ECE 385

Spring 2024

Final Project

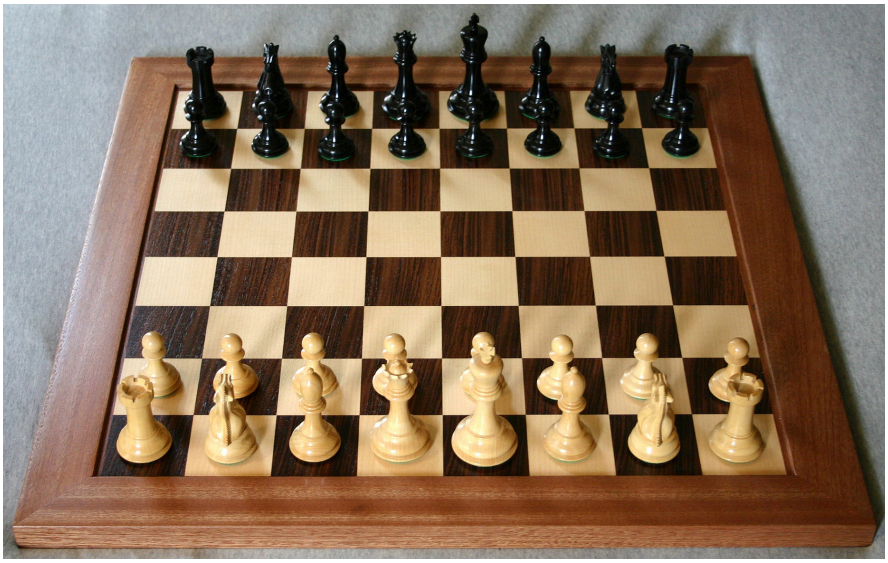
Chess

Ahmed Shafiuddin, Eddie Brenmark
XC / May 8 at 11:59pm
Shixin Chen

Introduction:

In our final project, our main goal was to design and implement Chess onto the FPGA using an SoC setup and hardware-software co-design principles by creating the graphics in hardware via SystemVerilog and dealing with the game logic on software (written in C). To interface with the chess game, we integrated a mouse into the design where a player can click to select a piece and click on its target square to move it. As an additional feature, we have started implementing an artificial opponent to play against instead of being limited to only human players. We also added an undo feature to allow for take backs and game status messages to indicate the current player's turn, if the position is in checkmate or a draw.

Chess is a commonly played board game on an 8 by 8 checkerboard where two opponents face off against each other with different colored pieces to denote a side (i.e. white and black). The goal of the game is to "checkmate" your opponent's king which means to attack your opponent's king in a way where it can't escape (or not be in check) on the next turn. A check is simply defined as a given side's king being under attack (prone to being captured on the next turn).



Initial setup of Chess

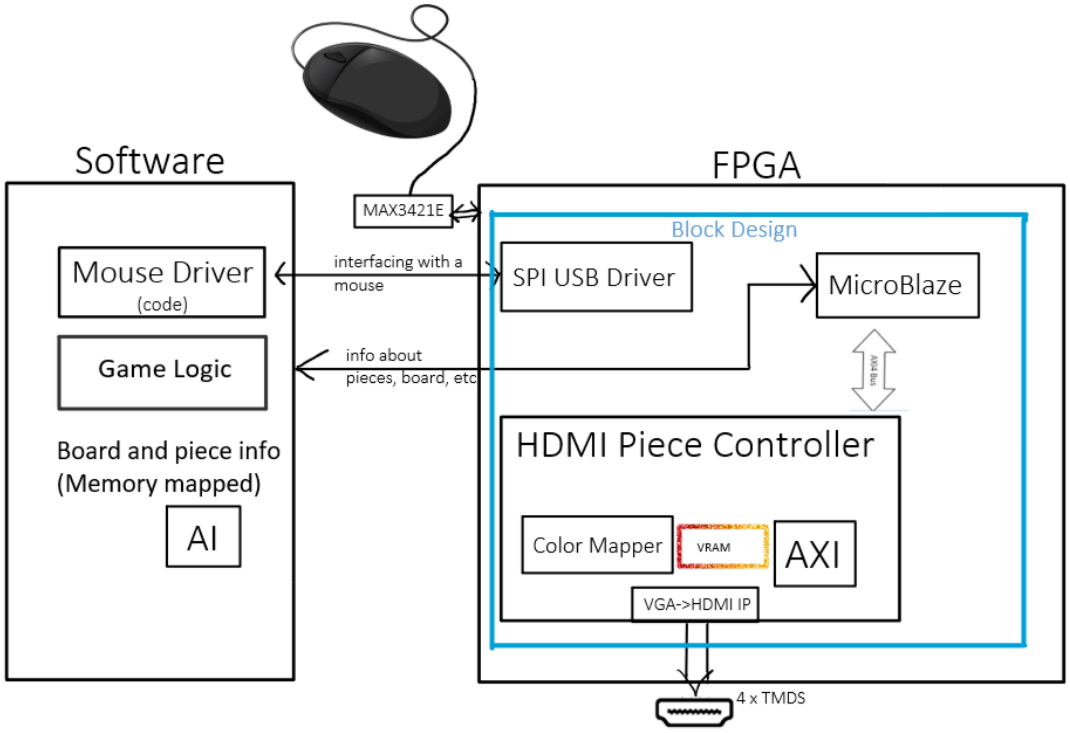
Our design of this project included the use of a memory-mapped HDMI piece controller which we created similar to that of the text controller in lab 7, however, we added more complex and nuanced sprites for each piece and used an entirely different positioning system to account for the 8 by 8 chess board instead of an 80 by 30 column text mode to display characters. It also utilized concepts from lab 6.2 where we effectively used SPI protocol and the USB drivers to receive mouse packets allowing us to display a moving cursor which would then be used to select and move pieces. Throughout building chess, we learned much more about the SoC design principles, how to interface between memory-mapped peripherals in software effectively, USB drivers, and about HID devices (specifically relating to the mouse). We began exploring move-generation and AIs by making a random (legal) move AI opponent.

