

FourSight: A Heuristic FPGA Connect 4 Solver

Final Report

1st Alex Horne

Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
alex_h@mit.edu

2nd Lucas Puig

Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
lucasp20@mit.edu

Abstract—This paper will detail the process for implementing a heuristic (imperfect) Connect 4 solver on an FPGA. The purpose of FourSight is to provide the player with a solver that advises moves during every turn of the game. Our goal was to enhance this process so that the user isn't stuck waiting for extended periods between plays, as we hoped for the game to feel smooth, quick, and interactive. A camera module is placed so that it can capture the entirety of the gameboard. From there, our system implements a pixel sampling technique in order to find the current game state. This is then passed into the Connect 4 solver module, which applies a heuristic solver to each possible board state that can be reached from the current one. The column played that results in a board state which produces the highest value of the heuristic scorer is deemed to be the best play.

Index Terms—Field Programmable Gate Array, Connect 4, alpha-beta, MiniMax

I. INTRODUCTION

Our project was inspired by the classic game Connect 4. This game features a two dimensional vertical board with six rows and seven columns. Players take turns dropping a chip down any of the columns, with the goal of creating a four in a row, either vertically, horizontally, or diagonally. We both grew up playing Connect 4 with our families, and when brainstorming potential final project ideas, we settled on this game due to its supposed ease of solving. This led to our idea of pointing a camera module at a Connect 4 board and having the system indicate what the next best column would be, or the perfect play. As it turns out, this is quite a hard task in any language or computer. What makes this a truly challenging task to solve are the 4.5 trillion possible game states, which prove difficult to store on most personal computers. A naive implementation would require 42 TB to store, and even with more efficient board representations, you would still be left with 25 TB of data.

On top of this, implementing such an idea on an FPGA board is even more challenging. To start, just programming a camera and HDMI port to display your results is a non-trivial task, although we fortunately had the code to do this from various labs throughout the semester. The first major issue that arises when programming in Verilog is recursion, which often leads the programmer toward iterative solutions. However, this approach turns out to also be challenging due to very limited lengths of lists possible in Verilog, especially

when comparing these length of lists to the number of pixels in an HD image. We first ran into this while trying to identify the current board state. After attempting various dynamically updating algorithms such as connected component labeling and even circle detectors, we decided to fix the points that the camera was sampling on the board and simply update the board state based on the color thresholds of those points. We wanted to make sure that we would have as much time as possible to work on the solving aspect of the system, so although this method of board classification is not as robust as dynamically doing so, it does exactly what we need it to.

We then had to work through the issue of the length of the potential move queue. Given an empty board, there are 4.5 trillion reachable board states. When running a perfect solve to find what the next best move is, one must be able to see all of these states simultaneously. To counter this, we decided to precalculate all of the possible game states after the first six moves, as that took a relatively small amount of space in memory while also drastically cutting down the maximum tree size that would be reachable later. This of course led to a different issue, in that we needed to come up with a list of all of these game states in our specific 49-bit representation, which required a lengthy Python script to do so. Once these state representations were made, we implemented a recursive function to create a plausible sequence of columns that could have been played to create them, and then passed the sequence into an open source solver made by Pascal Pons in C++ [1]. Pascal Pons' solver represents board states using a sequence of columns (e.g., "777" represents "yellow, red, yellow" in column 7), so we needed these sequences to pass into the solver. Once we managed to get these precomputed board states along with the corresponding next best move on the board, which we ended up managing to do by squeezing them into our BRAM, after first experimenting with writing to DRAM through UART and then realizing that switching to an HD camera would give a better all around product, we then created a way to look through these various board states. This was a demanding problem to solve, as the board state representations are too long to provide direct addressing options. Lastly, we needed an approach to solve the most difficult part of the project: actually creating a working solver on the board that could function past the first six moves and

also coming up with a solution to easily display the result on a monitor. For the solver, we ended up taking a heuristic approach on how to decide what the next best move was. This consisted of a module that could assign scores to boards based on their potential, which was measured by analyzing all possible 4-chip windows on the board.

