ECE385

Fall 2022

Final Project

2-Player Chess (Mouse Controlled, VGA Monitor Display, Status Bar, Evaluation WIP)

Shu Xian, Lai & Cindy, Yu (sxlai2, dexinyu2)
Section NL
TA: Nick Lu

Introduction

In our final project, I planned to write a chess game to be played by two human players by using a mouse. Our chess will be displayed on a VGA monitor and the mouse connected to the FPGA board. The rules for our chess game will ***follow any other classic 8x8 chess game**, **except for the fact that we did not include***:

1. pawn-promotion scenario
2. 50-move rule
3. Threefold-repeat move detection

To help determine a win-lose situation, we also implemented a simple scoring system where each piece captured by either player is worth a certain amount of points:

Pawn	10
Knight	20
Bishop	20
Rook	30
Queen	90
King	500

The scoring rubric above is based on the most common convention for relative chess pieces values found online from Wikipedia.com and Chess.com (both use the same metrics).

The score for king is almost irrelevant, because any capture of the king indicates a win.

The game also incorporates a simple status bar on the side.

The status bar's role is to display the captured pieces by both sides, the players' scores, some general options and a region that displays certain messages depending on game state. Aside from the basic chess board, we also include in the status bar 3 general options common in a chess game – forfeit, draw and restart. Players will be able to confirm their choices when they choose such settings.

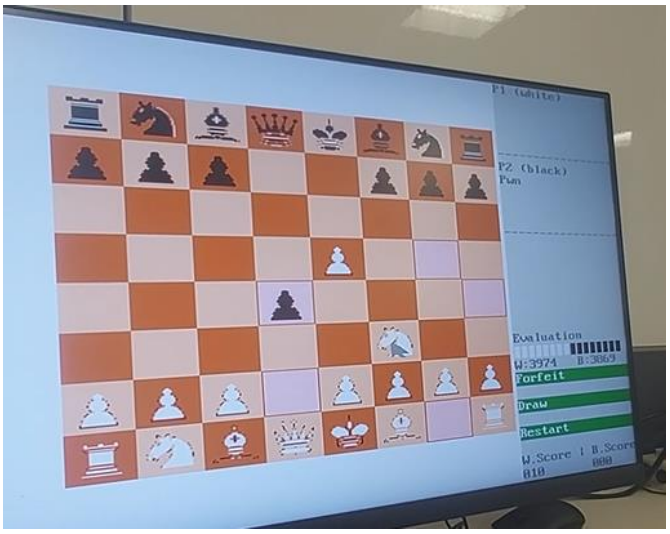
According to our initial final project proposal, our project successfully implemented all features that we promised, which are:

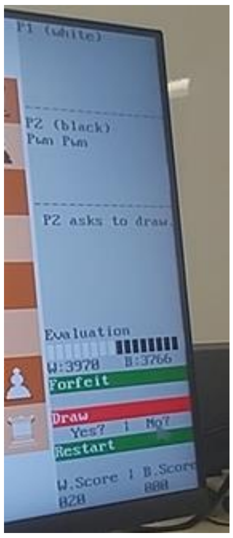
- The displaying of the chess board (and its pieces).
- A finite state machine that determines which player's turn to move, the legality of the moves (including some special moves like *en passant, castling*), the win/lose/draw conditions and the capturing of the pieces.
- The display of the pieces that have been captured on the side of the board.

