6.2050 Final Report - Othello

Andrew Spears
MIT
Cambridge, MA
amspears@mit.edu

William Liu
MIT
Cambridge, MA
wliu1@mit.edu

Clay Davis
MIT
Cambridge, MA
clayd@mit.edu

***Abstract*—We present an FPGA implementation of the classic game Othello. Given 4 corners to represent the board boundary and 1 game piece for the player, the FPGA projects the board and game state onto an HD live video feed in an augmented reality fashion. Internally, the solver engine implements the alpha-beta minimax algorithm to find the optimal move up to a certain depth, capable of responding nearly instantly while still easily beating strong players. We implement the design on a Real Urbana FPGA with an OV5460 camera and an HDMI monitor running at 1280x720. We evaluate its performance based on human player feedback and ability against online bots.**

Index Terms- FPGA, Othello, Search Algorithm, alpha-beta minimax, augmented reality

I. Overview

Many 6.2050 FPGA assignments revolve around image and live video processing. FPGAs are also well known for accelerating compute for certain types of classic algorithms. We combine these two features in our project Othello. The FPGA hardware recreates a live projection of an Othello board based on player-determined boundaries and concurrently accelerates an alpha-beta minimax search. The player only needs 1 game piece to experience Othello against a challenging but responsive computer opponent. Figure 1 below provides an intended overview of the player experience. Figure 2 provides a block diagram of the overall design.

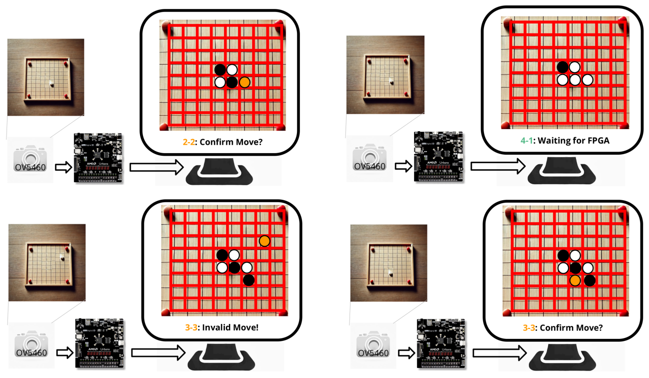


Fig. 1. Sample overview of player experience

II. Controls

As with any game, the player needs the ability to configure the game's settings. In our case, the player needs to define thresholds on color channels for the corner and Othello piece, as well as confirm moves. Utilizing the 16 switches and 4

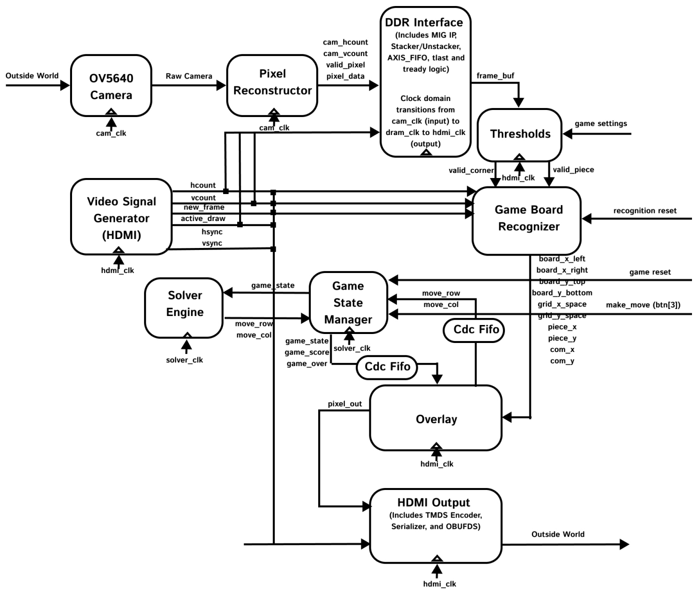


Fig. 2. Block diagram of overall design

buttons provided on the Real Urbana FPGA, we designed the following control scheme in Table I and Table II. Note that `btn[1]` sets the mode for configuring between the corner and board pieces, and in turn displays the current corresponding configuration values on the FPGA's seven segment display. Due to a limited number of buttons and switches, Table III combines pairings of the two for extra controls that we deemed necessary later on in the development cycle.

btn[3]	btn[2]	btn[1]	btn[0]
Confirm Move	Save Config	Game Piece Config	Reset

TABLE I
FPGA Button Controls

sw[15:12]	sw[11:8]	sw[7:4]	sw[3:1]	sw[0]
Upper Bound	Lower Bound	Debug Info	Channel	Camera Enable

