DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2016

Lab 3: Tic-Tac-Toe

Week of September 26 to 30 (due 11:59 PM the night before your lab session)

1 Introduction

In this lab you learn Verilog coding by looking at an existing hardware design for a Tic-Tac-Toe game and improving it in *two* ways. First, by detecting when someone has won, and second by adding logic so that the hardware makes more intelligent game play decisions. The Verilog we give you implements a Tic-Tac-Toe game on your DE1-SoC and includes a VGA output module to display the state of the game on the monitors in the MCLD 112 lab. You play versus a combinational logic game module. The biggest challenge in this lab is getting familiar enough with Verilog to understand the tic-tac-toe game logic. The amount of Verilog you need to write is relatively small but it may take you several hours to figure out how to write that code. If you are considering starting early on this lab (e.g., right after finishing Lab 2) please be aware we will cover the Verilog syntax needed to make this lab (relatively) easy during lecture on Thursday September 22, which will cover most of the rest of Slide Set 2. Thus, if you get stuck understanding the code given to you to start with you may want to come back to this lab after that lecture. Similarly, assuming the video recording works September 22 you may want to refer back to the video while working on this lab.

Some important notes and reminders before you get started: (1) You must follow the rules in the "Peer Help Policy" (a copy can be found on Piazza under Resources > Lab Handouts). (2) You must submit your code via "handin" by 11:59 pm the night before your assigned lab section for Lab 3 as that is the code that will be marked by your TA during your demo. To avoid any last minute difficulty with the submission process please do a trial submission as soon as possible and submit updates regularly as you improve your code. Since there are a very limited number of TA hours we cannot grant extension requests due to difficulties with using "handin". (3) If you were randomly assigned a lab partner for Lab 2 and you want to continue working with a partner (either the one we assigned you or a new partner) you must complete the "Second Chance Partner Signup" on Connect by 11:59 pm the night of your Lab 2 lab session as this signup form includes important terms you and your partner must agree to abide by. (4) If you previously signed up online to work with a partner and want to change or drop partners you must notify your prior partner via email one week before your assigned lab section for Lab 3 AND you must submit a signed partner add/drop/change form AND a copy of the above email at least 4 days prior to your assigned lab section.

2 Understanding The Code Provided (`tictactoe.v`)

Follow the steps below to get the starter Verilog code, simulate it and get familiar with what it does.

2.1 Step 1: Download.

Download the starter Verilog files provided on Piazza (lab3.zip) and unzip them. You should end up with several Verilog files in a folder called lab3. Some are used for driving the VGA ("`tictactoe_to_vga.v`", "`vga.v`", "`ff.v`", "`O.bin`", "`X.bin`"), and saving the state of the game ("`ff.v`" and "`game_state.v`"). We do not expect you to understand how these files work – they are included to make Lab 3 more fun! The file "`DE1_SoC.qsf`" is a new pin assignments file you should use for Labs 3 to 7. The file "`lab3_top.v`" contains synthesizable Verilog for connecting switches and LEDs to the game logic and it is explained in Section 4. The file "`tictactoe.v`" contains the actual game logic and you will need to understand and modify this file. The file "`tictactoe.v`" is explained below and in Section 9.4 of the Dally textbook. The file "`detectwin.v`" contains an empty module declaration for a module that you will fill in to detect when one player has won. DO NOT modify "`game_state.v`", "`tictactoe_to_vga.v`", "`vga.v`", "`ff.v`", "`O.bin`", "`X.bin`" or "`DE1_SoC.qsf`".

