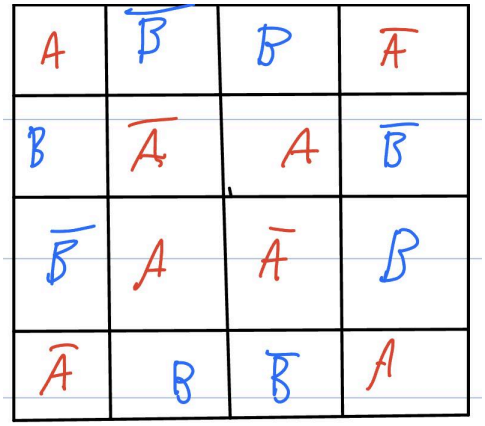
Group Members: Natalia Perey, Hong-Ye Wang, Gregory Chekler, Matthew Karazincir
Idea: **Sudoku Game**

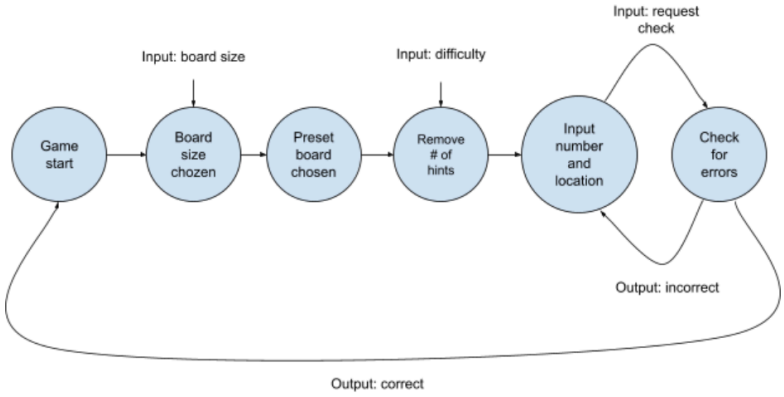
We propose designing a chip that allows the user to play a digital version of the game sudoku. On startup, the user will have the ability to choose their difficulty leveling for the following game. Once the game begins, they will be given a random 4x4 (or 6x6) sudoku board from the available preset boards that are hard coded in. This sudoku board will contain pre-placed numbers on the board ranging from one to three numbers, which will be determined by the selected difficulty level. The user will also be given a fail (attempt) counter that tracks the amount of mistakes the user has made in correctly placing the numbers. This initial fail count will be determined by the difficulty level chosen before the game as well, with the fail count ranging from one to three. There will be two conditions for the end of the game. The first will be if the user correctly places the final number on the digital sudoku board. The second will be if the user fails again after their fail counter reaches 0. If either of these conditions are met the game will end and the user will have the chance to select their desired difficulty level once again and play the game again. Some potential features may include the ability to request a limited number of hints and a timer that shows how long it took to complete the sudoku.

Based on current theories of implementation, we expect to have a word length of 2 bits. This is due to needing to represent the numbers one to four in our 4x4 sudoku game. To hold each of these distinct words in memory, we would need 16 internal registers because we have 16 boxes whose individual values need to be manipulated during the game. In this case, our total memory would be 32 bits for a single instance of our digital sudoku game. **Revised (2/17):** However, we can half our memory usage by determining pre-set values: What this means is that when we have our “subgrids” a.k.a. our smaller grids within the 4x4 sudoku board, we can pick two random spots in each of these 2x2 subgrids to which we assign values *A* and *B*: what value *A* and *B* are is determined by RNG, to which we can map a counterpart to each value by just inverting the value’s bit representation, which when scaled to the whole board, effectively halves the memory usage. *A* would either be given the value 00 or 11, and *B* would be either given the value 01 and 10; so by only having 2 of these values, we can cover all 4 in the 2 bit sequence space while only having memory for 2. Observe that this pattern can be scaled to a 6x6 or 8x8 sudoku grid. This allows us to generate presets that have definite solutions randomly.