Written Description:

Before delving into the details of the project, we first had to make a design decision on the implementation layout when making chess. For example, should the game logic and graphics be entirely handled in the hardware using SystemVerilog? Is it perhaps more advantageous to leave some of the slow updating features (i.e. moving a piece) to the software? The ultimate conclusion we came up with was to handle all of the game logic in software and all the graphics on the FPGA. This was because we realized that the game logic, which encompasses the piece movement, legality checks, and general flow of the game is only updated so infrequently in comparison to the universal 100 MHz clock. On the other hand, the graphics need to be updated quite frequently as each pixel's frequency rate is 25 MHz which means it is at least a fourth of the universal clock indicating a much higher required update rate. Additionally, we went this route since debugging the game on Vitis (software) took a lot less time as opposed to doing it in the hardware by regenerating the bitstream each time (taking ~20-30 minutes).

High level overview

As mentioned above, we dealt with the graphics on hardware (FPGA) and left the game logic to software. This means that the FPGA is oblivious to whatever logic is taking place on the chess board, but is responsible for actually handling the pieces, changing the game status text, and other graphic-related tasks commanded by software. The software, however, has no connection to the graphics but has an internal data structure which it uses to determine the validity of certain moves and almost acts as the "brain" of our project. Below is an illustration of what our design looks like:



Abstract block diagram

Graphics description

To draw the actual chess board and pieces as well as do anything graphic-related in this project, we used an "Image to COE" tool which allowed us to draw sophisticated sprites (in addition to the classic FONT_ROM) by essentially instantiating BRAM blocks of memory with a coefficient (COE) file that initialized a portion of the BRAM to the pixel contents of each image we needed. We did not create this tool and have credited the authors of it in the references section at the end. An important note is that due to space concerns, this tool also acts as a quantizer enabling us to fit the content of these images (which take up a lot of relative space) into the Urbana Board's BRAM by using the Lloyd-Max Quantization algorithm. While this is beneficial, there were also some drawbacks which we will shortly mention.

In order to use this tool, we first had to generate images. Unfortunately due to space constraints, we could not naively just pick any image for the board and pieces so when finding images, we used image compression tools prior to this tool to get it into a feasible image size (ex. 240 by 240 px for a board) and then apply the Image to COE tool. In the midst of doing this task, we realized that a chess piece's background needed to be transparent (or not drawn) so the solution we devised was to make each piece's background color hot pink - to then later filter out in our color mapping logic. A drawback of this was that some of the hot pink prior to applying the tool was being blended into other shades of pink making it challenging to filter out. The solution to this was for us to manually edit the generated …_palette.sv file to conform to a shade of gray (i.e. by filling a color like R=4,G=0,B=4 to R=4,G=4,B=4).

Black Bishop Image to COE output

To use the Image to COE tool effectively, we had to understand how to properly read in from the ROM which revolved around the following provided equation in the tool's output *..._example.sv* file.

```
// address into the rom = (x*xDim)/640 + ((y*yDim)/480) * xDim
// this will stretch out the sprite across the entire screen
assign rom_address = ((DrawX * 20) / 640) + (((DrawY * 40) / 480) * 20);
```

ROM Addressing for a 20 by 40 px image

As the comments mention, the default option stretches a sprite across an entire screen due to the division of 640 and 480 parameters. Then, by row-major addressing, you can access the current pixel based on the *DrawX* and *DrawY* timers. When applied to our project, we had to first draw the checkerboard which involved "stretching" a 240 by 240 px image to a 480 by 480 px image as we wanted to keep a consistent 1:1 aspect ratio for the board. This meant we needed to apply a 80 px offset to all of our sprites to center it. The following code shows the addressing of the board and a piece.

```
// Address into the appropriate shifted ROM, starts drawing at top left pixel (80, 0)
assign rom_address = (((DrawX-80) * 240) / 480) + (((DrawY * 240) / 480) * 240);
```

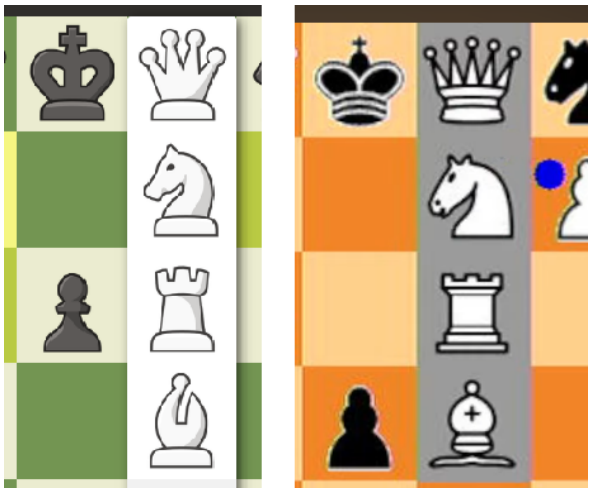
ROM Addressing for the board

```
assign rom_address = ((DrawX-80) - offsetX) + (DrawY - offsetY) * 60;
```

