EECS 470 Project #1

Note:

- This is an individual assignment. While you may discuss the specification and help one another with the SystemVerilog language, your solution – particularly the designs you submit – must be your own.
- This project is due by **Monday, September 8th**
- Project 1 is not graded for synthesis, but you should still implement your designs to produce correct output in both simulation and synthesis.

1 Introduction

In this project, you will be designing several different priority selectors. Then, you will use priority selectors to efficiently solve a parking spot problem (introduced in section four).

A priority selector, or priority arbiter, has n pairs of request and grant lines. As the names imply, the selector chooses one of the asserted request lines and asserts its corresponding grant line. In the most general case, the selector can assert k grant lines, where $k \leq n$. For your priority selector implementations, $k = 1$. We will provide you with a priority selector for the parking spot problem implementation without this limitation.

Priority selectors are heavily used in computer architecture, where we often have limited resources that need to be assigned optimally for best performance. The modules you write for this assignment will remind you about the concepts of digital design you first learned in EECS 270 and begin to prepare you for the final project, where they can be reused.

For this assignment, you will:

- Design combinational priority selectors in 3 styles:
  - A 4-bit selector using basic `assign` statements.
  - A 4-bit selector using `always` blocks with if-else statements.
  - An 8-bit and 4-bit selector using a hierarchy of smaller selectors as Verilog modules.
- Write a testbench for the 8-bit hierarchical priority selector.
- Analyze a design that uses a dependent for-loops to solve a parking spots problem.
- Use priority selectors to create a more efficient solution to a parking spots problem.

The starter code has two directories: `psels` and `parking`. The files for sections two and three are under `psels`, and the files for section four are under `parking`. Both directories have their own Makefile. You will start in the `psels` directory.

A set of example modules for the two hierarchical designs are the `and2` and `and4` modules in `and4.sv` in the `psels` directory. The Makefile starts ready to test these modules with `and4_test.sv`. You will need to update the `TESTBENCH` and `SOURCES` variables to run other tests, either by editing the Makefile itself or running with the updated files on the commandline.

You can set the variables at the commandline with: `make TESTBENCH=mod_test.sv SOURCES=mod.sv`

2 4-bit Combinational Priority Selector

For this section, you will design two different 4-bit combinational priority selectors. Both are to be declared as in figure 1

```
module ps4 (
    input        [3:0] req,  // lines being requested
    input              en,
    output logic [3:0] gnt  // lines to grant
);
```

Figure 1: ps4 module definition

The signal `req[3]` is the highest priority request, and priority goes down to `req[0]`, which is the lowest priority request. In all cases, no more than one of the grant lines should be asserted at any given time. If `en` is low, then no grant lines should be asserted. For example, if `en=1` and `req=4'b0101`, then `gnt=4'b0100`.

You will make two designs of this module.

In `ps4-assign.sv`: The first design should use four assign statements and Verilog logic operators to implement the selector.

In `ps4-if_else.sv`: The second design should use a single if-else chain in an `always_comb` block to implement the selector.

A testbench is provided in ps4_test.sv.

The Makefile for this project starts set-up to run the example in `and4.sv`. Update the `TESTBENCH` and `SOURCES` variables and run `make sim` to test your modules.

3 Hierarchical Priority Selectors

For this section, you will design 4-bit and 8-bit selectors using a hierarchy of modules. Both designs should go in `ps8.sv`. You will also need to write a testbench for the 8-bit module in `ps8_test.sv`. A blank file has been created for you.

Consider the Verilog you wrote in `ps4-assign.sv` and `ps4-if_else.sv`. If you were going to make a 128-bit priority selector, your design would be quite long and unreadable. One way to avoid this problem is to build your module in a way that it can be combined with itself to make a larger version. For example, it is fairly easy to use three 2-bit `and` gates to create a single 4-bit `and` gate. Think through how you would go about building this. Figure 2 shows how this is done in the case of the `and`. How would you go about doing this with priority selectors? It's not easy...

```
module and2 (
    input [1:0]  in,
    output logic out
);
    // a 2-bit AND operation
    assign out = in[0] & in[1];
endmodule

module and4 (
    input [3:0]  in,
    output logic out
);
    // use a variable to connect the left/right outputs to top
    logic [1:0] tmp;

    // reuse the logic inside the and2 module
    and2 left (.in(in[1:0]), .out(tmp[0]));
    and2 right(.in(in[3:2]), .out(tmp[1]));

    // combine the left and right to produce the final out signal
    and2 top(.in(tmp), .out(out));
endmodule
```

