MIT 6.111 2020

Team 27 Final Project Report

The Game of Go on Two FPGAs

Bill Kusters, Josh Talbot

System Overview and Block Diagram:

For our team's final project, we implemented the ancient game of Go, played individually across two Nexys-4 DDR FPGA development boards. If the reader is unfamiliar, the game of Go is based on two players placing stones onto the intersections of a set-size grid. Players win the game by controlling more territory than their opponent. Stones/groups of stones can be captured based on a few simple rules [https://en.wikipedia.org/wiki/Rules_of_Go#Concise_statement]. The simplicity of the rules lends itself to the challenge of handling all arising situations in the logic of the FPGA.

For this project specifically, each player plays on their own FPGA. Each FPGA renders the board state to the user via its own monitor, and will allow the user to render a possible move and place a stone if the move is valid. The FPGAs communicate to one another via Bluetooth modules, allowing only the player whose turn it is to make a move, then sending the move to the other player's FPGA.

This overview demonstrates four separate functional blocks that have been designed and implemented: a block handling all game and board logic (i.e. transitioning states, keeping state, game logic of valid moves, who wins), a block handling display logic, a block for user interfacing, and a block for communications between the two FPGAs. A full block diagram with all individual models is shown in Figure 1.

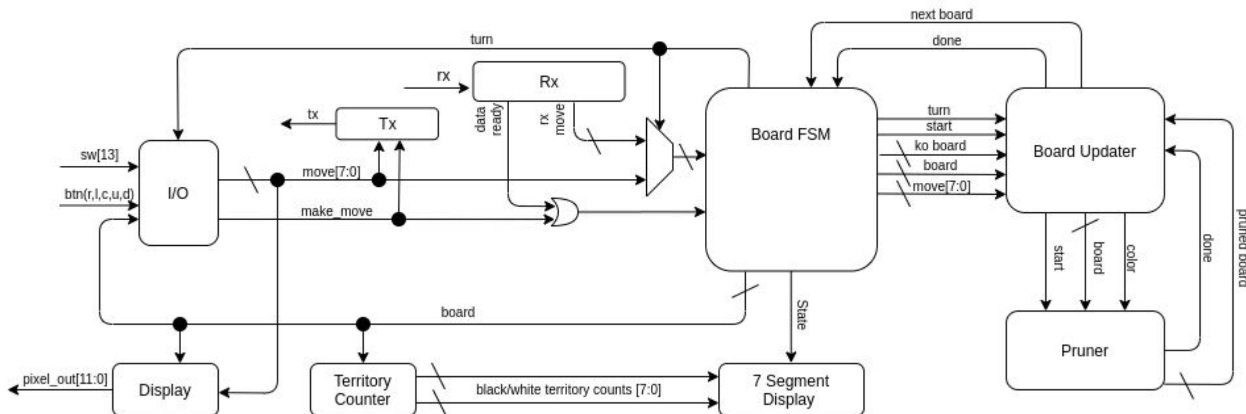

Figure 1: Block Diagram Demonstrating System Architecture

(Only significant signals shown; i.e. no level-to-pulse conversions,
All buses containing the board have size [1:0] board [8:0][8:0])

Moves were communicated via HC-05 Bluetooth modules instead of a physically connected UART. Go board sizes can be 9x9, 13x13, or 19x19 grid, and a set board size of 9x9 was chosen for this project. Automating the progression of board states proved to be the most interesting problem, as it logically requires the ability to capture variable size and shape groups of stones, count variable areas of territory, and maintain and transition the game state.

Display:
Designer: Bill

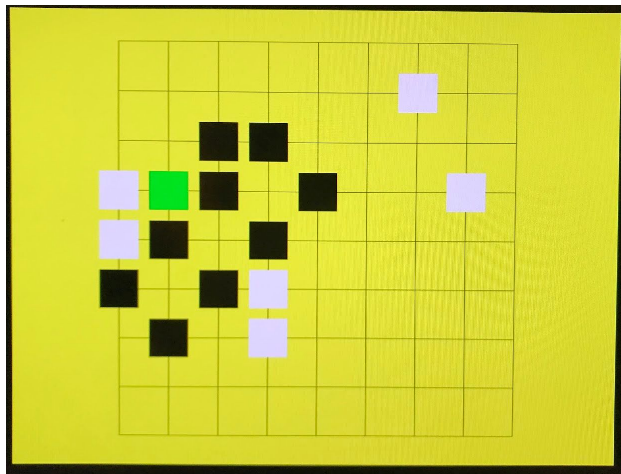

Figure 2: An Example Board State Displayed

The display module takes in as significant inputs both the board state and the position of the cursor. Lab 3 demonstrated how to render objects on the display via dedicated “blob” instances; if we were to use this same technique, a 9x9 board would require 81 separately instantiated and tracked blobs. Given the static nature of a Go board, a simpler approach was able to be taken that involved examining the board state as the pixel coordinate varied in position across the screen.

The pixel value of a given hcount, vcount pair should only be assigned one of a small number of discrete possible values. For instance, a pixel should either be the color of the background (yellow as chosen for the project), the grid (black), a tile (white or black), and the cursor. This fact lends itself to an implementation without blobs. The grid line positions are predetermined and parameterized to the appropriate values for a centered 9x9 board; if the current pixel intersects with one of these lines, it is set to black (as long as there is not a tile covering the grid).

To assess whether or not a given intersection is currently occupied, both a row and column index (named i, j respectively) are needed. These are both set to 0 upon a return of the pixel coordinate to (0,0), as traditional image row/column indexing is used. If the pixel ever encounters an intersection (where grid lines are both registered and intersecting), j is incremented, until j=8, at which point i is incremented. Both are reset after the final intersection is passed.

Now that the occupation status of a given intersection is known, the item remaining to be determined is whether or not the current pixel location needs to be evaluated as a tile. Stated another way, this is if the pixel location is within the bounds of the visual space of a tile. Instead of precomputing this, the following procedure is followed: parameterize the desired width of a tile (slightly less than the space between intersections works well). After a reset, use the first pass

of hcount and vcount across the screen to push intersection coordinate values into a 9x9 array; this is done by checking whenever the pixel coordinate coincides with a vertical and horizontal line simultaneously. Once these lower and upper bounds associated with each intersection have been stored, the system is ready to render a game state. Check if the pixel is within the bounds of the current (i,j) pair, and if board[i][j] is also occupied, render the corresponding tile color (black or white in the case of tiles, or green in the case of the cursor). This can be seen in the display module code as checking for whether or not the current pixel coordinate is "tilebound", "on_tile", and outputting "tilecolor".