ROM Addressing for a piece

Note the use of “offsetX” and “offsetY”. These are essentially the top left pixel positions that are subject to change (from the software) as a piece is allowed to move around. We then essentially repeated this process for all types of unique pieces. For a captured piece, we had an extra active high bit which erases a piece and highlights the background if another active high bit called *selected* was asserted. As a king can never be captured in chess (checkmated at most), we set the captured bit to the equivalent of that square’s border being outlined with red to denote a check.

Building the graphics for pawn promotion was a bit tricky as it was two-fold. I had to first update the provided *..._example.sv* file for the pawns to accommodate the change in type by instantiating the other modules and using a decoder (explained more later) based on the current type and also add in an entirely new ROM for promotion. As shown below, the design of the promotion prompt was similar to that of an online chess implementations as compared below:



chess.com implementation (left) versus our prompt (right)

Construction of the piece controller

As depicted in the high-level diagram, the piece controller integrates the entirety of the graphics within our project which is done by initializing all pieces and images through their generated COE files in our color mapper. As one could imagine this would cause a lot of repeated instantiations with very similar inputs so to circumvent this repeated code issue we made use of generate statements within SystemVerilog. The piece controller is also responsible for the crucial task of communicating with MicroBlaze through the memory-mapped AXI-4 Lite registers (VRAM or what we called PIECE_RAM) based on the following memory map:

Word Address	Description
0-7	White Pawns (48 to 55), *REG0 corresponds to 48*
8-9	White Rooks (56, 63)
10-11	White Knights (57, 62)
12-13	White Bishops (58, 61)
14	White Queen (59)
15	White King (60)
16-23	Black Pawns (8 to 15)
24-25	Black Rooks (0, 7)
26-27	Black Knights (1, 6)
28-29	Black Bishops (2, 5)
30	Black Queen (3)
31	Black King (4)
32	Game Status

AXI Peripheral Memory (all ranges are inclusive)

Each piece's bit encoding requires key pieces of information that need to be communicated to hardware and software (in both directions). The following bit encodings were used for the various chess pieces:

[31:24]	[23:16]	[15:8]	[7:0]
Captured (1-bit)	Type of piece (3-bit)	Highlight Piece (1-bit): bit 9 Color (1-bit): bit 8	Position (6-bit)

Bit encoding for all chess pieces

Note that the 'captured' bit for the king is instead used for a check (i.e. a red border on the king). The type of piece is imperative for pawn promotion to work. The 'color' bit is 0 if the piece is white or 1 if it is black. One will also note the presence of the "game status" memory location which is used for displaying whose turn it is to play and game flow indicators (i.e. if checkmate is on the board or draw). The bit encodings are shown below for these features:

000	pawn
001	knight
010	bishop
011	rook
100	queen
101	king
110	unused
111	unused

Bit representation for 'type of piece (3-bit)'

BIT 0: WHITE TO PLAY
BIT 1: BLACK TO PLAY
BIT 2: CHECKMATE! WHITE WINS
BIT 3: CHECKMATE! BLACK WINS
BIT 4: DRAW!

One-hot encoded 'Game Status' register

You'll notice that the game status register is one–hot encoded which just means that only one of those specified bits are meant to be high at a time (ex. if bit 1 is high and all others are low, "BLACK TO PLAY" would be printed). These representations and memory-mapped regions were crucial in helping us design the necessary software component to fully connect the hardware to the game logic.

Color mapping logic

For the position of each chess piece, we opted to use a representation 0-63 inclusive where 0 represents the top left square and 63 is the bottom right square which are both from white's perspective. The issue though was we needed some method to translate between this index and a pixel position for our pieces (sprites) to know their exact position. With the use of math, however, you find these simple equations:

```
offsetX := (POS % 8)*60
offsetY := (POS / 8)*60
```

These offsets are used as the top left pixel positions for a piece at a given *POS* (spans 0-63 inclusive). From this, we are able to instantiate each module with the appropriate memory-mapped regions as specified above and control piece movement. Something to note is that similar to lab 7, you can use bit slicing to save on the operations of modulo and divide as 8 is fortunately a power of two. Then we put all these modules together in a huge combinational block with the top-most element being the cursor, followed by the promotion prompts, game status messages, and finally the pieces.

Cursor Development

A considerable amount of time was spent on understanding the mouse functionality for this project. In essence, the memory-mapped features of the SPI mouse report format includes the 'button status' the x and y displacements. This is summarized in the table below obtained from the os dev's documentation (https://wiki.osdev.org/USB_Human_Interface_Devices#USB_mouse):

Offset	Size	Description
0	Byte	Button status.
1	Byte	X movement.
2	Byte	Y movement.

Report format of the mouse

Bit	Bit Length	Description
0	1	When set to 1, indicates the left mouse button is being clicked.
1	1	When set to 1, indicates the right mouse button is being clicked.
2	1	When set to 1, indicates the middle mouse button is being clicked.
3	5	These bits are reserved for device-specific features.