TABLE II
FPGA Switch Controls

sw[0] & btn[3]	btn[1] & btn[3]
Recognition Reset	Game Reset

TABLE III
FPGA Special Controls

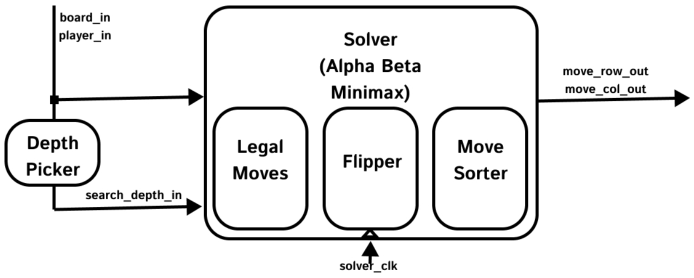


Fig. 3. Solver Module Block Diagram

III. Solver

As an opponent for the player, the design includes a solver (3) which uses a tree search to explore possible lines. Specifically, the solver implements alpha-beta pruning with variable depth, inspired by [1]. On a high level, the solver does the following:

1) Find legal moves from the current position.
2) Determine a heuristic ordering in which to try the legal moves, prioritizing stronger moves first.
3) Try the first move by searching possible lines. Evaluate the resulting position assuming that our opponent plays the best possible moves at each stage.
4) Repeat for all other moves.
5) Return the move/evaluation which was best for the current player.

This algorithm is known as Minimax, owing to the convention that the score/evaluation is represented as one value which white tries to maximize and black tries to minimize. Alpha-Beta pruning is a strict improvement on this algorithm which avoids searching redundant lines of gameplay, increasing search speed. Both of these algorithms are applicable to many 2-player, perfect-information games, including Othello, chess, checkers, and more.

The **solver** module computes the optimal move in a two-player board game based on the current game state. The module parameters include:

- **`N`** (default: 8): Board width (number of rows/columns in a square board).
- **`MAX_SEARCH_DEPTH`** (default: 10): Maximum search depth for the minimax algorithm.

The **`board_in`** input uses a row-major format for simplicity. Outputs provide the optimal move's position and evaluation score, ensuring valid synchronization through **`data_valid`** signals.

Our implementation is specifically tailored to both the game Othello and FPGA hardware in several ways. Firstly, although Alpha-Beta Minimax is most easily implemented recursively, our implementation is iterative, using a stack to organize which states to revisit after searching new moves. A result of this is that we want to avoid storing unnecessary information on this stack, as storing it in memory might be slow to access or write, and we need to push and pop data very frequently. Thus, we compress all state information as much as possible. Each element on our stack looks like the following:

board	player	d	revisit	v	α	β	moves_to_search

d is the depth and v is the current best estimate of the value of that state.

On each node visit, the solver does one of three actions.

1) **First visit**: If it is visiting a state for the first time, it will find legal moves using the **legal_moves** module, choose the highest priority move using the **move_sorter** module, and find the state resulting from that move using the **flipper** module. It then pushes a revisit call to the stack, recording that the remaining moves to search are the legal moves minus the chosen move, before finally moving to the chosen child node.
2) **Revisit**: If it has just updated a state which has more child moves to try, it will do the same as on a first visit, but refer to the remaining moves to search instead of computing legal moves.
3) **Final Revisit**: If it has just finished processing a state (tried all moves and found an evaluation OR hit the maximum depth and found a heuristic evaluation), it pops a revisit call off the stack, which corresponds to returning to the node's parent.

To save clock cycles in the search and avoid unnecessary computation, we very carefully avoided any 'tail call' pushes of the stack. A first visit to a node does not use the stack at all, but rather directly sets register values to those of the incoming node. Similarly, we avoid having to store any extra information from a child state when revisiting the parent by immediately updating the parent values from the stack.