2.2 Step 2: ModelSim Simulation.

Launch ModelSim. When ModelSim opens, it may automatically open your project from Lab 1. If this happens, close your Lab 1 project before continuing by going "File -> Close Project". Next, create a new project ("File->New->Project...") with Project Name "lab3" and set Project Location to the directory containing the files from Step 1 (click OK in the Create Project dialog window to continue). When the "Add items to the Project" dialog appears click "Add Existing File" and browse to find `tictactoe.v` then close the dialog window. This adds `tictactoe.v` to your new lab3 project. Next compile all (you should get no errors), and start the simulation by typing "`vsim TestTic`" in the transcript window. Add the `xin`, `xout`, `oin` and `oout` signals to the waveform viewer along with some internal signals (e.g., `TestTic/dut/block`, `TestTic/dut/empty`, `TestTic/dut/win`). If you don't recall how to do this, please review Figure 20(a) in the Lab 1 handout or watch the debugging video starting around time 11:36 (see https://youtu.be/2c3CZouKJKs?t=696) including the portion where it shows how to add *internal* signals starting at time 12:34 (see https://youtu.be/2c3CZouKJKs?t=754). Then, run the simulation by typing "`run -all`" in the transcript window. In general, if you have any difficulty setting up the simulation as described above please refer back to Section 2.3 of the Lab 1 handout (you can also search online for any error messages you get; for "vsim" error messages try the command `verror` in ModelSim transcript window to get a longer explanation). If you get stuck on getting the simulation to work you may ask another student (even if not your partner) for help, but bear in mind you need to be able to know how to create a new project on your own by Lab 4 and you may be tested on this during the coding proficiency test (Quiz 4).

2.3 Step 3: Understanding the code provided.

Next, to familiarize yourself with the code let's do a "top-down" walk through the code in `tictactoe.v`. Open "`tictactoe.v`" in ModelSim, which you can do by going to the project window and double clicking on `tictactoe.v` (if you are using Windows 8 and ModelSim crashes when you do this, then use Quartus to open your Verilog files instead).

2.3.1 TestTic module (testbench)

Start by examining the test bench code for module `TestTic` which starts at line 164 in `tictactoe.v`. The TestTic module is *not* synthesizable and we only use it in ModelSim. The TestTic test bench instantiates two copies of the synthesizable module `TicTacToe` on lines 169 and 170. The first is called `dut` (for device under test) and the second is called `opponent`. These module instances "exist" at the same time throughout the entire simulation. Module instantiation is *not* the same thing as a function call. For an explanation of the syntax see slides 38 to 40 in Slide Set 2 (slide number refers to slides posted before lecture), Section 7.1.1 and 7.1.2 in the Dally textbook or http://www.asic-world.com/verilog/syntax2.html.

The testbench script first checks some "corner cases" in lines 173 to 187. Then, it has the "dut" module play against "opponent". Notice these two modules are connected together via the test script with the 9-bit busses declared as "**`reg`** `[8:0] xin, oin ;`" for the *current* board position, and "**`wire`** `[8:0] xout, oout ;`" to carry the desired *next* positions. The 9-bit busses throughout `tictactoe.v` encode board positions as shown in Figure 1. For example bit 0 is the top-left square[1]. The **`repeat`** statement, which does NOT describe hardware, plays 6 turns of first X selecting a move, which "happens" at line 176 in the test script when the output `xout` of the combinational logic module `dut` is bitwise-ORed with the current set of positions occupied by X, `xin` to create a new set of board positions `xin`. The same process is performed for O at line 202. Again, note that the lines inside the initial block and repeat statement DO NOT describe hardware — they are part of a test script used in ModelSim to check the hardware described by the "design under test" module `TicTacToe`. Indeed, none of the code in TestTic module will be downloaded to your DE1-SoC in

[1] When we synthesize the code to put it on the DE1-SoC in Section 3 we will "reverse" this layout by the way we connect the `TicTacToe` module to the LEDs and switches in `lab3_top.v`. We do this to make playing the game on the DE1-SoC a bit more intuitive.

0	1	2
3	4	5
6	7	8

Figure 1: Board position versus bit position.

