Due date: Thursday February 22 11:59:59pm EST.

Turning in the lab: To turn in this lab, commit and push your changes you made to your git repository.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting within a week of the lab’s due date. See the course website for lab hours. This lab has restrictions on what subset of BSV you can use in your designs. This is not checked in the automatic tests, but we will be checking it while looking at your code in the check-off meeting.

Elevator Combinational Logic

Congratulations! You have been hired as a Junior Elevator Digital Logic Engineer at Grade A Tower Elevator Company, also known as GATE Co. Since you are only a junior elevator engineer, you will be tasked with designing a variety of combinational digital circuits necessary for a complete elevator system.

1 Elevator Location Decoder

Your first task as an engineer at GATE Co. is to produce an elevator location decoder. This elevator location decoder takes in sensor input from each floor and outputs the current floor of the elevator as a binary coded unsigned number.

Each elevator shaft has a sensor at each floor that detects if the elevator is at the floor. For a 4-floor building, the combined sensor signal is a 4-bit signal where bit 0 corresponds to the first floor and bit 3 corresponds to the fourth floor. Each bit is 1 if the floor’s sensor detects the elevator, and 0 otherwise. This 4-bit signal is the input to your elevator location decoder.

The output of the elevator location decoder is a binary-coded unsigned number. For a 4-floor building, the output is a 3-bit signal to encode the values from 1 to 4. In the case where no sensors are triggered, the decoder should output the value 0.

The distance between floors varies from building to building. As a result, when an elevator is between floors, sometimes no sensors are triggered and sometimes both the sensor below and the sensor above are triggered. In these cases, you need the elevator location decoder to return 0, signaling that the elevator is not at an exact floor.

```verilog
function Bit#(3) elevator_location_decoder(Bit#(4) floor_sensors);
    Bit#(3) ret = 0;
    // your code here
    return ret;
endfunction
```

Figure 1: Skeleton code for `elevator_location_decoder`

Exercise 1 (20 points): Implement an elevator location decoder for a 4-floor building in BSV using only the bitwise logic operators ~, &, |, and ^. The initial code for this exercise is found in `LocationDecoder.bsv` and is also shown in Figure 1. To test the design, compile the simulation executable by running `make location_decoder_test` and simulate the circuit by running `./location_decoder_test`. To check that the circuit is running as expected, look at test cases 0 through 7 and make sure your design produces the correct results. If your circuit works as expected, you will see Elevator Location Decoder: PASSED at the bottom of the test output. (It’s OK if you see Floor Sensor Error: FAILED for now.) Please refer to the [getting started guide](#) to get the initial code for this lab.

When an elevator is stopped at a floor, only the one floor should have its sensor triggered. When an elevator travels between floors, it either triggers the two adjacent sensors or it triggers no sensors. If any two
non-consecutive floors ever have their sensors triggered at the same time, then there is an error somewhere
and an emergency stop should be triggered.

```haskell
function Bit#(1) floor_sensor_error(Bit#(4) floor_sensors);
    Bit#(1) ret = 0;
    // your code here
    return ret;
endfunction
```

Figure 2: Skeleton code for `floor_sensor_error`.

Exercise 2 (20 points): Implement the `floor_sensor_error` function in BSV using only the bitwise logic
operators ~, &, |, and ^. This function should return 1 if there are two non-consecutive sensors triggered,
and 0 otherwise. The initial code for this function is found in `LocationDecoder.bsv` and is also shown in
Figure 2. To test the design, compile the simulation executable running `make location_decoder_test` and
simulate the circuit by running `./location_decoder_test`. If your circuit works as expected, you will see
`Floor Sensor Error: PASSED` at the bottom of the test output.

2 Seven-Segment Decoder

GATE Co. is installing digital displays inside their elevators to show which floor the elevator is at. To reduce
costs, the lead elevator designer has decided to use standard seven-segment LED displays.

The seven-segment displays show 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.