A. *Solver memory usage*

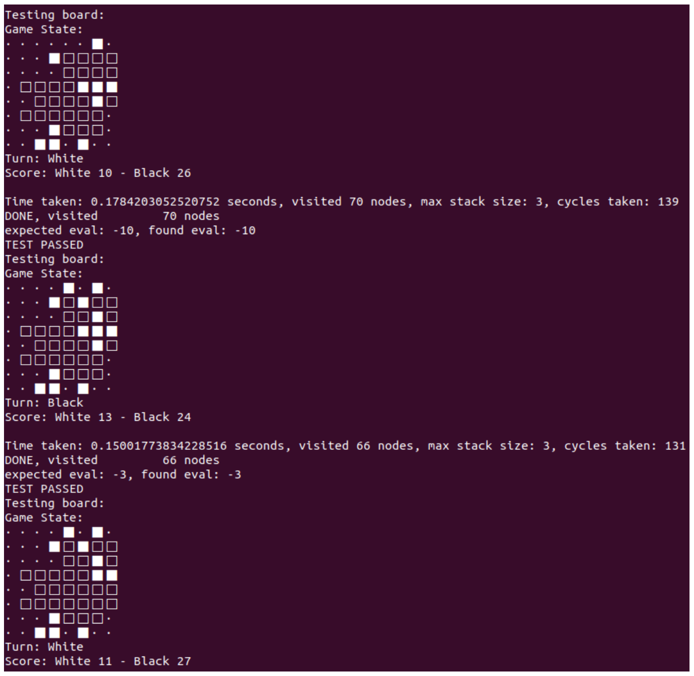
All in all, these optimizations ensure that our search algorithm's memory usage is at most a couple of hundred bits times the search depth[1], allowing us to rely only on distributed RAM.

We now give an overview of the significant modules used in the solver and game state updating:

B. *Legal_moves*

To find legal moves combinationally, we split each square into its own problem. For a square to be a legal move, it must have some direction in which there is a line of opponent's tiles followed by our own tile. This allows for a very parallel computation - for each of the 8 directions, and for each distance, check if this exact pattern is found. If any distance and direction succeed, this is a legal move. Thus, every direction and distance combination does a simple check, and the results are all OR-ed together to give the square's legality.

[1]Not including a constant number of bits used for legal move finding, state updating, and similar modules



```
Testing board:
Game State:
· · · · · · ■ ·
· · · ■ □ □ □ □
· · · · □ □ □ □
· □ □ □ □ ■ ■ ■
· · □ □ □ □ ■ □
· □ □ □ □ □ □ ·
· · · ■ □ □ □ ·
· · ■ ■ · ■ · ·
Turn: White
Score: White 10 - Black 26

Time taken: 0.1784203052520752 seconds, visited 70 nodes, max stack size: 3, cycles taken: 139
DONE, visited         70 nodes
expected eval: -10, found eval: -10
TEST PASSED
Testing board:
Game State:
· · · · ■ · ■ ·
· · · ■ □ ■ □ □
· · · · □ □ ■ □
· □ □ □ □ ■ ■ ■
· · □ □ □ □ ■ □
· □ □ □ □ □ □ ·
· · · ■ □ □ □ ·
· · ■ ■ · ■ · ·
Turn: Black
Score: White 13 - Black 24

Time taken: 0.15001773834228516 seconds, visited 66 nodes, max stack size: 3, cycles taken: 131
DONE, visited         66 nodes
expected eval: -3, found eval: -3
TEST PASSED
Testing board:
Game State:
· · · · ■ · ■ ·
· · · ■ □ ■ □ □
· · · · □ □ ■ □
· □ □ □ □ □ ■ ■
· · □ □ □ □ □ □
· □ □ □ □ □ □ □
· · · ■ □ □ □ ·
· · ■ ■ · ■ · ·
Turn: White
Score: White 11 - Black 27
```

Fig. 4. Solver simulation results

C. *Move_sorter*

Alpha-beta pruning is vastly improved by searching stronger moves first, since we prune redundant branches more quickly. We sort moves based on a simple location-based priority (e.g. corners are very high priority, since they cannot be flanked by the opponent).

To find the first move to try out of the remaining legal moves, we hard code a re-ordering schema which returns an array of 1's for moves to try and 0's for illegal or already-searched moves, with the rightmost indices being the highest priority. This then goes into a recursive tree-based **priority_encoder** module, which finds the index of the leftmost 1. This is finally translated back into a coordinate in the board, which becomes our next move to search.

D. *Flipper*

The flipper works similarly to the legal move finder. For every direction from the square where a move is being played, and for every distance in that direction, we check for the exact pattern required to flip the opponent's discs. If this pattern is found, we flip those discs and combine the results of these processes. This returns the resulting board.

E. *Solver timing constraints*

Experimentation with Vivado shows that the major combinational modules in the solver have to be split into 2 cycles to meet timing constraints. Specifically, each visit of the solver can be split into two stages:

1) **Find legal moves and sort:** The **Legal_moves** module takes the board state as input and feeds serially into the **Move_sorter** module (on a first visit) or the remaining moves to search (on a revisit). Together, these two operations fit in one cycle of a 74 MHz clock.[2]
2) **Flip:** The **Flipper** module takes input from **Move_sorter** and the board state. The propagation delay of the flipper alone fits in one cycle of a 74 MHz clock.