Table of Features

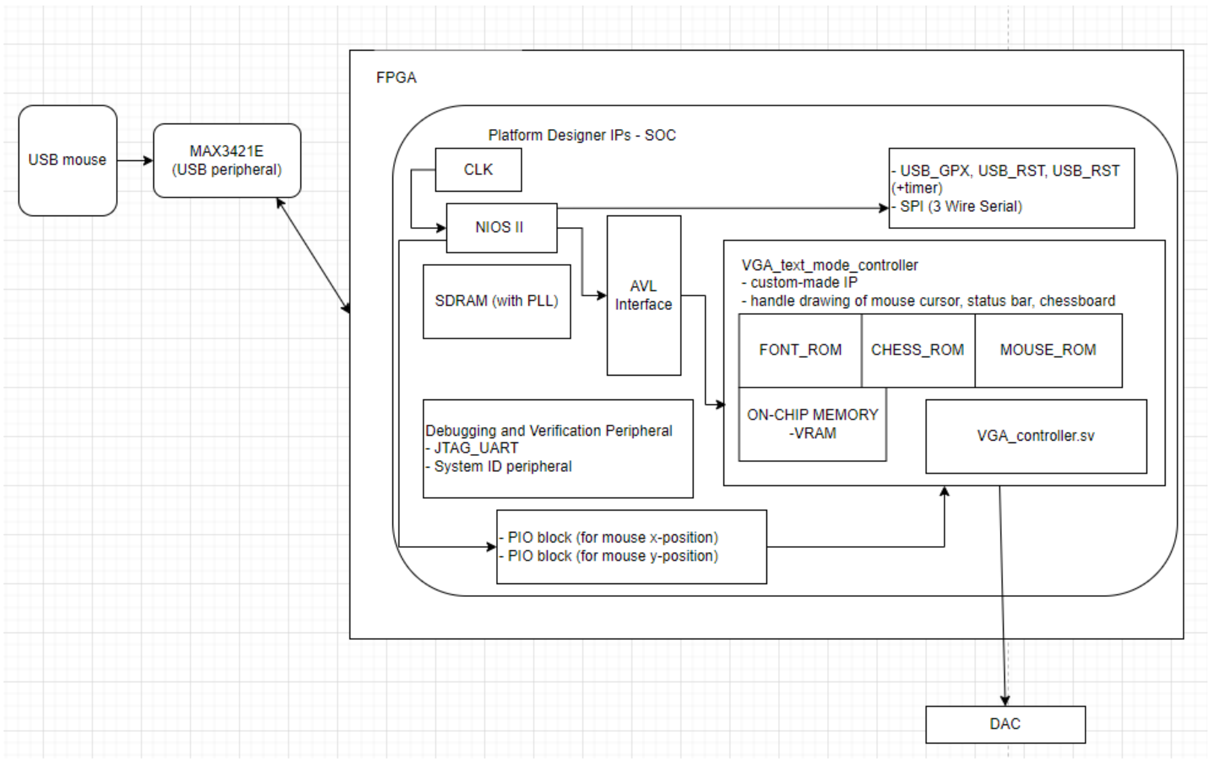
Features	Explanation
Visual cue for possible moves of a piece	When a player clicks on a chess tile that contains a piece, the tiles that correspond to legal moves should have some sort of a visual cue (lighting up, having a different color etc.).
Displaying the captured pieces of both players	When a piece has been captured by either player, the piece will be displayed on the side of the board. This is just to make it easier for players to keep track of the pieces they lost (and what pieces they can trade when pawns reach the other end of the board).
Additional basic features of a chess game (reset, forfeit, win-lose-draw, 50-move checking)	These will be additional but basic settings to keep track of our chess game. The settings should be self-explanatory to players familiar with the game.

Snapshot of what our game looks like





Design Block Diagram


FPGA
USB mouse
MAX3421E (USB peripheral)
Platform Designer IPs - SOC
CLK
NIOS II
- USB_GPX, USB_RST, USB_RST (+timer)
- SPI (3 Wire Serial)
SDRAM (with PLL)
AVL Interface
VGA_text_mode_controller
- custom-made IP
- handle drawing of mouse cursor, status bar, chessboard
FONT_ROM
CHESS_ROM
MOUSE_ROM
ON-CHIP MEMORY -VRAM
VGA_controller.sv
Debugging and Verification Peripheral
- JTAG_UART
- System ID peripheral
- PIO block (for mouse x-position)
- PIO block (for mouse y-position)
DAC

Written Description (Game play + Evaluation Feature in progress)

<u>Game play</u>

Our 8x8 chess game is played with the control of a mouse, where players take turns to click on the piece they want to move. We enabled the featured where all possible moves for a clicked chess piece will light up, indicating that that particular piece can move to those 'lighted up' tiles.

The chess game starts automatically with White's turn to play first. Player can also, for sanity wise, just restart the game board anytime they wish, meaning to start a new game. Any pieces that are captured by both sides will be displayed on the status bar.

Whenever a King of either side is captured, the status bar will display a message indicating a certain player won. After that, players will need to choose the restart option to be able to play a new game.

