PyRTLSweeper: An attempt at automated transpilation of digital circuits to Minesweeper boards

Edwin Chang
College of Creative Studies
University of California, Santa Barbara
edwinchang@ucsb.edu

***Abstract*—In this paper I describe how to construct a set of digital circuit components in Minesweeper, and how to automate their generation. Then, I discuss an approach for laying out the circuit components on a Minesweeper board to make a circuit described using the Python library PyRTL, and automate this as well. The boards produced by the final result are really big and some of them don't work, but some of them do work which is pretty cool.**

***Index Terms*—Digital circuits, logic circuits, logic gates**

I. Note for those not in the course

This paper was written as part of a class project in Exploring the Hardware/Software Interface (CMPTGCS 130E Spring 2025) at UC Santa Barbara, taught by Zach Sisco.[1] I spent about 4 weeks on this project, and this paper documents my findings and what I managed to build in that time.

II. Introduction

Anyone reading this paper has probably played Minesweeper before. The player is given a board on which mines are hidden, and must deduce which cells are safe based on the number of mines each safe cell is surrounded with. A useful tool for analyzing Minesweeper boards is David Hill's JSMinesweeper.[2]

PyRTL[3] is a hardware description language created by UC Santa Barbara's ArchLab.[4] Using it, one can write digital circuits in Python with relatively simple syntax and do all sorts of analysis on them.

The goal of this project was to write a Python package that provides an algorithm to produce a Minesweeper board representing a circuit written with PyRTL. Since registers and memories seem to be a difficult task given that they require state preservation, I didn't implement them in this project. In this paper, I go through the necessary components needed to make PyRTLSweeper work, and how I wrote the final algorithm. The algorithm does not work in general for unknown reasons, but it does seem to work on some circuits.

[1]https://ccs.ucsb.edu/courses/2025/spring/exploring-hardwaresoftware-interface

[2]https://davidnhill.github.io/JSMinesweeper/

[3]https://ucsbarchlab.github.io/PyRTL/

[4]https://www.arch.cs.ucsb.edu/

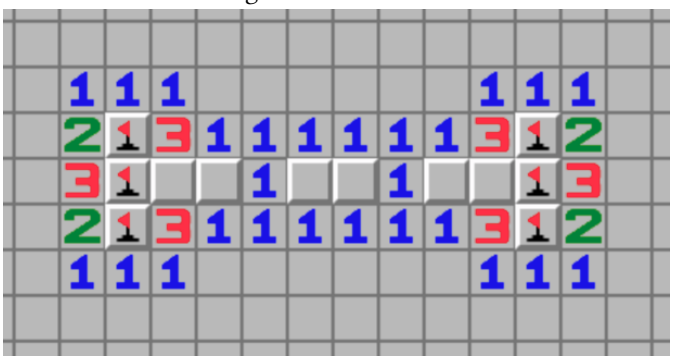
III. Background

There is prior work on building digital logic circuits in Minesweeper. Kirby703 demonstrated in her 2025 SIGBOVIK paper how she built Minesweeper inside Minesweeper using a repeated logic gate [1]. Richard Kaye wrote a couple papers that used logic circuits in Minesweeper to show that Minesweeper is NP-complete [2], [3]. Michiel de Bondt also did work using logic gates to analyze Minesweeper's complexity [4], and Seunghoon Lee found improved logical components in a hexagonal variant of Minesweeper [5]. I drew most of my inspiration from Kirby703's work in my project.

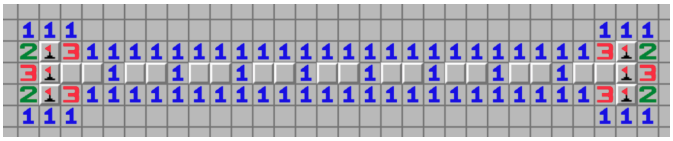
IV. Board components

A. Creating wires

Consider the following board.