This allows us to visit a node every 2 cycles, regardless of the type of visit. We can now compute the time required to search based on an estimate of the average number of legal moves (the branching factor of the game tree). This branching factor can be roughly approximated as a function of the number of moves made in the game thus far, and found numerically by simulating lots of random game states. Using this method, the estimated search times for various depths and numbers of moves made are given as a heatmap in Figure 5.

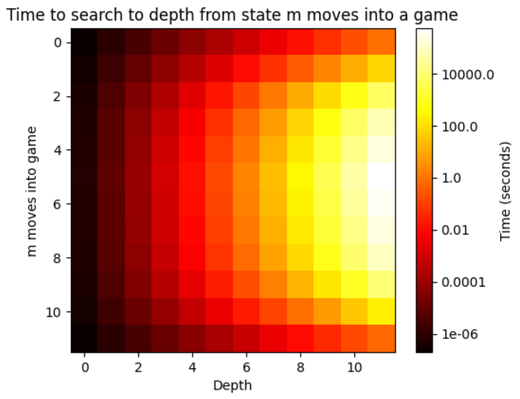


Fig. 5. The times to search various depths from a state m moves into the game for a range of m from 0 to 60. The middlegame is slower to search because there are more legal moves, while the opening and endgames are relatively quick. Our solver uses depths which follow the orange-red band to keep search time approximately 1 second.

To keep search times fairly consistent, we implemented a **Depth_picker** module, which uses a case statement to pick search depth based on the number of moves played thus far. Testing shows that the search time is indeed on the order of one second using these depths.

IV. Recognition

The purpose of the recognition module as shown in Figure 6 is to identify the player-defined game board boundaries and the currently active player piece. Given the game settings as defined by the controls, the threshold modules filter for pixels considered part of the corner pieces and the central game piece.

The k-means center of mass module provides coordinates for 4 points that it considers to be the corners and the lone center of mass module computes for the game piece itself. The corner data is then used to compute the bounding rectangle and grid spacing for the game board, which is then used to compute the coordinates of the game piece through the following formulas:

$$piece\_x = (coord_x - board_{xl}) * N/(board_{xr} - board_{xl})$$

[2]Notice this is slightly slower than the clock speed of the HDMI output. See the FPGA timing section for details

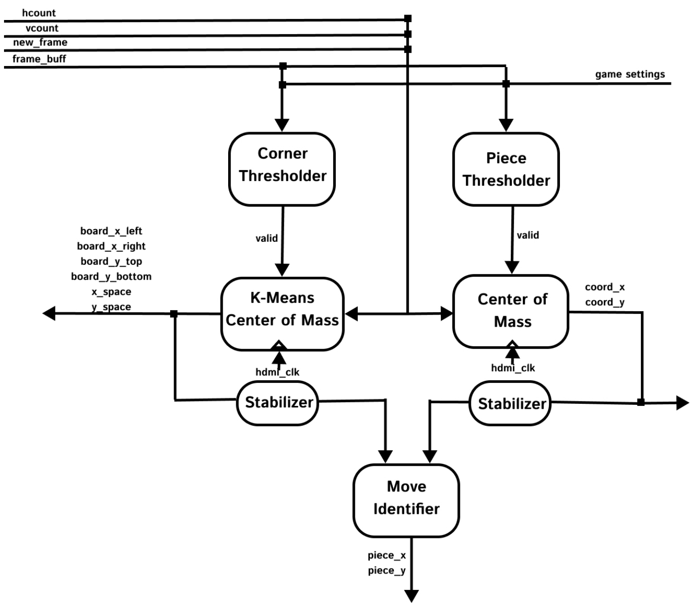


Fig. 6. Recognition Module Block Diagram

$$piece\_y = (coord_y - board_{yt}) * N/(board_{yb} - board_{yt})$$

In our k-means module, we process over a stream of threshold marked pixels to classify which corner they are closest to. To take advantage of FPGA parallelism, we compute the Euclidean distance from each such point to all k centers in parallel similar to [2], before computing the minimum. During each frame, the module tracks a sum and count of coordinates for pixels belonging to each center. Upon receiving a new frame signal, the module computes the new k centers. If the corners all remain within a certain threshold after a few frames, we consider the k center of masses as stabilized and valid, before sending it down the design pipeline. Figure 7 provides rendered scenes during RTL simulation of this module.