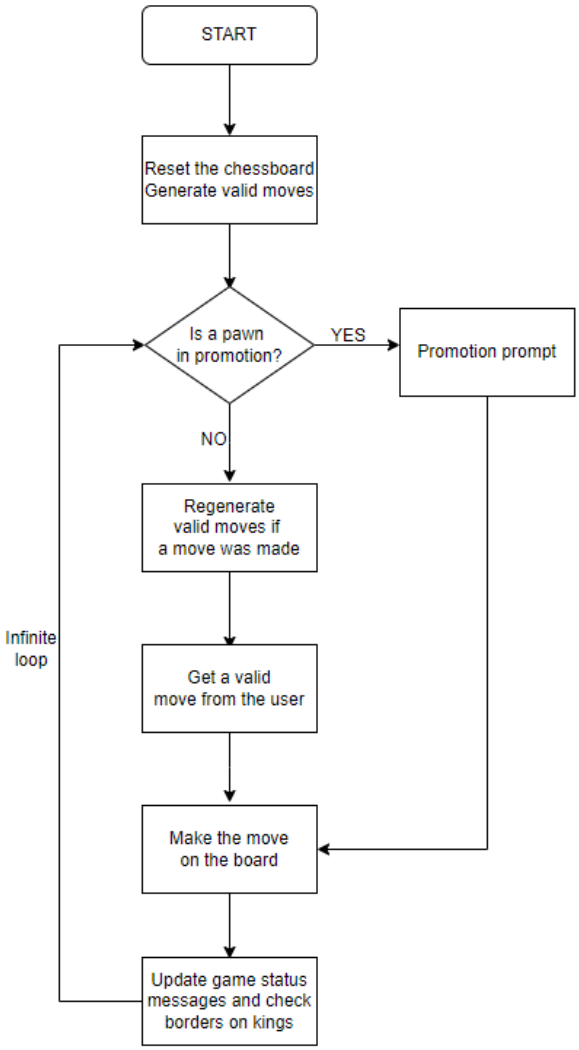
One-hot encoded button status bit representation

The HID mouse polling report code that was provided to us from the lab 6.2 code follows this format where the X and Y movements are reported as displacements from last the given time that the mouse moved. At first we attempted to create a hardware implementation of a cursor but after learning about the shortcomings due to the inherent discretization associated with the mouse displacement packets as well as the unsigned comparisons of SystemVerilog we thought of an alternative which was to just use create it in the software and then send over the cursor coordinates as an external input into the piece controller.

This was done by creating local variables in C that held the current *x, y* coordinates of the cursor and then adding on the displacement on each iteration of the infinite while loop (inside main()) while enforcing boundary conditions of staying within the screen. These coordinates were then sent to the color mapper where we had a circle or ball as the cursor combinationally outputted and changed colors if any button was pressed.

FSM

As our implementation for the game logic was entirely software based, we did not have a specific hardware implementation of an FSM. That being said, we did follow a rough algorithm that took form in the following flowchart.