Finally, the cursor position is wired to the display module from the IO block, and is registered as a game tile with the color green. Now that the display module is rendering board states, we can begin to interact with the board states and see them change.

Board Pruner:

Designer: Josh/Bill

To progress a game state forward, the tiles that have been captured need to be found and subsequently removed from the board. Software-based Go games retain an array of representations of all disjoint living groups on the board per player, as well as the number of liberties for each group. This allows algorithms to quickly determine whether or not a group goes to 0 liberties and needs to be pruned from the board after a move. This representation of groups is difficult to implement on the FPGA, so a novel approach was generated.

The pruner follows an implementation of what is essentially a breadth-first search built within the constraints of the FPGA. What needs to be computed in deciding whether or not a tile must be pruned is whether or not that tile has access to a liberty, either directly or through its own connected group. The difficulty lies in determining the liberty status for a member of a living group far away from any of the liberties of the group. Logically, the rule is simple: if a tile is directly connected to a liberty, it has a liberty. Otherwise, a tile has a liberty if any of its same color neighbors are connected to a liberty. If neither of these cases is true, the stone has no liberties and should be pruned from the board.

At the time of project proposal, our team thought of a circuit using commonly available lab components that could determine a tile's prune state. The circuit can be found in Figure 3. The idea was to have a circuit that automatically set an intersection voltage. For a given intersection A, with adjacent intersection C: A should have its prune level (voltage) tied to C's if either A or C are empties, or if they are stones of the same color. This directly follows the stated logic, and will tie the voltages of A and C to the same value under the conditions provided. Thus input B in Figure 3 would be the output of a mux which is high only when the stones are the same color or when either intersection is empty. Inputs A and C would be high when A or C are empty intersections. This circuit could have its operation demonstrated in the following scenario; if A were empty and C was filled, the transistor at B would act as a low resistance connection. The transistor at A would do the same, setting the voltage at the node above the resistor on A's side high. This high level is connected to the same node on C through the low resistance

connection of the transistor at B, so the level is high there as well. Intersections at a high level would not be pruned on checking by design, and the circuit correctly ties the stone at C to a safe value. The pulldown resistors tie an occupied intersection low unless tied to a high voltage through the transistor of an adjacent intersection (either via a same-color stone or an empty). If no paths to a liberty's high voltage are made, an intersection sets at a low voltage. Occupied stones with a low voltage would be removed from the board. Transistor B does not connect stones of different colors to prevent the sharing of a prune voltage as desired.

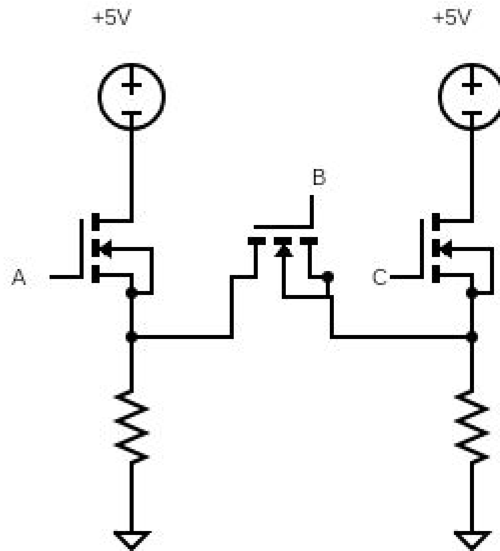

Figure 3: Original Hardware Concept for Prune Determination

A, B, C represent mux outputs depending on the values of the intersections in question.

This circuit has not been built and tested, as this is not within the design of implementing the Go board on an FPGA. It served as a useful mental model for the desired characteristics of a working digital abstraction throughout the implementation process.

At first glance, it seems a combinational circuit at each intersection should mimic the described behavior of Figure 3. If the prune values of all adjacent tiles are set, simply OR the prune values of each adjacent tile to determine that tile's prune value. However, this only works to set the prune values of a group high at connection time. If the liberties of the group are fully removed, the ORing holds each tile high even after the removal. This is the result of the circular dependence of each prune value, caused by a combinational loop. Thus, pruning is not an easily defined problem using the values of tiles, as once set high, they will be floating once closed off from any liberties. Floating prune values could possibly be implemented as a use case for three-state (high impedance) logic, but the doing requires greater knowledge of Vivado constraint manipulation than was present at the time of this project.

An insight from attempting this approach was that it might be possible to use the state of the wires connecting each intersection to determine prune value. This relates to the original

circuit in Figure 3, which essentially decides whether to connect adjacent intersections via a wire. A wire would be "hot" if it connects tiles to empties or other tiles of the same color, or "cold" if it connects pieces of opposite color, where "hot" and "cold" are used as terms to represent connection and disconnection, respectively. If a wire connects a tile to empty, it can only be hot. If a wire connects tiles of the same color, the wire's hot/cold state is the logical OR of all its neighbors, as shown in Figure 5.

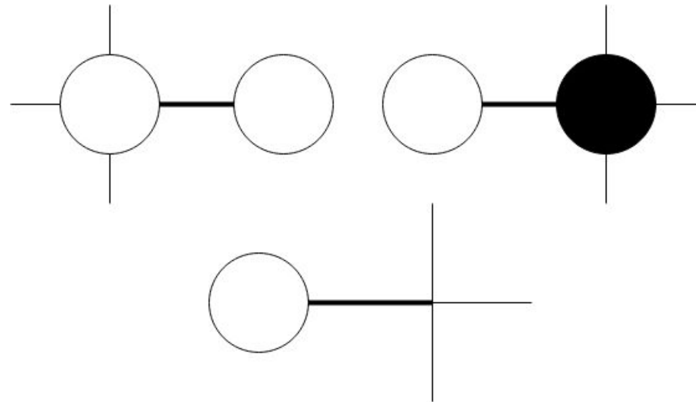

Figure 4: Three Possible States of a Wire

Top Left: wire connects same color
Top Right: wire connects opposite colors
Bottom: wire connects tile to empty

