Deep Blue and Gold

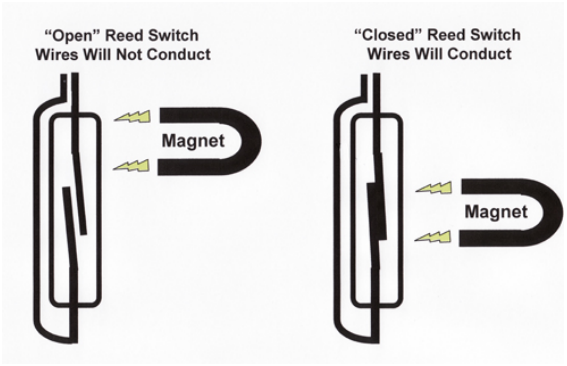
EE149 Final Project by Sean Scofield, Frank Lu, and Arthur Jeng

Project description:
We built a smart chess board that provides both an AI for a human player to play against as well as legal moves and hints for the player. The transparent chessboard has 8 Adafruit Neopixel strips attached to its underside that are used to light up each of the squares on the board.

Hardware:

- Arduino Mega 2560
- 2' by 2' Lexan polycarbonate board
- 64 magnetic reed switches
- 8 Adafruit Neopixel strips with 24 lights each
- 5V, 4A power supply
- 32 3d-printed chess pieces
- 32 neodymium magnets
- 8 8-stage static shift register (CD4021B)

Sensors and Inputs:
Reed Switches
The magnetic reed switches make up the entirety of the **sensors** for this project. Each reed switch is attached beneath a square on the board, and each row of 8 reed switches is wired to one of the 8 daisy-chained shift registers.



How a magnetic reed switch works

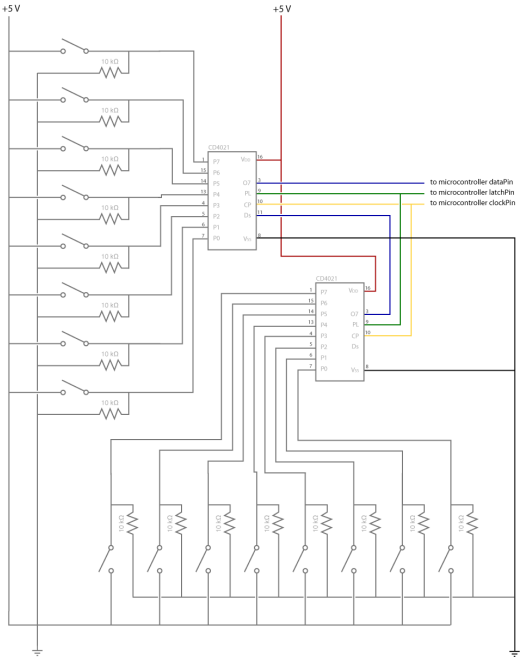
Each chess piece has a magnet in it, so whenever a piece is on a square, the switch beneath that square closes. The state of the reed switches (and therefore information about the current state of the pieces on the board) is then read through the shift registers, and sent to the Arduino.

The state of the chess board is read once per iteration of the *Loop* function in the Arduino (each iteration takes roughly 0.1 to a few seconds. The Arduino, which is the actuator in this case, receives each reading as a 64-bit binary string (i.e. 1001010100100010…). This is the **input** to our FSM.

Each of the chess pieces are 3d-printed such that magnets can be placed in each one.

Shift Registers
We need to use shift registers because of the issue of scalability. We have 64 reed switches but not 64 GPIO pins. Limited GPIO pins can either be resolved by a multiplexer shield on our Arduino or shift registers. We decided to use shift registers because

1. Shift registers can be daisy chained, meaning that we only need 3 GPIO pins whereas multiplexers will take more pins.
2. we can get all the data pins at the same time simultaneously despite multiplexers being faster.



Snippet of the circuit diagram of the shift registers and voltage dividers.

Each shift register consists of 8 digital input pins, 3 serial-out pins, 1 serial-in pin, 1 clock pin, 1 serial control pin, 1 supply voltage pin, and 1 ground pin.

The input pins are **asynchronous parallel input** pins. It is "parallel" because the shift registers collect the information on the pins all at once; it is "asynchronous" because the shift register does all the data collection at its own pace without coordinating with the Arduino.

The **clock** pin is the metronome of the conversation between the shift register and the Arduino. We set the clock from from LOW to HIGH to notify the shift register to change the state of the serial output pin.

The **parallel to serial output** pin is a "**latch**" pin. When the latch pin is HIGH, the shift register collects the data on the 8 pins. When it's LOW, it listens to the clock pin and sends the data serially.

Real-Time Behavior:

The real time behavior aspect of this project is making sure that our model represents the real world as closely as possible. For our project, this means that we continually poll the state of the board. In each *loop()* of our code, we continually call *getReading()* to get the 64 bit state of the board.