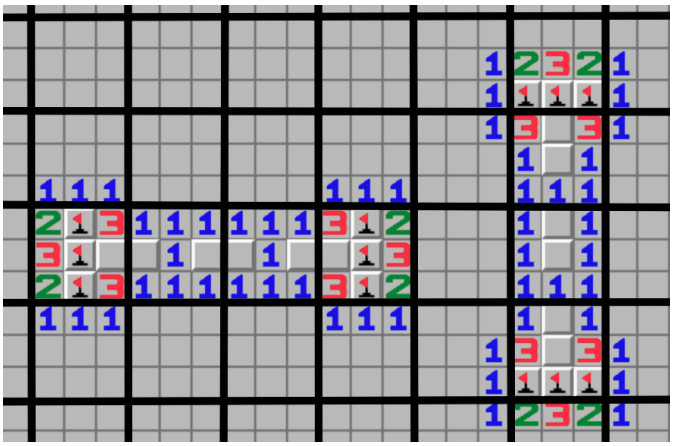
Given that there are 3 unflagged mines on the board, there are two possible ways in which the mines can be placed: one mine on the left side of each pair of hidden cells or one mine on the right side of each. Note how placing one mine determines the positions of the remaining mines. It follows that we can extend this pattern to be as long as we want, and there will always be exactly two possible sets of mine positions.



Since placing one mine on one side determines the mine positions all the way down the chain to the other side, this structure works like a wire.

B. Phases

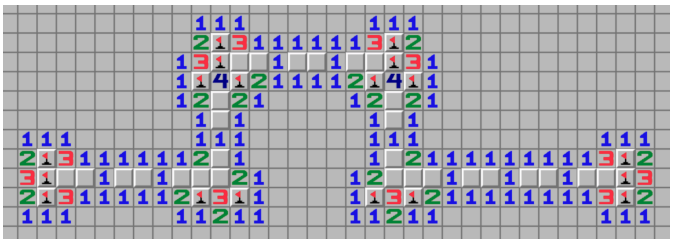
An important thing about wires is that the repeated wire component is repeated every 3 cells. This means that when we put together wires and logic gates, they need to be in the correct phase to connect to each other. Dividing the board into 3×3 squares simplifies constructing boards. For the purposes of my project, I divided the board and placed wires as follows:



With this setup, each 3×3 square can contain a repeatable wire component. Notice how the horizontal wire components have two separated hidden cells within the 3×3 so that touching hidden cells will touch over a border. I'll explain the reason for this later.

C. Turns

With wires and phases defined, we can make turn components:



If you ignore the numbers for now, each turn component is defined by hidden cells and flags contained within a 3×3 square. In fact, these can be tightly packed, so turn components can be directly next to other wire/turn components without interference.

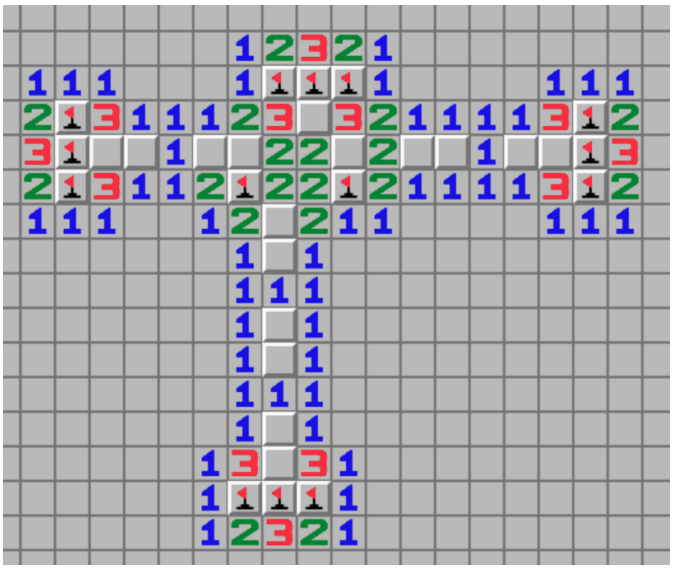
D. Crossovers

Possibly the most surprising part of wiring is that it's possible to make wires cross over each other without inteference with a 3×3 component.

This is why the horizontal wire component has two separated hidden cells: so crossovers work. I couldn't find a way to make them work with other horizontal wire phases while keeping the wires centered on the cross axis.

E. Splitters

This design takes an input signal from the left and reproduces the signal downwards and to the right.