Given a wire's state is directly assigned if it connects to an empty or to two opposite color tiles, the wire connecting two same color tiles needs to be determined. A wire in this case is the OR of all of its directly connected neighbors, illustrated in Figure 5:

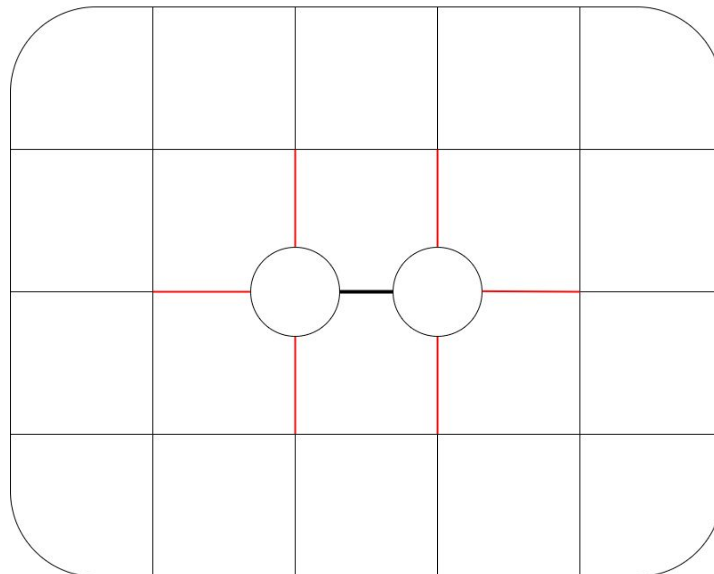

Figure 5: Wire Neighbor Illustration

The OR of wires in red determine the value of the bold wire in the center.

The horizontal case is shown. To compute a vertical wire's state, the illustration can just be rotated 90 degrees. In an edge/corner case where there are fewer than six neighbors, an individual wire's state determination module is wired to a cold value for each missing neighbor. A tile's liberty status is then also the logical OR of all of its connected wires. Using this approach, enclosed groups will always have a clearly defined value as a part of their OR expression, but will be held high when transitioned from one liberty to none as seen in Figure 6:

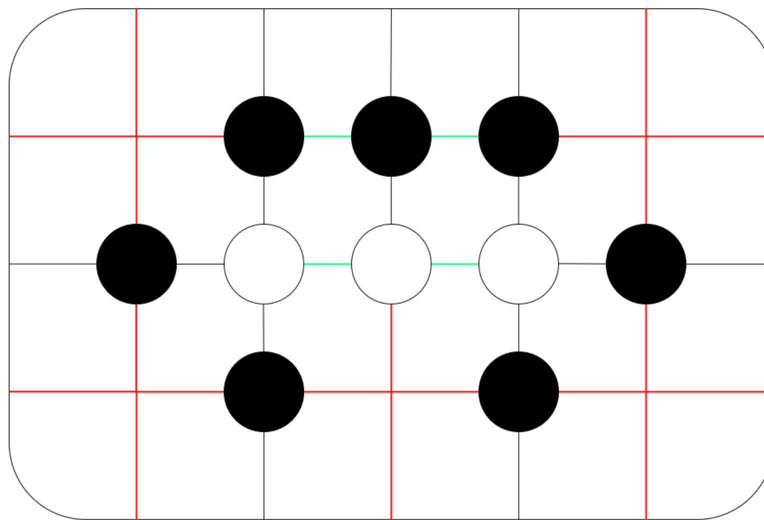

Figure 6a: White Group Alive

Red wires evaluate directly to "safe". Green wires are the result of the OR to all neighbors, also computing to "safe". Edge wire states only colored if definitively known from given state.

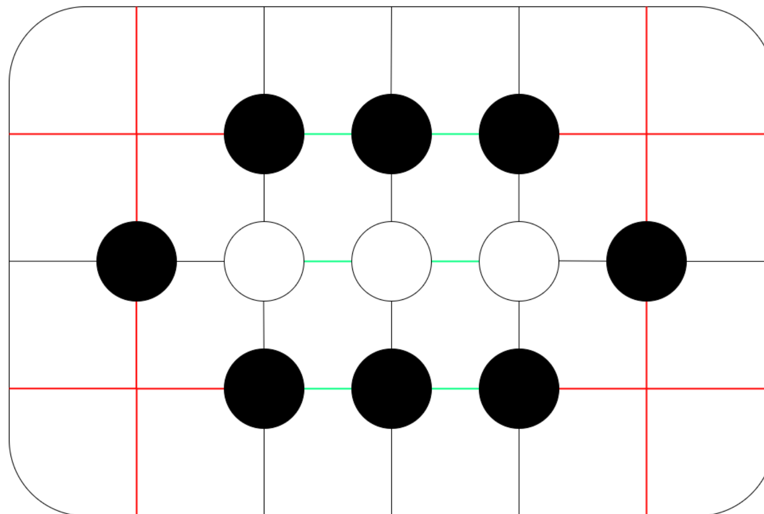

Figure 6b: White Group Dead, Held Safe

The white group demonstrates the combinational loop; even after removing any set values, lhe OR holds the inside group high.

This dashed our hopes and dreams of building a purely combinational pruning module. However, this circuit was not without merit. The model is functional as a prune value setter if it is possible to set each wire's value to 0 briefly upon a new move being made, before allowing

wires connected to an empty intersection to take on their usual value of 1 at the next clock cycle. To do this, the wire calculation modules must take samples of their neighbors' module outputs as inputs as opposed to the module outputs directly. Each register is pulsed low on the clock cycle a turn is made, and then set to their computed value on the next clock cycle. By sampling the output of each wire calculation module, this allows the empty intersection to propagate in their outputs with each clock cycle. This process is demonstrated in Figure 7.