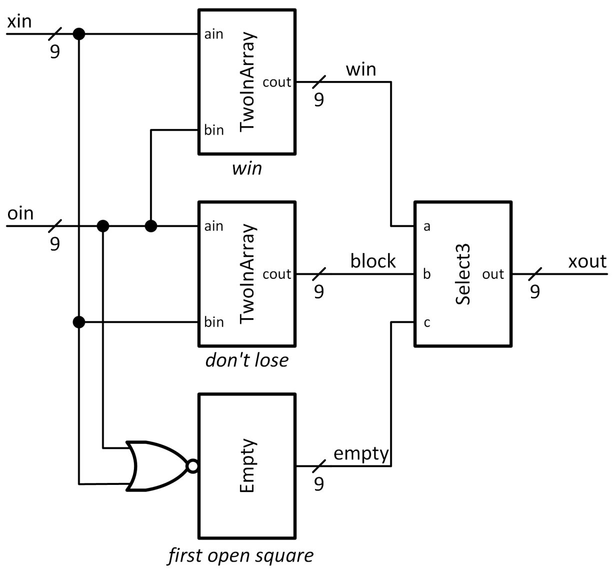


Figure 2: High-level design of the tictactoe module (reproduces Figure 9.11 in Dally)

Section 3. Note "opponent" is connected differently from "dut" to enable it to play for O instead of X.

2.3.2 TicTacToe module

Next, look at the code for the module `TicTacToe` on lines 88 to 98 which corresponds to Figure 2. `TicTacToe` instantiates two copies of module `TwoInArray` called `winx` and `blockx`. These two instances will exist *at the same time* immediately after you download your design to the FPGA on the DE1-SoC and thereafter until you disconnect the power on your DE1-SoC. The first module looks for an open square to play for X to win and the second looks for an open square that X needs to play to block O from winning. The module instance `emptyx` selects the square to play if it is the first move. The module instance `comb` selects which of these three strategies to actually use.

2.3.3 TwoInArray module

Next, look at the code for `TwoInArray` on lines 100 to 132. This large module checks for the various ways one can win at tic-tac-toe when one has played in two squares in a row, column or diagonal. The inputs are called A and B to keep the code general. Concretely, this generality helps because if we are designing hardware to play as X then to select a good move that hardware must consider how it could complete three X's in a row to win, or how it's opponent O might win by doing completing three O's in a row. Each instance of module `TwoInRow` selects if player A has two squares out of three where the third square is open. For example, the instance `topr` on line 109 checks whether player A has two squares in the "top row" and the third square in the top row is empty.

The curly braces "`{}`" syntax used on lines 114 to 122, 125, 126, 130 and 131 means "concatenation" of signals. The concatenation operation `{a,b}` creates a larger bus out of the two smaller busses `a` and `b` by including wires from both `a` and `b` where the wires from `a` come before the wires for `b`. For example, the code "`{ain[6],ain[3],ain[0]}`" combines three 1-bit signals into a 3-bit bus in which the leftmost wire's value is determined by `ain[6]`, the middle wire's value is determined by `ain[3]`, and the leftmost wire's value is determined by `ain[0]`. The concatenation syntax is also explained on slide 33 of Slide Set 2 (slide number refers to slides posted before lecture), Section 7.3 of Dally and online here http://www.asic-world.com/verilog/operators2.html. Furthermore, as "`{ain[6],ain[3],ain[0]}`" appears inside the module instantiation statement "`TwoInRow leftc({ain[6],ain[3],ain[0]},...`" the 3-bit bus defined by the concatenation "`{ain[6],ain[3],ain[0]}`" is connected to the first input, `ain`, of the instance `leftc` of module `TwoInRow` declared as "**`input`** `[2:0] ain`" on line 136.

A point that can be very confusing if you have not had a lot of experience programming is that in Verilog we can use the same name, `ain`, in *different* modules to refer to *different* signals. For example, in the case above we have a 9-bit signal `ain` declared on line 102 which is completely separate from the 3-bit signal with the same name `ain` declared on line 136. Indeed, both exist simultaneously in the design as they are part of different module instances – e.g., the 9-bit signal `/TestTic/dut/winx/ain` and the 3-bit signal `/TestTic/dut/winx/topr/ain`, which you can plot together in the wave viewer in ModelSim (see the video links above in Step 2 if you didn't already).

Finally, the assign statement on line 129 bitwise-ORs all possible tic-tac-toe positions where player A can complete three in a row.