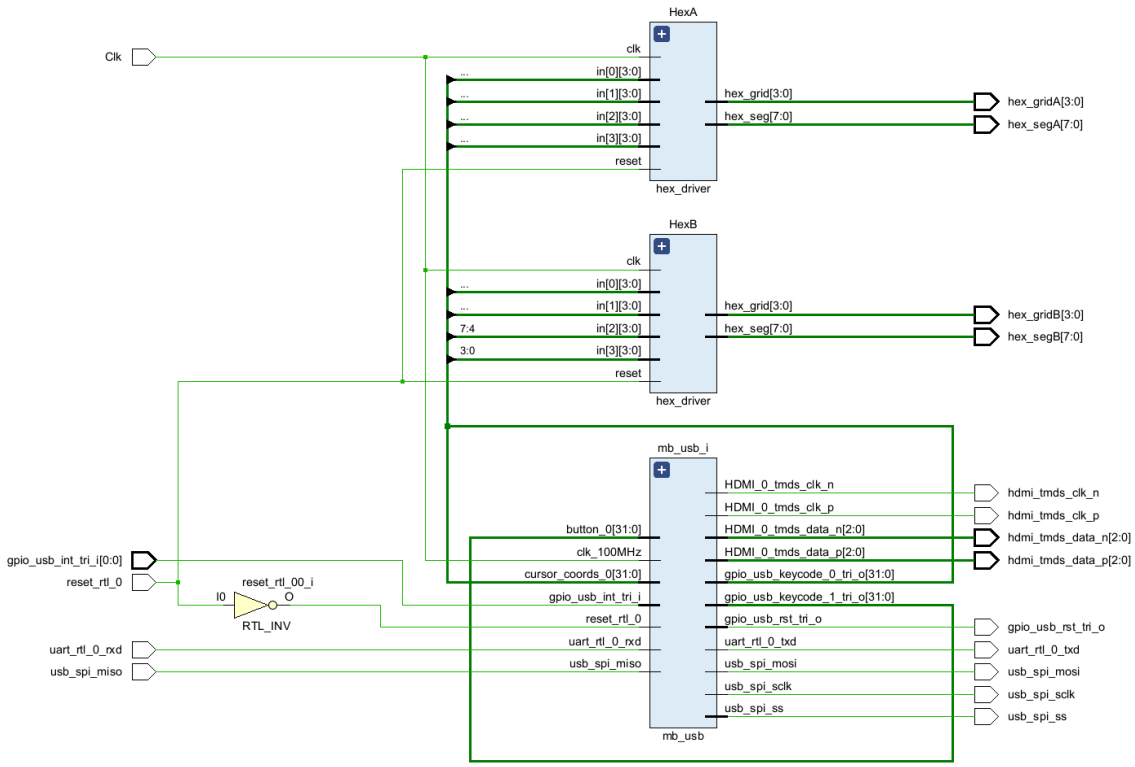


Software-based state diagram

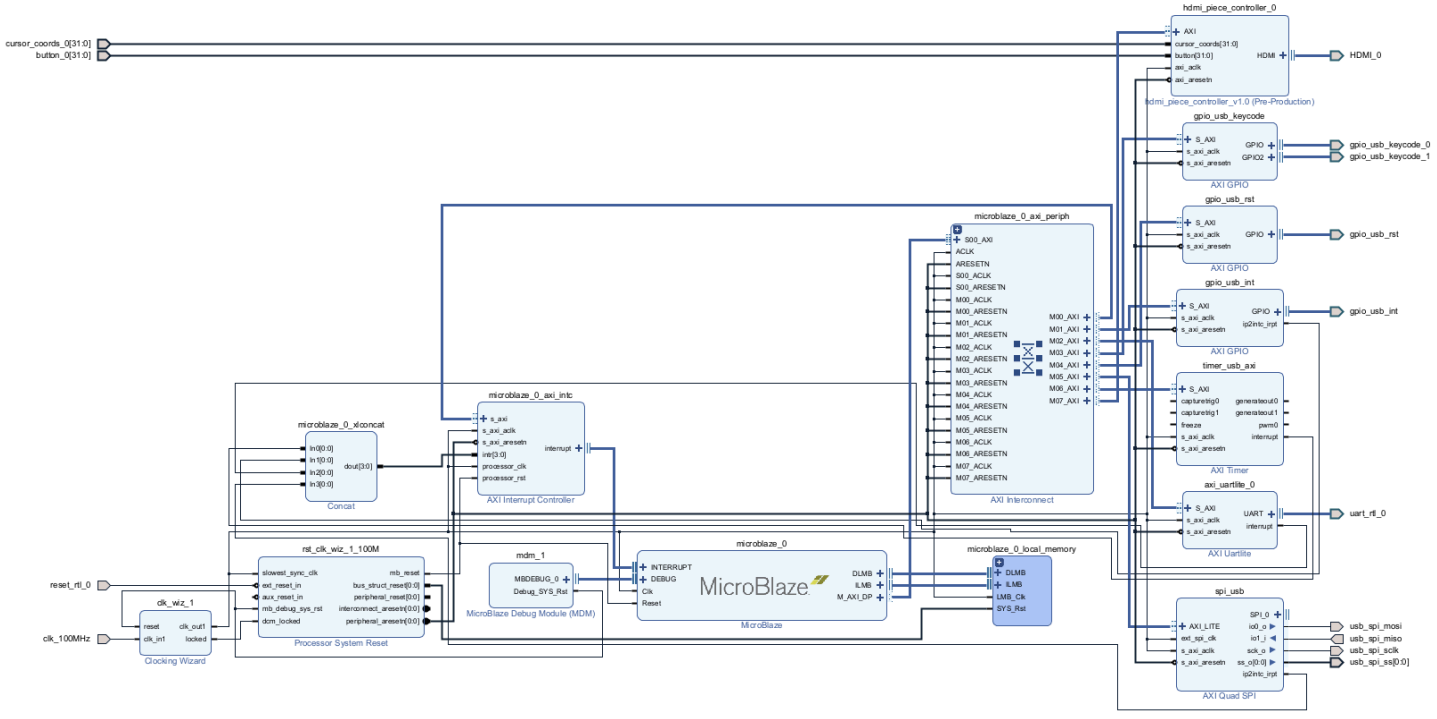
Note that this flowchart does not encompass the presence of an AI opponent and more extensive details about the tasks specified within these states can be found in 'Software Component Description'. Overall, the algorithm we chose for our chess engine makes use of a naive approach which checks for legality by first playing out all possible pseudo legal moves, checking if that move leaves the king endangered, and if it does, removes it from the list of valid moves. A pseudo-legal move is defined as one where a piece can move with no regard for the king's safety but in the general movement of that piece. For example, a knight can only move in L shapes, so all pseudo-legal moves would encompass all of these types of moves (while enforcing boundary conditions). This is a naive approach as one can observe that it is very computationally expensive and there are other ways of doing this, but we opted for simplicity.