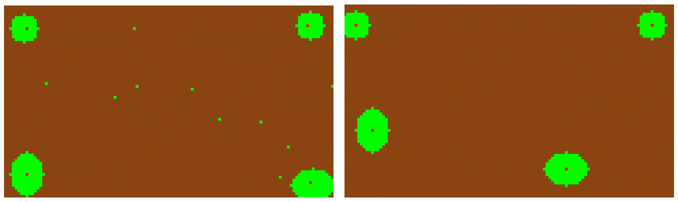


Fig. 7. Scene rendering simulation for k-means center of mass

Note that all the data accumulates when the HDMI active draw signal is high, and all the remaining computation occurs after the new frame signal arrives. This design takes advantage of the very long HDMI blank and sync periods to compute the new k center of masses - specifically, we have nearly 50,000 cycles given 1280 x 720 resolution HDMI specifications for blank and sync cycles, which is more than enough for these modules, especially as the longest path consists of 2 32-cycle dividers in series for computing the coordinates from the corners.

During our early test runs under real world conditions, the recognition's bounding box and piece location were very jittery. We tackled this problem through a stabilizer module as seen in Lab 6 of 6.2050, where the output value is determined as a weighted average of 25% of the new value and 75% of the previous value. This weighted account of history led to decent stabilization. Additionally, after multiple rounds of failed corner detection, we reset the k-means values to the 4 corners of the screen to help the computation escape a poor convergence.

V. Overlay

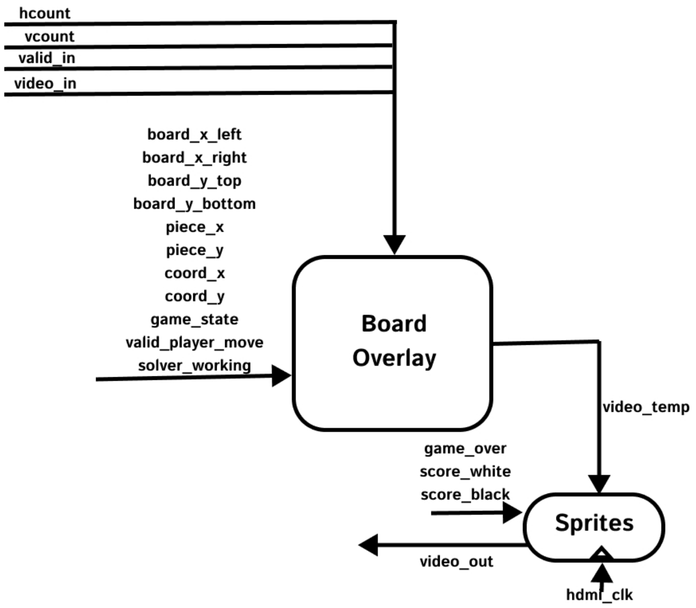


Fig. 8. Overlay Module Block Diagram

The inputs to this module as seen in Figure 8 are:

- **clk_in, rst_in (1-bit):** Clock and reset signals.
- **hcount (12-bit):** HDMI hcount signal.
- **vcount (11-bit):** HDMI vcount signal.
- **video_in (24-bit):** RGB pixel data from the video input stream.
- **valid_in (1-bit):** Indicates valid pixel data.
- **game_state (array, N_PIECES):** Bitmap indicating active game pieces.
- **valid_player_move (1-bit):** Indicates if a move is valid.
- **solver_working (1-bit):** Indicates if the solver is working
- **game_over (1-bit):** Indicates if a game is over.
- **board_x_left, board_x_right (12-bit):** Horizontal boundaries of the board in pixel coordinates.
- **board_y_top, board_y_bottom (11-bit):** Vertical boundaries of the board in pixel coordinates.
- **piece_x, piece_y (3-bit):** Coordinate locations of the player game piece on the board.
- **coord_x, coord_y (12-bit/11-bit):** Actual pixel coordinates of the detected player piece for determining whether the player piece is in bounds.
- **score_white, score_black (6-bit):** Scores for respective players.

The outputs are:

- **video_out_data (24-bit):** Modified RGB pixel data with overlaid game elements.

The overlay and related modules take game state information and the position of the board within the input video in order to ”draw” game pieces and grid spacing. By intercepting the incoming video stream, the module modifies the pixel data to render game elements in real-time. The design allows for flexible rendering of pieces across the board, with the ability to overlay sprites or draw via changing pixels based on the current game state and piece positions. Each piece is represented by a circular sprite centered within its corresponding board cell, providing a clean and visually intuitive representation of the game board. Game state pieces are represented by white or black, invalid moves are represented as red, valid moves are represented as green, and player moves during solver engine execution are highlighted as gray. The images of the board and the pieces will also be stabilized using a minimum movement threshold for our previous k means calculation. This will ensure the overlay is stable and not susceptible to small changes in the camera input.