2.3.4 TwoInRow module

Next, look at the code for `TwoInRow` on lines 134 to 142. This code uses Boolean expressions to check each of three possible ways in which player A can have two squares out of three with the third square left open (not held by player B). It outputs a 1 in the bit positions where A can play to complete three in a row. For example, the assign statement on line 139 checks whether neither A nor B has played at position 0 and that A has played at positions 1 and 2.

2.3.5 Empty, Select3 and RArb modules

Next look at the code for `Empty` on lines 145 to 151 and `Select3` on lines 153 to 162. Both modules instantiate module `RArb` which is a "reverse arbiter". To understand how these modules work, it first helps to understand the hardware behavior described by the code for module `RArb` on lines 80-86.

The module `RArb` is a reverse priority encoder similar to the priority encoder described in slide 50 of Slide Set 2 (slide number refers to slides posted *before* lecture). The syntax "**`parameter`** `n=8 ;`" declares a module parameter `n` which acts like a constant for any given module instance. The value 8 can be overridden with different values during module instantiation as is done on line 149, where the value of `n` is set to 9 via the syntax "`#(9)`" and on line 159 where the value of `n` is set to 27 via the syntax "`#(27)`". Parameters are described on slide 48 of Slide Set 2, Section 8.2 of Dally, and online here: http://www.asic-world.com/verilog/para_modules1.html. The two assign statements on lines 84 and 85 both describe combinational logic. Each bit position of the bus on the left-hand side of the equals sign is assigned to the value of corresponding bit in the expression on the right-hand side on the corresponding line. For example, if `n` is 8, then `c[7]` is assigned the value `1'b1`, and in parallel `c[6]` is assigned the value "`~r[7] & c[7]`", and so on with the final parallel assignment for line 84 being `c[0]` being assigned the value "`~r[1] & c[1]`". The way module `RArb` works is best understood by drawing out the logic with individual AND and NOT gates. The result should look similar (though not identical) to the arbiter circuit in Slide Set 2 on slide 49 and 50.

At a high level, the operation of RArb is as follows: The input is a set of "requests" `r`—one request per bit of `r`. The output `g` is a set of "grant" signals. If `r` is not all zeros, then a single bit of `g` will be set to 1. Which bit? The bit of `g` that will be set to 1 will be the first bit of `r` that is set to one starting from the highest index bit position in `r`. Note that `r` is declared as "**`input`** `[n-1:0]`". This means it contains n bits with index values from $n-1$ for the leftmost bit down to 0 for the right most bit. By default n is set to 8, but we can change n when we instantiate the `RArb` module. For example, using the notation

"`RArb #(9)`" we change n to 9 when we instantiate `RArb` inside the module `Empty`. Suppose now that input `r = 8'b00101111`. Then, the bit with highest index, bit 7, has a value of `1'b0` and the bit with lowest index has value `1'b1`. The output `g` will be `8'b00100000`. Notice that this is a so-called *one-hot* code, which is a binary code with only one bit position set to logic value 1. If all request inputs `r` are logic value 0, the output grant `g` will also be all 0. You might want to try creating a testbench script and simulating different inputs to just the `RArb` module to check if you understand how the output `g` depends upon the input `r`. The Verilog for `RArb` is also described in Section 8.5 and Figure 8.31 of the Dally textbook.

Returning to module `Empty` we see it will output a one in the first position with a 1 on input `in`. The code "`{in[4],in[0]`..." ensures that if position 4 is *not* occupied it will be selected first, then position 0, and so on. Position 4 corresponds to the middle of the tic-tac-toe board, which is typically considered the best opening move. The module `Select3` prioritizes a move from input `a` over a move from `b` over a move from `c`. At line 97 we see `a` is connected to `win` and `b` is connected to input `block`. So, `Select3` will prioritize winning over blocking.

At this point you should be able to understand the simulation waveforms in the ModelSim waves window. A reminder the code in `tictactoe.v` is also explained in Section 9.4 in the Dally textbook and you are welcome to discuss this starter code (i.e., code provided for Lab 3 by the instructor) with any other CPEN 211 student (even if they are not your lab partner).