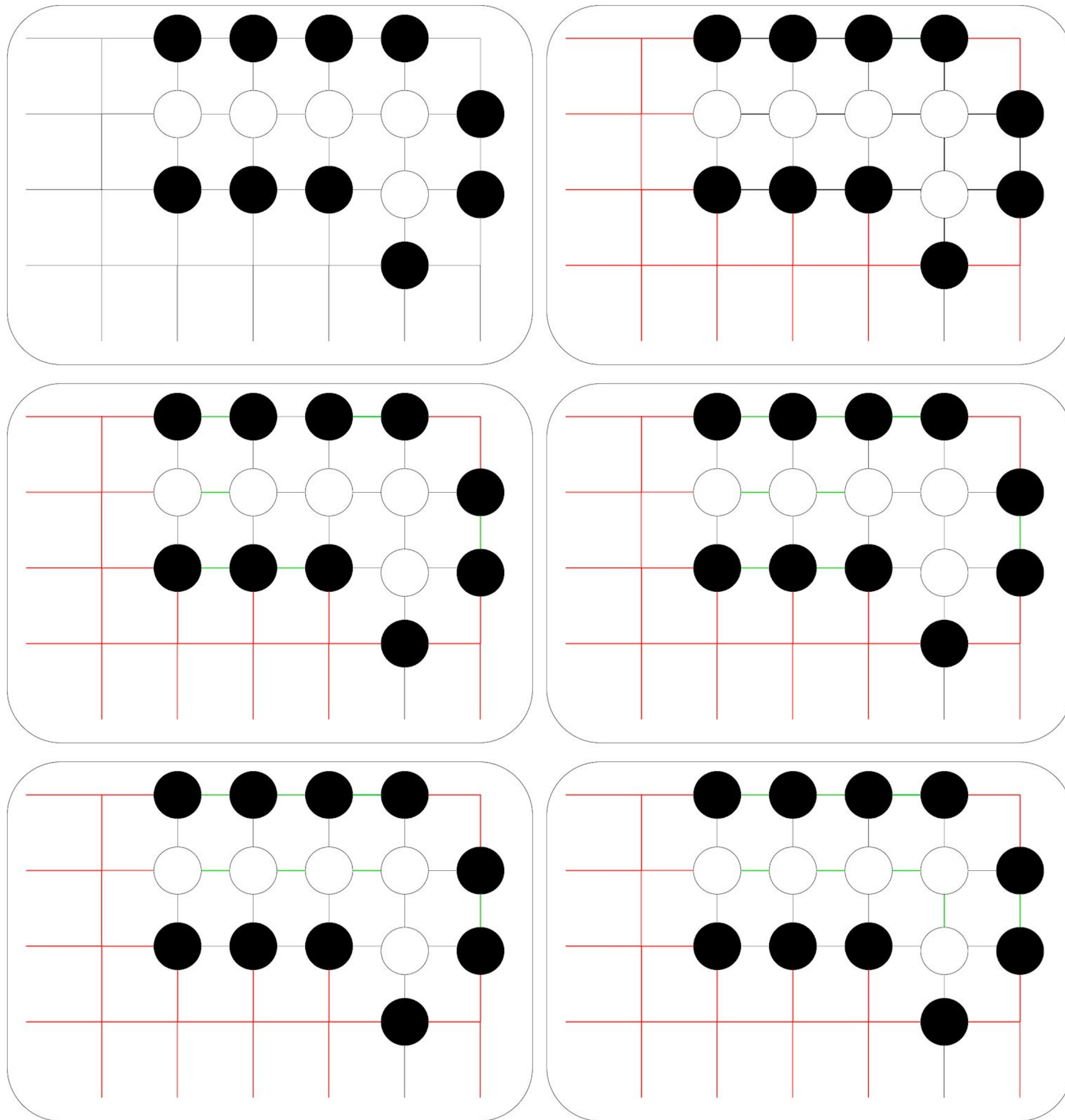

Figure 7: Propagation of Liberty Into Groups

From top left to bottom in order: all wire latches are assigned a low value on the first clock (black). On the next clock, wires are assigned the result of their wire calculation module (empty connections go hot, shown as red). Then, the wire values propagate by sampling the wire calculation results at every clock. The results of a hot wire from an OR are shown in green. The green wires move inwards to a group at every clock cycle due to the latch value propagation. End this process after a delay of worst-case propagation time to ensure stones furthest from a liberty have their correct value set before examining their state.

Figure 8 goes on to show how this same pruning/propagation process would work if black placed a stone enclosing the white group. The original issue of the combinational loop of a group holding itself high is fixed due to the pulsed low assignment to the wire state latches, and the wires internal to the white group never become hot. The examination of the OR of each white stone's wires results in their state as staying in "prune", and the pruner holds the correct board on its output.