<u>Evaluation Bar</u>

As a feature in progress to experiment on (not included in project proposal, and still work in progress), we implemented an evaluation bar that displays how well both Black and White development their gameboards, depending on the phases of the game. Most commonly, chess games are split into two phases, the early-middle game and the endgame.

This feature (evaluation itself is completed) however, does not incorporate a search function, rendering the evaluation itself somehow meaningless. This is because the evaluation is static (uses the piece table evaluation) and only considers the position of a certain piece and assign the piece a certain score (for example, a knight is strongest when placed in a middle position compared to the edge of the board, hence a knight in the middle position will be assigned a greater score).

Therefore, the evaluation does not consider potential moves of the opponent when assigning the scores. More details of how piece tables work can be looked up online and details on the specific evaluation we used is explained below.

PeSTO's Evaluation Function

PeSTO's evaluation function is an evaluation function by Ronald Friederich and used in several chess engines including RofChade and PeSTO. The evaluation function utilizes piece-square tables to calculate a heuristic score (call this S) based on the current configuration of the chess board. For every configuration of a chess board, there is an evaluation score based on the position of White pieces (call this W), as well as an evaluation score based on the position of the Black pieces (call this B).

The computation of the evaluation score also takes into account the phase of the game (early-middle game vs end game for chess) by observing the chess board. Such consideration is commonly known as "tapered evaluation" in the chess programming community.

Depending on which player's turn it is to move (and incorporating tapered evaluation),

e.g. if it is White's turn to move:

$$S = W - B$$

If it is Black's turn to move:

$$S = B - W$$

S is then finetuned by the tapered evaluation calculation which is not included here, interested readers can easily look up the formulas behind to understand the details.

Again, the end-goal of the evaluation function is to return the heuristic score, S. This score is integral when incorporating searching algorithms such as minimax, alpha-beta pruning to fully program a chess engine.

As of our current submitted project, we did not manage to utilize S, but we only display, W and B with a bar.

Written Description of .sv Modules

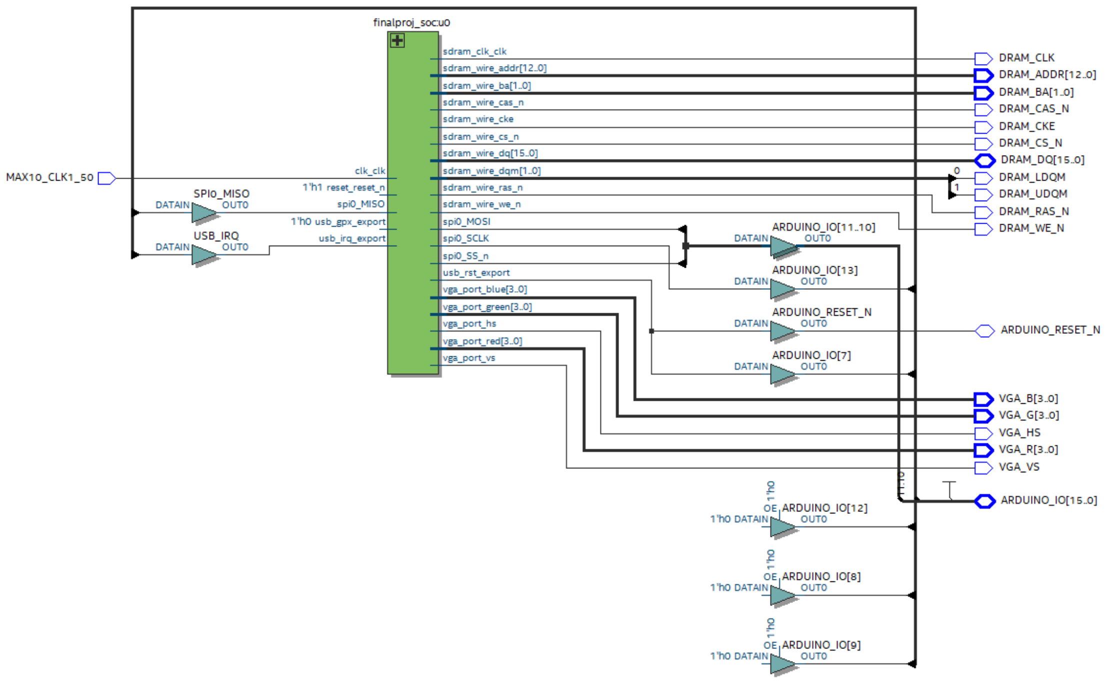
Module: vga_text_avl_interface.sv Inputs: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, AVL_BYTE_EN, AVL_ADDR, AVL_WRITEDATA Outputs: AVL_READDATA	Description: Custom-made IP to be loaded onto SOC. Enwraps all the on chip memory and ROM and helper functions. Purpose: Handles the drawing of the chess sprites, mouse sprites, and status bar.
Module: finalproj_soc Inputs: input wire clk_clk, input wire [15:0] mouse_port_button_status, input wire [15:0] mouse_port_x_displacement, input wire [15:0] mouse_port_y_displacement, input wire reset_reset_n, Outputs: output wire sdram_clk_clk, output wire [12:0] sdram_wire_addr, output wire [1:0] sdram_wire_ba, output wire sdram_wire_cas_n, output wire sdram_wire_cke, output wire sdram_wire_cs_n, output wire [1:0] sdram_wire_dqm, output wire sdram_wire_ras_n, output wire sdram_wire_we_n, output wire spi0_MOSI, output wire spi0_SCLK, output wire spi0_SS_n, output wire usb_rst_export, output wire [3:0] vga_port_red, output wire [3:0] vga_port_green, output wire vga_port_hs, output wire [3:0] vga_port_blue, output wire vga_port_vs, output wire [15:0] x_displacement_export, output wire [15:0] y_displacement_export output wire [15:0] button_status_export,	Description: System generated VHDL file based on Platform Designer components. Purpose: The main purpose of this module is to enwrap some readily available IPs that we included in Platform Designer (e.g. NIOS II processor, SDRAM) as well as the custom-made IPs, specifically our VGA_TEXT_AVL_Interface.sv
Module: font_rom.sv Inputs: addr Outputs: data	Description: Read-only memory file that contain sprite bytes for ASCII characters from code 0 to code 127