2.4 Step 4: Quartus and DE1-SoC.

Launch Quartus and create a new project (refer to the Lab 1 handout if needed). Be sure to use the directory lab3 created when unzipping the provided files as the "working directory". Otherwise, you will get errors about missing memory initialization files, and you may miss some files when you submit your code for marking. Call the project "lab3" and be sure to set the top-level module to "`lab3_top`". Add all the provided Verilog (.v) files to the project.

Once you have created the project, load the new pin assignments file, "`DE1_SoC.qsf`", that is provided with Lab 3. You will be using this new pin assignments file for the rest of the term and likely for the rest of your courses that use your DE1-SoC.

Compile the project and download the design to your DE1-SoC as you did for Lab 1. Connect your DE1-SoC to a VGA monitor (e.g., in MCLD 112) as we walk through the user interface below.

3 Understanding the Tic-Tac-Toe DE1-SoC user interface

The user interface has two display components, a VGA output which you can connect to any VGA monitor including the ones in the lab in MCLD 112, and a simpler interface using the red LEDs on the DE1-SoC. The switches SW8-SW0 provide the input. Initially, all the switches should be in the "off" position (down). In that case you should see an empty 3x3 grid on the VGA monitor like that shown in Figure 3a. On the DE1-SoC itself, you should see LEDR8-LEDR0 on.

The 9 lights indicate all 9 board positions are available to us. The correspondance between switches, LEDS and board positions for X (you) are given n Figure 3b. The numbering is reversed relative to Figure 1 to make playing the game more intuitive and fun on your DE1-SoC.

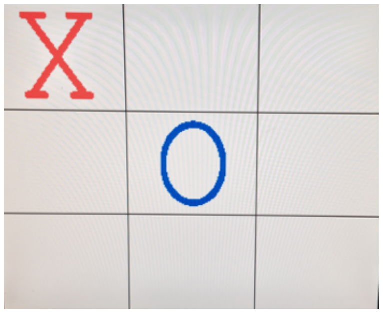
Suppose we decide to start by playing an X to the upper left position on the board. This corresponds to SW8. Moving SW8 to the "on" position, we end up with the VGA looking like Figure 3c. Note there is a red X in the top left corner of the grid representing our move. However, there is also a blue O in the center square on the VGA. The blue O in center was played by the game logic (i.e., module `TicTacToe`).

Continuing along we can defeat the game logic using the following set of moves: First, put an X in the bottom right corner by setting SW0, then the top right corner (SW6) and finally middle right (SW3). See Figure 3d through Figure 3f. Later in this lab you will implement two changes: detecting when either X or O wins and improving the game logic so that O does not lose so easily. To reset the board and play again, move all the switches back to down and press KEY0.

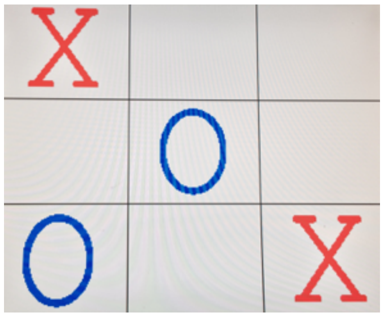
(a) Initial state on VGA. Ensure that SW8-SW0 are all off (down).

SW8	SW7	SW6
SW5	SW4	SW3
SW2	SW1	SW0

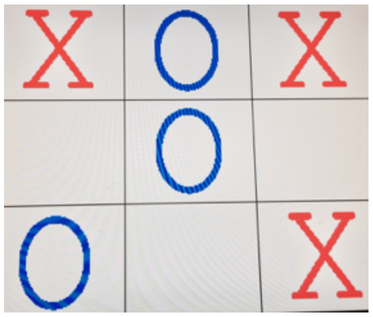
(b) Board position versus switches (NOTE: you play "X" and move first).



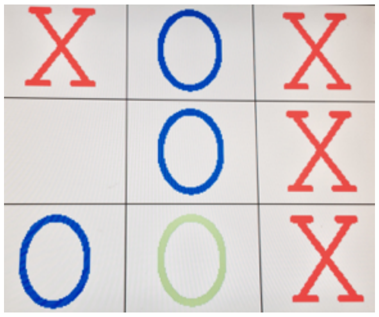
(c) You play X. If you place your first X in the top left (moving SW8 to the up position) the game logic responds by playing O in center.