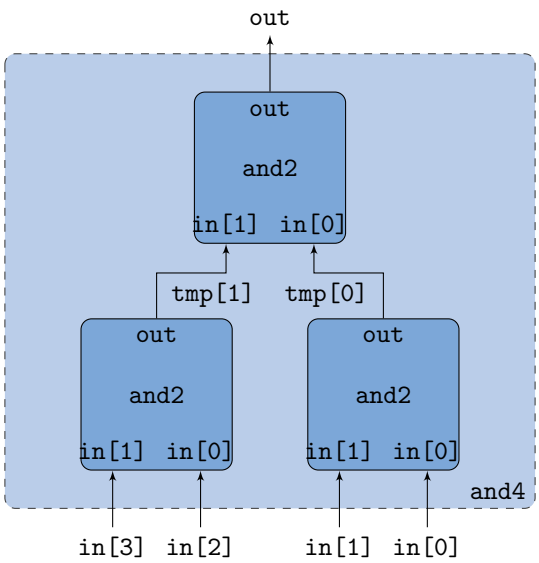

Figure 2: Hierarchical Module Example: a 4-bit `and` built with three 2-bit `and` modules.

Let's consider a 2-bit priority selector as a building block. In order to make a 4-bit selector, you can use three 2-bit selectors in a tree structure. The right selector might get the two lowest priority request and grant lines, while the left gets the two highest priority request and grant lines. Then the left and right would ask the top to choose which of them is enabled to output the grant.

In our previous 4-bit module we had a way to be told if we *could* grant to anyone, the enable line (`en`). But we didn't have a way of saying "Hey, I have a request that would like a grant." We will add that functionality, as the `req_up` output for "requesting something from the device above me." This signal should be asserted if either request is asserted no matter the value of the enable.

You will need three modules, one for the base 2-bit priority selector and others for the the 4-bit and 8-bit selectors. These modules should be declared as follows:

```
module ps2 (
    input        [1:0] req,
    input              en,
    output logic [1:0] gnt,
    output logic       req_up
);

module ps4 (
    input        [3:0] req,
    input              en,
    output logic [3:0] gnt,
    output logic       req_up
);

module ps8 (
    input        [7:0] req,
    input              en,
    output logic [7:0] gnt,
    output logic       req_up
);
```

The 2-bit selector should be implemented like either `ps4-assign.sv` or `ps4-if_else.sv`, but with extra logic for implementing the `req_up` signal (high if anything is requesting).

Once you've made the 2-bit selector, implement the 4-bit and 8-bit selectors by using fig. 2 as an example. You must use the minimum number of child modules for each design.

Your designs should output the final `gnt` lines directly from the module instantiations using the outputs of those modules. **They should not be the result of an assign statement or assigned to in a combinational block. You may use an assign statement to output the `req_up` signal.**

You will also need to write a testbench for the ps8 module. Write this in `ps8_test.sv` using `ps4_test.sv` as a reference (Hint: make sure to test all outputs of the module, including any new ones).

Your testbench must print "`@@@ Incorrect`" for an incorrect ps8 implementation and must print "`@@@ Passed`" for a correct implementation. Ensure that these strings are printed exactly as shown in your testbench.

Update the `TESTBENCH` and `SOURCES` variables and run `make sim` to test your modules.

3.1 Additional Thoughts

If you get extra time, here are some things you might want to look into. These will not be graded.

- Using a counter you can do exhaustive testing (test every possible case) of the combinational priority selectors. In general, when is exhaustive testing *not* a good idea?
- There is a `$random` function provided in Verilog for testing, which can be used to generate random test vectors. Read about it online and try to understand how it works. Why would you want to use this function?

4 Wonka's Parking Spots

For this section, you will analyze a solution to a parking spots problem (introduced in the next subsection) that uses dependent for-loops. Afterward, you will develop a more efficient implementation using a priority selector. All the files for this section are in the `parking` directory.

4.1 Parking Spots Problem

Congratulations! Willy Wonka has hired you to manage his chocolate factory's parking lot. The parking lot has 64 parking spots, three entrance gates, and three exit gates. To ensure smooth operation, you want to design a system that prevents cars from wandering the lot in search of an open spot. Instead, cars will only be allowed to enter if a parking spot is available, and each car will be assigned a specific spot upon entry.

Cars can enter the lot through any of the three entrance gates. For a car to be admitted, there must be an available parking spot. If admitted, the car is provided a specific spot to park. With three entrance gates, up to three cars can enter simultaneously.

Cars can exit the lot through any of the three exit gates. When a car exits, the system is notified of the specific spot that was vacated, allowing that spot to later be reassigned to an incoming car. With three exit gates, up to three cars can exit simultaneously.