Block Diagrams:

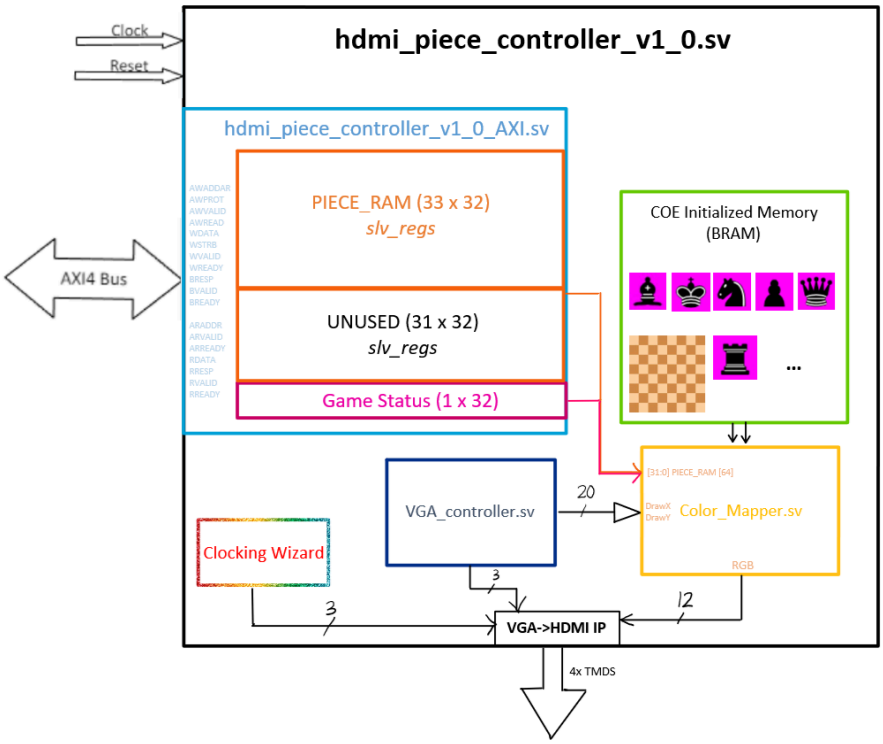
We have provided the main project's elaborated design schematic and block design. The piece controller's hardware ended up being too cumbersome to deal with (making it practically illegible) so we manually created one instead, ensuring the important signals are captured.



Top-level diagram



Block Design Diagram



Piece controller block diagram

Hardware Module Descriptions:

chess_top.sv

Module: `chess_top`

Inputs: `Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd`

Outputs: `gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB`

Description: Takes in all input signals into all the subcomponents of the project. Starting with universal signals, *Clk* and *reset_rtl_0*, these are responsible for connecting throughout the entire design and providing a synchronization and reset for all modules. Next, the USB + SPI signals included *[0:0] gpio_usb_int_tri_i, gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_miso, usb_spi_sclk, usb_spi_ss* which consisted of all the signals connecting to the AXI Quad SPI IP block as well as I/O signals which included common SPI lines (MOSI, MISO, SCLK, SS). Then the UART signals simply were *uart_rtl_0_rxd* and *uart_rtl_0_txd* which are responsible for receiving and transmitting data within this protocol respectively. For the HDMI portion, the necessary differential signals *hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p* are used (further explained in VGA-HDMI question response). Finally, the hex display signals *[7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB* are used to connect to Hex_Driver modules and display the current keycode being pressed.

Purpose: Top-most hierarchical module which combines all IPs, including the piece controller, the SPI interface, and MicroBlaze within the block design.

hex_driver.sv

Module: `hex_driver`
Inputs: `clk, reset, [3:0] in[4]`
Outputs: `[7:0] hex_seg, [3:0] hex_grid`

Description: For input, takes in four 4-bit sequences and converts them into the appropriate 7-bit hex segment display based on arbitrary constants.

Purpose: The hex_driver module is responsible for taking some sort of numerical input and displaying the hexadecimal output onto the 7-segment display LEDs.

bB_example.sv (repeated for bN, bQ, bR, wB, wN, wQ, wR)

Module: `bB_example`
Inputs: `vga_clk, [9:0] DrawX, [9:0] DrawY, blank, [9:0] offsetX, [9:0] offsetY, captured, selected`
Outputs: `bB_on, [3:0] red, [3:0] green, [3:0] blue`

Description: See section titled ‘Graphics description’. Sets the *bB_on* to be high whenever the piece is within the relevant 60 by 60 pixel block.

Purpose: Enables you to print a black bishop piece somewhere on the screen.

bP_example.sv (repeated for wP)

Module: `bP_example`
Inputs: `vga_clk, [9:0] DrawX, [9:0] DrawY, blank, [9:0] offsetX, [9:0] offsetY, captured, selected, [2:0] bP_type`
Outputs: `bP_on, [3:0] red, [3:0] green, [3:0] blue`

Description: In addition to the bB_example code, instantiates the other pieces that it can promote to (bQ,bR,bB,bN) and makes use of a decoder to combinationally assign the current type of piece associated with this pawn as per the piece bit representation (0 for pawn by default).

Purpose: Enables you to print a black pawn piece somewhere on the screen. Changes type depending on if it reached the back rank of a chessboard.

bB_palette.sv (repeated for all image-related modules)

Module: `bB_palette`
Inputs: `[3:0] index`
Outputs: `[3:0] red, [3:0] green, [3:0] blue`

Description: Instantiates a distributed memory block based on a simple palette look up table. Supports asynchronous reading.

Purpose: Gives the RGB intensities needed in bB_example to draw a certain pixel based on an index.

bCheckmate_example (repeated for bToPlay, draw, wCheckmate, wToPlay)

Module: `bCheckmate_example`
Inputs: `vga_clk, [9:0] DrawX, [9:0] DrawY, blank`
Outputs: `bCheckmate_on, [3:0] red, [3:0] green, [3:0] blue`

Description: Hard-coded to display the message if present in the game status register at top left pixel position 560,434

Purpose: Displays the CHECKMATE! BLACK WINS message on the bottom right of the screen.

bPromote_example.sv (repeated for wPromote)

Module: `bPromote_example`
Inputs: `vga_clk, [9:0] DrawX, [9:0] DrawY, blank, [9:0] offsetX, [9:0] offsetY, in_promotion`
Outputs: `bPromote_on, [3:0] red, [3:0] green, [3:0] blue`

Description: Uses the similar offsets to determine which file (column) to place the promotion prompt on based on if a pawn reaches the back rank of a chessboard.