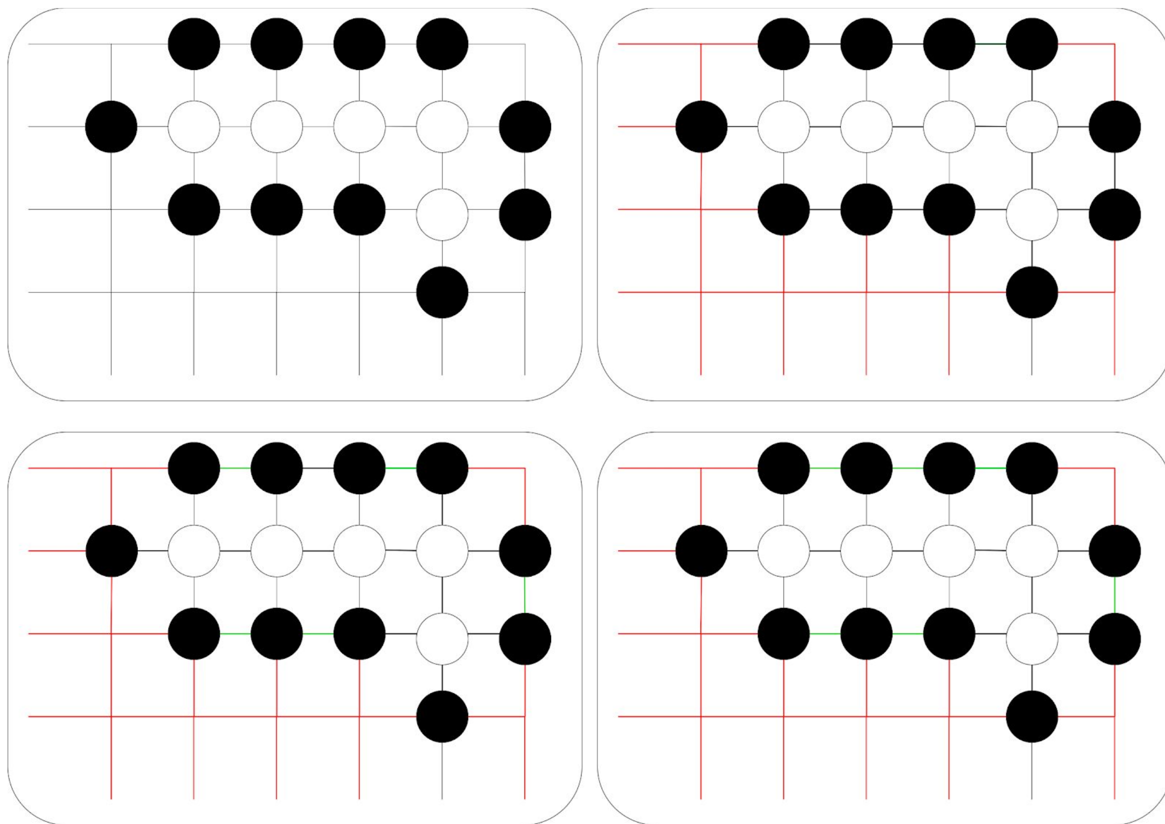

Figure 8: Pulsed Propagation with an Unreachable Group
The diagram is colored as in Figure 7.

This propagation can be viewed computationally as a breadth-first search with empties set to be source nodes. A BFS is a common technique in software implementations of Go to determine if tiles are connected to liberties, but such graph search algorithms are difficult to implement directly on digital logic. The pulse-latch propagation technique allows empties to propagate their values in without a directed search; information about the graph in question could be retained as in a traditional BFS, but for the purposes of a Go board simply knowing whether or not the nodes (intersections) are connected to a liberty is enough.

The BFS executes neatly in parallel, taking advantage of the FPGA by computing each BFS sourced from every intersection concurrently. A traditional software implementation would need to iteratively set all BFS attempts, so a speedup is obtained.

In order to execute the steps outlined in this module description, the pruner has been implemented as an FSM. The pruner waits for a signal to examine the board on its input bus,

pulses the latches low as described, allows the wire values to propagate, and then raises a done flag before returning to the waiting state and resetting the prune delay count. The FSM itself is simple, with its state transition diagram shown in Figure 9.

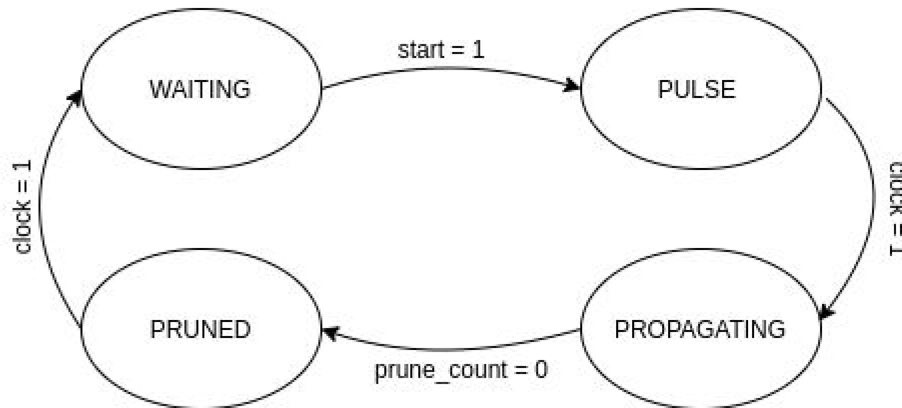

Figure 9: Prune Module FSM Transition Diagram

Game FSM:

Designer: Josh

The game state FSM acts as a way to maintain and progress the board state. It maintains information such as who's turn it is, the previous board state (to prevent violation of the ko rule), if the last move was a pass, and whether or not the game is over. Computation is modularized into the modules of the board updater and the pruner so the FSM implementation can remain simple. The transition diagram is shown in Figure 10.

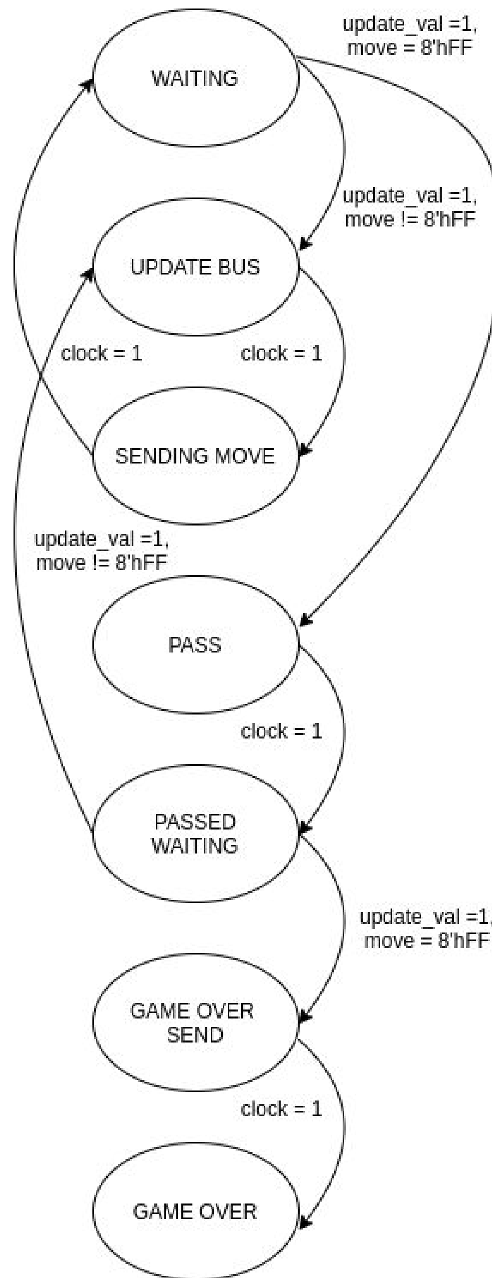

Figure 10: Game FSM Transition Diagram

Upon a reset, the game FSM assigns an empty board to the board on the output bus, an empty board to the ko registers, the turn starting with black, and a low on the tx ready line. Once this is done, the FSM enters the waiting state. Once either the user IO module or the rx ready line pulse, the FSM continues waiting until the board updater has computed the next board state and verified its validity, at which point the FSM transitions into the update bus state unless the move was a pass. Passes must be handled as special cases, as they do not change the board state and two consecutive passes will end the game.