4.1.1 Module Interface

```
module parking #(parameter SPOTS=64, ENTRY_GATES=3) (
    input                                       reset,
    input                                       clock,
    input  logic [ENTRY_GATES-1:0]              car_incoming,
    input  logic [SPOTS-1:0]                    car_exiting_spots,
    output logic [ENTRY_GATES-1:0][SPOTS-1:0]   car_assigned_spot
);
```

Your system is implemented using a parameterized module with the above declaration.

The module has two parameters:

- `SPOTS`: The number of parking spots (64).
- `ENTRY_GATES`: The number of entrance gates (3).

The module has four inputs:

- `clock`
- `reset`: When reset is asserted, all parking spots should be initialized as empty. The reset is synchronous.
- `car_incoming`: A bit vector indicating whether a particular entrance gate currently has a car requesting admission. `car_incoming[i] == 1` means that the `ith` entrance gate has a requesting car. **The entrance gates are not necessarily filled one after another**; i.e. the `ith` gate being unoccupied does not imply that the `(i+1)th` gate is also unoccupied.
- `car_exiting_spots`: A bit vector indicating which spots (if any) are being vacated. `car_exiting_spots[i]` means that the `ith` parking spot is being vacated. No cars are vacating if all the bits of `car_exiting_spots` are 0. The maximum number of simultaneous `1s` in `car_exiting_spots` equals the number of exit gates.

The module has one output:

- `car_assigned_spot`: A two-dimensional bus of one-hot lines for each entry lane indicating which spot, if any, an incoming car is allocated. `car_assigned_spot[i][j] == 1` means that the car in the `ith` entry

lane is allocated the `jth` parking spot. If a car requesting entry on the `ith` lane cannot be admitted due to a lack of open parking spots, all the bits of `car_assigned_spot[i]` should be 0. If `car_incoming[i] == 0` (i.e. there are no cars requesting entry at the `ith` gate), the state of `car_assigned_spot[i]` is ignored (it can be arbitrary).

4.2 Analyzing a Naive Implementation

After reading your system requirements, an Oompa Loompa engineer attempts to implement it. You can find their implementation in `parking_naive.sv`.

4.2.1 Behavioral Analysis

We have given you a testbench that verifies whether an implementation of the parking spots problem is behaviorally correct. To run this testbench on the Oompa Loompa's solution, set `TESTBENCH=parking_test.sv`, `SOURCES=parking_naive.sv`, and `SYNTH_FILES=parking.vg` in the Makefile. Then, run the command `make sim` to compile and execute the testbench.

The implementation is behaviorally correct. Answer the following questions about the implementation:

Question 1. Removing the `break` from the Oompa Loompa implementation makes it incorrect. Why is the `break` necessary? Answer in at most three sentences.

Question 2. Replacing `if (next_parking_lot[j])` with `if (parking_lot[j])` makes the Oompa Loompa implementation incorrect. Why is it necessary for the conditional to use the next state? Answer in at most three sentences.

Question 3. The for-loop on line 55 of the Oompa Loompa's implementation is an independent for-loop because each iteration of the loop operates independently of the others. In other words, the outcome or execution of any given iteration does not rely on the results or progress of previous iterations. The for-loop on line 39 is dependent. Why is this so? Answer in at most three sentences.

4.2.2 Synthesis Introduction

Unlike in software development where code gets compiled into executable machine code, in hardware design, Verilog code is synthesized into physical circuits.

Synthesis is the process of converting a behavioral Verilog design into a structural design that could be taped out on real silicon. Specifically, the synthesis tool (Synopsys Design Compiler in our case) replaces larger operations (like multiplication and addition) with standard designs for modules, and attempts to build larger logic components out of a set of standard cells (our standard cells are in the `lec25dscc25.v` file included by the Makefile).

Just as software can be optimized by how a programmer writes the code to reduce execution time or memory usage, the way a hardware designer writes Verilog can impact the performance of the synthesized hardware, resulting in lower latency, reduced power consumption, and/or a smaller chip area.

4.2.3 Synthesis Analysis

Although the Oompa Loompa's solution is behaviorally correct (it passes the testbench), we want to evaluate whether the synthesized hardware is efficient.

As discussed earlier, the Oompa Loompa's solution uses dependent for-loops in its implementation. Dependent for-loops can lead to long chains of hardware after synthesis because each iteration of the for-loop depends on the previous one, introducing sequential dependencies. When synthesized, these sequential dependencies translate into a series of connected hardware elements where a particular element can only complete its work once the previous element in the series has finished. On the other hand, independent for-loops are synthesized into hardware where the result of each iteration is computed in parallel.

Question 4. Fill in the blank: The long chains of hardware generated by dependent for-loops can increase the _____ path, leading to a longer minimum clock period.