In addition to the rendering of game pieces and outlines, a subtle green highlighting effect is applied to the area under the game board. This is done by checking if the current pixel is within the bounds of the board and not in a game piece or a gridline. If this is true, the R and B values of the pixel are decreased, creating the green highlight. This maintains the visual hierarchy of the game while providing a much more clear boundary for the overlay. An example overlay testbench render can be seen in Figure 9.

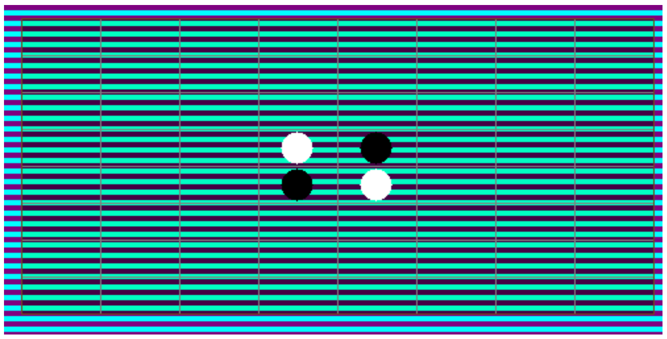


Fig. 9. Scene rendering simulation for the overlay module of default game state

To further enhance the user experience, the overlay module relies on the aforementioned averaging module that stabilizes the board's motion against minor camera jitter and lost identifications. Each new frame's computed positions are blended with the data from several preceding frames. This multi-frame integration smooths out transient shifts in the board's detected coordinates and reduces flicker or jitter in the displayed overlay. As a result, even if the camera or the board slightly moves, the overlaid graphics will remain steady, ensuring players can focus on the game play rather than being distracted by wobbly graphics or momentary misalignment.

Lastly, we encode sprites with the digits 00 to 64 along with a victory and loss message for the purpose of score display. These are encoded as bitstrings where each pixel of the original image is represented by a 0 or a 1, and the overlay module can decide on the color to display for the sprite.

VI. Game State Manager

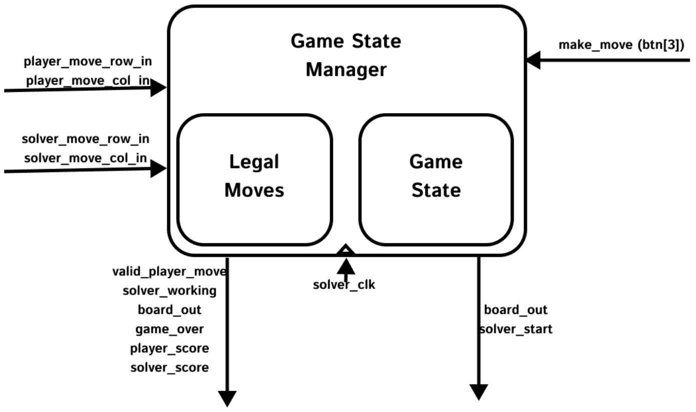


Fig. 10. State Manager Block Diagram

The game state module shuttles stateful information between the recognition system, solver engine, and the overlay module.

Given a player coordinate, it determines if the move is valid with an internal legal moves module instance and upon confirmation, uses an internal flipper module instance to update the game state. Afterwards, it signals the solver to begin solving. Once the solver finishes, it sends its move back to this game state manager for updates. This state manager also keeps track of the score and whether the game has ended, which occurs when no players can make legal moves. The state of the game, the validity of a player's move, and work status of the solver engine are all wired as outputs to other modules.

VII. Other Design Aspects

The remaining aspects of the design come mostly from lab 6 in 6.2050. To enable HD camera quality (1280x720 at 60 fps), we rely on storing video data with help from the AXI protocol, MIG, and FIFO IP to arbitrate with DRAM. For each pixel that comes out, we run it through the same color channel and threshold modules seen in previous labs to guide the later recognition modules. Our system was composed of multiple clocks derived from the Vivado clocking wizard, consisting of a camera, camera sampling, MIG, HDMI, HDMI TMDS, and solver engine clock. To arbitrate between the different clock domains from the game engine and the recognition as well as overlay system, we rely on Vivado FIFO Generator IP for synchronous clock domain crossing transfers.

An interesting design choice that frequently popped up was taking the sum of an 8x8 array representing the Othello board, as in the case of heuristics and score tracking. We noticed that a linear accumulation through a for loop of each row in parallel chained with a final linear summation in RTL code led to better timing when compared to a fully recursive tree adder. Such a result goes against expectation, but we suspect that Vivado optimizations are a factor in play here.