(d) 2nd move (SW0)



(e) 3rd move (SW6)



(f) 4th move (SW3)

Figure 3

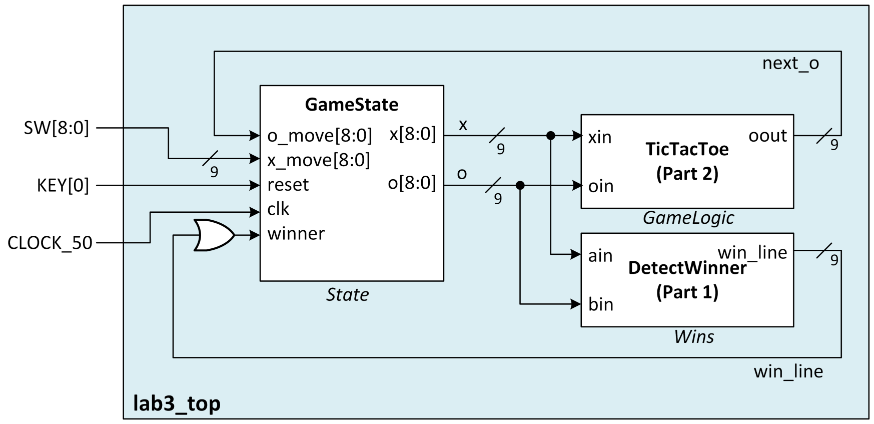


Figure 4: lab3_top module logic (VGA and LED logic omitted)

4 Understanding `lab3_top.v`

Before completing the first part of the assignment (detecting a win), it helps to understand how the code you will write is interfaced to the `TicTacToe` module when it is instantiated on your DE1-SoC.

Figure 4 illustrates the logic inside `lab3_top.v` in the module `lab3_top` which is the top-level module you should be using when downloading your design to the DE1-SoC. This figure omits the details of the VGA interface and output to the red LEDs as you do not need to understand these to complete the lab. The block labeled "State" has logic that remembers the moves played previously by both X and O. You do NOT need to understand that code that implements the State block (it is in `game_state.v`).

The block labeled "Wins" is a combinational logic block that you will design! It needs to detect whether *any* row, column, or diagonal has three X's or three O's in a row. If so, it outputs a logic '1' on a wire corresponding to that row, column or diagonal. The code for "Wins" is inside `detectwin.v` and you will add Verilog in that file for the first part of the lab.

Given the current positions of X and Y on the board, the GameLogic block outputs a new move for O on the wire labeled "`next_o`". You will modify GameLogic so it plays better in the second part of the lab.

5 Lab Procedure and Marking Scheme

This section outlines the changes you need to make to the code provided along with how you will be graded. You must submit your code via `handin` by 11:59 PM the night before your assigned lab section as described in Section 6 AND both partners must show up to your assigned lab section prepared to demo as described in Section 7. If you find yourself spending significant time due to "bugs" please review the video from Lab 1 describing how to find syntax and simulation (design) errors here: https://youtu.be/2c3CZouKJKs. Note you *must* use the *same* ModelSim and Quartus project files for Part 1, Part 2 and the Bonus (if you do it) or marks will be deducted due to the additional storage and marking overheads that otherwise result.

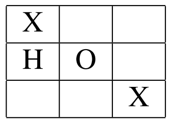
Part 1 [6 marks]: Detecting a "Win" The top level module, `lab3_top` (in `lab3_top.v`), has a bus called "`win_line`" that can be used to display a line through three adjacent squares of the same type (X or O). We have added an empty module declaration for "DetectWin" in detectwin.v that you should fill in for this purpose. You must create a testbench `detectwin_tb.v` to thoroughly test your implementation and in the wave window use "File -> Save Format..." to save the resulting waveform format as "`part1_wave.do`" in your top level project folder. Either you or the TA will re-run the simulation during your marking session using your submitted code; to save time you must have a `part1_wave.do` file. See the examples of creating a testbench in Slide Set 2, the module "TestTic" in tictactoe.v and/or Sections 3.6 and 7.2 in Dally. The 6 marks for this part will be computed as follows:

1 mark For answering the TA questions about how the TicTacToe Verilog we gave you works.