The update bus state pushes the computed bus state from the board updater onto the output bus, and retains the board on the output bus previously as the ko board for the updater to use in future checks. The FSM never stays in this state longer than one clock cycle, immediately progressing into the sending move state once the buses have been updated. The sending move state raises the tx ready line if the move happened on the FPGA player's turn, otherwise it was a received move and should not be sent back to the opposing player. The turn is inverted in this state as well. This state lasts for only a clock cycle before returning to the waiting state.

If a pass was made, the FSM enters a special pass state that operates similarly to the update bus state without changing the buses, but switching the turn. On the next clock cycle, the FSM progresses into the passed waiting state, at which point, if a second pass is received, the game will end after relaying the game over signal to the opposing player. If a regular move is received, the FSM reenters the update bus state in a similar fashion to the original waiting state.

While moving through these states, the game FSM module acts as a synchronizer for the rest of the system, ensuring the turn is available to any other modules that require it and transmitting moves after verifying they do not result in a ko or suicide as reported by the board updater. By modularizing the computation into submodules, the FSM can be simplified into a module focused on handling transitions rather than computation.

Territory Counter:

Designer: Josh

The groundwork for the territory counting module is laid by the pruner. By altering the state of the board on the input bus into a variety of forms and pruning, the territory counter is able to determine exactly how much territory each player fully encloses with their stones. A simple FSM similar to the pruner is used to decide when to examine the board state on its input bus. The transition diagram is shown in Figure 11.

In order to determine territory counts of each player, a few alterations of the board state are maintained combinationally. The board states used match what is on the module's input bus except for one characteristic difference each. To determine black's territory count for instance, we hold the following two boards combinationally: a board where blacks are swapped with empties and vice versa, and a board where all whites are removed but blacks remain and empties are unaltered. The first board described is pruned and subsequently XOR'd with the unpruned board. This resulting board and the board where whites were simply removed are both one tallied, with the results added together to determine black's total territory count (number of black stones plus number of intersections that only connect to black's stones).

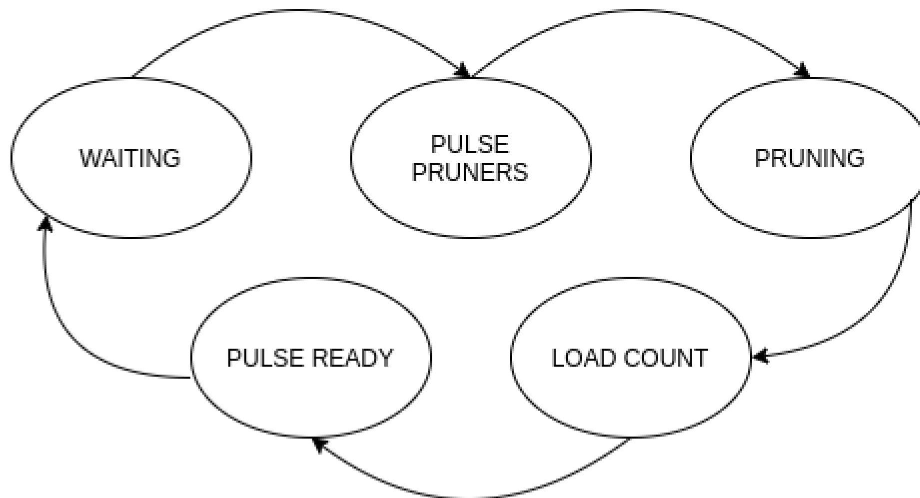

Figure 11: Territory Counter FSM

The management of this process is done via an FSM. When the start pulse is received, the territory counter exits the waiting state and pulses the start pulse of its pruners. The FSM then immediately transitions into the pruning state, waiting for the doubly swapped board (opposite color swapped with empties, empties with opposite color) to finish pruning. Once this occurs, the counts of the one tallier are pushed into the output regs before entering the pulse ready state that asserts data available to the rest of the system.

Board Updater:

Designer: Josh

The board updater acts as a computation tool for the FSM. The main board pruner is instantiated within this module. Due to the event based nature of the input signals and the actions required upon assertions, an FSM is again used to implement this module. The transition diagram can be found in Figure 12. The primary functionality of this module is to start the pruner after a new move has been applied and to hold the next board state after determining its validity.