![Diagram of seven-segment display](image)

Figure 3: Seven-segment display

```
Figure 4: Digit representations on 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.

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

Figure 5: Skeleton code for `seven_segment_decoder`

**Exercise 3 (30 points):** Implement the `seven_segment_decoder` function in BSV. Your design is allowed to use bitwise logic operators and case statements. The initial code for this function is found in `SevenSegment.bsv` and is also shown in Figure 5. This initial code currently displays an $E$ for all inputs. To test the design, compile the simulation executable by running `make seven_segment_test`, and simulate the circuit by running `./seven_segment_test`.

3 Priority Selector

When an elevator is called from multiple floors at once, all requesting to go down, the elevator should go to the highest floor requesting to go down. This will allow for the elevator to pick up everyone wanting to go down in a single trip.

To figure out what floor to go to, we need a priority selector circuit to take in an n-bit input of requests (one per floor), and the circuit should return an n-bit output with only the highest request enabled. For example, if the input is $0'b0101$ the returned value should be $0'b0100$.

One way to do this is to first construct the n-bit signal `higher_floor_has_req` where each bit is 1 if there is an input request on a higher floor. The easiest way to construct this signal is to figure out a formula for `higher_floor_has_req[i]` that can be written in terms of `higher_floor_has_req[i+1]` and `floor_signals[i+1]`, and apply the formula in a for loop.

```haskell
function Bit#(n) priority_selector(Bit#(n) floor_signals);
    // your code here, you can use for-loops like the ones below for
    // iterating through the bits in floor_signals, although you may
    // need to modify them
    for (Integer i = 0 ; i < valueOf(n) ; i = i+1) begin
        // loop from 0 to n-1
    end
    for (Integer i = valueOf(n) - 1 ; i >= 0 ; i = i-1) begin
        // loop from n-1 to 0
    end
endfunction
```

Figure 6: Skeleton code for `priority_selector`

**Exercise 4 (30 points):** Implement the `priority_selector` function in BSV. The initial code for this function is found in `PrioritySelector.bsv` and is also shown in Figure 6. You can use bitwise logic operators for single-bit operations and for multi-bit operations, and you can use for-loops to deal with
the parameterized input width. To test the design, compile the simulation executable by running `make priority_selector_test` and simulate the circuit by running `./priority_selector_test`.

## 4 Population Count and Comparator

When people are calling the elevator to go different directions, the elevator controller needs to choose whether to handle the down requests or the up requests first. To do this, the elevator controller needs to count how many up and down requests there are, and compare them to see which is larger.

The hardware for doing this computation is done in two stages. First the number of requests going in each direction are counted using a population count circuit. Second the number of up requests is compared to the number of down requests using a comparator.

### 4.1 Population Count

First you are in charge of implementing a population count circuit. For simplicity, your population count circuit will only take in an 8-bit input and return an output representing the number of 1s in the input signal.

```verbatim
function Bit#(4) population_count(Bit#(8) in);
    // your code here
endfunction
```

Figure 7: Skeleton code for `population_count`

**Discussion Question 1 (5 points):** Why is the return type 4 bits wide?

**Exercise 5 (30 points):** Implement the `population_count` function in BSV. The initial code for this function is found in `PopulationCount.bsv` and is also shown in Figure 7. You can use any BSV shown in lectures, recitations, or labs up to this point. You are not allowed to use any Bluespec-provided population count libraries. To test the design, compile the simulation executable by running `make population_count_test` and simulate the circuit by running `./population_count_test`.

### 4.2 Comparator

As your final task, you are in charge of implementing an n-bit comparator. Even though the `population_count` only outputs a 4-bit signal, there are many other signals in an elevator system that need to be compared, and they are of many different widths. Therefore your comparator needs to work for any sized input signals.

Your comparator, `is_geq` will take in two n-bit binary unsigned numbers `a` and `b` and return `True` if `a` is greater than or equal to `b`.

```verbatim
function Bool is_geq(Bit#(n) a, Bit#(n) b);
    // your code here
endfunction
```

Figure 8: Skeleton code for `is_geq`

**Exercise 6 (30 points):** Implement the `is_geq` function in BSV. The initial code for this function is found in `IsGeq.bsv` and is also shown in Figure 8. You are free to use any BSV shown in lectures, recitations, or labs except for the built-in comparator functions provided by BSV like `<`, `>`, etc.) (although you can use `==` to convert a `Bit#(1)` to a `Bool`). To test the design, compile the simulation executable by running `make is_geq_test` and simulate the circuit by running `./is_geq_test`. 