	Purpose: Allow hardware to perform sprite base drawing based on ASCII character codes in VRAM.
Module: ocm.sv (stands for on-chip memory) Inputs: [7:0] avl_writedata, vga_writedata, Input[9:0] avl_addr, vga_addr Input avl_we, vga_we, clk, Outputs: Output [7:0] avl_readout, vga_readout	Description: Use infer logic to create a synchronous on-chip memory module that has a data width of 8, and an address width of 10 bits. Calculation is based on the needed VRAM contents to represent the chess game. Purpose: Utilizes the M9K blocks on DE-10 FPGA as our on chip memory to store VRAM contents while providing fast access without wasting resources, compared to using registers.
Module: mouse_cursor.sv Inputs: input logic [9:0] mouse_x, mouse_y, draw_x, draw_y, Outputs: Pixel_type, inside_chess	Description: Based on the dx, dy position of the VGA gun, and the mouse_cursor x, y position sent from by our USB driver, determines whether we currently need to draw a mouse cursor. Purpose: Performs an accurate mouse cursor drawing on the VGA screen.
Module: mouse_rom.sv Inputs: [3:0]addr Outputs: [15:0]data	Description: Read-only memory that stores the sprite byte of a 16x16 mouse cursor sprite. Purpose: To be accessed based on position of cursor and position of electron gun on vga screen.
Module:mouse.sv Inputs:Reset, frame_clk, x_displacement, y_displacement Outputs: mouse_x, mouse_y	Description: Does not achieve anything, a garbage module that I initially thought I need, but disregarded later. Purpose: For future users to clean up.
Module: chess_column_lookup.sv Inputs:	Description: Based on the dx, dy position of the electron gun, determine which row and column of the chessboard we land on.