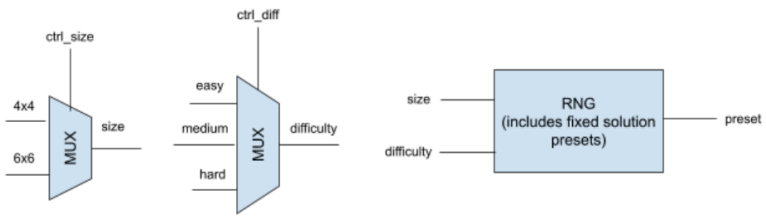


An algorithm we are thinking of is taking 3 existing values and finding our fourth by doing a 3 bit XOR: Take A and B in the top left and not(A) in the third row, and we can find C in the first column by doing the XOR of A, B, and not(A). This algorithm is generalizable to any instance, since it works for every combination of 3 unique 2 bit sequences out of the 4 that exist.

A way to check the solution would be to perform logic comparison between the user input and the preset solution at the cell. Inputs to the data path will include both the user input at every turn and input for the difficult setting prior to game start. Outputs to the data path will include the visual display representing the current state of the game and a visual display of the fail count for users to know if the previous input was valid. Due to the selection of a 4x4 sudoku board, the total memory used will only be 32 bits theoretically. However, because there are other valid sudoku boards sizes we can expand our design to a 6x6 board. In this case, our word length would increase to 3 bits because we will need to represent the numbers one through six in our 6x6 sudoku game. To hold each of these distinct words in memory, 36 internal registers would be necessary. The total memory would be 108 bits for a single instance of this 6x6 sudoku game. **However, since we are using presets, we can actually make sure that our number of registers is significantly smaller. Due to this, we can make the size larger and include more functionality. Thus, the sudoku game can go through all of the stages of generating the game, giving an input, and solving.** The memory inputs, and memory outputs for the 6x6 sudoku game would remain nearly the same as the 4x4 sudoku game. In terms of I/O pins necessary for the user number input, we will need 2 I/O pins for each digit since there are only two significant bits in terms of representing the numbers one to four. This means in terms of simply representing the numbers one to four we will need 8 I/O pins. We will also need 6 I/O pins for the difficulty level, 2 I/O pins for the size of the game, and 6 I/O pins for the fail count. Additional input pins would also be needed to specify the location of the grid, which would be 4 bits for one through sixteen or 6 bits for one through thirty-six depending on the game size. Essential pins also include the output pins that would be responsible for the display of the board and all of its intricacies. As we are nearing the end of the design, we plan to test the design using a more intricate display that allows for players to see the already placed numbers, the currently selected number, and the failure count. Based on current estimation, the amount of total pins exceeds the amount allotted by the project. How the game is generally played is below.



