This lab explores the connection between propositional logic and computation. It is due in class Friday, March 13 but earlier handins are welcome and encouraged.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Adders were discussed in class. The first exercise has you implement a 4-bit adder to get familiar with the tool you'll be using to create and test your circuits. The other exercises introduce several other computationally useful circuits — for each, you'll first build a truth table defining the computation and then implement it as a circuit.

- Use the xLogicCircuits tool to draw and test your logic circuits. URL: `https://math.hws.edu/eck/js/xLogicCircuits/xLogicCircuits.html`

  Click the link at the top of the page for instructions for using the tool — look over the Brief Instructions, Logic Gates and Logic Circuits, Subcircuits, and Buttons sections.

- Your diagrams should be neat and organized. You do not need to spend lots of time making them perfect, but try to avoid tons of crossing lines and follow the conventions that inputs go on the left side and outputs on the right.

- It is recommended that you create a directory `~/cs229` to hold your files for this class, and a subdirectory `~/cs229/lab1` to hold your files for this assignment.

- Create a new circuit (use the "New" button) for each exercise, then add on to it for each part within one exercise.

- Name your circuits using the "Title" box before saving. Name them exactly as directed and *do not* include spaces. By default, files are saved to your `~/Downloads` directory. It is a good idea to promptly copy them to your `~/cs229/lab1` directory.

- Hand in a hardcopy of your written answers. Hand in your circuits by copying your `~/cs229/lab1` directory (and its contents) to your handin folder `/classes/cs229/`*`username`*.

1. *Adders* were discussed in class, but the implementation described made use of XOR gates. xLogicCircuits has only AND, OR, and NOT gates.

   (a) Create a logic circuit for the XOR ($\oplus$) operation:

      (i) Write a complete truth table for the XOR operation.

      (ii) Write the DNF (disjunctive normal form) proposition corresponding to this table.

      (iii) If possible, simplify this proposition without introducing operations other than $\wedge$, $\vee$, and $\neg$. Show your work as a series of logical equivalences with justifications.

      (iv) Create a logic circuit corresponding to your (simplified) proposition. Test your circuit by turning the power on (the "Power" checkbox is checked) and making sure that the output is correct for every combination of inputs. Red indicates ON/1 values.

      (v) When your circuit is correct, save it with the title `xor` (all lowercase). Note that the saved filename will be `xor.txt`.

   (b) Iconify your XOR circuit, then use it to create a logic circuit for the full adder discussed in class. You can find the propositions on the second "Full Adder" slide from Wednesday's class. Save the full adder with the title `adder1`.

   (c) Iconify your full adder circuit, then use it to create a logic circuit for a 4-bit adder. (The "Adders" slide from Wednesday's class shows how to use two full adders to create a 2-bit adder — extend this idea.) Save your 4-bit adder with the title `adder4`.

2. *Comparators* are another fundamental building block in computer systems. They are used to tell whether one value is larger than, smaller than, or equal to another. This kind of comparison is useful in many familiar programming tasks, such as deciding which `if` statement branch to take, ordering items in a list, or choosing which task should be handled first. At the hardware level, multi-bit comparators are built by combining 1-bit comparators, making the 1-bit case an essential starting point.

   Consider a 1-bit comparator with inputs A and B. There are three outputs:

   - L: 1 if $A < B$ and 0 otherwise
   - E: 1 if $A = B$ and 0 otherwise
   - G: 1 if $A > B$ and 0 otherwise

   (a) Write a complete truth table which includes output columns L, E, and G.

   (b) Write the DNF proposition resulting from this truth table.

(c) If possible, simplify this proposition without introducing operations other than $\wedge$, $\vee$, $\oplus$, and $\neg$. Show your work as a series of logical equivalences with justifications.

(d) Create a logic circuit corresponding to your (simplified) proposition, test it, and save it with the title `comparator1` (all lowercase). If your proposition makes use of $\oplus$, start a new circuit (with the "New" button), load your `xor` file from the previous problem, iconify it, and go from there.

To compare two 4-bit numbers $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$:

- Starting with $i = 3$, compare A_i and B_i. If $A_i < B_i$ (i.e. $A_i = 0$ and $B_i = 1$, then $A < B$. If $A_i > B_i$ (i.e. $A_i = 1$ and $B_i = 0$), then $A > B$. Otherwise decrement i and repeat.
- If $A_0 = B_0$, $A = B$.

(e) Complete the truth table below to reflect this algorithm — fill in the L, E, and G output columns for the rows listed and add the missing rows. Omit rows with impossible combinations of inputs (for example, only one of L_3, E_3, and G_3 can be 1 at a time). It is OK to combine rows with the same output as shown — for example, if $A_3 < B_3$ (so L_3 is 1), then $A < B$ no matter what the other A_i and B_i values are. This is shown with the "0,1" entries rather than separate rows covering all the combinations.

L_3	E_3	G_3	L_2	E_2	G_2	L_1	E_1	G_1	L_0	E_0	G_0	L	E	G
1	0	0	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	1	0	0
0	0	1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1			
0	1	0	1	0	0	0,1	0,1	0,1	0,1	0,1	0,1			
0	1	0	0	0	1	0,1	0,1	0,1	0,1	0,1	0,1			

(f) Write the DNF proposition resulting from this truth table.

(g) Iconify your 1-bit comparator circuit, then use it to create a 4-bit comparator.

Create a logic circuit corresponding to this proposition, test your circuit, then save the result with the title `comparator4`.

3. *Parity checking* is a simple but widely used technique for detecting errors in digital systems. Memory systems, communication buses, and storage devices often use parity bits to detect whether a single bit has been flipped due to noise or hardware faults. While parity cannot correct errors, it provides a low-cost way to detect many common failures.

Parity checking works by computing an extra bit, called a *parity bit*, from the data bits so that the total number of 1s in the combined data and parity bits is even. This means that the parity bit will be 1 if the data bits involve an odd number of 1s and 0 otherwise. When the data is later read or received, the system recomputes the parity from the data bits and compares it to the stored parity bit.

(a) Write a complete truth table to compute parity for three data bits A, B, C. The output should be 1 if A, B, and C involve an odd number of 1s and 0 otherwise.

(b) Write the DNF proposition resulting from this truth table.

(c) Implement and test this circuit, saving it as `parity3`.

(d) Write a truth table which uses this parity computation to check the parity bit. There will be four inputs (three data bits A, B, C and a parity bit P). Add a column $parity(A, B, C)$ whose value is the computed parity for A, B, and C. The output should be 1 if $parity(A, B, C)$ and P are the same and 0 otherwise.

(e) Write the DNF proposition resulting from this truth table. Use P and the compound proposition $parity(A, B, C)$ as building blocks rather than A, B, and C directly.

(f) Iconify your parity computation circuit, then use it to create the parity checker circuit. Test your circuit, then save the result with the title `pcheck3`.

(g) Use your `parity3` circuit to build a 9-bit parity checker (for 9 data bits $ABCDEFGHI$ and one parity bit P): first create a circuit which uses four copies of `parity3` to compute $parity(parity(A, B, C), parity(D, E, F), parity(G, H, I))$, then compare that result to P as you did for `pcheck3`. Test your circuit, then save the result with the title `pcheck9`.