When synthesized under medium effort, the minimum clock period the Oompa Loompa's solution achieves is about 3.3 nanoseconds with 64 parking spots. You will learn how to synthesize and find the minimum clock period yourself in future labs/projects.

4.3 Using a Priority Selector

You decide to write your own solution to the parking problem that avoids dependent for-loops in hopes of creating more efficient and parallelized hardware. For this problem, **you can avoid dependent for-loops by using priority selectors.**

In `parking.sv`, we have provided you with some starter code and scaffolding for your solution to the parking problem that uses a priority selector. You are to finish the implementation, following the `TODOs`. Your solution should not contain any dependent for-loops.

To check the correctness of your solution using the given testbench, set `SOURCES=parking.sv psel_gen.sv` in the Makefile (with the other variables kept the same as what was set for Oompa Loompa testing). To compile and execute the testbench, run `make sim`. If your solution fails, the testbench will provide an error message to hopefully aid your debugging efforts.

4.3.1 K-Grant Priority Selector

For your solution, you will need a priority selector that can grant k simultaneous requests. We have given you an efficient implementation of such a priority selector in `psel_gen.sv`. The module has two parameters:

- `WIDTH`: The number of input request lines.
- `REQS`: The number of requests that can be granted simultaneously (i.e. k).

Along with the input signal `req` and the output signal `gnt` that you used in your priority selector implementations, the given priority selector has two more outputs:

- `gnt_bus`: A two-dimensional output bus of one-hot lines for each of the k simultaneous grants.
- `empty`: A single-bit output that is high if there are no input requests.

The priority selectors are already instantiated for you in the given starter code.

See the block comment at the top of `psel_gen.sv` for example behavior of this module under various inputs.

4.3.2 Analyzing the Priority Selector Implementation

By eliminating the dependent for-loops and using a priority selector, our solution synthesizes into more parallelized hardware than the Oompa Loompa's solution.

When the priority selector implementation is synthesized, the minimum clock period achieved is 2.8 nanoseconds with 64 parking spots (compared to the Oompa Loompa's 3.6 nanoseconds). The difference in performance is even more pronounced when we increase the number of parking spots. With 256 parking spots and 12 entry gates, the Oompa Loompa's minimum clock period is about 13.6 nanoseconds (∼73.5 MHz) whereas the priority selector implementation is only about 8.9 nanoseconds (∼112 MHz)! A difference of 4.7 nanoseconds is significant for hardware components, especially when placed in the critical path of a larger system.

Optional: To synthesize yourself, run the following command: `make -B synth/parking.vg CLOCK_PERIOD=X`, where `X` is replaced with your desired clock period. After synthesis completes, you can check if your implementation was able to meet the desired clock period by running `make slack`; if `VIOLATED` appears in the output, your implementation was unable to meet the desired clock period. We will cover synthesis in more detail in future labs/projects.

Question 5. In your own words, explain why using priority selectors resulted in a smaller minimum clock period compared to the dependent for-loop implementation. Answer in at most two sentences.

5 Comprehension Questions

Answer the following questions and submit your answers in the provided `answers.txt` file:

Question 1. Removing the `break` from the Oompa Loompa implementation makes it incorrect. Why is the `break` necessary? Answer in at most three sentences.

Question 2. Replacing `if (next_parking_lot[j])` with `if (parking_lot[j])` makes the Oompa Loompa implementation incorrect. Why is it necessary for the conditional to use the next state? Answer in at most three sentences.

Question 3. The for-loop on line 55 of the Oompa Loompa's implementation is an independent for-loop because each iteration of the loop operates independently of the others. In other words, the outcome or execution of any given iteration does not rely on the results or progress of previous iterations. The for-loop on line 39 is dependent. Why is this so? Answer in at most three sentences.

Question 4. Fill in the blank: The long chains of hardware generated by dependent for-loops can increase the _____ path, leading to a longer minimum clock period.

Question 5. In your own words, explain why using priority selectors resulted in a smaller minimum clock period compared to the dependent for-loop implementation. Answer in at most two sentences.

6 Submission

To submit your project to the EECS 470 autograder, upload your solution files to the `main` branch of your GitHub repository and run the project submission script for project 1:

```
/usr/caen/misc/class/eecs470/submit 1
```

Soon after your submission you will receive an email with a summary of the public output of the autograder. This will contain the public results of correctness tests and whether the autograder encountered any errors and is not representative of your final grade.

Email the instructors or create a private Piazza post to debug any issues with the autograder.

The following files will be graded:

- `answers.txt`
- `parking.sv`
- `ps4-assign.sv`
- `ps4-if_else.sv`
- `ps8.sv`
- `ps8_test.sv`