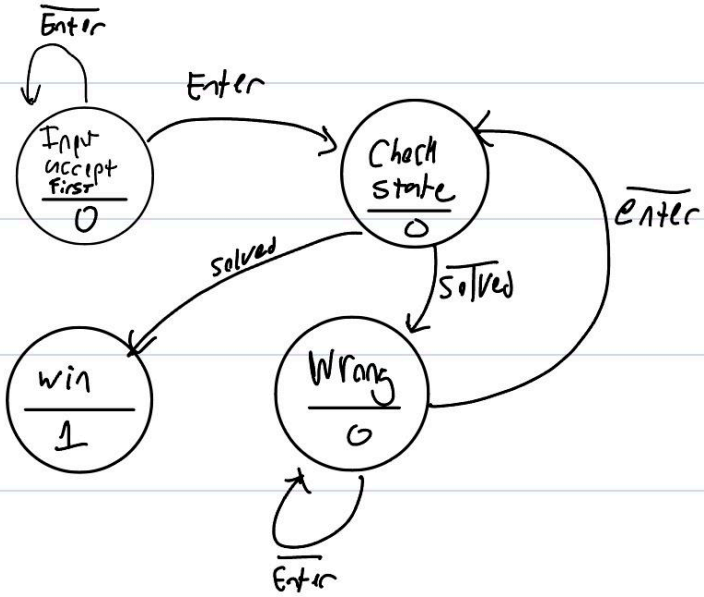
State Diagram Rough Draft



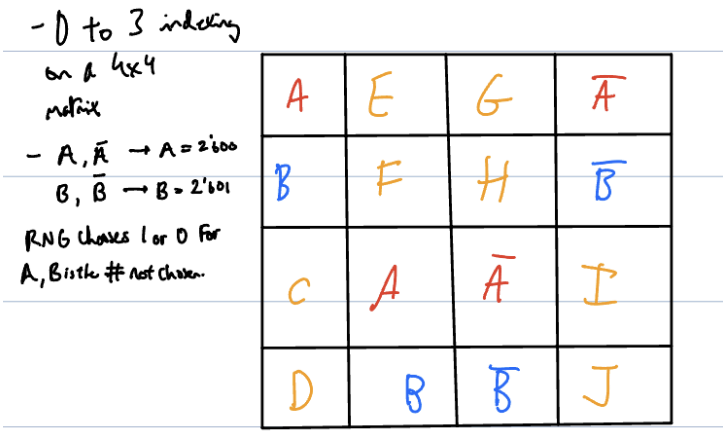
User input FSM

Enter = 1 ← when user inputs data

output 1 if win



We can start with a “maximum” A and B (Red and Blue) grid which is solvable and increase difficulty by randomly removing 0 to 3 of these “base numbers” - difficulty increments as we increment the number of red/blues taken away. 0 taken away corresponds to Easy, 1 to Medium, 2 to Hard, 3 to Very Hard.



This again can be scaled for 6x6 and 8x8 grids.

FSM Controller

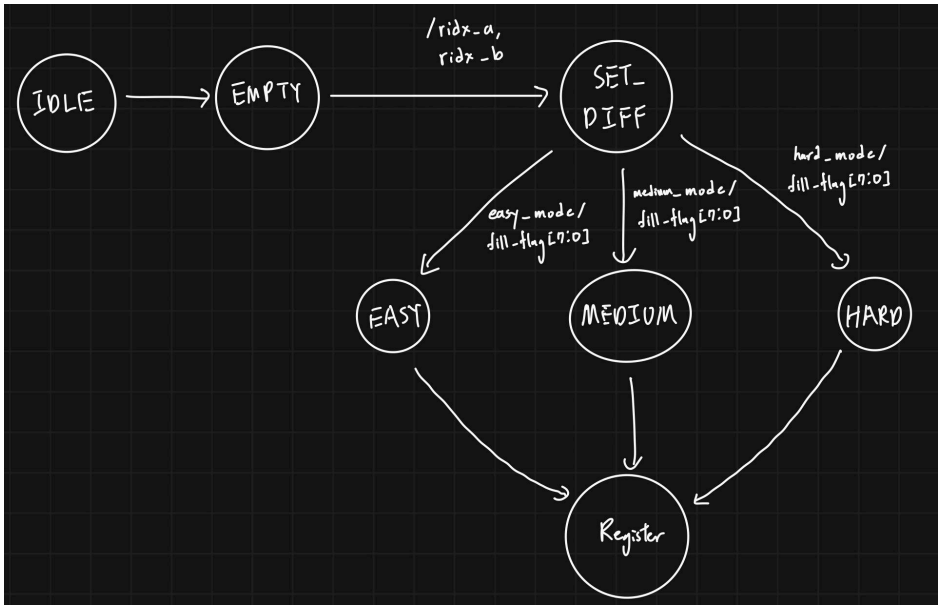


Figure 1. Set Difficulty State Diagram

From IDLE, the FSM moves to the state EMPTY where the solutions of the board (A,B) would be generated already in the datapath. It will then move to SET_DIFF, where two control signals *ridx_a, ridx_b* are sent to our two RNGs to generate two random numbers *i, j* within the range 0 (00) to 3 (11) from the two RNGs used to generate A, B. They will decide the location of the hints, which will be symmetric for rows 1, 2 and rows 3, 4. Depending on the user input *difficulty*, we will move to the EASY, MEDIUM, or HARD state. The output will be 8 bit *fill_flag* where 1's indicate the indices in the first two rows where a hint will be revealed. The state diagram then moves to the Register state that begins game play, continued in Figure 2. Note that there is a restart signal that will send the state back to IDLE.