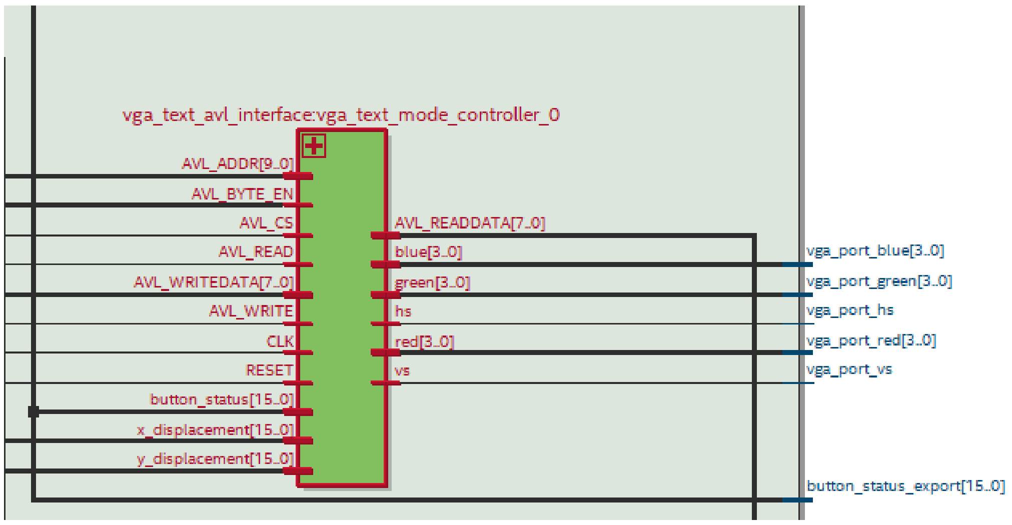
input logic [9:0] x_coordinate, y_coordinate, Outputs: output int unsigned tile_row, tile_col	Purpose: As stated.
Module: mapper_45_64 Inputs: dx, dy Outputs: dxout, dyout	Description: Perform simple arithmetic on inputs and return the output. Purpose: Exists as helper function.
Module: statusbar_rowcol Inputs: dx, dy Outputs: row, col	Description: Performs simple arithmetic on the dx, dy coordinates and return which row and column of the status bar we are on. Purpose: Exists as helper function.
Module: frameRAM.sv (adapted from RISHIS tool) Inputs: [14:0] read_address, CLK Outputs: [11:0]data_Out	Description: Store the 13 unique chess tiles (6 black, 6 white and 1 empty tile) to represent a chess board. Each chess tile is a 32x32 sprite. Purpose: Using infer logic readily from Quartus to infer an on-chip memory block. The sprite bytes are read from a series of .txt files converted from PNG downloaded online. The conversion code from PNG to TXT is course resources

System Level Block Diagram

Netlist Viewer of top-level design entity:

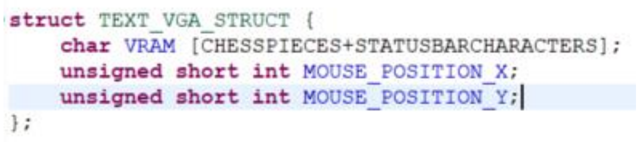


Netlist viewer of the integral components vga_avl_text_interface.sv that wraps up the rest of the other helper modules:



Note: The breakdown of the Netlist Viewer of the VGA_TextAVL_Interface module is attached in another file for better viewing purposes. See submission on Canvas.

VRAM structure (related to how we store the byte formats to represent chess pieces and game state, piece states)



```
struct TEXT_VGA_STRUCT {
    char VRAM [CHESSPIECES+STATUSBARCHARACTERS];
    unsigned short int MOUSE_POSITION_X;
    unsigned short int MOUSE_POSITION_Y;
};
```

Figure above shows the structure we allocate in C code that represents our VRAM. We can exploit this structure because the memory of a struct in C is allocated contiguously.