Finite State Machine:

Before going into detail on our FSM, it is worth noting that our system is **discrete**, meaning it has a countable number of states. The inputs to our machine are piece positions, and there is a only a finite number of chess positions. The outputs are integers representing indices of Neopixel LEDs to light.

The overall problem is a conceptually simple one: two players (one being an AI) make moves completely sequentially, with the guard checks being implemented as *isPieceLifted, isPiecePlaced, isLegalMove*. However, as we soon found, we were met with many challenges in our FSM designs and redesigns to account for "real-world behavior" being accounted for in our model.

As shown in Figure x, our FSM is deterministic: the same sequence of moves will lead to the same sequence of states. An example run of our model once turning on our microprocessor would be:

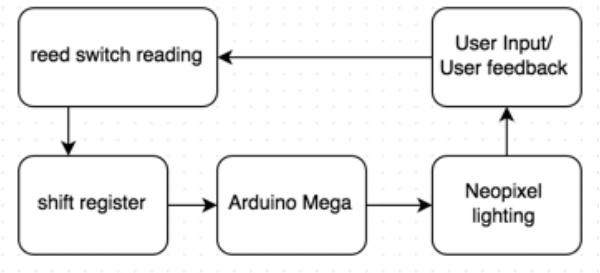
- The pieces are not set up correctly, so the model continually stays the *Initial* state.
- Once *isLegalStart* returns true, we move into the *Ready* state. If the ready switch is on, we go into the meat of the FSM.
- *StartTurn* directs our model into a human or AI sequence based off the *turn* global variable.
- In human sequence, we must be able to know the real world state at all times. Thus, a given move is broken into *PieceLifted* and *PiecePlaced*. This gives the model more insight into what is actually happening.
- The AI sequence lights up squares where it wants to move, and it is up to the human player to move it.
- At any point, invalid moves or checkmates make the model go into *InvalidMove* and *Checkmate* respectively.

This is a relatively simple path conceptually, but there are many ways the model can be out of sync with what is happening. For example, the reed switches will sometimes flick on and off when a piece is being set. At the end of the first flick, the model will be going into the AI sequence, but this state's getReading() call could be reading an unwanted flick of the reed.

We encountered many (seriously, a lot!) of these "real-world" problems, and we had to tackle these in creative ways in our FSM. To tackle the problem described above, we had a delay going into the *PiecePlaced* to wait for any flickering to die down.

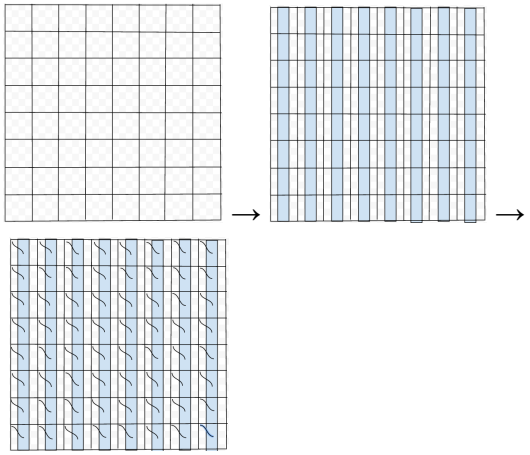
parallel/ **Design Methodologies**

Model



We begin this project by constructing a model that translates serial inputs of the shift registers from reed switch readings to neopixel lighting on certain segments. We then use discrete dynamics with emphasis on the FSM to model different modes of operations such as piece lifted, piece placed, generating AI move, and so on.

Design



Starting from an 8x8 square chess board, we first attached a Neopixel strip to each column (represented in blue). Then, we attached a magnetic reed switch to the bottom of each square (represented by the curved lines)

In our design phase, we focus heavily on scalability, immediate feedback, and modularity. In terms of scalability, we use neopixel strips instead of LED's so we can control the entire row instead of a single square. We also use shift registers to conserve our digital IO pins. We decided to use lexan polycarbonate to make sure the lighting is properly dispersed to give the user a clear output. In terms of immediate feedback, we continuously poll the entire board on command with parallel input pins. Finally, our choice in using the Arduino Mega gives us a lot more library support for neopixel strips, tutorials for shift-in registers, more GPIO pins, and more hardware capabilities (processing power and memory).

Testing

We verify the system workflow modularly by unit testing each step. For example, we first made each reed switch is not defective by using ammeter to detect a closed reed switch. After that, we checked if each shift register is functional by writing unit tests that output bitstreams of each rows' readings. Then, the mapping of the bitstream to the square on the board is tested with neopixel lighting. Finally, knowing that individual components all work together flawlessly, we are able to test our finite state machines. The lighting conveniently plays a role of a live debugger for each state transitions.