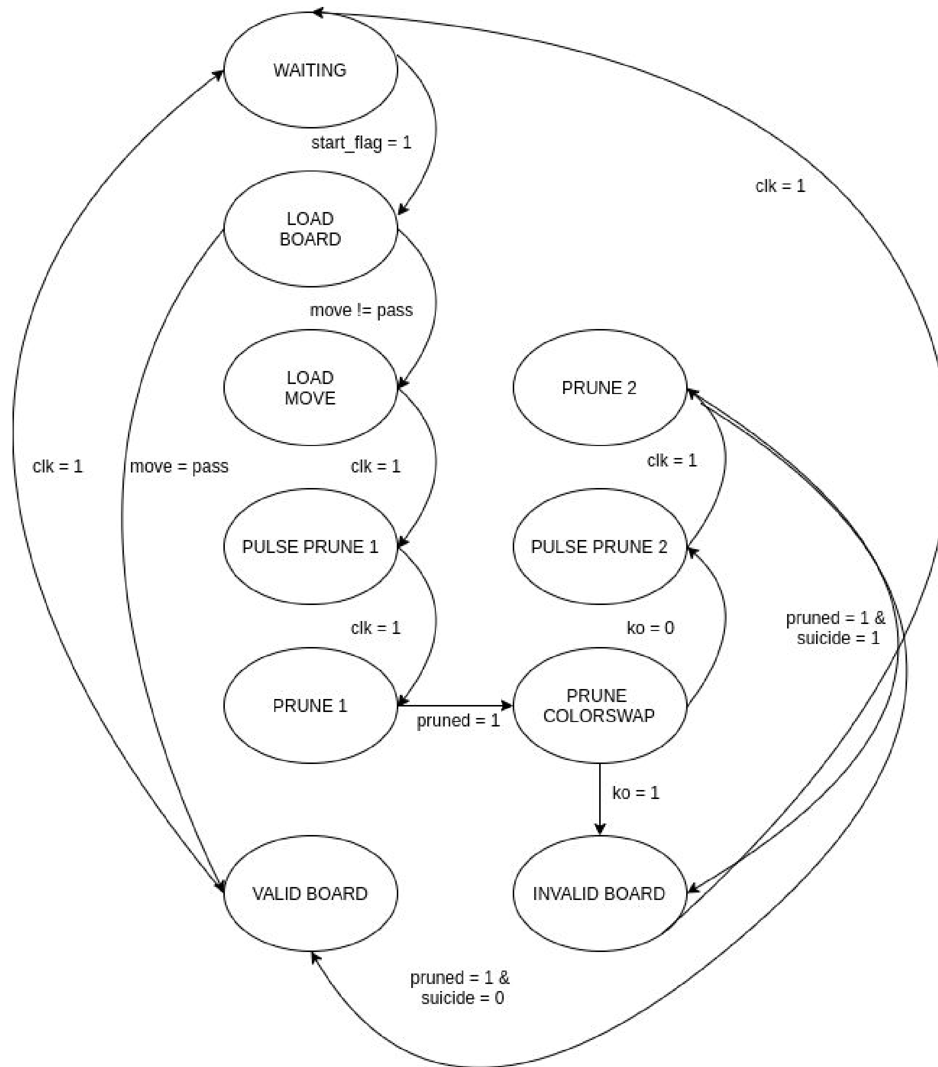

Figure 12: Board Updater FSM Transition Diagram

On reset, the board updater sets all registers to 0 to hold an empty board on its bus and validity flags to 0. When in the waiting state and a start flag is asserted, the board updater begins its computation. In the load board state, the board updater checks for a pass, which immediately transitions to the valid board state. Otherwise, it applies the move in the load move state. After applying the move to the board, it prunes the board by pulsing its pruner and waiting until the ready flag is asserted from the pruner. Once this is complete, we need to try pruning the color of the player who played the move. In the prune color swap state, we swap the color we are attempting to prune and check that the first prune did not result in a ko with an equality check between the pruned board and the ko board. If it did not, we attempt to prune the color of the player who played the move. If any stones are pruned, this results in an invalid board as there was a suicide. The validity states exist to pulse their corresponding flags before being reset in the waiting state. If the invalid flag is asserted, the game FSM does not progress to the next turn and

accepts a new user input. Otherwise, the game FSM will accept the board from the board updater and progress to the next turn.

TX:

Designer: Bill

The implementation of a UART transmission module follows the 6.111 lab 2 implementation nearly exactly. A shift buffer is used to load the 8-bit move in from the user IO module, and is parameterized to send a bit at a time at 9600 baud to the HC-05 Bluetooth modules.

RX:

Designer: Josh

The UART receiver samples bits at 16-times the baud rate of 9600. An FSM is used to wait after reset or completion of receiving a packet to prevent sampling after a spurious low value on the transmission line. After receiving a byte in full, the rx ready signal is pulsed to the FSM so the FSM will parse what is on the move bus as the next move to be computed.

Use of the HC-05s:

A note on the use of the HC-05 Bluetooth modules; they interface directly with a UART at a standard 9600 baud. The tutorial shown here [https://www.youtube.com/watch?v=BXXAcFOTnBo&t=130s] was used to initialize the modules and pair them. They do not have a guarantee on inter-byte spacing, so a delay is needed to be used if, for instance, whole board states were to be sent.

User IO:

Designer: Bill

The user IO module interfaces with the asynchronous signals generated by a user to change a board state. Considering this, its intent is to register a possible move to the user and allow the user to change what that possible move is. Once the desired move is chosen, a separate input is made to register the choice to the game FSM. A clear design choice to satisfy these requirements is a cursor.

The cursor does not allow the user to make an invalid input, but enables the user to move between valid spaces to play a stone around the board. Due to the asynchronous nature of the input signals to the module, and the states of the system that arise when waiting for these signals, an FSM is used. The FSM either waits for user input, processes the user input, raises a send flag, or locks input when it is not the current user's turn. The state transition diagram is shown in Figure 13:

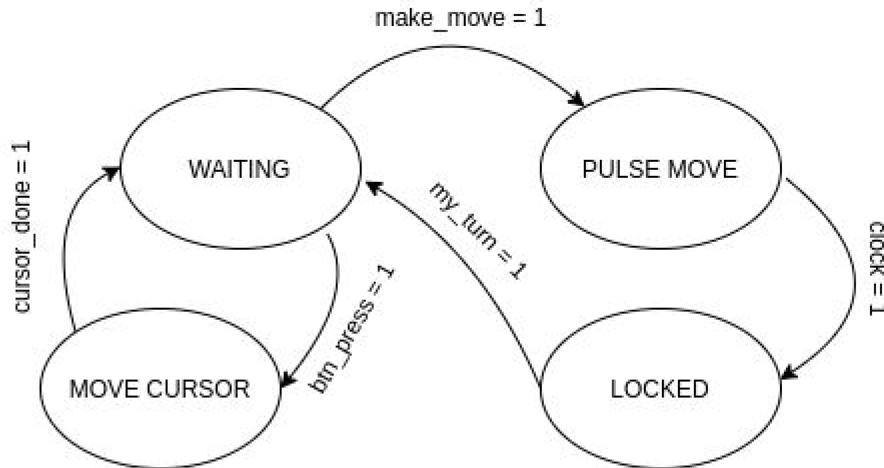

Figure 13: User IO FSM Transition Diagram

The interesting state in this module is "move cursor". The cursor is designed to skip over existing tiles if directed to move across a tile so as to never be in an invalid position while accepting user input to send the move. It should also never allow the user to move the cursor off the board. For instance, if the cursor were positioned immediately to the left of two adjacent stones in the middle row of the board, and the user were to press the right button, the cursor will move to the closest position to the right that is a valid move; i.e. after the consecutive stones.