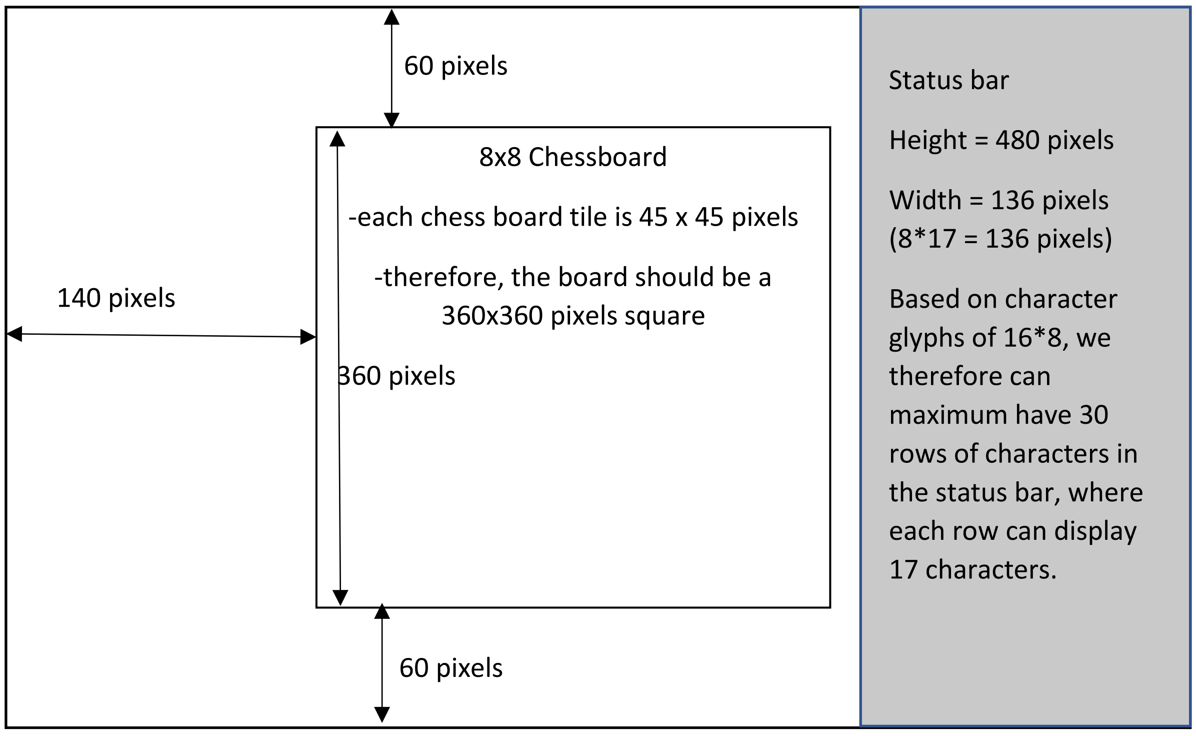
The relevant constants:

- CHESSPIECES = 64
- STATUSBARCHARACTERS = 510 (the total number of characters we can represent in our status bar, not all are utilized)

Note that the number of status bar characters was completely custom and manually calculated to fully utilize the blank space to the right of the chess board on screen. There shouldn't be any space left for the current position of the chess board, but one can definitely customize the status bar to have more/less characters.

Visualization of the board and status bar on a VGA screen

To visualize, below is the orientation of the elements on a 640X480 VGA monitor, we intentionally left out a very small stripe between the status bar and the chessboard for better display purposes.


60 pixels
8x8 Chessboard
-each chess board tile is 45 x 45 pixels
-therefore, the board should be a 360x360 pixels square
140 pixels
360 pixels
60 pixels
Status bar
Height = 480 pixels
Width = 136 pixels (8*17 = 136 pixels)
Based on character glyphs of 16*8, we therefore can maximum have 30 rows of characters in the status bar, where each row can display 17 characters.

VRAM Register Breakdown (continued from VRAM structure)

For the first 64 entries in VRAM, each byte represent a chess tile on the chessboard, the byte information is structured as below:

Bit[7] Special code	Bit [6] Special code	Bit [5] glow code	Bit[4] parity bit	Bit[3:0] Piece type - need 4 bits to represent 13 values. (6 black unique chess piece, 6 white unique chess piece and an empty tile)

- Two special codes are used to help distinguish certain piece states such that the chess rules are applied in the game. Special piece states include the state for castling, whether a pawn is being moved for the first time etc.
- Glow code is used to distinguish whether a chess tile is being lightened up as visual cue for potential moves.
- Parity bit is used to distinguish whether the chess tile background is dark brown/light brown.

For the remaining 510 entries in VRAM, the most significant bit is used as an invisible bit – if invisible bit is high, the character is not drawn and only background of the status bar is shown.

The remaining bits (bits 7 to 0) are ASCII codes that represent regular characters in the FONT ROM.

Special note on the two *unsigned short int* – they are specifically created such that the processor can transfer the mouse position data back to our hardware. The hardware system Verilog code uses this information to determine drawing of the cursor.