Modeling our project:

Chess AI:

After perusing various chess libraries, we decided we needed to build our own chess AI to expose certain functions needed in intermediate steps of deciding optimal moves. This allowed our FSM to react accordingly to as small discrete steps as possible.

In order to generate hints and AI moves, we use a minimax game tree representation and give each chess piece relative value corresponding to its relative strength in potential exchanges.

Symbol					
Piece	pawn	knight	bishop	rook	queen
Value	1	3	3	5	9

Then we incorporate alpha-beta pruning to improve the naive minimax.

In order to generate hints, we built a function into our engine called *generateMovesForSquare*, which takes as arguments a square, a color (black or white), and a game board (represented as a string), and returns a list of possible moves.

Takeaway and Future Work:

This project put to practice many of the concepts we have encountered in this course (most notably, FSM design), and showed us firsthand the difficulty of building a high confidence system with real-time and concurrent behaviors. We also gained experience in writing clean and modular code with a group of people.

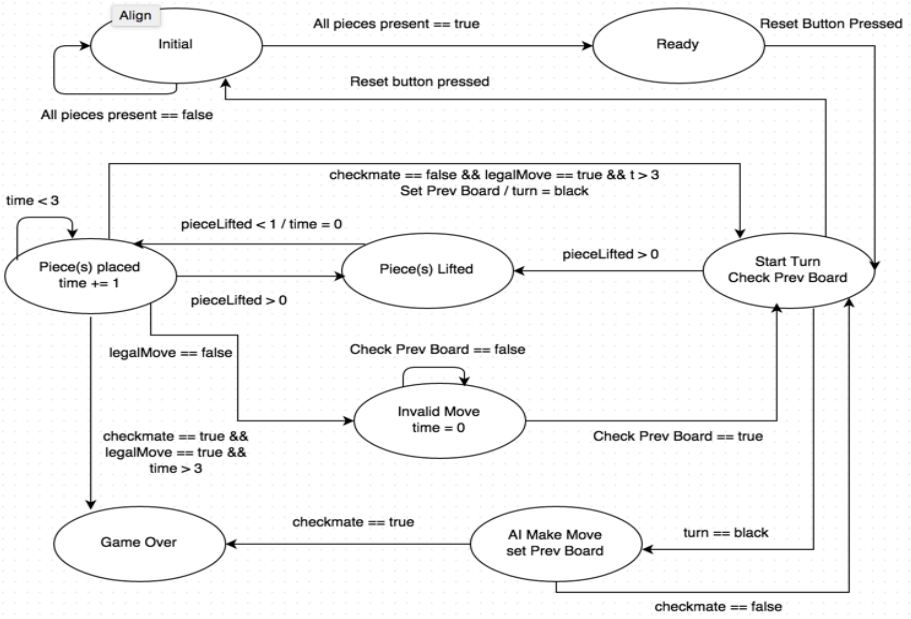
One big takeaway we got from this project was that hardware decisions that may initially seem minor can have a large impact on the difficulty of implementing a hardy finite state machine. One example of this was the size of the squares on the chessboard. We sometimes encountered issues in which one piece was activating two magnetic reed switches because it was close enough to another square to do so, thereby confusing our FSM.

In addition, using RFID instead of magnets in order to detect pieces would provide us with more precise information, making our FSM implementation easier and more reliable.

In the future, we still have an interest in building actuation around our chessboard. We also started

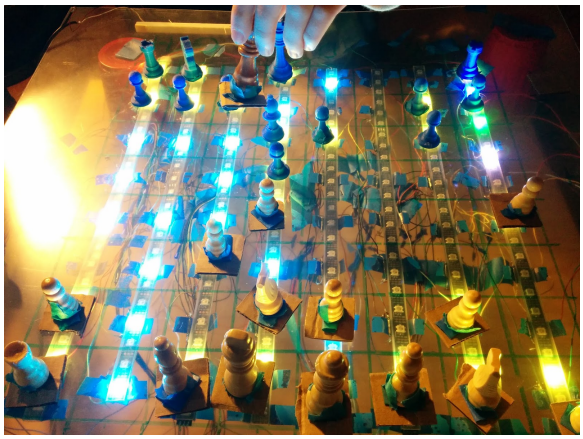
creating a web application that can run the AI portion of our code. The web application could also allow a user to play moves even if they are not near the board.

Thanks to everyone in the class who helped us throughout the project: Professors Edward A. Lee and Alberto Sangiovanni Vincentelli, GSI's Antonio Iannopollo, Ben Zhang, and John Finn, and the classmates around us!



Inputs: reading: { 64-bit binary string }
Outputs: Neopixel Lighting: { AI Move, Legal Moves, Error Message}

Note: the guards displayed in our FSM above do not explicitly mention the 64-bit binary string reading. Implicitly, the boolean values associated with these guards (i.e. pieceLifted) are calculated behind the scenes using the reading at each tick. There was simply no room in the diagram to show all of this.



Final Pictures