VIII. Evaluation

A. Game Play

Image 11 and Image 12 demonstrate the augmented reality display capabilities of our FPGA design for Othello, based on the simplistic real life setup of 4 corners of 1 color and 1 piece of another color in Image 13. Image 14 showcases the game over scene, from which a player must reset the game state to start a new game. As of now, we have yet to ever see the victory message as the solver engine's lookahead simply makes it too powerful. A video of our functioning design can be found at https://www.youtube.com/watch?v=4q_LQUODgKs.

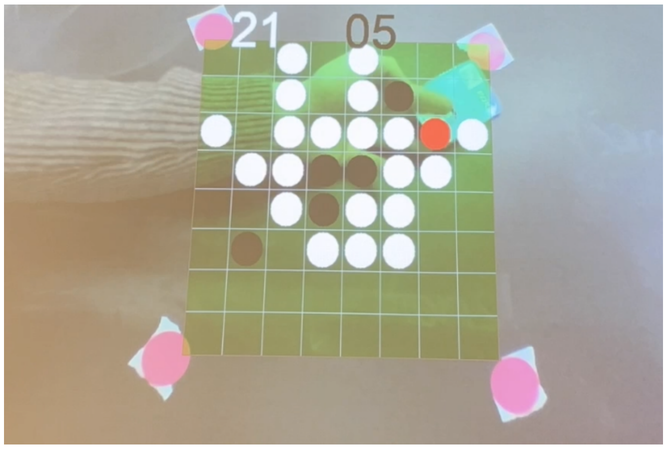


Fig. 11. Invalid and Valid Move Display

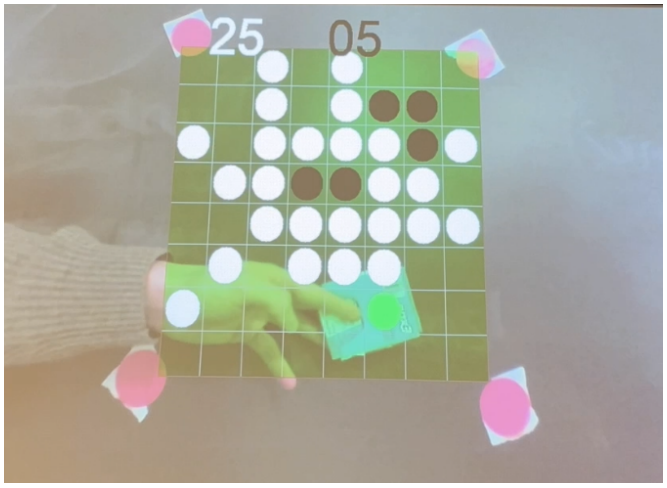


Fig. 12. Valid Move Display

B. Solver Performance

To evaluate the effectiveness of our solver engine, we conducted a series of matches against established alpha-beta minimax implementations. We ran multiple matches against an open-source online bot[3]. Our bot cleanly won every game in a 10-game match, with substantial score differentials. In addition to this bot vs. bot testing, we were unable to beat the solver ourselves after substantial practice.

Our solver also definitely performs better when compared to the original paper from 2004[1]. Their node processing took anywhere from 3 to 16 cycles at 50 MHz, while ours always took 2 cycles at 74 MHz.

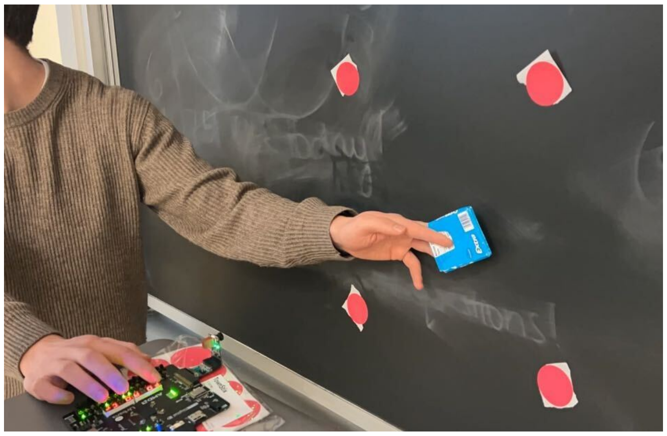


Fig. 13. FPGA Design Testing in Real Life

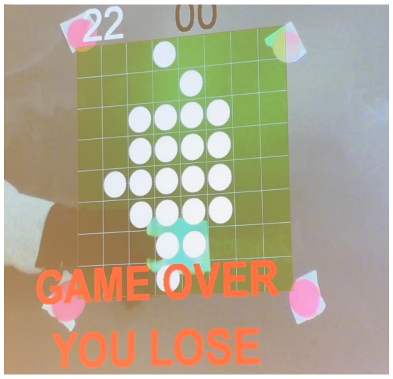


Fig. 14. Game Over Display (We have yet to reach the winning display)