II. SYSTEM DESCRIPTION

A. Camera Input

The first major component of our system is the camera module. On the physical side, an OV5640 camera sensor is placed directly in front of the Connect 4 board, optimally with a blue or black background set behind the board to minimize interference from colors similar to the game pieces; backgrounds containing significant red or yellow content tend to degrade detection accuracy. An example of how this would look can be seen in figure 1. The camera's output stream is fed into the pixel_reconstruct module from lab 5, which converts the data into pixel indices and 16-bit pixel values, with 5 bits representing red, 6 representing green, and 5 representing blue.

These values are then written into the DDR memory system. The data first passes through the stacker module, which aggregates them and combines the 16-bit pixel RGB values into 128-bit messages. These messages then flow into the AXIS FIFO, which then gives them to the traffic generator module. The traffic generator interfaces with the UBER DDR3 memory controller, which ultimately controls the DDR. This is a complete description of the pixel-write pipeline, and it also mentions all modules driven by our camera clock.

On the read side, a dedicated FIFO brings pixel data from the DDR into the pixel-clock domain. The pixels are then unstacked back into their 16-bit RGB values and then sent on to the rest of the module.

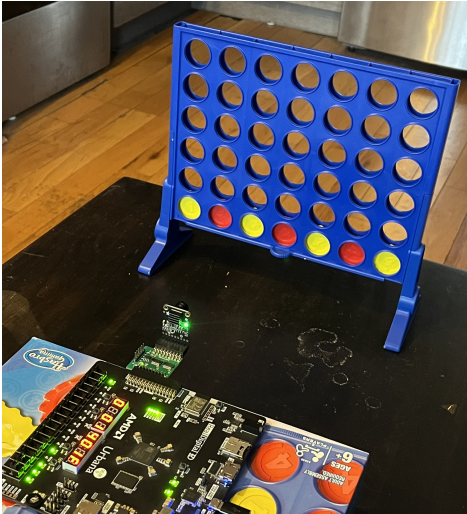


Fig. 1. Physical Setup of FourSight

B. Board Classification

Now that the device is able to take in images through the camera module, the next step is for it to classify the current

board state that is being passed in through the camera. Our system utilizes a thresholding system on 42 fixed points in order to classify the board state. The system samples at each of these 42 fixed pixel locations, as each one represents a slot on the Connect 4 board.

Although this fixed sampling approach unfortunately requires better camera positioning than our initial attempt at a connected component algorithm, this choice has two major advantages. The first is the ease of displaying an HD video stream on a monitor. To run the connect components algorithm, one must rely on a stack that is made up of a substantial length. Interfacing to the DRAM to do reads for both displaying the image and for this algorithm would have likely required making the screen go black for a while as there would be a pause for it to search. Another major limitation of the connected component labeling algorithm is that it relies on consistently identifying all, or at least most, of the pixels that make up a piece as the true color of that piece. This issue is harder than we initially expected to overcome due to the lower quality grade of the camera we are using.

By thresholding, FourSight creates an 84-bit gameboard which is used to tell the solver what the state of the board is. This representation uses two bits to represent each piece, where 2'b00 represents an empty piece, 2'b01 is a yellow piece, and 2'b10 is a red piece. An example board representation can be seen in figures 2 and 3. For simplicity and conciseness, each number in figure 2 represents two bits, and it can be 0, 1, or 2. We use the least significant bits to designate the bottom right corner of the board and the most significant bits for the top left. This is in row-major order. This board is then passed on to our solving module to find the optimal move to play.

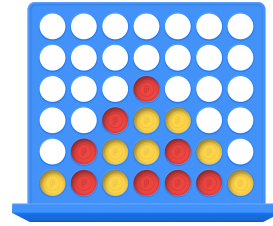


Fig. 2. Example Connect 4 Board [2]

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	2	0	0	0
0	0	2	1	1	0	0
0	2	1	1	2	1	0
1	2	1	2	2	2	1

Fig. 3. Bit Representation of Connect 4 Board

C. Solver - Opening Book

For the solver, FourSight consists of a detailed opening book for early game states as well as a heuristic module that evaluates the next best move for board states that have seven or more moves. This opening book is a database that stores the precomputed best plays for the first six moves of the game.

