Course Name: EMBEDDED SYSTEMS III

Course Number and Section: **16:332:579:05**

Year: Spring 2023

Final Project Report

Instructor: Milton Diaz

Student Name and RUID: Junior Brice, [redacted]

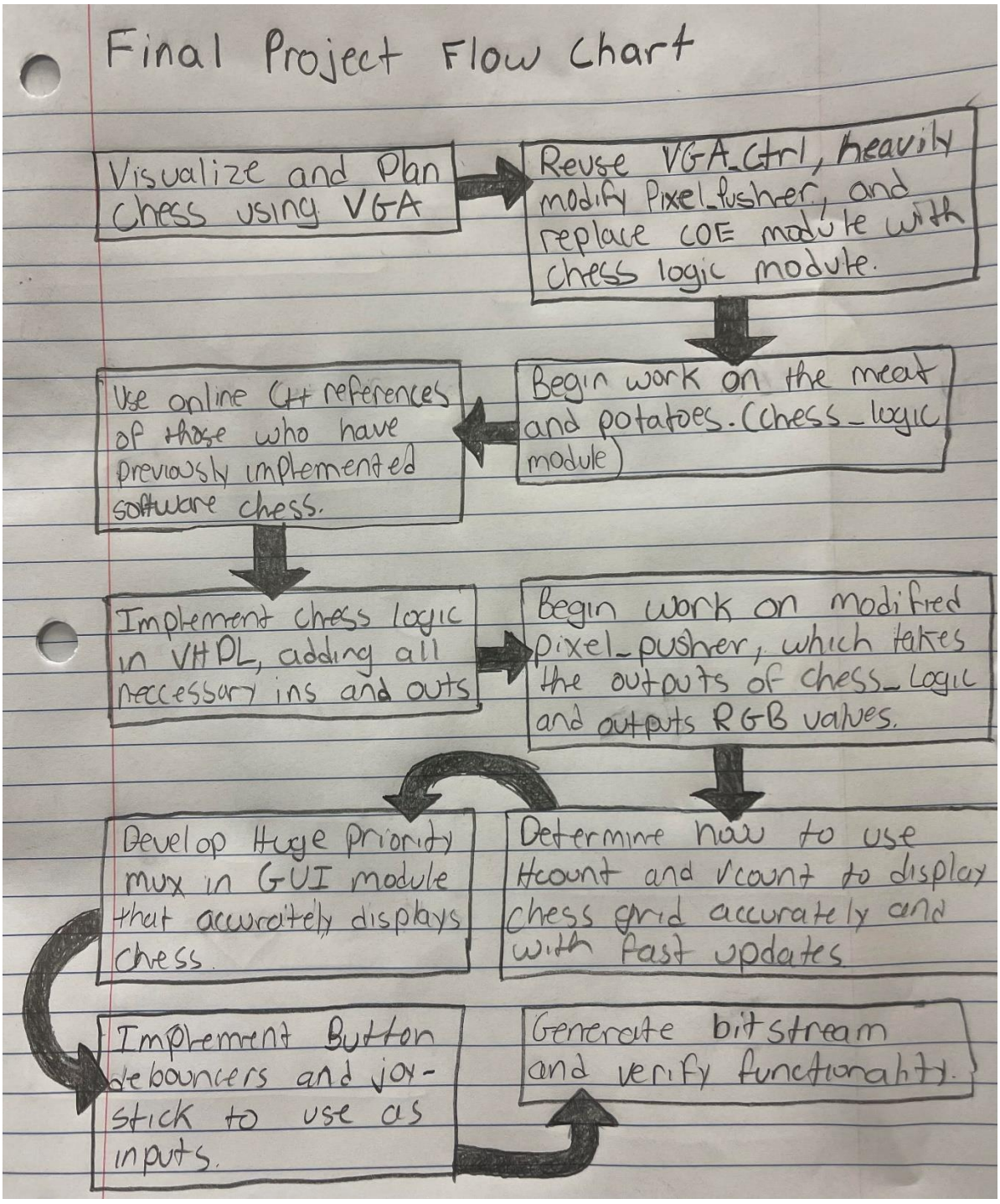
Date Submitted: 05/07/2023

GitHub Link: [redacted]

Purpose/Objective:

The purpose of my final project was to implement a functional version of chess on the Zybo Zynq-7000 board.

Theory of Operation:



Disclaimer:

This project was admittedly complicated to realize. There are a significant amount moving signals and components that make it difficult to keep track of everything unless you have a higher understanding of the main dataflow. This report will aim to explain said dataflow to you through examples, but keep in mind it is by no means a substitution for looking at the code yourself, if that is what you choose to do. For that reason, as I was designing this project, I made sure to frequently add comments to nearly every aspect of it, to make sure it is somewhat easy to follow. I will do my best to explain here, but if you want the best explanation or have any questions, the RTL code is your best friend!

Basic Explanation and Diagrams:

This project is broken up into 4 main blocks. The Input block, the Chess Logic block, the VGA Control Block, and the GUI block. In total, there are 4 inputs to the system and 8 outputs. Please have a look below. Inputs are in **RED**, **outputs in BLUE**, and intermediates in **BLACK.**

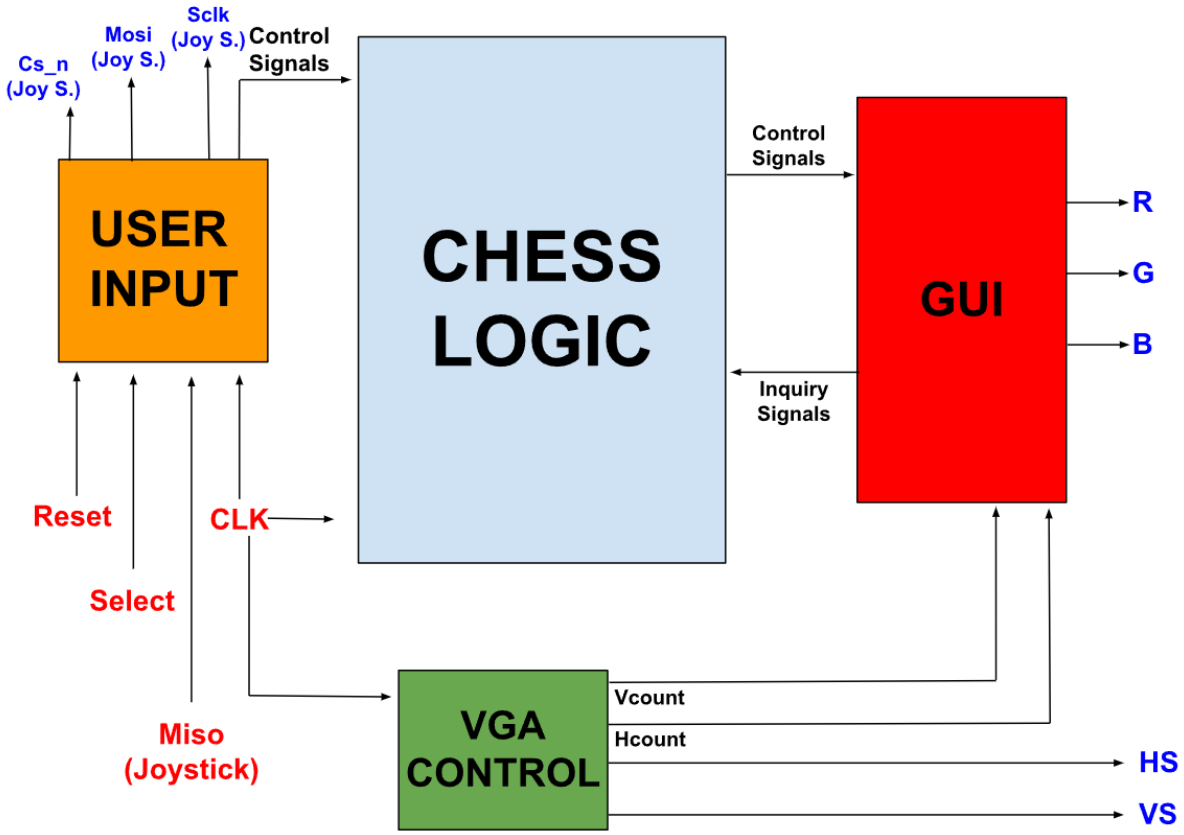


Figure 1: Main Block Diagram

Let us analyze a basic instance of the dataflow. We start in the **USER INPUT** block, where external signals from the user's joystick, reset button, and select button are passed through debouncers and a joystick control module, then are sent to the **CHESS LOGIC** block as control signals. The debouncers are of the simple counter style, and the joystick control module uses the SPI protocol to communicate with the it (hence the dedicated 4 inputs/outputs for the joystick alone).