Software Component of the Lab

We created from scratch two major components of the code suite, there is a suite that handles the playing of the chessboard, and there is another suite that handles the display of the status bar.

We reused some of the readily provided/written code from previous labs – USB drivers, the Avalon interface peripherals.

We also incorporated and slightly modified a PeSTO evaluation code that we retrieved online for the evaluation function.

Note that some symmetrical functions are included in the same entry – symmetrical meaning they functionally the same thing but slightly differ in certain constant values because handling the black and white sides of the chess. Details of the code implementation is left for the readers to look at the submitted code.

For the chessboard:

void Clear_whitepawn_sp() void Clear_blackpawn_sp()	clears the special bit for pawns – used for enpassant
void populate_Chessboard()	populate the VRAM structure first 64 entries with all special codes set to the proper value (a new game)
alt_u8 tile_clicked(mx, my)	given the x, y position of a mouse and if left clicked, whether the click lands on a chess tile on the screen
int within_board(tile_index) int empty_tile(tile_index) int same_color(tile_index)	Simple Boolean functions, names are self explanatory.
void turn_off_glow()	setting the glow bit of all chess piece entries on the board to low
void glow(int tile_index, int source)	glowing a certain tile
int valid_turn(chesscode, whiteturn, blackturn)	checks whether it is the user turn to move when a mouse is clicked on the tile

void*
 	
generate_black_pawn_move()
generate_white_pawn_move()
generate_queen_move()
generate_king_move()
generate_bishop_move()
generate_knight_move()
generate_pawn_move()
	glow up tiles depending on which chess piece is clicked on, mainly change VRAM content glow bit accordingly

int black_check_attacked(tile_index) int white_check_attacked(tile_index) void white_castling() void black_castling()	Helper functions to reinforce chess special rules, mainly to help determine special conditions whether certain tiles need to be light up.

For the status bar:

populateFRAM(string, start)	write a certain string into the VRAM contents that belongs to the status bar
populate_statusbar()	initialize the original settings of the status bar

update_evalbar(int difference)
update_evalscore(whitescore, blackscore)
balance_evalbar()
	Function series that perform different functionalities related to the evaluation bar.

populatemessage(player, clear, forfeit, draw)	Populate a certain message on the status bar region depending on settings chosen
settings_clicked(mx, my)	determine whether user clicked on a settings option
yesno_clicked(mx, my)	determine whether user clicked on a yes/no response

invi_row(row)	hide a row when needed
reveal_deadpiece(index, count, player)	reveal a piece that has been attacked by a player
captured_piece_index(piece)	helper function
glow_row(row, glow_bool)	change the color of a settings bar when clicked

Design Resource and Statistics

LUT	6179
DSP	0
Memory (BRAM)	55296
Flip-flop	2580
Frequency	61.36MHz
Static Power	96.73mW
Dynamic Power	108.20mW
Total Power	223.68mW

Bug log

Bug Description	Resolve
When implementing the movement of the mouse cursor, the cursor's movement behaviour was sometimes indeterministic. This was due to bad design choice of having the software sends only the mouse (x_displacement, y_displacement) to the hardware.	I added two additional information in VRAM that represents the x_position and y_position of the mouse, and send that information from the software end to hardware. This makes the mouse movement seamless and behave as expected.
When adding the status bar display to the side of the chess board, the display of the status bar was some garbage ASCII, and the chessboard display was also affected. This was because I only extended the VRAM capacity on the software end to include the memory for the status bar, but did not modify the hardware address width to match that.	Extend the hardware addressability to match the VRAM space.
During some heavy modification of the VRAM (such as completely restarting the board), the code that was supposed to execute the modification seems to not being run. Based on my observations, this seems to be due to the fact that it takes some time between the software and hardware to communicate the exchange.	Added some empty functions that only runs a for-loop that does nothing (for a reasonable number of iterations, say 100), in between the chunks of code that modify a huge chunk of the memory.

There were some minor bugs in implementing the toggling of the status bar “settings” option, as well as some game play implementation bugs of the chess. These bugs were mainly logic mistakes and missing some edge cases.	Go through the game flow logic and test it out on the screen, then fix the logic bugs

Conclusion

Thank you.

References

PeSTO Evaluation Function (very slight modification to meet project needs)

https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function