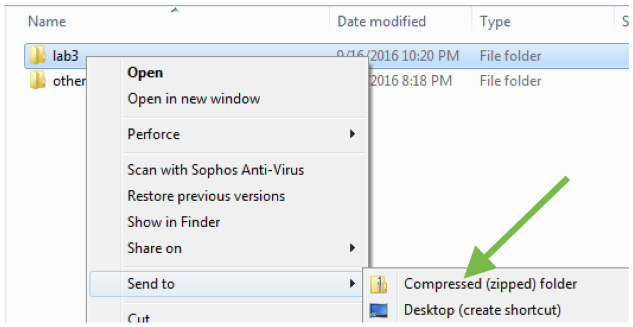
2 marks For explaining the synthesizable Verilog code you wrote in `detectwin.v`. NOTE: Your `detectwin.v` must conform to the style guidelines for synthesizable Verilog in the slide sets and must be free of inferred latch warnings (see summary at end of Slide Set 1 and in the appendix of Dally). Include a one (or more) line comment before every assign statement, always block, or module instantiation that you add or modify to explain what your Verilog code does (up to 1 mark may be deducted from this part if your code is not commented).

2 marks For explaining your testbench Verilog in detectwin_tb.v and your simulation waveforms and if your test script checks for all winning board configurations and at least five cases where there is no winner. Include a one (or more) line comment before every input test pattern in your test script explaining what is being tested and the expected outcome (up to 1 mark may be deducted for this part if this requirement is not met).

1 marks For demonstrating your design works on the DE1-SoC.



(a) Game logic improvement



(b) Creating lab3.zip.

Figure 5

Part 2 [4 marks]: Game logic improvement Add a `PlayAdjacentEdge` module with inputs `ain` and `bin` that, on a board that is empty except for two of your X's in two opposite corners and the hardware's O in the middle, predicts the best choice is to play O to an adjacent edge space (labeled H in Figure 5a). The goal of making this change is to improve the game logic so we cannot beat it like we did in the example above. The resulting game logic should either play you to a draw or beat you if you make a mistake. You will need to change the `Select3` module to a `Select4` module. The 4 marks for this part will be computed as follows:

2 marks Explaining your synthesizable Verilog code (must follow style guidelines and include comments).

2 marks Create a file `gameplay_tb.v` to test your implementation. Save your waveform format in `part2_wave.do`. You get up to 2 marks for these plus your explanation of your testbench and simulation waveforms and demonstrating your modified synthesizable code "works" on the DE1-SoC while satisfying the requirements for part 2 describe above. NOTE: You will need to think of how to convince the TA your design "works". You *must* add comments in `gameplay_tb.v`.

BONUS [1 mark] Add a component to the tic-tac-toe module that outputs a signal when the game is over and indicates the outcome. The signal should encode the options: playing, win, lose, draw. Hook these outputs up to seven segment LEDs on your DE1-SoC (they have signal names "HEX0" through "HEX5" in the `DE1_SoC.qsf` file).

6 Submission Instructions

ONLY Partner 1 should submit code for Lab 3 to 11—marks may be deducted if both partners submit as this requires additional teaching staff time to identify and address. The following assumes Partner 1 as indicated in the partner sign up form is logged into a computer in the lab in MCLD 112. Submitted files may be stored on servers outside of Canada; you may omit personal information (e.g., name, SN) from your code.

Create a directory "`Z:\cpen211\Lab3-<section>`" (e.g., `Z:\cpen211\Lab3-L1A` if you are in section L1A) and copy your entire project directory. You *must* include *all* files in your ModelSim and Quartus project folders (this will be 10's of MB and that is fine).