Once the signals are done being transformed in the **USER INPUT** block, we send them into the **CHESS LOGIC** block. They are used to control the game of chess being played within the **CHESS LOGIC** block. Inside said block we have only a single module, and within said module, we perform operations on an 8 x 8 memory array in VHDL which represents our 8 x 8 checkered chessboard. Instantiated as integers, this array could house any integer value between -6 to 6 in one of its 64 total spots. It looks something like this:

```
Board <= ((-5,-4,-3,-2,-1,-3,-4,-5),    --This is the board we will be modifying!
          (-6,-6,-6,-6,-6,-6,-6,-6),    --The negative numbers are the dark pieces,
          (0,0,0,0,0,0,0,0),            --and the positive are the white pieces.
          (0,0,0,0,0,0,0,0),            -- 6 is PAWN
          (0,0,0,0,0,0,0,0),            -- 5 is ROOK
          (0,0,0,0,0,0,0,0),            -- 4 is KNIGHT
          (6,6,6,6,6,6,6,6),            -- 3 is BISHOP
          (5,4,3,2,1,3,4,5));           -- 2 is QUEEN
                                        -- 1 is KING
                                        -- 0 is EMPTY
```

Figure 2: 8 x 8 Chess Memory Array in VHDL

Notice, we have selected each of the integers in the range of -6 to 6 to represent a different type of piece or an empty spot.

Throughout the entirety of the **CHESS LOGIC** block, we modify this memory array based on the input control signals that initially originated from the user. I will provide a basic example. Say a user (playing white) wants to modify the original memory array layout and perform one of the most widely used opening moves in chess, pawn to D4, otherwise known as a Queen's Pawn Game opening (see below).

Figure 3: Pawn to D4

Well, to make this move, the user will have to use the joystick to select the pawn, then select the D4 square. Let's see how this might look in VHDL:

```
elsif(Sel_Xreg = Prev_Xreg and board(Prev_Yreg-1,Prev_Xreg) = 0
        and board(Prev_Yreg-2,Prev_Xreg) = 0
        and Sel_Yreg = Prev_Yreg-2 and Prev_Yreg = 6) then --mechanism for inital 2
                                                           --space pawn progression.
            board(Sel_Yreg,Sel_Xreg) <= temp_piece;
            Piece_Selreg <= '0';
            temp_piece <= 0;
```

Figure 4: White Pawn Initial 2 Space Movement

Here, we are in the VHDL ***if*** statement which dictates white pawn movement. *Sel_Xreg* and *Sel_Yreg* indicate the selected x and y position of the user cursor (respectively) to control the game. *Prev_Xreg* and *Prev_Yreg* indicate the position of a selected piece.

- ➔ Notice, since pawns can only move forward when not capturing, the first condition of this branch of the if statement requires that *Sel_Xreg = Prev_Xreg*.
- ➔ The statement then proceeds to check that the two spaces ahead of the pawn are both empty (i.e., *Board[Prev_Yreg – 1, Prev_Xreg] = 0 and Board[Prev_Yreg – 2, Prev_Xreg] = 0*; where if you remember, 0 indicates an empty spot).
- ➔ It also checks that the pawn is currently located in the 7^{th} rank, since pawns can only perform this double space move if they had not been previously moved (so *Prev_Yreg* must equal 6, as 6 indicates the 7^{th} rank on an integer array of 0 to 7).
- ➔ And finally, the newly selected position of Y position of the pawn move must in fact be two spaces ahead of the pawn (*Sel_Yreg = Prev_Yreg-2).*

If all these conditions are true, the selected board position gets updated with the value stored in *temp_piece*, which due to logic earlier in the module will always be the same value as the piece previously selected. We also exit *select mode* (*Piece_Selreg* gets updated from a '1' to a '0'), and the value in *temp_piece* gets cleared. If none of the conditions are true along the *entire* if statement for white pawn movement, the pawn's position will not be updated and *select mode* will be exited.

Now that you have a taste of how piece movement operates in the **CHESS LOGIC** block, you can imagine how complex it got for some of the more intricately moving pieces! Because of this complexity, the logic module was over 900 lines long! Continuing, as the user continues to select and move pieces using the joystick and select button, the **CHESS LOGIC** block will update accordingly. It also generates 6 output signals, 2 for the current cursor position (*Sel_X* and *Sel_Y*), 2 for a previously selected position *(Prev_X* and *Prev_Y*), 1 to express if a piece has been selected or not (*Piece_Selected*), and 1 to express the value of the current piece located at a particular X and Y position (*Cur_piece*). It sends all of these outputs to the **GUI BLOCK** as control signals, where they get used to generate the VGA RGB signals. It should also be noted the **GUI BLOCK** sends back 2 inquiry signals to the chess logic block (*Grid_X* and *Grid_Y*), which are used to drive *Cur_piece* as such:

```
Cur_piece <= board(Grid_Y,Grid_X);
```

Figure 5: Driving Current Piece

So, the GUI block can inquire at any moment in time how each position on the board is populated and use that information to generate the appropriate RGB signals.

Now, let's go to the **GUI BLOCK**. As mentioned, this block accepts several control signals, one of which is *Cur_piece*, which is a direct result of its own inquiry signals to check every position of the board. It also accepts *Hcount* and *Vcount*, which comes from the **VGA CONTROL** block. Using the current value of *Hcount* and *Vcount*, the **GUI BLOCK** will make the appropriate requests to the **CHESS LOGIC** block, in order to ensure the GUI's output RGB signals are correct on a per pixel basis.

There is a master ***if*** statement (priority mux) in the GUI top level model, which will print the blue boundaries of the game, the checkered chess grid, and the chess board's brown outlines regardless of any external signals. It is only once a piece is detected from the **CHESS LOGIC**

block that the **GUI BLOCK** samples *Cur_piece* and uses the received value to access a hand drawn 12x12 pixel map (shown below)

```
constant piece_icon:pix_map :=
        (    (x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"0000",x"0000",x"2945"),
             (x"2945",x"0000",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"7BEF",x"0000",x"2945"),
             (x"2945",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"0000",x"2945"),
             (x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945",x"2945"));
```

Figure 6: Black Knight Pixel Map (Highlighted for Visibility)

and proceeds to manipulate the incoming values of *Hcount* and *Vcount* to know exactly where to display said piece on the screen, using the RBG color codes in its 12x12 array. In the end, we end up with something like this (shown below)!

Figure 6: Finished Chess Board

The board itself was chosen to be a green and beige color, and the pieces are dictated by their pixel map, which I explained earlier. The outer border was chosen to be a blue color, but the

board's outlines are a light brown. The cursor was chosen to be a light blue color but leaves behind a burgundy/coffee/brown color on a square the was previously selected until a move has been made, or it has been deselected. For completeness, below you can see an image of a game in progress, where the H2 pawn has been selected, and the cursor is preparing to push it to H4!

Figure 7: Game in Progress!

And with that, I hope you enjoyed the dataflow explanation of my chess game!

Capabilities/Features:

In this section I aim to point out some capabilities and features of this chess game when compared to the real thing.

- ➔ Pieces are constrained to movements only allowed by their type. (I.e., Knights can only move like Knights, and Queens can only move like Queens.)
- ➔ Pawns that have not been previously moved can in fact move the initial two squares as usual, if there is nothing blocking their way. They cannot move two squares otherwise.
- ➔ Pieces of opposite color can be captured. Pieces of the same color cannot be captured.
- ➔ Pawns may not move diagonally unless in the act of capturing, as usual.