The main advantage of an opening book is that it provides the user with perfect game play early on in the game. In the case of a perfect solver, this would drastically cut down the size of the game tree as it would eliminate a massive amount of possible branches. In order to save space in BRAM, the board state is implemented differently in the opening book than it is anywhere else in the system. The boards are represented as 49-bit packed arrays, but can be thought of as being a 7x7 grid, where the 7 most significant bits are the top row and the 7 least significant bits are the bottom row. The top 1 in each column represents the first empty cell, and any bits below that are valid game chips, either red or yellow. Any 1 below the top 1 in a given column denotes a yellow piece and a 0 denotes a red piece. An example of this is denoted in figures 4 and 5. This requires a 7x7 gameboard because if a column is full, then there will be a 1 above the top row of that column, signifying that there are no empty spaces left. This is the chosen representation as it uses fewer bits than encoding each cell state as two bits (yellow, red, or empty), which would take 84 bits.

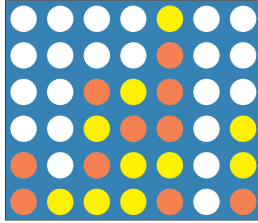


Fig. 4. Example Connect 4 Board [3]

0	0	0	0	1	0	0
0	0	0	0	1	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1
1	0	1	0	0	0	1
0	1	0	1	1	0	1
0	1	1	1	0	1	0

Fig. 5. Bit Representation of Connect 4 Board

In the first six moves, there are exactly 22,100 possible unique board states, so storing these boards, alongside the next best move, takes $22,100 * 52 = 1,149,200$ bits, or 0.14365 MB. Due its small size, the opening is stored in BRAM in order to provide quick lookups for the first six moves. In fact, this allows for the first six moves to appear so quickly that the calculation appears instantaneous. The opening book file

was created with a Python script that made every possible board state for the first six moves. From there, using a Python function, every board state was matched with a valid sequence of moves that could have created it, and this was fed into an existing open source Connect 4 Solver made by Pascal Pons [1]. However, Pons' solver provides a score given a valid sequence of moves rather than an actual column, so a Python function written by us was used to interpret the score and choose the next best move for each of the board states.

In terms of the Verilog implementation, this lookup is implemented using an iterative search process. This takes the form of a Finite State Machine (FSM), where the searching function is waiting for the game_board_valid signal to be high, and once it is, it then transitions to searching. It then searches until it finds the matching board and outputs the move as well as a valid signal to show the next move. Lastly, it transitions back to waiting.

D. Solver - Heuristic Solving Algorithm

Rather than providing users with perfect play through deep recursion into a game tree, FourSight offers guidance based solely on the current board state. In order to do this, the heuristic solver consists of various modules that factor into how it is able to function as desired. First, the terminal_board_check module is utilized to determine whether a board has reached a terminal state or not, which is defined as when the game has either been won by either of the colors, or when a draw has occurred, meaning there are no valid moves left to make because the board is completely full. The input to this module is an 84-bit unpacked array gameboard, where each index is a 2-bit value, and the output is is_terminal flag that represents the terminality condition of the input board. The valid_spots module decides which columns are still able to be played. It does this by checking if the top space in each column is empty. If it is, then this column can still be played. Otherwise, it can not be. The input to this module is an 84-bit unpacked array gameboard, and the output is a 7-bit number, where each bit represents a column. A 1 means that the column can still be played, and a 0 means that it cannot. The is_game_won module determines if a game has been won, and it does so by checking if there are any 4 in a row sequences of the same color, either vertically, horizontally, or diagonally. The input to this module is an 84-bit unpacked array gameboard, and the outputs are a game_won flag as well as a 2-bit number representing the color of the winner, given that there is one. The play_column module is used to produce a new gameboard after a move has been played. The inputs to this module are an 84-bit unpacked array gameboard and a 2-bit number representing which color is being played, and the output is an 84-bit unpacked array that shows the resulting gameboard.