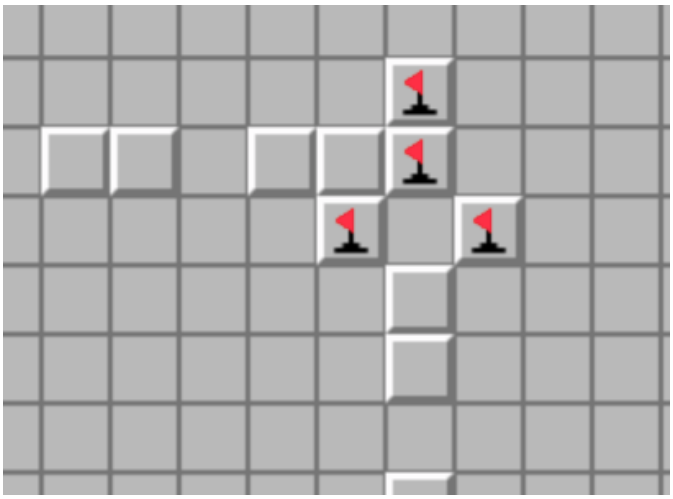


It looks like it takes more than a 3×3 square of space, but as I'll explain in the next section, it only takes one to represent. It might not be very packable though.

F. Auto generation

I still haven't talked about how to automate making any of this, which is clearly important to building actual circuitry. I wanted to make these boards viewable in David Hill's JSMinesweeper, so I had to indicate whether each cell in a board is hidden, has a flag, or is open, and its number if open.

My first attempt at scripting the generation of these wires was, for each component, to store the positions of hidden cells and flagged cells within the 3×3 square. A turn, for example, would look like this:



Then, note that whenever open cells are next to hidden cells, it is only for the purpose of 'connecting' the hidden cells together (forcing one to have a mine if the other doesn't and vice versa). Therefore, we have the following formula for computing the number to mark an open cell with:

$$\text{cell } \# = \text{adjacent flags} + \left\lfloor \frac{\text{adjacent hidden cells}}{2} \right\rfloor \quad (1)$$

Using this formula, we can fill in the numbers for all the open cells.

Then I realized that most of the flags can be computed on the fly as well. My second (and final) attempt involved storing any of the following for each position in a 3×3:

- A hidden cell
- A required flag
- A required open cell
- Nothing

Then the algorithm becomes as follows:

1) Place all hidden cells, required flags, and required open cells (open cells are differentiated from unknown cells).
2) For each unknown cell, if there is an even number of neighboring cells, make the cell open. Else, place a flag there.
3) Use the above formula to compute the number in each open cell.

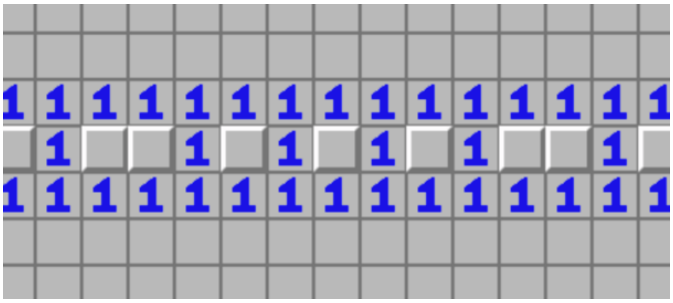
The advantage of the second approach is that for the most part, each component only needs to correspond to a set of hidden positions. Required flags and required open cells are only used in some special cases. In the splitter design in the previous section, the splitter component doesn't need to specify where any flags are, only the hiddens. The flags don't even need to be in the 3×3 square to get automatically computed. Since the hidden cells fit in the 3×3, I consider it a 3×3 component.

JSMinesweeper also wants to know how many mines are in a board, in order to analyze it. We calculate this from a final board using a similar formula to the one above:

$$\text{total mines} = \text{total flags} + \left\lfloor \frac{\text{total hidden cells}}{2} \right\rfloor \quad (2)$$

G. NOT Gate

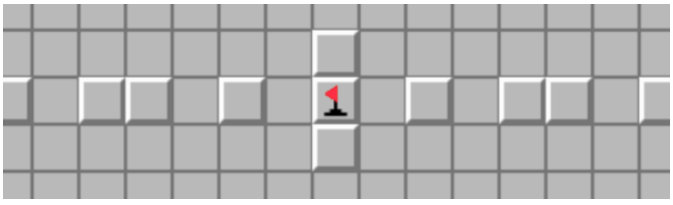
We now consider building an inverter. A naive implementation looks like this:



However, there's a problem here. When playing Minesweeper, the game tells you how many mines are left, so the number of mines needs to be constant. In this design, the center hidden cell is a mine in one board state and is safe in the other board state, but the hidden cells on its sides are both mines or both safe. This means that if the number of mines is known, the positions of the mines are determined, which is bad.