- ➔ Pawn promotion is functional; however, pawns auto promote to Queen.
- ➔ Diagonal movement selection allowed by the joystick.

Drawbacks/Shortcomings:

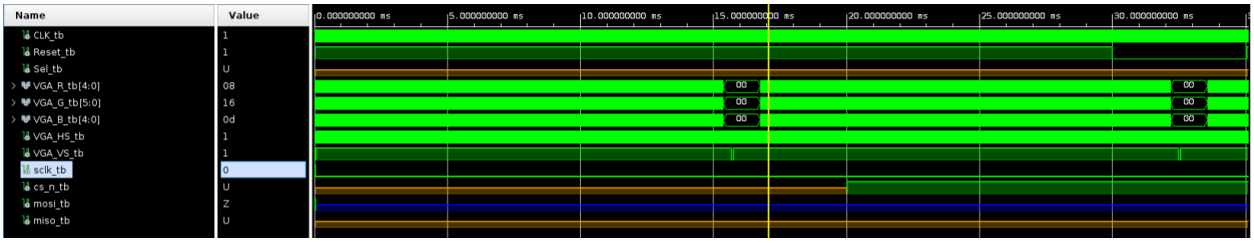
Now I will point out some drawbacks and shortcomings.

- ➔ No turn order, players can move pieces however they wish regardless of proper turn.
- ➔ No castling functionality.
- ➔ No En passant functionality.
- ➔ No check or checkmate functionality.
- ➔ No collision detection for the Queen, Bishop, or Rook pieces. (I.e., these pieces may behave like a knight and jump over other pieces in its path of legal moves, as shown below.)

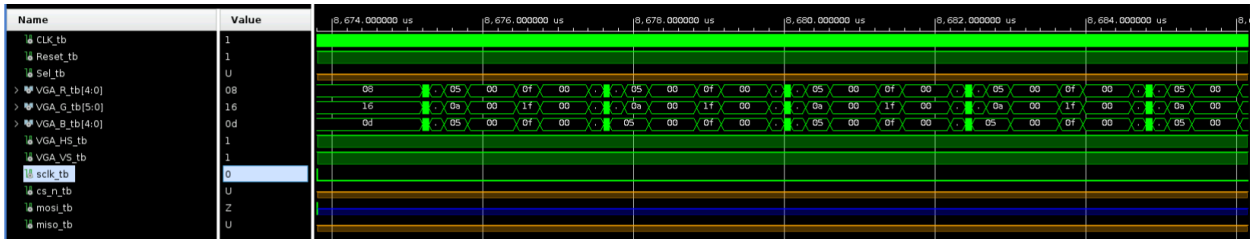
Figure 8: Illegal Bishop Jump Move!

Simulation Waveforms:

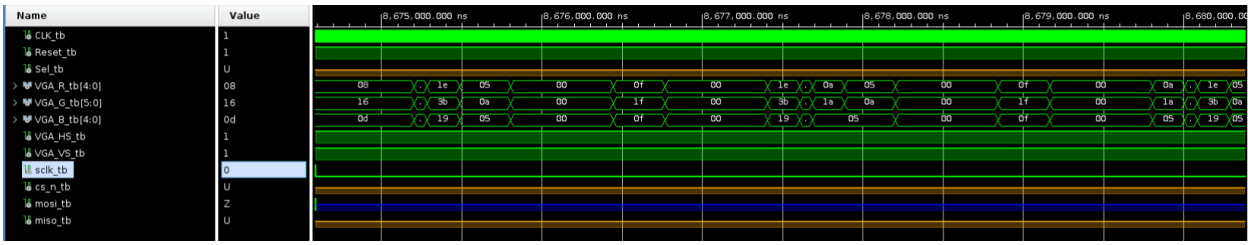
During my debugging stage I created only a single testbench for my top-level module, to ensure it was sending over RGB information, regardless of what it was. My reasoning for this is that it would be impossible for a human to decipher if the frames sent over by VGA Control and GUI were correct by simply looking at the RGB values on a simulation waveform over the course of what would have to be about nine-hundred and two milliseconds. Instead, I simulated for thirty-five milliseconds and just verified that the RGB signals were in fact being sent, then I proceeded to debug by generating bitstreams to view in real life as I went along. Below you can find my thirty-five milliseconds of simulation!



Zoomed in at about eight milliseconds to verify RGB:



Zoomed in even more at the same spot:



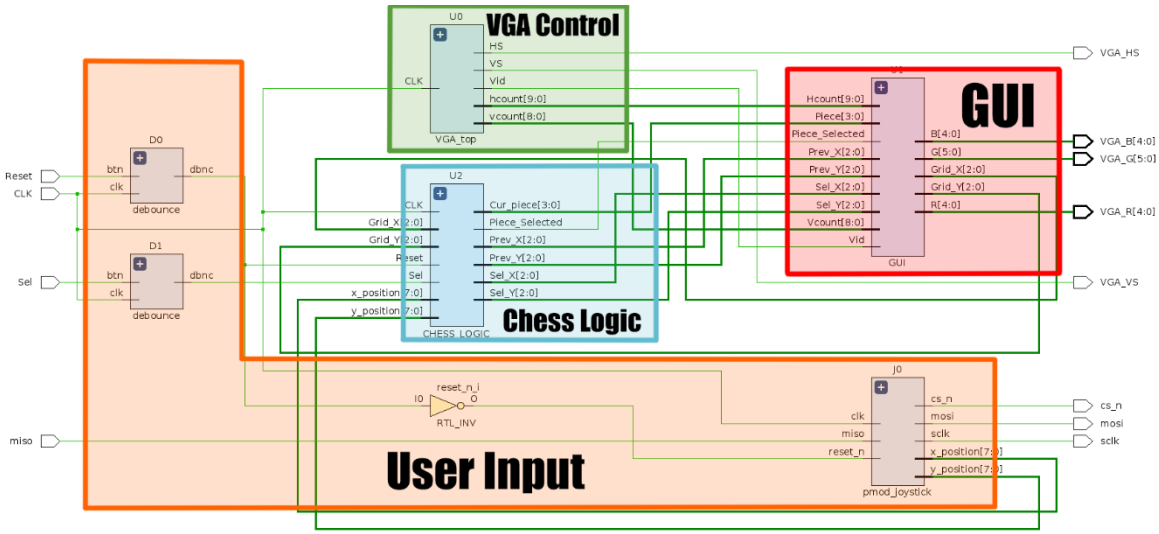
So, my VGA Control block and GUI block were at least sending over values that I was able to see when I generated my bitstream and began debugging!

Vivado Schematics:

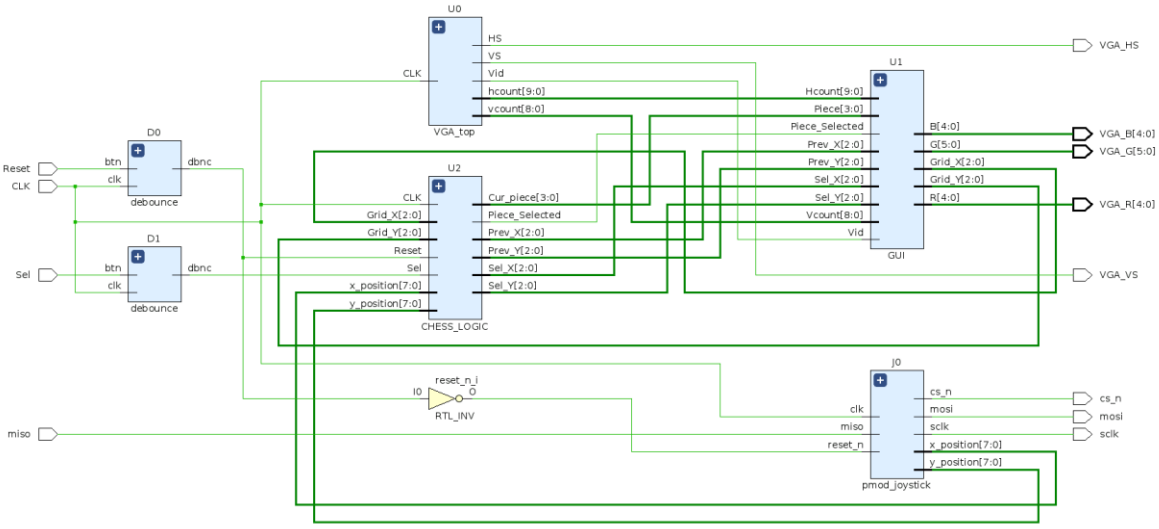
a) Vivado Elaboration Schematics

CHESS_TOP:

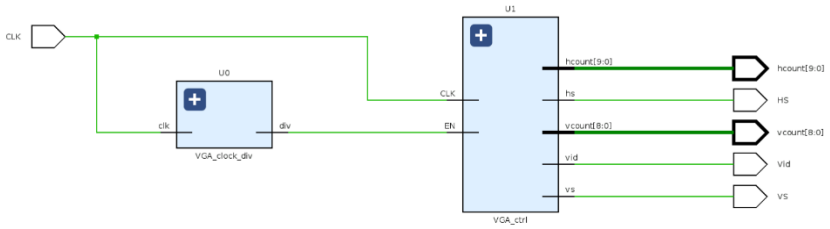
Block Diagram Style:



Regular Diagram:

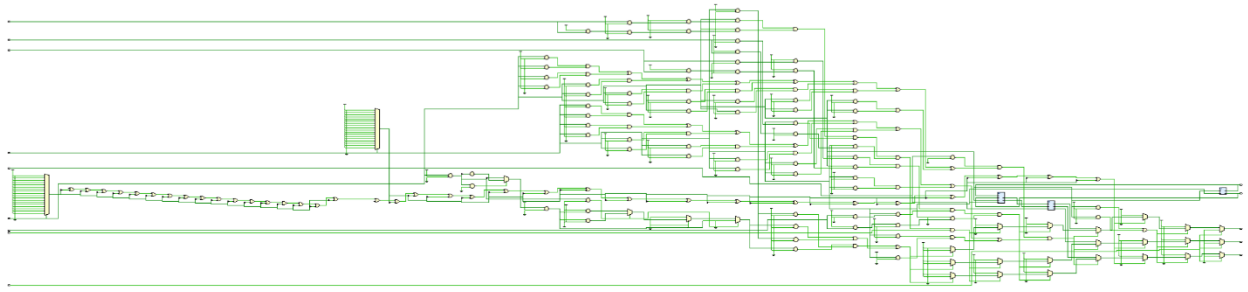


VGA_TOP:

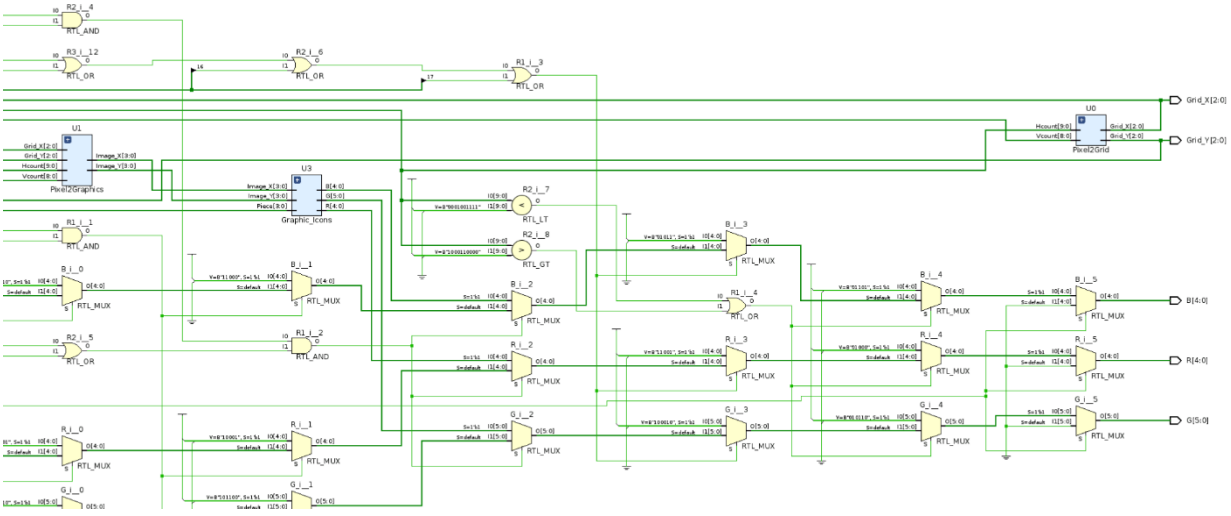


GUI:

Zoomed Out:



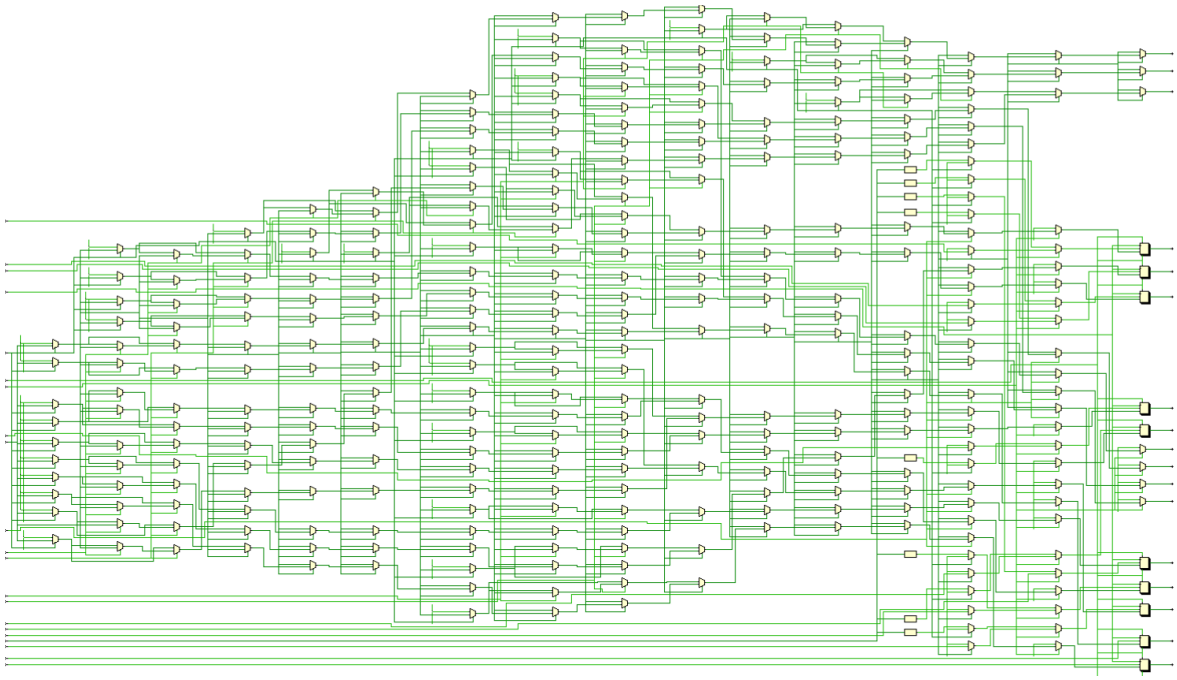
Zoomed In (to the part that matters):



CHESS_LOGIC:

There were ***sixteen*** pages of elaboration images generated by Vivado for this module (makes sense given how long it was)! Since it is not feasible for a human to comprehend all of them in a

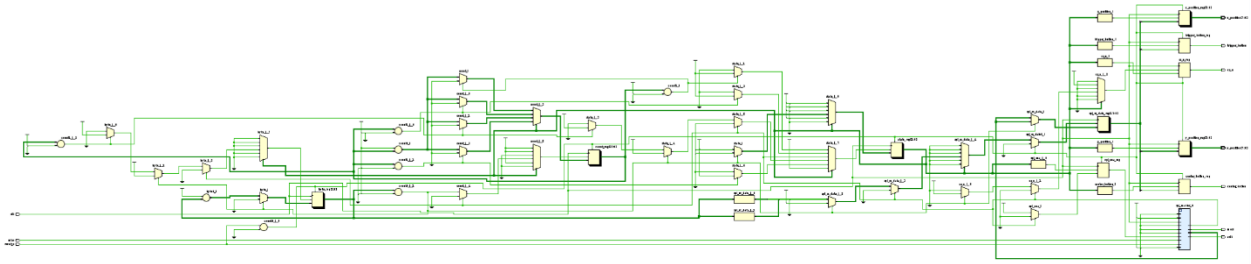
reasonable amount of time, I will provide only the final page for simplicity's sake. But rest assured, they exist!