Purpose: Shows the promotion prompt (image displayed earlier) for users to select a piece for their pawn to promote to.

Color_Mapper.sv

Module: `color_mapper`
Inputs: `[31:0] PIECE_RAM[64], [9:0] drawX, [9:0] drawY, [9:0] cursorX, [9:0] cursorY, click, clk_25MHz`

Outputs: `[3:0] red, [3:0] green, [3:0] blue`

Description:To deal with the cursor, takes in the cursor coordinates and uses the formula of a circle ($x^2 + y^2 <= r^2$) to create a “cursor_on” logic variable which is used to be either blue or red if a click is pressed. Next, it instantiates all of the …_example.sv files which we would generate using statements in SystemVerilog for repeated instantiations like pawns, bishops, etc. When making these instantiations , we assigned the memory-mapped regions of our PIECE_RAM into the corresponding …_example.sv instantiations to effectively control the pieces. Finally, we had a big combinational block that utilized if and else if statements to allow for overlapping in the following order (top to bottom): cursor, game status messages, pawn promotions, and finally pieces.

Purpose: Responsible for displaying everything on the screen. Sends out the relevant RGB intensities for the current pixel on the screen determined entirely by drawX and drawY to VGA-> HDMI IP.

VGA_controller.sv

Module: `vga_controller`
Inputs: `pixel_clk, reset`
Outputs: `hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY`

Description: The VGA controller module has outputs “drawX” and “drawY”, which are timers to keep track of the vertical and horizontal position of the pixels being drawn to the screen. When the position reaches the largest allowed X coordinate value, the next pixel to be drawn resets back to the far left of the screen when *hsync* is pulled low. Similarly, when the Y position reaches past the bottom edge of the boundary, the vertical timer resets when *vsync* is pulled low (which happens 60 times a second). Each pixel is redrawn based on the frequency specified in *pixel_clk* (typically 25 MHz). Additionally, when the drawX and drawY counters reach a pixel limit, there is a blanking interval (needed time for *hsync*/*vsync* to properly reassert itself).

Purpose: Maps a timer to spatial coordinates, used to essentially direct “an electron beam” to the monitor. Used in conjunction with other modules (like a color mapper) to create graphical signals in terms of analog voltages.

hdmi_piece_controller_v1_0.sv

Module: `hdmi_piece_controller_v1_0`
Parameters: `C_AXI_DATA_WIDTH=32, C_AXI_ADDR_WIDTH=8`

Inputs: `[31:0] cursor_coords, [31:0] button, axi_aclk, axi_aresetn, axi_awaddr[31:0], axi_awprot[2:0], axi_awvalid, axi_wdata[31:0], axi_wstrb[3:0], axi_wvalid, axi_araddr[31:0], axi_arprot[2:0], axi_arvalid, axi_rready`

Outputs: `axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_bready, axi_arready, axi_rdata[31:0], axi_rresp[1:0], axi_rvalid`

Description: Top-level for the piece controller. Sends the button and cursor coordinate information straight to the color mapper for display and performs common AXI4-Lite protocol usage as in lab 7. An address width of 8 was chosen as we required $\log_4(64)$ memory locations.

Purpose: This module is essentially the top-level module that allows for AXI communication between the internal modules of the IP block.

hdmi_piece_controller_v1_0_AXI.sv

Module: `hdmi_piece_controller_v1_0_AXI`
Parameters: `C_S_AXI_DATA_WIDTH=32, C_S_AXI_ADDR_WIDTH=8`

Inputs: `S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR[11:0], S_AXI_AWPROT[2:0], S_AXI_AWVALID, S_AXI_WDATA[31:0], S_AXI_WSTRB[3:0], S_AXI_WVALID, S_AXI_ARADDR[11:0], S_AXI_ARPROT[2:0], S_AXI_ARVALID, S_AXI_RREADY`
Outputs: `[31:0] regs[64], S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP[1:0], S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA[31:0], S_AXI_RRESP[1:0], S_AXI_RVALID`

Description: Creates a “piece ram” of up to sixty 32-bit registers that are able to read and write from the AXI protocol. A more extensive description of this module can be found in lab 7. An address width of 8 was chosen as we required $\log_4(64)$ memory locations.

Purpose: This module contains the functionality of the AXI communication and is used in the previous module to provide AXI connection between the rest of the components in the IP block. The AXI implementation consists of five separate channels: Read Address, Write Address, Read Data, Write Data, and Write Response. Each channel works together to form successful data transfer between master and slave and this AXI functionality is implemented in this module.

Software Component Descriptions:

While the project is running and constantly polling for mouse clicks, the game logic initializes values like the player turn, castle rights, checkmate/stalemate flags, and other values that require a starting value.

During the execution of the chess game, the program continuously monitors input events, particularly mouse clicks, to interact with the user interface. At the outset, it sets flags indicating the player's turn, whether it's a human player's turn or an AI player's turn, based on the color of the pieces and the player

settings. If it's the human player's turn and a promotion is pending, the program prompts the user for piece promotion, handles the selection, and executes the move accordingly.

