# Lab 1

*Combinational Logic: Extension of the Tic Tac Toe Game*

# Introduction/Objectives

This lab is designed to familiarize you with the process of designing, implementing, and verifying a combinational logic circuit. In particular you will get an overview of the tools we'll be using throughout the quarter and how they interact. For this lab you will be taking the provided Tic Tac Toe game and extending it as in problem 9.20 in the reader. Please make sure you have read and studied chapter 9.4 of the reader before starting this lab. The code provided here is that the code described in the book with a bit of glue code to take input in from the pushbuttons and output it to the screen. If you don't understand the design of the game from the book you are going to have a very hard time making this lab work.

## What you'll be doing

Logic design consists of three main parts: verifying, verifying, and verifying. Well, technically you design something, then you verify it, then you implement it, and then you validate, but most of your time will be spent making sure it works the way you designed it. In this lab we've provided all of the code for a working Tic Tac Toe game with the Win and Block (two_in_row.v) and Empty (empty.v) strategies as outlined in the book, and a test bench (lab1_testbench.v) for verifying correct functionality in simulation. Your job is to write a Play Adjacent Edge module (probably called something creative like play_adjacent_edge.v) and a Select-4 module and integrate them into the top-level model (tictactoe.v) as described in problem 9.20.

Excerpt 9.20:

> Tic-Tac-Toe strategy, III. Extend the tic-tac-toe module of Section 9.4 by adding a strategy module that, on a board that is empty except for an opponent O in two opposite corners and your X in the middle, play to an adjacent edge space. In the following example, H is the move played by player X.
>
> ```
> O - -
> H X -
> - - O
> ```

In addition to adding the `play_adjacent_edge` module and hooking it up, you will need to modify the top-level simulation model to test this new functionality, and verify that it works in the simulator. Once you are very sure that it is functioning as required, you will come into lab and demonstrate it running on the FPGA. You will quickly (and quite possibly painfully) learn that unless you have extensively tested a module it will not work, so plan on putting more thought into how you are going to test your design as how you're going to build it.

## Important things to remember

- The majority of your time will be spent simulating your design to verify its functionality. Debugging your design on the actual FPGA is enormously more complex than in simulation, so try to make sure everything works before you start downloading to the FPGA.
- When you simulate a model you will use a *testbench* module which provides input to the *device*

*under test* (dut) to see how it responds. In this lab the lab1_testbench.v module instantiates two instances of the TicTacToe module, one for testing (called dut) and one to play against it (called opponent).

- The testbench is never actually run on the FPGA, so it can do things like print messages to the screen and use for-loops. A separate top-level module "lab1_top.v" is used for synthesizing the design for the FPGA. That top-level module hooks up the "tictactoe.v" module to the glue code to take input from the buttons and output to the display.

# Getting started (Prelab portion): Simulation

## Creating the project in ISE

To work on your project, you first need to create a new ISE project and import your source files.

1. Run ISE by typing `ise` in the terminal.
2. Create a new project by choosing File → New Project…
3. As in lab 0, first set the location and then the name to `lab1`. Finish the wizard. You now have a lab1 directory in your group folder.
4. Now that you have a blank project, you need to add all of the files. This time, we have provided starter files for you, so you do not need to create the project from scratch. Choose Project → Add Copy of Source..., then navigate to your extracted lab folder and select all of the files. Since you selected "Add **Copy** of Source," these files will be copied into your group folder so that you can modify them.
5. The resulting dialog should show some files associated with "implementation," some with "simulation," and some with "all." If you're not sure why some files are implementation and some are simulation, look at the synthesis portion of Lab 0, or ask your TA. Hit OK.
6. Finally, the first thing that we're interested in is simulation, so switch the view to Simulation.

You should now see all the simulation-related files listed in their hierarchy on the left. Note that while there is one interesting testbench module, `lab1_testbench`, most of the modules at the top of the hierarchy are NOT testbenches, but are listed there because there is no testbench that instantiates them (directly or indirectly). In particular, you can see that `lab1_top` is at the top of the hierarchy, which means that there is currently no testbench for it. Don't worry about that; you will be using the `lab1_top` module only when you synthesize in lab—so don't modify it this time around.

You can view and edit the files by double-clicking on them. At this point you should go through each file in the lab1_testbench part of the hierarchy and familiarize yourself with what is in it. Be aware that modules will be listed in the hierarchy each time they are instantiated somewhere, so there will be duplicates, which is normal. What you see in the files should be basically the same as what is in the book in chapter 9.4.

## Running the provided code

Once you are familiar with the contents of the files (which is important given that you're going to be modifying them) you should simulate the whole design to see how it works.

Remember that when you run a simulation you are using a *testbench* to control the simulation. In this case the testbench is the top-level file "lab1_testbench.v". It is setup to run a bunch of tests on the design and then stop. As the simulation runs, the testbench uses the $display() command to print out messages to the log window. Remember that since the testbench is only used for simulation it can

include commands like $display() or loops.

Running a simulation is easy:
1. Select "`lab1_testbench`" in the hierarchy, as that is the one we want to simulate.
2. Under "Modelsim Simulator" in the processes pane below, choose "Simulate Behavioral Model." Just like in lab 0, you will see a simulation bench and timing diagram pop up where you can view the results.