The final module that is utilized in the heuristic solving implementation is the heuristic scoring module, which assigns a score to a board state that is passed in. This works in a similar fashion as a convolution; a window of size four is slid across a gameboard, and patterns in that window that have impactful potential contribute positively to the overall score of that board

state. This window is slid across the board four different times, as there is one to check for horizontal patterns, vertical patterns, positive diagonal patterns, and negative diagonal patterns. As an example, a horizontal pattern of 3 yellow chips and 1 empty chip in a window is very good for the yellow player, so this would contribute a positive value to the overall score. Thus, the solver_heuristic module utilizes all of these in order to choose the next best move given a board state. At a high level, it works by applying a move at each of the seven columns (if applicable), checking if that caused the game to terminate, and if not, applying the heuristic scoring module on the new board state. A visual for this can be seen in figure 6. If the game terminates after playing a column, this is immediately determined to be the next best move, as this would cause the player to win or be the only possible move. In the case that none of the columns result in termination after being played, the column that produced a board state with the highest heuristic score based on our scoring module is determined to be the next best move. This module is applied to all gameboards that have seven or more chips on them. Something to note is that this heuristic solving module originally had very deep layers of combinational logic, and as a result, the slack was extremely negative. To fix this, we implemented a few layers of pipelining including processing only one row per cycle to guarantee that it was meeting timing. An overall block diagram for the solver can be found in figure 7.