If your files are on your laptop, transfer them to this directory by first creating a ".zip" archive by opening "Windows Explorer" (Windows 7) or "File Explorer" (Windows 10), then clicking on the folder (e.g., lab3) so it is highlighted, then right-clicking on the folder, then selecting "Send to", then "Compressed (zipped) folder" on Windows (see Figure 5b). Then, send the resulting ".zip" file containing all files (including files generated by ModelSim and Quartus) to yourself at an online email account such as Gmail then opening a web browser on the lab computers and download and save the lab3.zip file to "`Z:\cpen211\Lab3-<section>`". [2] The .zip file will be several MB in size but we need all the compo-

[2] If you know Linux/UNIX you can also use the unix/linux program "scp" to transfer a file to your ECE account under the

nents to speed up the marking process during your lab session. Gmail and many other online email providers allow for attachments up to 25MB which should be large enough for a zipped copy of the directory containing all of your Lab 3 files. If your zip file is larger than your email provider allows, then use a USB key to transfer your files (but do not forget to take it when you leave MCLD 112).

Next, open up Cygwin on a lab computer by going to the start menu and typing "cygwin" and hitting enter or clicking on the Cygwin icon. In the Cygwin window type: `ssh <username>@ssh.ece.ubc.ca` after replacing "<username>" with your ECE account username. E.g., "ssh aamodt@ece.ubc.ca"). If you see a message such as:

```
The authenticity of host 'ssh-linux.ece.ubc.ca (142.103.83.22)' can't be established.
RSA key fingerprint is 8e:95:cc:cf:66:9b:da:0f:67:72:28:94:a1:f7:33:1a.
Are you sure you want to continue connecting (yes/no)?
```

type "yes" and hit enter. Type "`chmod go-rx ~/cpen211`" to secure your files. To submit type:

```
handin cpen211 Lab3-<section>
```

For example, if you are in section L1A, type: "`handin cpen211 Lab3-L1A`". You will be asked to acknowledge the following prompt:

```
PLEASE READ THIS STATEMENT CAREFULLY AND ENSURE THAT YOU UNDERSTAND IT BEFORE
SUBMITTING YOUR WORK.
By submitting these files, I indicate that I am fully aware of the rules and
consequences of plagiarism, as set forth by the Department of Electrical and
Computer Engineering and the University of British Columbia. I hereby certify
that the work in the submitted file(s) was performed *only* by me (the owner of
the account used to submit this work), except as acknowledged in the work
submitted.

Are you sure you want to continue? (y/n)
```

To overwrite your previous and/or trial submission: "`handin -o cpen211 Lab3-<section>`" (e.g., "`handin -o cpen211 Lab3-L1A`" if you are in section L1A)

7 Lab Demonstration Procedure

To reduce congestion in the lab we will be dividing each lab section into two one hour sessions. For example, for L1A the first session will run from 9 am to 10 am and the second session will run from 10 am to 11 am. We request that you show up no more than 10 minutes before the start of your assigned one hour "Lab 3 Marking Time", which will be posted on Connect at least 24 hours before your lab section along with your "Lab 3 TA". The TAs will have a randomly ordered list of lab partners and will start working their way down the list marking. If you and your partner are not present when they ask to mark you and you have not told them where you are beforehand, your name will be put to the end of the list. If this happens the TA will be under no obligation to mark you, but may do so at their own discretion and if time permits. Please note the TAs, several of whom are 3rd year ECE students who did very well in CPEN 211 last year, are not expected to stay past the end of their assigned lab sections and many may have classes they are attending, which start on the hour. We do not have enough TA hours to allow for any sort of "make up" sessions.

Your TA will likely have your submitted code with them and have setup a "TA marking station" where you will go when it is your turn to be marked. However, we still require that you bring your DE1-SoC, submitted code, and (if you have one) laptop to MCLD 112. This is in case either your TA did not get your submitted code in time for the lab (it takes time to organize the files), or because they ask you to use another workstation or your laptop for your demo to better manage time. If they ask you to demo using your own workstation or laptop, then you **must** demo the exact same code you submitted via handin so be sure you have a copy with you.

directory ~/`cpen211/Lab3-<section>` or "rsync" to transfer an entire directory and its contents. If you want to submit your code remotely from home before the lab, this is the best way.