C. FPGA Timing Requirements and Throughput

Our design had to primarily handle meeting the timing requirements of two clocks - the HDMI pixel clock and the solver engine clock. The HDMI pixel clock requires a rate of 74.25 MHz. Whenever we failed to meet this timing, we had to introduce additional pipeline stages. In the end, there is only a 5 cycle delay from pixel read to output. The overlay module takes 3 cycles to compute the proper projections, which then feeds to a sprite module that experiences a 2 cycle delay due to BRAM reads. Additionally, the display projection is technically always a frame behind as the recognition module performs its computations after the HDMI new frame signal, but the human eye cannot notice this delay.

As mentioned previously, the solver clock is set at 74 MHz - we can run this at an arbitrary clock but would like to push the speeds as high as possible. This is approximately as high as we can push it for timing to pass among every path during routing (at this clock rate, the timing does not pass during synthesis and placement). However, an FPGA design with just our solver engine can definitely reach slightly higher clock speeds due to lower resource contention.

We have to also ensure against FIFO overflows in the HDMI to solver clock domain crossings. The recognition module simply feeds the game state engine a new potential player coordinate every frame, so this FIFO will never overflow. The game state engine feeds a constant stream of state to the overlay module, but an overflow will also never occur as the HDMI clock is higher. FIFOs were specifically used to guarantee data synchronization of the data bits.

D. FPGA Resource Usage

Our design used in total 19918 LUTs. The solver engine used 5928 and the game state manager used 4443 - most of the usage comes from the legal moves checker and flipper modules due to their massive parallelism. The MIG itself was also a heavy user of LUTs at 5346. The recognition system only used 2011 and the overlay system only used 465.

Our design used 23 DSP units, with 8 of those used in the recognition engine for k-means calculation and 8 used in the overlay system for projection as well as sprite drawing.

The score digits and game over messages heavily utilized BRAM. Despite being stored as a bitstring of 0s and 1s for each pixel, they utilized 47 36 kb BRAM blocks. This could definitely be optimized, but we were not too concerned as the FIFOs and other BRAM-reliant modules still had more than enough of the remaining 28 BRAM36 and many BRAM18 blocks to work with.

Lastly, Vivado reports that our design uses 1.235 W, with the DRAM MIG system being the dominant user at 0.613 W.

E. Goals Reached

We have satisfied all but one goal in our original proposal for the commitment, the goal, and the stretch. Based on the commitment, we have a working recognition, solver, and overlay projection system. Based on the goal, we can robustly identify boards based on 1 color for the corners, search several layers deep within a second, and have textual overlays as well as smooth video stabilization. Based on the stretch, we have a recognition system based on k-means and can automatically adjust search depth based on the state of the game. Our one missing goal is for a parallelized search, but we realized that alpha-beta minimax isn't a particularly parallelization algorithm due to the propagation of alpha and beta dependencies.

For future work, we suspect that a better control system combined with taking in a 24 bit color camera feed would improve the robustness and filtering of the recognition system to provide an even better game time experience. Additionally, our overlay module could use anti-aliasing, which a simple 3x3 line buffer and convolution module utilizing Gaussian Blur could achieve at the expense of a few more cycles of delay in the video feed.

IX. Source Code

All code relevant to this project can be found at [redacted].

X. Contributions and Acknowledgments

Initially, William worked on the recognition system, Andrew worked on the solver system, and Clay worked on the overlay module. During integration, William and Andrew focused on optimizing the solver engine and everyone contributed equally to the final graphical displays in the design. Thank you to Kiran Vuksanaj and Joe Steinmeyer for their feedback and guidance throughout our project design, and to Kailas Kahler for helping us with Vivado IP usage.

References

[1] C. Wong, K. Lo, and P. Leong, "An fpga-based othello endgame solver," in *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, 2004, pp. 81–88.

[2] D. Lavenier, "Fpga implementation of the k-means clustering algorithm for hyperspectral images," Los Alamos National Laboratory, Los Alamos - NM 87545 - USA, Tech. Rep. LA-UR # 00-3079, Jul. 2000.

[3] M. Tajwar, "Othello game ai." [Online]. Available: https://othello-game-ai.netlify.app/