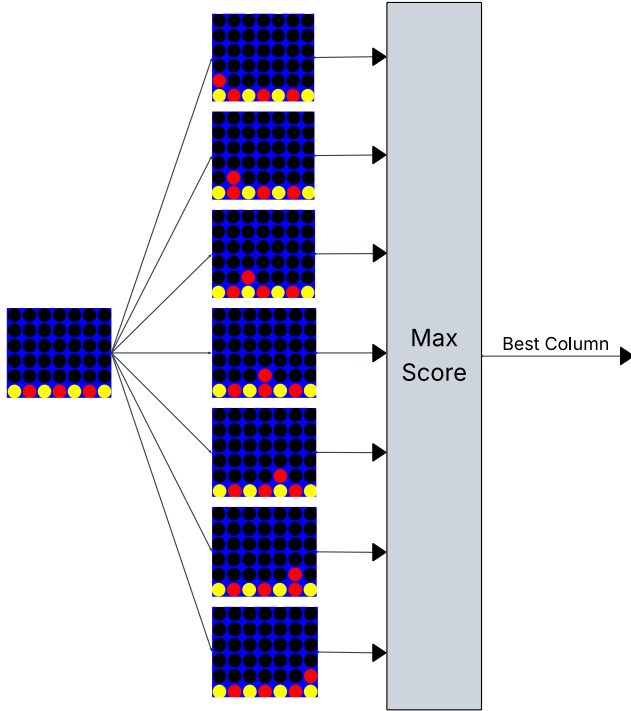


Fig. 6. Best Column Selection Flow Chart

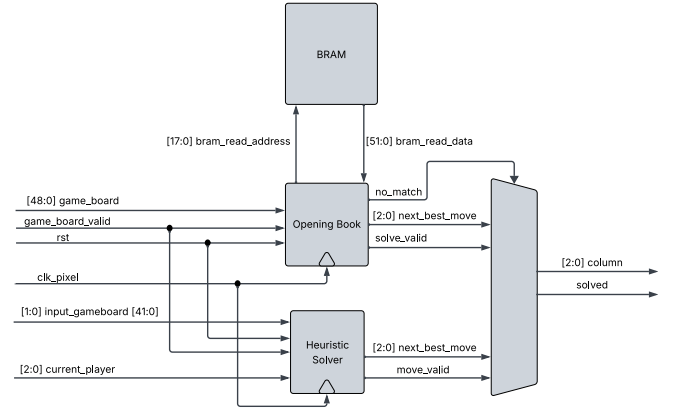


Fig. 7. Solver Block Diagram

E. Displaying the Final Move

The final logic module in FourSight is the image modifier. This module displays two things: the column to play in and tiny crosses that show both the color of the pieces and where they are on the board. This is implemented as a 3-cycle sequential module that takes a single pixel and its *h_count* and *v_count* as inputs. The first cycle checks whether the pixel would be included in the arrow that points to the best column to play in. The second cycle is used to check if the pixel would be in one of the 42 crosses that show where the pieces are in the image. The final cycle decides whether the pixel should be its original color, black (to be a part of the arrow), or a part of the cross on each piece, which means it would be red, yellow, or black (representing an empty position). Figure 8 shows how the best move is displayed in hardware.

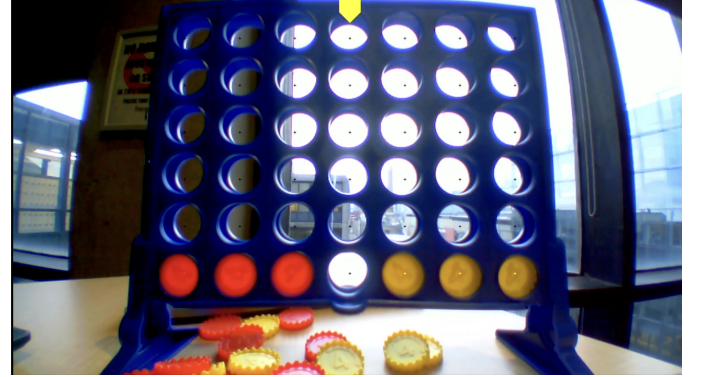


Fig. 8. Best Move Being Shown in Real Time

III. OVERALL SYSTEM EVALUATION

A. Throughout, Latency, and Timing

The latency and throughput of our modules may not be optimized perfectly, but the human eye can not tell the difference as it runs within the timing of finding a single frame and so we believe that the latency and throughput were made sufficiently fast. One potential speed up could be a faster look up algorithm in the opening book, but again this

optimization would have no impact on the system, so is likely not worth doing. At a low level, our design must meet the timing constraints of three separate clocks. The camera clock runs at 200MHz, the DRAM is run at 666 MHz, and the HDMI as well as the solving algorithms clock is run at 74.25 MHz. All of our logic must, and does, meet the requirement that the combinational logic must have a shorter critical path than the clocks period. This gives our system a WNS of 0.519 ns. On a high level, we planned our system so that it solves the board rapidly enough to be usable in a game. We have achieved this as it solves the board in less than the time it takes to write a single frame to the monitor.

B. Memory Usage

Our main form of on-chip memory usage comes in the form of the opening book, which is stored in the BRAM. We store exactly 22,100 unique board states in the opening book, and each is represented using 49 bits. Along with that, we allocate 3 bits to indicate the next best move. We therefore utilize $22,100 * 52 = 1,149,200$ bits, or 0.14365 MB of the BRAM. We also use the DRAM as a buffer between the camera and the HDMI output. This takes $1280 * 720 * 16$ bits, or 1.8432 MB of space. This is a small proportion of the available space in the DRAM, and so is not a large usage of memory.

C. Checklist

FourSight is able to provide guidance to a user on what move to play given any valid Connect 4 board state. This can either be done in live-video, or through freezing frames and solving them individually. A camera simply has to be set in front of the board such that the 42 crosshairs are each on top of the correct spaces slots on the gameboard, and from there the game can be played with a solver ready to help. Our project checklist outlined various goals that we wanted to meet by the end of the semester. The checklist can be broken down into four main components: accurate board state recognition, displaying the next best move, fast solving ability, and solving capabilities.