Assuming we can only generate two random numbers i, j, we plan to implement the hint location as follows:

Difficulty State	Number of hints	Location of hints
EASY	8	00i, 00$\bar{i}$, 01j, 01$\bar{j}$, 10j, 10$\bar{j}$, 11i, 11$\bar{i}$
MEDIUM	6	00i, 00$\bar{i}$, 01j, 10$\bar{j}$, 11i, 11$\bar{i}$
HARD	4	00i, 01j, 10$\bar{j}$, 11$\bar{i}$

Board grid index:

0000	0001	0010	0011
0100	0101	0110	0111

1000	1001	1010	1011
1100	1101	1110	1111

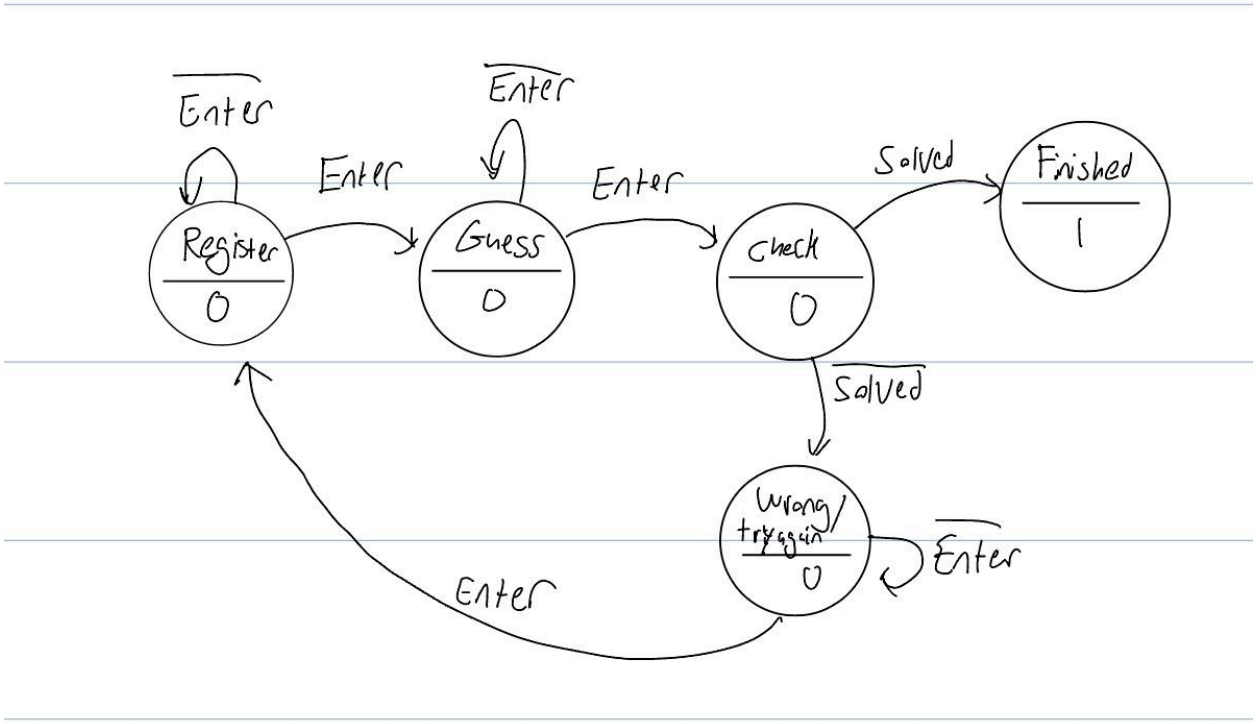


Figure 2. User input state diagram

Updated user input state diagram. The first state Register is to input which register one wants the number to be inserted in. After enter is hit, the Guess state is inputting the number you want to insert into that register. Once enter is hit, it will go into the Check state which will go for one cycle and depending if the combinational logic in the register signals that it was solved or not, it will either go to a finished state or a wrong state. If enter is hit in the wrong state, it will allow you to insert a number into a register again. This will be done until the puzzle is solved.