This works as follows: when the user presses any direction button, the FSM transitions into the "move cursor" state after sampling the input direction into a register. A register is used to hold the pulsed direction value throughout the duration of the computation. The "move cursor" state checks the position immediately adjacent in the direction sampled; if that position is valid (not off the board and unoccupied), it moves the cursor into that position and assigns the "cursor done" signal to high, exiting the state. If the position is not valid but is not past the edge of the board, the module assigns the move, not the cursor position, to that space and stays in the "move cursor" state. It reenters the state now checking the position two spaces over in the same fashion. This process continues until either an unoccupied intersection is found or the move position increments off the edge of the board, at which point it resets to the cursor value and raises "cursor done", exiting the state.

This probing for open positions ensures the cursor is never moved into an invalid position, enabling the designer to make the assumption throughout the rest of the system that the FSM will only need to handle legal moves coming from the input module.

Seven Segment Display:

Designer: Josh

The seven-segment display driver of this project is similar to a standard implementation. Alphanumeric states are parameterized so messages can be displayed to the user, such as if the most recent state is a pass, whether or not a pass move is selected, the territory counts, and the winner/loser of the game on a double pass. The driver receives the territory counts of both

players and the state of the game to decide whether to display “winner” or “loser” based on the territory counts at the end of the game.

AI:

Designer: Bill

The FPGA interface to the AI requires no modification of the source code used for bitstream generation. Due to the use of the UART to communicate moves, the FPGA is agnostic as to what is sending moves to its FSM from the outside world. As long as the move is valid, the system will progress as usual. Go has been an object of study in artificial intelligence since antiquity, resulting in a number of open source bot implementations across GitHub. Despite this, finding one to interface with to parse moves and examine board states was nontrivial.

Fortunately, one library followed a state progression logic similar to one implemented on the FPGA. This library (https://github.com/maxpumperla/deep_learning_and_the_game_of_go) is intended for use alongside a book written by the creators to walk the reader through training their own implementation of the DeepMind AlphaGo bot. To do this, the library contains code to accept input from a human in the format of a row and column to verify behavior of the available bots. This is the functionality we used to interface to the FPGA. Instead of typing the move, the move input waited for a received move from the UART on the FPGA, then parsed the byte received into a row, column pair which was printed and used to update the board state. The bot then computed its move, which was written to USB and received by the FPGA UART, which updated the FSM as normally receiving a byte would. This enables the user to quickly play moves using the FPGA. Figure 14 shows the displayed board state of the library.

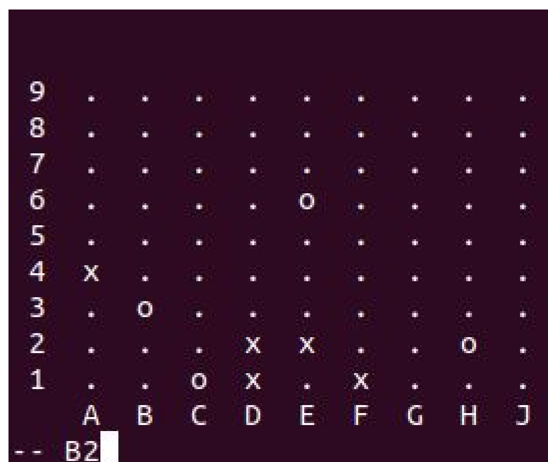

Figure 14: Bot Interface shown on Linux Terminal

The Python script written to handle the interfacing is named “serbot.py” in the top level of the project GitHub repository. The script must be running to begin a game with the AI. An easy way to build and run the needed libraries and script are through the use of the container

platform Docker. The library itself is attached with a Dockerfile, which when built gets the user ready to run the AI. serbot.py should be made available within the container, and the pyserial library installed. If the container is correctly configured with the correct USB port access, the AI is set up and ready to go!

The FPGA needs its UART wired to an Arduino as in Lab 2 when sending ASCII characters to a computer. Wire the FPGA's tx/rx line to the corresponding lines on the Arduino Leonardo, and from there connect the Arduino to the PC with a USB cable, and run the code for looped serial writing and reading available on the website (write to USB what's received on the rx line, and write to the tx line what's received on USB). After this setup is complete, the user should be able to interface with the running AI from their FPGA as if they were simply playing another player.

Reflections and Future Work:

This project was very rewarding. Spawned from a night of working on 6.111, the question was posed as to how one could one use the constraints of hardware to create a simple way to prune the captured pieces from a Go board, i.e. by differentiating captured pieces with a logic high/low when living pieces were always held in the opposite state. That led to the design of the circuit presented early in this report. The team still has yet to test that implementation but is curious to see it through.

Naturally, the most difficult component of this project was pruning the board. All other tasks felt more naturally within the design paradigms presented throughout the class. At first, the project was started with the idea that the next Go board state could be determined with entirely combinational logic, leading to a computation only limited by propagation delay. More than a day was spent trying to make the combinational solutions presented in the report functional. It seems to be the case that there may be a way to do so; the combinational circuit using the wires of the board could work if the wires could reliably be pulsed (or glitched significantly) into the cold state. A solution working within two clock cycles, where latches are pulsed but a compound combinational circuit from those latches is used might work and is of interest to the team. The words "Fastest Go Board in the West" were spoken at the beginning of the project, but were stowed away once the sequential logic ended in a runtime scaling with the size of the board more directly.

A boon to the team throughout development was heavy use of version control in the form of Git. Good branching practices in order to have a known working implementation of subsets of modules was reassuring at times and vital in others. It is encouraged that anyone looking to follow the implementation in this report do the same; the habit saved many headaches.

The HC-05 modules have the property of no guarantee on inter-byte spacing for transmission. Unfortunately, despite examining many datasheets, this was not known to the team. Originally, to test the communication modules, we attempted to send entire board states back and forth. This means 162 bits in one packet, which worked beautifully over a directly wired UART. This was not the case over Bluetooth. We never planned to send whole board states in the final

implementation, but did plan to as part of development. Actionable advice from this experience can be condensed as follows: ask about how parts work if you know people familiar with the parts.

Interfacing with the AI, from a technical perspective, was more about appropriately parsing the move byte format into a format usable by the AI. At the end of the day however, it was rewarding to be able to play the game of Go from just an FPGA and a computer as a standalone game. The AI was a fun stretch goal; it would be interested to attempt to implement a full bot on the FPGA that could be activated with the flick of a switch.

Go is a beautiful game even to novice eyes, and we loved the experience of getting a bit closer to the game through computation. The source code for the project is available at the following link:

https://github.com/wmkusters/fgbitw