Here's the fixed inverter:

In this case, the number of mines in each board state is the same, which satisfies our needs. Also note that this makes use of a required flag (the one in the center) because even though it neighbors two hidden cells, those hidden cells shouldn't be connected. The other flags are implicitly placed because they neighbor an odd number of hidden cells. This gate is represented as something like this when scripting:



H. AND Gate

The most complex component needed is an AND gate. In theory we could use OR/NAND/NOR gates instead to make logic, but PyRTL provides a nice algorithm to convert everything in a circuit to AND gates. I took the design from Kirby703's paper and modified it to fit my constraints:

This takes up a 4×3 of 3×3 squares. I'm not going to get into how this exactly works, because I don't entirely understand it either. An important thing, though, is that the hidden cell at the far top right may look like an unnecessary dead end, but it's actually necessary to keep the number of mines constant, just like in the NOT gate.

V. Board Layout

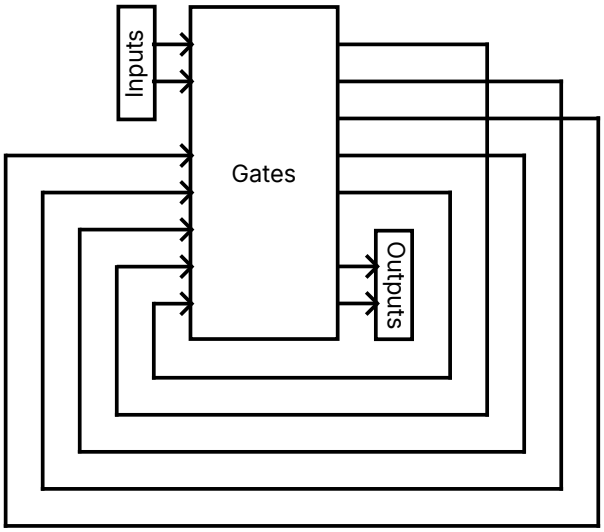
A. The problem

PyRTL provides nice APIs for analyzing circuits. Given a PyRTL circuit, PyRTL can produce the list of logic gates, list of wires, map of wires to logic gates powering them, and map of wires to logic gates they are input for. Additionally, PyRTL can reduce a circuit to 1-bit wires and logic gates, and further reduce it to only NOT and AND gates.

All we need to do is project the graph of logic gates and wires onto a Minesweeper board using the previously built components.

B. Laying out the components

The easiest way, to my knowledge, to project a directed graph onto a board like this is to put all the nodes in a line. Here's what the design roughly looks like:

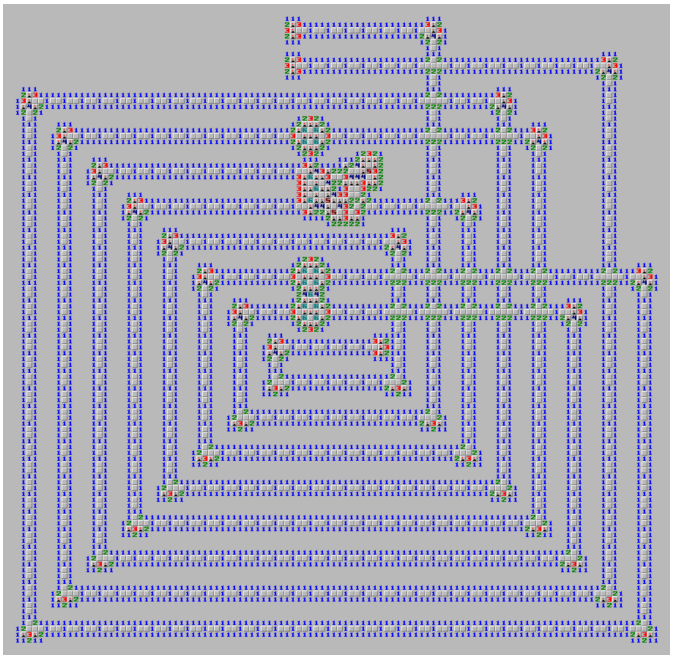


It's not pictured here, but wires can also split off in the top right if multiple gates take one wire as an argument.

