). Note that `bit_scan_reverse(0)` is undefined, so we will not check your output in this case. For example,

- `bit_scan_reverse(4'b1000) = 3`
- `bit_scan_reverse(4'b0110) = 2`
- `bit_scan_reverse(4'b0001) = 0`

```
function Bit#(2) bit_scan_reverse(Bit#(4) a);
    Bit#(2) ret = 0;
    // your code
return ret;
```

Figure 3: Skeleton code for **bit_scan_reverse**.

Exercise 1 (10%): In **CombDigitalSystems.ms**, implement the **bit_scan_reverse** function for a 4-bit input using only the basic gates provided in Common.ms. Figure 3 shows the skeleton code for this function.

Hint: 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 Minispec code.

You can build and test your circuit by running:

```
make bit_scan_reverse_test
./bit_scan_reverse_test
```

Once you're done implementing `bit_scan_reverse`, count up the number of each type of gate your implementation uses, and record it in the commented section above the function in the skeleton code.

4 Power of 2

The `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%): In **CombDigitalSystems.ms**, implement `is_power_of_2` for a 4-bit input, which returns 1 if an input is a power of 2 or 0 otherwise.

You can build and test your circuit by running:

```
make is_power_of_2_test
./is_power_of_2_test
```

Once you're done implementing `is_power_of_2`, count up the number of each type of gate your implementation uses, and record it in the commented section above the function in the skeleton code.

5 $\log_2$ of Powers of 2

We now want to combine the previous two circuits. We want a circuit `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 (15%): In **CombDigitalSystems.ms**, implement `log_of_power_of_2` for a 4-bit input using your previous two circuits as well as any of the basic gates.

Hint: Think about how you can simplify the implementation by importing `bit_scan_reverse` and `is_power_of_2` functions and expanding the given 1-bit multiplexer function to a 2-bit multiplexer.

You can build and test your circuit by running:

```
make log_of_power_of_2_test
./log_of_power_of_2_test
```

Once you're done implementing `log_of_power_of_2`, count up the number of each type of gate your implementation uses, and record it in the commented section above the function in the skeleton code.

6 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%): Implement the `equal` function in **`CombDigitalSystems.ms`** using only the basic gates.

You can build and test your circuit by running:

```
make equal_test
./equal_test
```

7 Vector Equality Testing

We are often interested in comparing groups of 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,

- `vector_equal(0xaabbccdd, 0xaa11ccdd) = 0b1011`
- `vector_equal(0xaabbccdd, 0x0011ccdd) = 0b0011`
- `vector_equal(0xaabbcc00, 0x0011ccdd) = 0b0010`

Exercise 5 (10%): Implement the `vector_equal` function using your `equal` circuit as well as the basic gates in **`CombDigitalSystems.ms`**.

You can build and test your circuit by running:

```
make vector_equal_test
./vector_equal_test
```

8 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 4, and each segment has its own input signal for enabling and disabling the segment. Figure 5 shows how the numbers should be displayed on the seven-segment display.

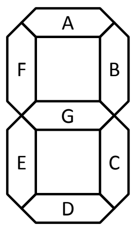


Figure 4: Seven-segment display.

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 `abcdefg` 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.

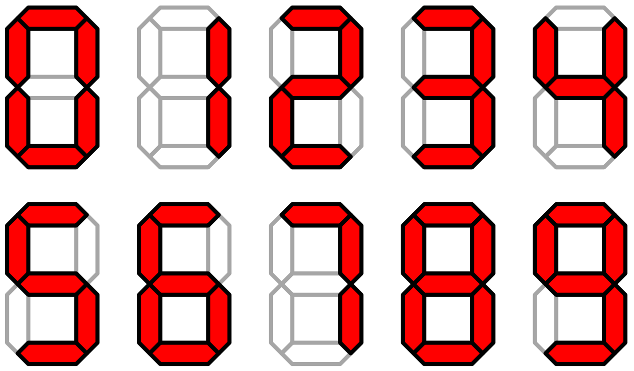


Figure 5: Digit representations on the seven-segment display.

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

Figure 6: Skeleton code for `seven_segment_decoder`.

Exercise 6 (15%): Implement the `seven_segment_decoder` function in **`CombDigitalSystems`**.ms. **For this exercise only**, in addition to the basic gates in Figure 1 you can also use a `case` expression in your Minispec implementation. *Case expressions are described in Section 6 of the Minispec combinational logic tutorial.*

You can build and test your circuit by running:

```
make seven_segment_test
./seven_segment_test
```

9 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.

Exercise 7 (15%): Implement the `population_count` function in **`CombDigitalSystems.ms`** using only the basic gates.

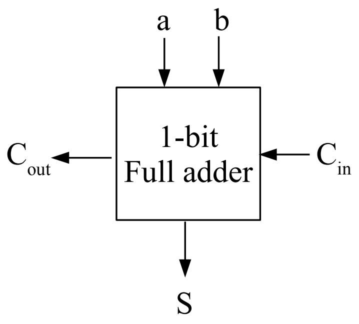
Hint: You may want to use a full adder function to count the number of 1's in the input. An example 1-bit full adder and its truth table are shown in Figure 7.