In regards to board state recognition, we were able to meet our stretch goal, as FourSight can recognize the board in live-video, and it also has the ability to perform freeze frame board state recognition (we provide the ability to toggle between these viewing styles using button three on the board). We decided to include both of these options in the implementation as we felt that they both provided utility in their own way depending on the context of the game. After thoroughly testing the system, we determined that it was able to accurately classify the gameboard in front of it. In terms of displaying the next best move, we reached our stretch goal by not only putting an arrow on top of the recommended move, but also having it be the same color of whoever's turn it is. Along with that, the user has the ability to turn off the arrow at any point in the game using the switches on the board. Next, we consider fast solving ability, in which we hit our stretch goal. Visually, the solves appear to happen instantaneously, and the user does not have to sit and wait for the recommended move to appear.

This is both because of how easy it is to access the opening book and return a move, as well as because the solving module does not consist of any sort of lengthy recursion. Lastly, we hit our commitment goal in terms of solving capability. Our stretch goal was to have a perfect Connect 4 solver using a recursive alpha-beta MiniMax algorithm, but ultimately, we decided to implement a heuristic solver using a scoring system to ensure a functioning final project, which was part of our initial commitment.

Something that we could have easily implemented in our design, but we did not deem necessary at the time, is the ability to have the solver indicate moves for just one of the colors throughout the whole game. Instead, we made it such that the solver can be turned on and off during the game, but this would have been a more dynamic version of that. The switches could have been used to indicate which color we wanted to show moves for, and it would have been easy to implement as we already had a module called `whose_turn` that determined whose turn it was given a board state. Also, our standard convention in designing this project was that yellow always plays first. This factors into the opening book, as it only stores board states where yellow played first. We could have made the opening book twice as large in order to accommodate for both colors being able to play first, which would have been a simple adjustment to make.

IV. REFLECTIONS AND INSIGHTS

When thinking about the journey of our project from the beginning to the end, there were a few roadblocks that we hit along the way that required us to pivot our plan. The game tree for Connect 4 is enormous (about 4.5 trillion possible states), so we knew that this was not going to be an easy task to figure out. After reading various papers and blogs about different solver implementations online, we had a better understanding of the approach to take, and we began to make the Verilog implementation. However, we had a hard time finding other implementations of MiniMax recursion on an FPGA, and this is certainly not a trivial task. After butting our heads at this for a few weeks, we determined that we simply weren't going to have enough time to finish the project unless we switched to an imperfect solver, so that is the decision we ended up making.

Here are some of our key takeaways from working on this project:

- Starting the project by working on an implementation in an easier language like Python had its pros and cons. In many aspects, it was nice to have a version in Python, as we used this as a basis for the rest of our project. It provided us with direction and created an outline for us of what modules needed to be made. However, it is very easy to implement something in Python and think that it will be just as easy to do so in Verilog. This is indeed not the case, and it is important to remember that things we take for granted in higher level programming languages, like recursion, are not simple tasks in hardware.

- More is not always better. At the start of the semester, we had the idea of making a board state classifier using a connected components algorithm. We spent a lot of the semester trying to get that to work, but it ended up being a very difficult task. Instead, we switched to a much simpler idea: a pixel in the middle of each slot to decide if it was yellow, red, or empty. This module was more straightforward to develop and served the exact same use case. Thus, it is best to consider all options before jumping into the most complex one.
- It is important to use a convention that can be easily simulated and testbenched. Over the course of the semester, we wrote various testbenches to ensure that our modules were exhibiting the correct behavior. Because we saw it as being easier to work with, our gameboard was represented as an 84-bit unpacked array. However, this caused headaches when it came to analyzing waveforms, as unpacked arrays do not appear in Surfer. This required us to constantly make test variables that we packed our gameboards into, and it started to get confusing when pipelining came into play. Looking back, it would have been wiser to go with a more robust gameboard representation to avoid this issue completely.