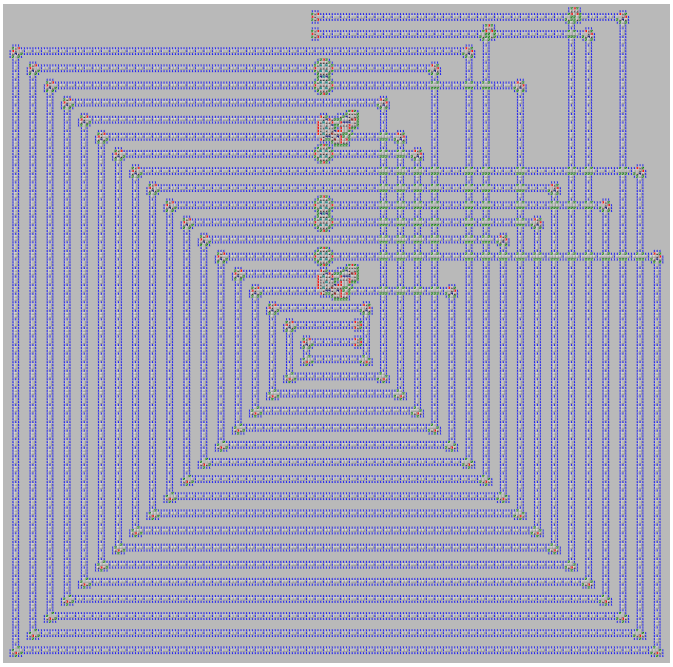
On the actual Minesweeper board, I add 3-square padding between the wires and gates, which are sized with 3×3 units, to make things easier.

C. Putting everything together

Using everything discussed before, I wrote a Python script to generate Minesweeper boards from PyRTL circuits. Here's what `c <<= a | b` looks like:



A more complicated circuit, `c <<= ~(~a | ~b); d <<= a | b`:



Board generation is not deterministic since it depends on ordering within Python sets, but the same graph is represented on each run.

On testing the circuits, most circuits I tried worked correctly, but for some reason `c <<= a ^ b` gives the wrong

outputs. I don't know why, but at least simple circuits made from a few AND, OR, and NOT gates seem to work.

VI. Results

PyRTLSweeper can generate Minesweeper boards from PyRTL circuits! Some circuits don't work for some reason, though, notably an XOR gate, and I haven't figured out why. It's probably either a small issue with my code making the wires get connected incorrectly, or a fundamental issue with the algorithm I've overlooked. If you can figure out what I did wrong, please reach out to me and I'll be happy to update this paper.

Also, the boards turn out to be quite monstrously large. But hey, some circuits work.

VII. Applications

If you find any, please let me know.

VIII. Future direction

There are a few ways in which I'd like to continue this project:

- Adding registers/memories (if possible?)
- Making the board more compact, by using a better graph projection algorithm and/or tightly packing wires and gates
- Making a custom editor/analyzer for circuits, since my web browser does not enjoy analyzing boards with 3000 mines
- Making a Typst[5] package for typesetting Minesweeper boards so I can format future papers better

Acknowledgements

I thank Zach Sisco for being an awesome teacher this past quarter; I had a lot of fun working on this project and exploring other weird ways to compute things. I also thank Kirby703 for submitting her work on Minesweeper circuits to SIGBOVIK, as it was among my favorite papers in the conference this year and I learned a lot of this stuff from that paper. In addition, I thank David Hill for creating JSMinesweeper, which I used extensively in this project. Finally, I thank Jonathan Balkind for introducing me to PyRTL, and UCSB's ArchLab for creating PyRTL.

X. Appendix: Running the code

The source code is available online at `https://github.com/EdwinChang24/pyrtlsweeper`. To try it, clone the repo and run `examples/circuit.py` with a Python runtime (I recommend `uv`[6]). It will produce a `.mine` file which you can drag and drop into JSMinesweeper. The main source code is in the `src` directory, although it may not be very readable. If you have any questions, feel free to open an issue on GitHub or email me.

[5] https://typst.app

[6] https://docs.astral.sh/uv/

References

[1] Kirby703, "Building Minesweeper in Minesweeper," in *SIGBOVIK 2025*, Mar. 2025. [Online]. Available: https://sigbovik.org/2025/proceedings.pdf

[2] Richard Kaye, "Minesweeper is NP-complete," *The Mathematical Intelligencer*, vol. 22, no. 2, 2000.

[3] Richard Kaye, "Some Minesweeper Configurations," May 31, 2007. [Online]. Available: https://web.archive.org/web/20231102222553/https://web.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[4] Michiel de Bondt, "The computational complexity of Minesweeper," Nov. 27, 2024.

[5] Seunghoon Lee, "A Short Note on Improved Logic Circuits in a Hexagonal Minesweeper," Nov. 14, 2021.