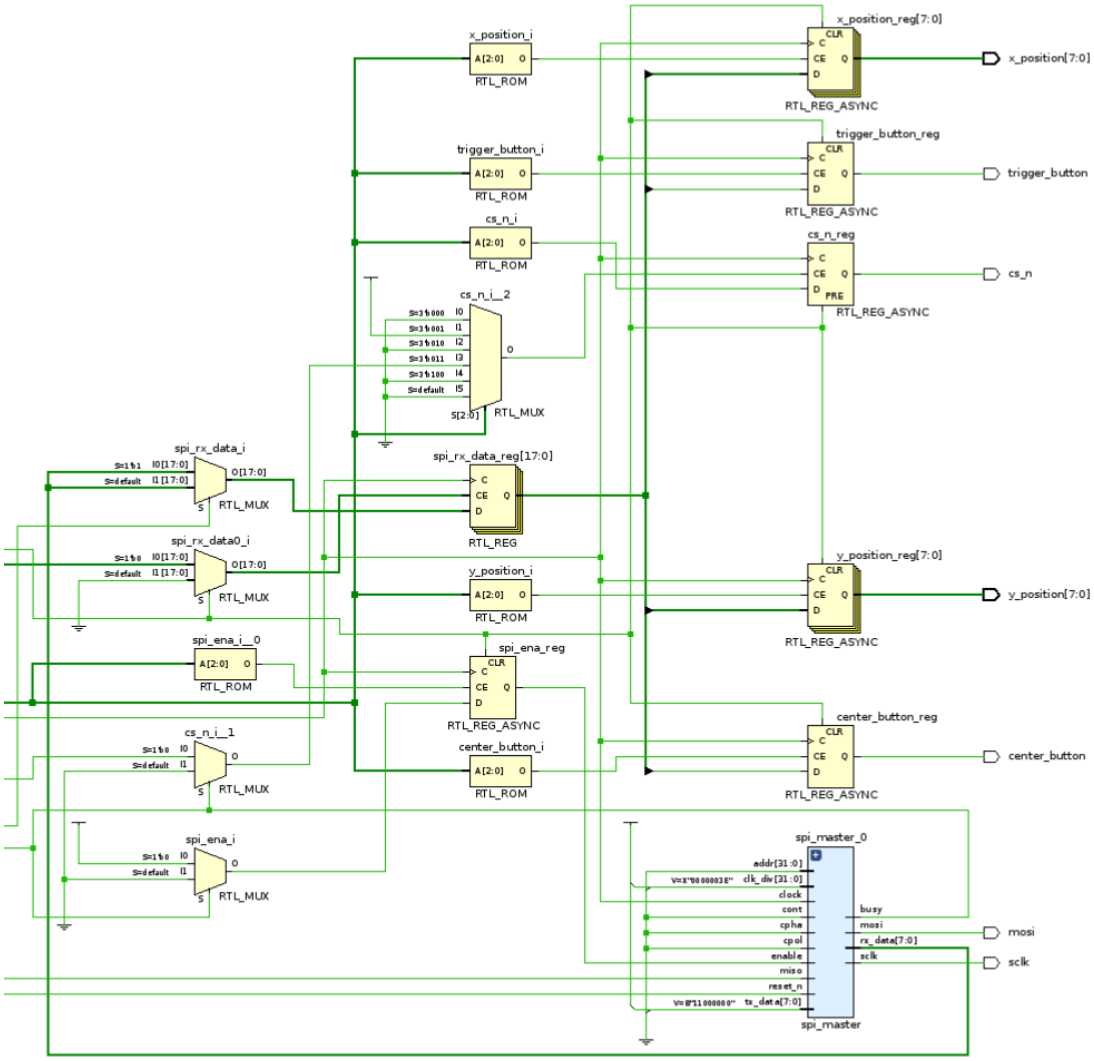
Yes, there are fifteen other pages just like this! The signals on the left are coming in from multiple different pages, and the signals on the right are either outputs of CHESS_LOGIC or signals being routed to other pages as well.

PMOD_JOYSTICK:

Zoomed Out:



Zoomed In (to the SPI Master instantiated module and Outputs):

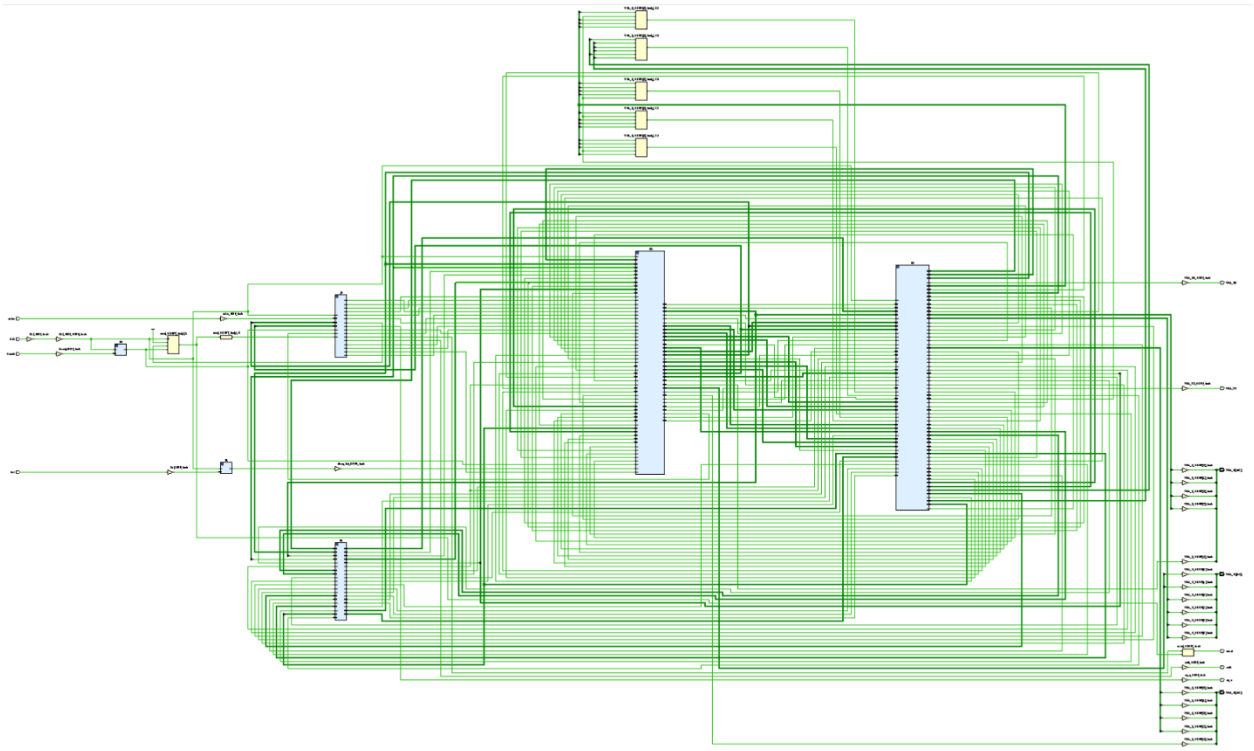


Notice how in our overall design we never used the *center_button* or *trigger_button* of the joystick. They are shown as outputs here, but they are left floating when elaborating the top-level module.