You can build and test your circuit by running:

```
make population_count_test
./population_count_test
```

10 Comparator

As your final design 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.



a	b	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 7: 1-bit full adder and its truth table.

Exercise 8 (15%): Implement the `is_geq` function in **`CombDigitalSystems.ms`** using only the basic gates.

Hint: You may want to implement a function that performs a bit-wise comparison and then extend it to perform a 4-bit comparison.

You can build and test your circuit by running:

```
make is_geq_test
./is_geq_test
```

11 Circuit Performance Analysis

To build good digital systems, it is crucial to analyze the performance of your implementations and explore implementation trade-offs, so you can find ways to further improve your design. To do this we will use `synth`, a synthesis tool that automatically translates your Minispec designs into optimized gate-level implementations using basic logic gates.

We recommend you complete Section 7 of the Minispec combinational logic tutorial, which covers the basics of `synth`.

Synthesis tools require three types of input: the circuit to be synthesized, the standard cell library of gates to use to implement the circuit, and the optimization objective (area, delay, power, or some combination of them). `synth` comes with several standard cell libraries, all of them derived from the open-source FreePDK45 library (`https://research.ece.ncsu.edu/eda/freepdk/freepdk45/`). This is representative of 45 nm silicon fabrication technology introduced in 2008. By default, `synth` optimizes for delay, but this is controllable by specifying a target delay. `synth` will try to reduce the circuit's area as long as its delay is under the target. Under the hood, `synth` uses the Minispec compiler and the Yosys open synthesis suite (`http://www.clifford.at/yosys/`).

In this part of the lab, we will first learn how to use `synth` by synthesizing and analyzing several circuits. These exercises will build your intuition of delay and area implementation trade-offs and of the optimizations `synth` can perform, which will be very useful when it comes to designing more complex digital systems. Specifically, we will analyze two circuits from the ones you have built in this lab: `bit_scan_reverse` and `seven_segment_decoder`.

First, your `bit_scan_reverse` function can be synthesized by running:

```
synth CombDigitalSystems.ms bit_scan_reverse
```

`synth` reports three pieces of information. First, it reports three summary statistics: the number of gates, the total area these gates take (in square micrometers), and the circuit's critical-path delay (i.e., the longest propagation time between any input-output pair) in picoseconds. Second, it reports the delay across the different gates of the critical path. Third, it shows a breakdown of the types of basic gates utilized in your program and their area. `synth` can also produce circuit diagrams which allows you to see the circuit implementation of your program. To do that, run:

```
synth CombDigitalSystems.ms bit_scan_reverse -v
```

This will produce a diagram in `bit_scan_reverse.svg`.

If you are using an Athena workstation directly, or are connected remotely via SSH and have X11 forwarding working (i.e., `ssh -X` or `ssh -Y`), you can display this file by running:

```
inkview bit_scan_reverse.svg
```

If you are connected to Athena remotely and do not have X11 forwarding, you'll need to copy the file from Athena to your local machine using scp to view the diagram. On Linux or OSX, this command will look something like:

```
scp {YourMITUsername}@athena.dialup.mit.edu:~/lab3/bit_scan_reverse.svg ~/
```

On Windows, the command will differ depending on what program you use to connect to Athena. For PuTTY, you'll need to use `pscp`, for SecureCRT `vcp` is required. If you have troubles getting X11Forwarding or any of the above commands to work, please come to office hours for assistance.

Once you've copied the file to your computer, you can open it by running `inkview bit_scan_reverse.svg` (or use another SVG viewer like Firefox, Chrome, `display`, or others).

Discussion Question 1: Does the circuit diagram shown in `bit_scan_reverse.svg` match your Minispec implementation of `bit_scan_reverse`? Prove that is the case by using Boolean algebra.

By default, `synth` uses the `basic` standard cell library, which only has a buffer (the identity function: 0 returns 0, and 1 returns 1), an inverter, and two-input NAND and NOR gates. You can control the library used with the `-l` flag. Let's try using the `extended` library, which also has AND, OR, XOR, and XNOR gates, as well as 2, 3, and 4-input gates. To see the difference, we will analyze a more complex digital system, `seven_segment_decoder` for which is not intuitive for us to count the number of basic gates. To do this, run:

```
synth CombDigitalSystems.ms seven_segment_decoder -l extended
```

Discussion Question 2: How does `seven_segment_decoder` change when synthesized with the extended library? What gates are used now vs. with the basic library? How does this affect area and delay?

You can also visualize the new circuit by creating another **`.svg`** file via

```
synth CombDigitalSystems.ms seven_segment_decoder -l extended -v
```

By default, `synth` tries to minimize delay by setting a target delay of 1 ps, which is obviously unachievable in this technology. You can control the target delay with the `-d` flag. A relaxed delay requirement will cause the synthesis tool to optimize for area instead. This way, you can trade off delay for area (in current technology power is also a crucial consideration, even more so than area; but power analysis is a complex topic, so in this course we will focus on area and delay). For example, the following command uses a target delay of 1000 ps:

```
synth CombDigitalSystems.ms seven_segment_decoder -l extended -d 1000
```

Discussion Question 3: Synthesize `seven_segment_decoder` using the command above. How do its area and delay change vs. the previous (delay-optimized) circuit?