For this lab you'll mostly want to see the signals at the top-level of the design in your timing diagram, so click on "`lab1_testbench`" in the SST (signal search tree) pane, then select all the signals from the signals pane and click "append" to add them to the signal viewer. Now you can see the values of those signals over time (you may need to zoom out). You can save this setup for future use by simply choosing File → Write Save File, or hit Ctrl+S. The signal setup will be automatically loaded next time you run the same testbench. This will definitely come in handy for your final project and the later labs!

Make sure you take a look at the top-level `lab1_testbench` module and understand what those signals are as you will need them when you write and test your own module. At this point you should make sure you understand the output you're seeing and how it relates to the testbench.

## Your job: Extending TicTacToe

Now that you've got your simulation environment all set *and you understand how the original design works*, it's time for you to extend it. What you need to do consists basically of:
1. Add a new `play_adjacent_edge` module that meets the specifications of problem 9.20.
2. Change the `select3` module to a `select4` module capable of prioritizing amongst 4 inputs.
3. Integrate the new modules into the `tictactoe` module. (We've even specified where you should do this.)
4. Update the testbench to verify the functionality of your module.

Please remember that verifying your module's functionality is the most important part of this whole process. For a small design like this it isn't so difficult to do, but as your designs become larger you will find that the more time you spend testing the easier everything goes!

## Synthesis

The final part of the lab is to synthesize your Verilog into the bit file that you will download to the FPGA to configure your FPGA properly for your desired functionality. Specifically, the bit file tells the FPGA how to initialize logical slices, connect the reconfigurable interconnects, and to setup the chip I/Os. Every time you make a modification to the Verilog for your project, you need to completely redo the process for making the bit file; specifically, you need to re-synthesize and map your Verilog (Xilinx ISE takes your design and maps it to logic available on the FPGA), and place and route your design (Xilinx ISE takes the logic it mapped your design to, and figures out where it fits on the FPGA). This process can take a long time, so try to avoid fixing your design through iterative resynthesis.

You've already set up your ISE project file, so open it up and switch to the "implementation" view. There are a couple files of interest that you didn't deal with when you were simulating:

## Top-level module

Unlike in simulation, the top-level module (the module highest in the hierarchy) is not a testbench, but is a normal module that interfaces with the hardware as defined by the signals in the UCF file (more on that in a moment). Our top module in this lab is lab1_top, implemented in lab1_top.v. The top module pretty much just instantiates our tic-tac-toe module and feeds signals from the outside world to the module and vice versa. It also instantiates and configures the VGA driver to enable display on the computer monitor.

Look over lab1_top.v and the other modules. You should understand at a basic level what lab1_top.v is doing, but you do not need to concern yourself over the inner workings of the VGA driver.

## Top-level module

The User Constraints File is a file used by Xilinx ISE to apply constraints to the FPGA design. These constraints can be as simple as what pin on the chip a signal should connect to and what logic family (voltages) this pin should use, to complicated constraints relating to the behavior of the clock the chip will use (for a chip implementing sequential logic). Although you won't modify the UCF in this lab, it is essential to become familiar with UCFs, and it would be a good idea to look over the provided UCF to become more familiar with the syntax used by UCFs.

## Generating your .bit file

The process list will change depending on which part of the design you select in the Sources list. To generate a .bit file you need to select the top-level portion of the project "lab1_top" and double-click on "Generate Programming File" at the bottom of the Processes list or right-click on it and choose "Rerun All". After Xilinx finishes synthesizing, mapping, placing, and routing your design, double click "configure target device". From this point on, it's the same as lab 0, so reference that handout to find out what you need to do next.

## Top-level module

Now that your design is running on the FPGA you can try it out by flipping the DIP switches around, hitting buttons, and viewing the display output. The game state is set using the eight DIP switches and the down button to mask out the spots on the board (total of 9 locations), and then the left button to set X or right button to set O. The letter C (conflict) will be displayed if you've set X and O to the same board spot. The up button resets the game. The board on the left is the current state as you've entered it, and on the right is the move that X would take.

Make sure you test your edge cases for Play Adjacent Edge. Once you've got your design working you need to verify that your new strategy behaves as expected and show the TA.

# Submissions

## Submission Instructions:

Create a directory called **lastname-firstname-lab1** and make a directory inside of it called **source.** Then copy the following files to that folder, placing verilog code in the source directory and all other files in the top level:

1. Verilog code (that is, **.v files**) of:
    - the modified `tictactoe` module
    - your new strategy module
    - the `select4` module and
    - your modified testbench and any other test benches that you created
    - DO NOT include your entire project folder. Doing so will earn you a 10% deduction.
2. A screenshot containing GTKWave simulation diagrams of your extended game as well as its display outputs. These **must** be annotated with text to point out where your new module is being tested and how the displayed results match what you expect. You may use any program to complete this, but make sure you save it in a reasonable format (png,jpeg,pdf)
3. A text file with answers to the following questions.
    - How did you develop your design and how is it structured?
    - What strategy did you use to verify your design in the testbench and what modifications did you make?
    - Did you encounter any problems during the design and did you understand those problems?
    - Explain the difference between synthesizable and non-synthesizable code. Give examples of blocks/constructs that you have used already that are non-synthesizable (you know of at least two).

Zip the folder and submit it to blackboard.