V. CONCLUSION

This paper outlines FourSight, an FPGA-based Connect 4 solver that determines game states using the OV5640 camera sensor and utilizes a heuristic solver in order to predict the next best move. As a summary, our system takes in image frames of the Connect 4 board and samples pixels at 42 different locations, one for each spot on the board, in order to create an 84-bit unpacked representation of the current game state. Following this, the board state is used by the solver module to actually compute what the next best move is. If the board state can be created using six or less moves, then the opening book is utilized to provide the next best move. Otherwise, this is provided through the heuristic solving module. The last step is overlaying the optimal choice on the screen so that the user can see where to put the next chip. This is done with an arrow that sits on top of the correct column. The arrow can either be yellow or red, and this depends on whose turn it is in the game, yellow meaning it is yellow's turn, and red meaning that it is red's. A full system diagram of FourSight can be found on the final page.

VI. INDIVIDUAL CONTRIBUTIONS

A. Technical Implementation

Alex wrote the board state classifier, the module that reads the opening book, and the logic for being able to display an arrow over any of the columns after the board has been solved. He also played a crucial role in organizing the top level file that set the overall structure of the system and optimizing all modules to meet timing.

Lucas made a Python implementation of a heuristic solver utilizing Keith Galli's Youtube series [4] for inspiration, wrote the Verilog implementation of the heuristic solver as

well as its various submodules that it instantiates within it (is_game_won.sv, terminal_board_check.sv, etc.), and developed a Python script to make the opening book that is stored in BRAM.

We both helped one another in developing test benches, debugging code, and pipelining the system to make sure that it made timing.

B. Report

We both collaborated on sections I, II, and VI. We both also contributed to all figures and block diagrams in the document. Alex wrote section III and Lucas wrote section IV.

VII. ACKNOWLEDGMENT

We would like to thank Joe Steinmeyer and the rest of the 6.2050 staff for all of the guidance and support that they provided us with throughout the semester. This project would not have been possible without their help.

VIII. CODE REPOSITORY

Please find all of our code at the following repository: <https://github.mit.edu/6205F25/fa25-6205-team35>

REFERENCES

- [1] P. Pons, "Solving Connect 4: How to Build a Perfect AI", 2016.
- [2] J. Chui, Figma, Connect 4.
- [3] J. Hamkins, "Infinite Connect Four", 2025.
- [4] K. Galli, "How to Program Connect 4 in Python!", 2017.

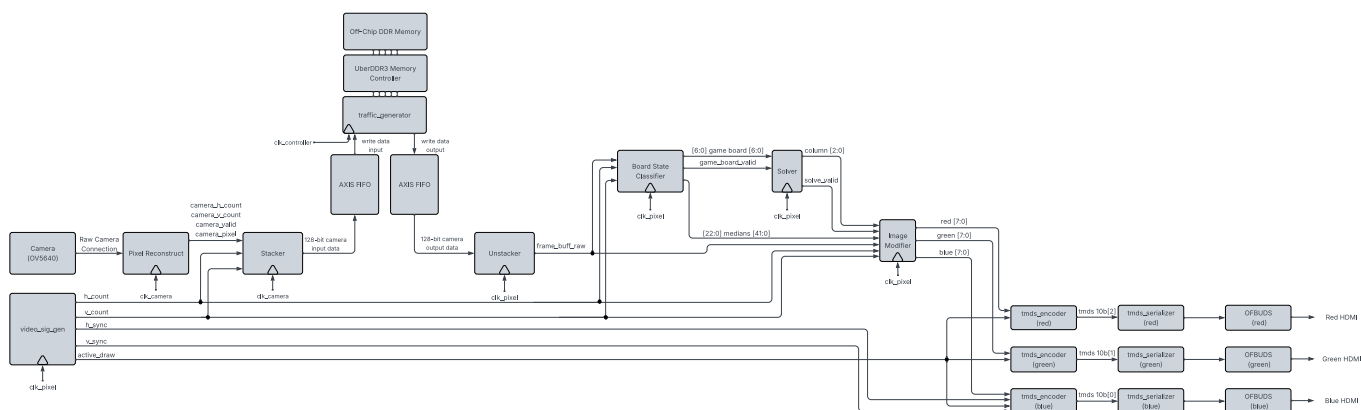


Fig. 9. Full System Block Diagram