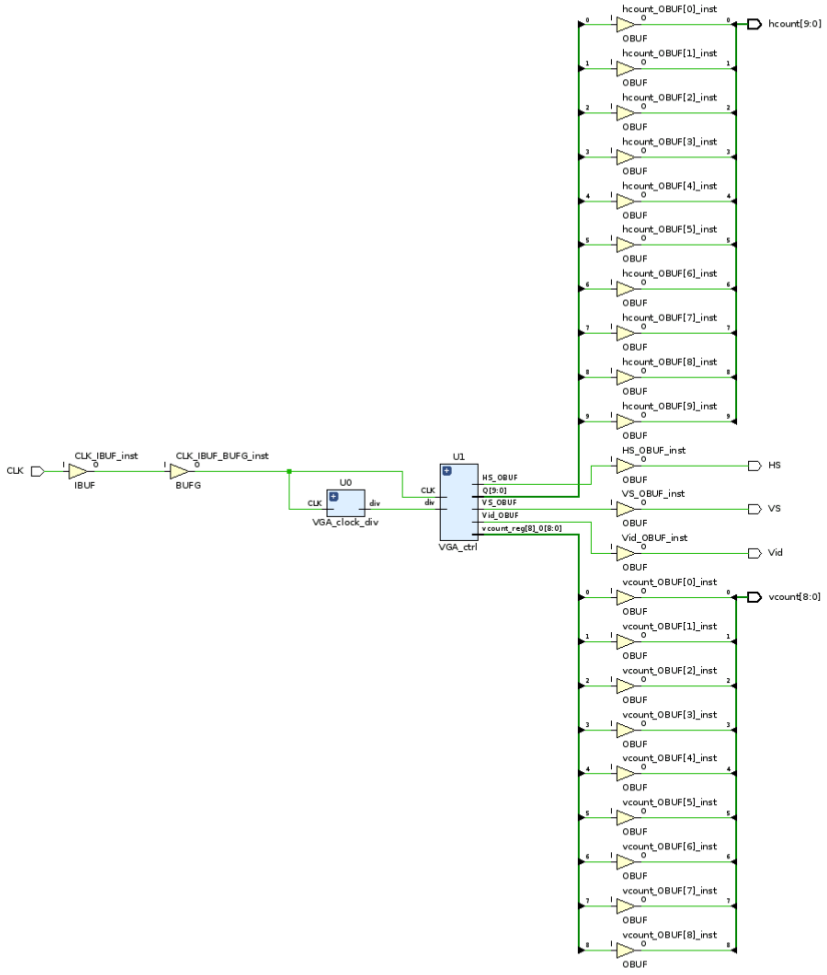
b) Vivado Synthesis Schematics

Some synthesis schematics for this section will be complex, so much so that it is pointless to zoom in on inputs/outputs/instantiations without losing critical information anyway. So please excuse any far zoom shots.

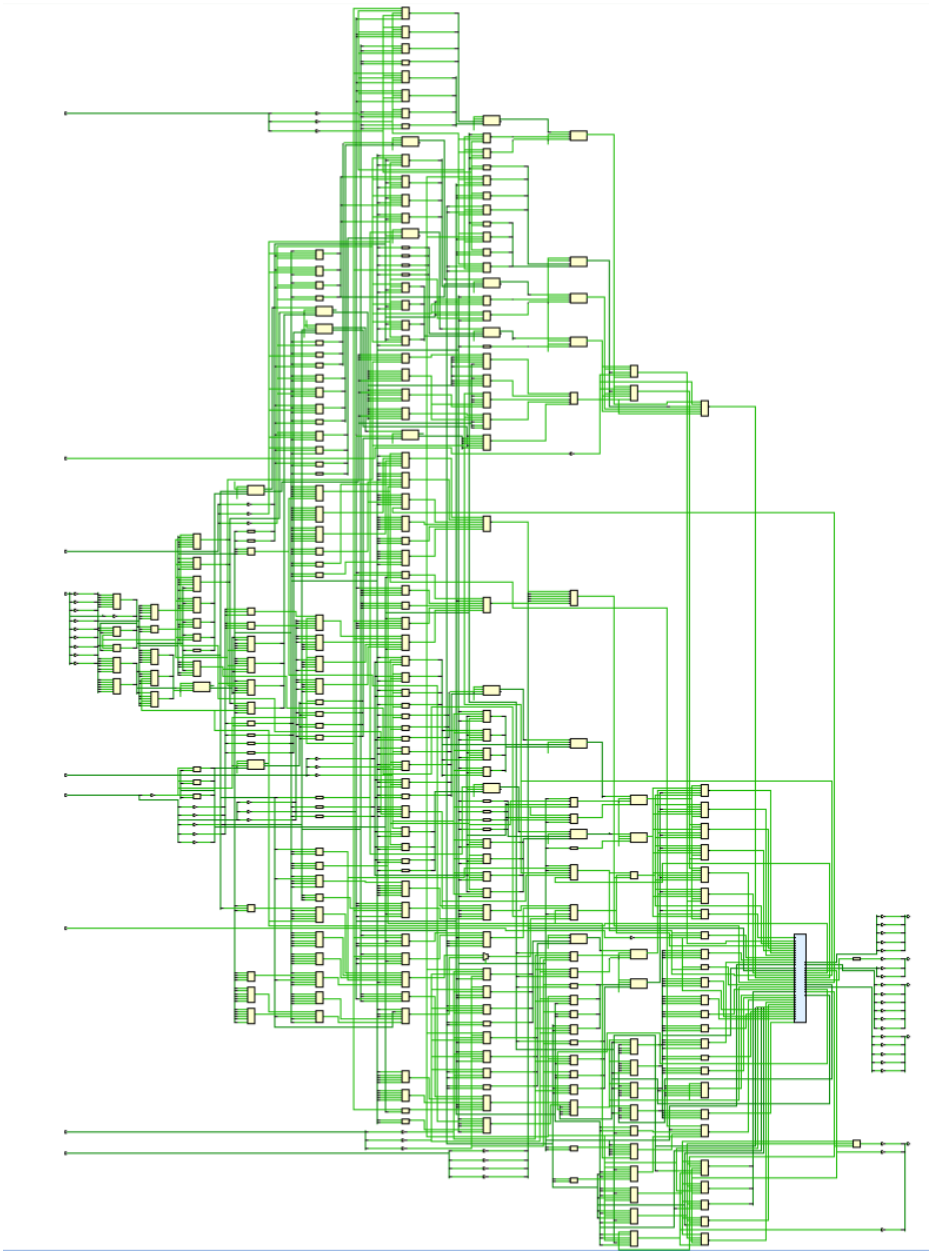
CHESS_TOP:



VGA_TOP:

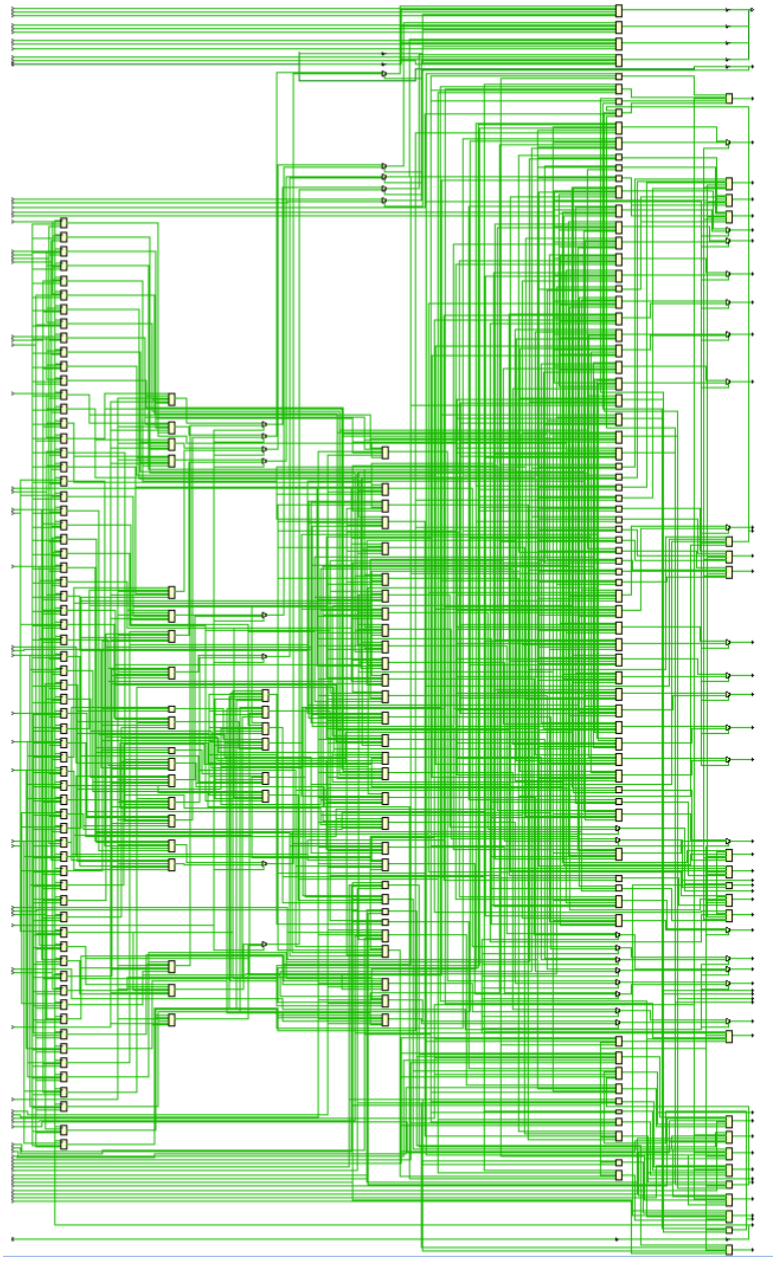

CLK
CLK_IBUF_inst
IBUF
CLK_IBUF_BUFG_inst
BUFG
U0
CLK
div
VGA_clock_div
U1
CLK
div
HS_OBUF
O[9:0]
VS_OBUF
Vid_OBUF
vcount_reg[8]_0[8:0]
VGA_ctrl
hcount_OBUF[0]_inst
hcount_OBUF[1]_inst
hcount_OBUF[2]_inst
hcount_OBUF[3]_inst
hcount_OBUF[4]_inst
hcount_OBUF[5]_inst
hcount_OBUF[6]_inst
hcount_OBUF[7]_inst
hcount_OBUF[8]_inst
hcount_OBUF[9]_inst
OBUF
hcount[9:0]
HS_OBUF_inst
HS
VS_OBUF_inst
VS
Vid_OBUF_inst
Vid
vcount_OBUF[0]_inst
vcount_OBUF[1]_inst
vcount_OBUF[2]_inst
vcount_OBUF[3]_inst
vcount_OBUF[4]_inst
vcount_OBUF[5]_inst
vcount_OBUF[6]_inst
vcount_OBUF[7]_inst
vcount_OBUF[8]_inst
vcount[8:0]

GUI:



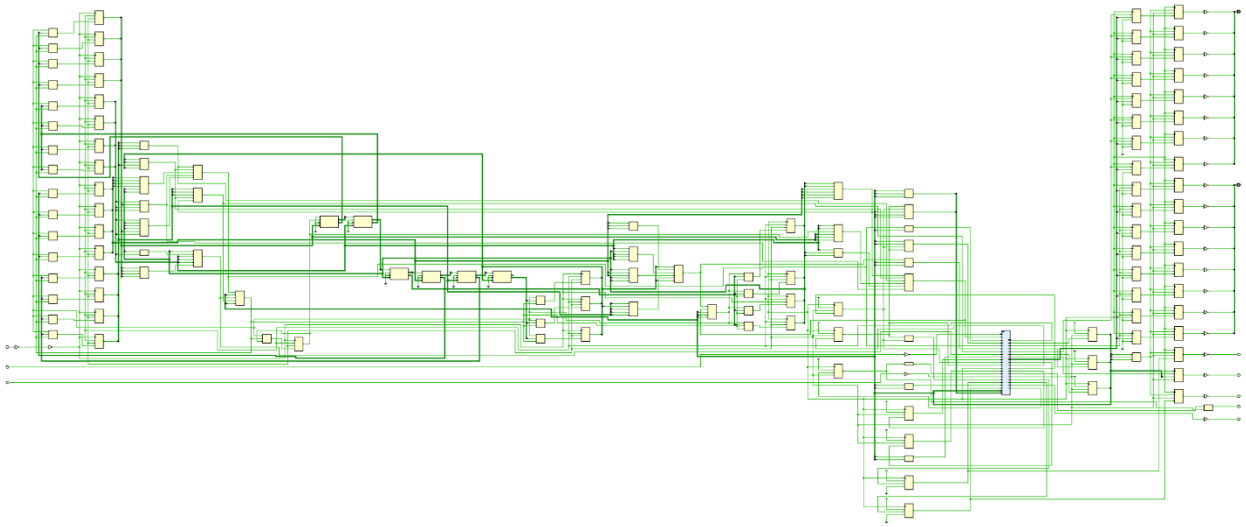
CHESS_LOGIC:

Once again, there were ***eleven*** pages of synthesis images generated by Vivado for this module (makes sense given how long it was)! Since it is not feasible for a human to comprehend all of them in a reasonable amount of time, I will provide only the final page for simplicity's sake. But rest assured, they exist!



Yes, there are ten other pages just like this! The signals on the left are coming in from multiple different pages, and the signals on the right are either outputs of CHESS_LOGIC or signals being routed to other pages as well.

PMOD_JOYSTICK:



c) Post- Synthesis Utilization Tables

CHESS_TOP:

Resource	Estimation	Available	Utilization %
LUT	2273	17600	12.91
FF	537	35200	1.53
IO	25	100	25.00
BUFG	2	32	6.25

VGA_TOP:

Resource	Estimation	Available	Utilization %
LUT	31	17600	0.18
FF	46	35200	0.13
IO	23	100	23.00
BUFG	1	32	3.13

GUI:

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	311	17600	1.77
IO	59	100	59.00

CHESS_LOGIC:

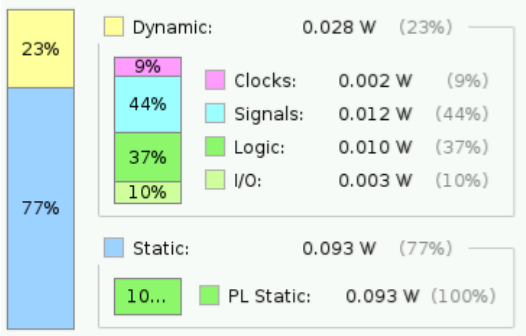
Resource	Estimation	Available	Utilization %
LUT	1943	17600	11.04
FF	313	35200	0.89
IO	42	100	42.00
BUFG	2	32	6.25

PMOD_JOYSTICK:

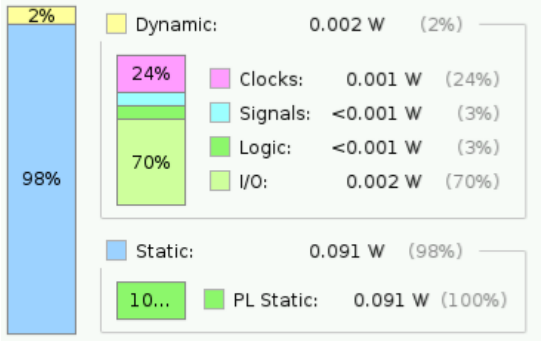
Resource	Estimation	Available	Utilization %
LUT	92	17600	0.52
FF	130	35200	0.37
IO	24	100	24.00
BUFG	1	32	3.13

d) On-Chip Power Graphs

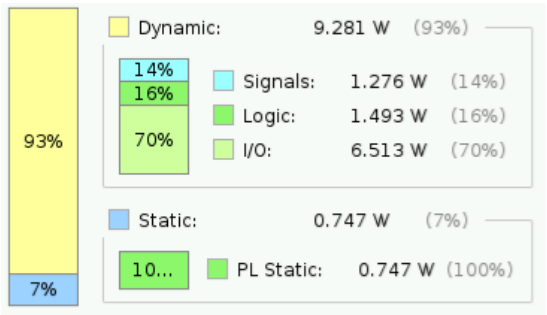
CHESS_TOP:



VGA_TOP:

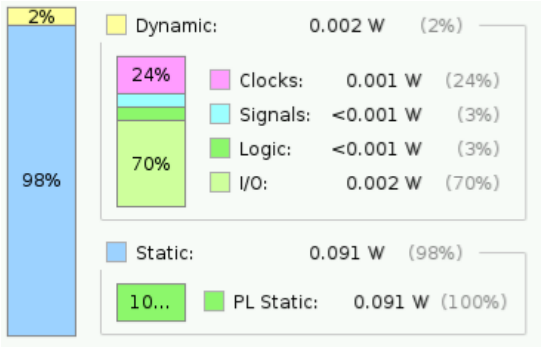


GUI:

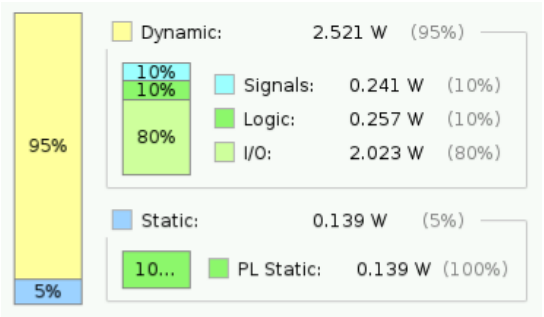


Please note, there were some critical warnings that resulted in low confidence power ratings for this device. (Which is why the GUI is drawing 9.3 watts, even though we know this to be false.) Namely, there are uncertainties in Vivado regarding the signals that would be driving with GUI's inputs. Since it is such a fast-operating module, it makes sense that the rate at which its inputs are changing would drastically affect power draw.

CHESS_LOGIC:



PMOD_JOYSTICK:



Looks like the high sampling frequency of the joystick caused the same uncertainty that we had with the GUI!

e) XDC File Modifications

```
##Clock signal
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { CLK }]; #IO_L11P_T1_SRCC_35 Sch=sysclk
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { CLK }];


##Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { Reset }]; #IO_L20N_T3_34 Sch=BTN0
#set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { Down_btn }]; #IO_L24N_T3_34 Sch=BTN1
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { Up_btn }]; #IO_L18P_T2_34 Sch=BTN2
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { Sel }]; #IO_L7P_T1_34 Sch=BTN3


##Pmod Header JA (XADC)
set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { cs_n }]; #IO_L21P_T3_DQS_AD14P_35 Sch=JA1_R_p
set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 } [get_ports { mosi }]; #IO_L22P_T3_AD7P_35 Sch=JA2_R_P
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { miso }]; #IO_L24P_T3_AD15P_35 Sch=JA3_R_P
set_property -dict { PACKAGE_PIN K14   IOSTANDARD LVCMOS33 } [get_ports { sclk }]; #IO_L20P_T3_AD6P_35 Sch=JA4_R_P


##VGA Connector
set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { VGA_R[0] }]; #IO_L7P_T1_AD2P_35 Sch=VGA_R1
set_property -dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { VGA_R[1] }]; #IO_L9N_T1_DQS_AD3N_35 Sch=VGA_R2
set_property -dict { PACKAGE_PIN J20   IOSTANDARD LVCMOS33 } [get_ports { VGA_R[2] }]; #IO_L17P_T2_AD5P_35 Sch=VGA_R3
set_property -dict { PACKAGE_PIN G20   IOSTANDARD LVCMOS33 } [get_ports { VGA_R[3] }]; #IO_L18N_T2_AD13N_35 Sch=VGA_R4
set_property -dict { PACKAGE_PIN F19   IOSTANDARD LVCMOS33 } [get_ports { VGA_R[4] }]; #IO_L15P_T2_DQS_AD12P_35 Sch=VGA_R5
set_property -dict { PACKAGE_PIN H18   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[0] }]; #IO_L14N_T2_AD4N_SRCC_35 Sch=VGA_G0
set_property -dict { PACKAGE_PIN N20   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[1] }]; #IO_L14P_T2_SRCC_34 Sch=VGA_G1
set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[2] }]; #IO_L9P_T1_DQS_AD3P_35 Sch=VGA_G2
set_property -dict { PACKAGE_PIN J19   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[3] }]; #IO_L10N_T1_AD11N_35 Sch=VGA_G3
set_property -dict { PACKAGE_PIN H20   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[4] }]; #IO_L17N_T2_AD5N_35 Sch=VGA_G4
set_property -dict { PACKAGE_PIN F20   IOSTANDARD LVCMOS33 } [get_ports { VGA_G[5] }]; #IO_L15N_T2_DQS_AD12N_35 Sch=VGA=G5
set_property -dict { PACKAGE_PIN P20   IOSTANDARD LVCMOS33 } [get_ports { VGA_B[0] }]; #IO_L14N_T2_SRCC_34 Sch=VGA_B1
set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports { VGA_B[1] }]; #IO_L7N_T1_AD2N_35 Sch=VGA_B2
set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports { VGA_B[2] }]; #IO_L10P_T1_AD11P_35 Sch=VGA_B3
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { VGA_B[3] }]; #IO_L14P_T2_AD4P_SRCC_35 Sch=VGA_B4
set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports { VGA_B[4] }]; #IO_L18P_T2_AD13P_35 Sch=VGA_B5
set_property -dict { PACKAGE_PIN P19   IOSTANDARD LVCMOS33 } [get_ports VGA_HS]; #IO_L13N_T2_MRCC_34 Sch=VGA_HS
set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports VGA_VS]; #IO_0_34 Sch=VGA_VS
```

I modified these aspects of the XDC file to account for the board's input clock signal, my select and reset buttons, the PMOD header for the joystick, as well as all the appropriate VGA signals.

Answers to Additional Questions and Extra Credit:

Not Applicable.

Non-Functional Requirements:

- ➔ The performance of this game is fast when compared to software counterparts, response is near instant. Since it is meant to interface with humans, it more than meets its performance requirements.
- ➔ Size is also exceptionally small given how complex the game logic was. In fact, we only used 13% of the LUTs (~2200 used) on the Zybo board, less than 3% of the flops (~540 used), and less than 7% of the buffered global clock signals used for clock distribution (2 used). All of which were less than most labs for this class! I anticipate this would translate exceptionally well to sizing on any dedicated chip.

- ➔ The on-chip power-draw is also small and uses only about 121 milliwatts (0.121 watts) during operation. The peak operation temperature was also around room temperature (around 40 C), meaning that it can be kept cool without active cooling.
- ➔ Overall, an adequate power source and cool room would still need to be provided in order for the chip to operate properly.

Follow Up:

As a follow up to this project, I would for sure try to address some of the drawbacks/shortcomings of my implementation! The first and most important thing that would need to be fixed is collision. Then I would tackle castling, En passant, check/checkmate and turn order in that order. If all of those were to be fixed, we would have a great chess game on our hands!

Conclusion:

In conclusion, it was interesting to complete this project. Though it took a lot of time, I learned a lot when putting it together. The performance and operation turned out to be about what was expected, and the game is completely playable. Though it is difficult for a human to understand the simulation waveforms (since we are generating video frames) they seem to behave as intended. It was interesting to implement the pixel maps and translate that to a video RGB signal, and we can in fact see this pixel map activation in the simulations. Overall, let's play some chess!