In the case of AI moves, the program regenerates all possible moves for the current player and evaluates the game state. If it's not the human player's turn and the game is not in a checkmate state, the AI generates its move. During this process, if a promotion is pending, the AI handles it automatically. Once the AI generates a move, it executes it and updates the game state. For human moves, the program waits for mouse clicks indicating the player's selection of a piece to move and the destination square. It validates the move, executes it, and updates the game state accordingly. After each move, the program updates the visual indication of check status for both kings on the board. Additionally, it checks for game-ending conditions such as checkmate or stalemate and updates the game status accordingly, potentially ending the game or allowing it to continue based on the outcome. Throughout this process, the program continually monitors user input and updates the game state to provide a seamless and interactive chess-playing experience. This game logic portion of our project in C was largely guided by Eddie Sharick's *Creating Chess in Python* tutorial playlist on YouTube.

hdmi_piece_controller.c / hdmi_piece_controller.h:

void resetBoard();

Parameters: `N/A`
Returns: `N/A`

Purpose: Repositions all pieces to starting squares.

Description: First, resetBoard() clears all memory-mapped piece registers. Then, it goes through each piece by type and color and sets each type and default starting position in the piece's memory-mapped register.

void movePiece(uint8_t idx, uint8_t target);

Parameters: `uint8_t idx, uint8_t target`
Returns: `N/A`

Purpose: Moves a piece located at an initial position to the target position in the memory-mapped registers.

Description: Given "`idx`" (the word address of the piece to be moved) and "`target`" (the new position for the piece), the piece's position portion of its register contents are cleared and updated to the target position value.

void deletePiece(uint8_t idx);

Parameters: `uint8_t idx`

Returns: `N/A`

Purpose: Removes the specified piece from the board.

Description: Given the index of a piece to be removed from the board, deletePiece() sets the capture bit of the piece's memory-mapped register to high in order to remove the piece from the board.

void revivePiece(uint8_t idx);

Parameters: `uint8_t idx`
Returns: `N/A`

Purpose: Brings back the specified piece from the board (undo).

Description: Given the index of a piece to be brought back, revivePiece() sets the capture bit of the piece's memory-mapped register to low in order to bring back the piece to the board.

void transformPawn(uint8_t pawn_idx, uint8_t type);

Parameters: `uint8_t pawn_idx, uint8_t type`
Returns: `N/A`

Purpose: Promotes a pawn on the piece controller by changing its type.

Description: Given the index of a pawn to be promoted and a piece type to change it to, transformPawn() clears the piece type bits of the specified pawn's register and updates it to the piece type passed in by the parameter "`type`".

void highlightPiece(uint8_t idx, uint8_t sel);

Parameters: `uint8_t idx, uint8_t sel`
Returns: `N/A`

Purpose: Highlights or unhighlights a piece, typically used if a piece is in selection.

Description: Given the word address of a piece to highlight and a select bit, highlightPiece() clears and updates the bit responsible for keeping track of highlighting in the piece's register. If "`sel`" is high, the piece will be highlighted.

void checkHighlight(uint8_t color, uint8_t sel);

Parameters: `uint8_t color, uint8_t sel`

Returns: `N/A`

Purpose: Draws a red border (check-highlight) around a king.

Description: Given the king's color and a select bit, checkHighlight() clears and updates the bit responsible for keeping track of check highlighting in the piece's register. If "`sel`" is high, the piece will be highlighted.

void whiteToPlayMessage(uint8_t sel);
Parameters: `uint8_t sel`
Returns: `N/A`

Purpose/Description: Displays the "WHITE TO PLAY" text if sel is 1, disables it if sel is 0.

void blackToPlayMessage(uint8_t sel);
Parameters: `uint8_t sel`
Returns: `N/A`

Purpose/Description: Displays the "BLACK TO PLAY" text if sel is 1, disables it if sel is 0.

void whiteCheckmateMessage(uint8_t sel);
Parameters: `uint8_t sel`
Returns: `N/A`

Purpose/Description: Displays the "CHECKMATE! WHITE WON" text if sel is 1, disables it if sel is 0.

void blackCheckmateMessage(uint8_t sel);
Parameters: `uint8_t sel`
Returns: `N/A`

Purpose/Description: Displays the "CHECKMATE! BLACK WON" text if sel is 1, disables it if sel is 0.

void stalemateMessage(uint8_t sel);
Parameters: `uint8_t sel`
Returns: `N/A`

Purpose/Description: Displays the "DRAW!" text if "`sel`" is 1, disables it if "`sel`" is 0.

game.c / game.h:

void makeMove();

Parameters: `N/A`
Returns: `N/A`

Purpose: Performs a move based on the respective player's turn. Assumes 'current_move' is NOT valid (does legality-checking). Updates the game board structure and memory-mapped piece.

Description: If the game's current state is "VALID" and depending on the type of move to be made (castle, en passant, promotion, normal move, etc), if the move is valid, both the software and hardware boards are updated to reflect the move. In the case of capturing, the piece getting captured is deleted from its position by setting its capture bit in its memory-mapped register high. The game's state is then updated to "IDLE" to prepare for a new move to be made. The move is also added to the "played_moves" list to allow for undo capabilities, the move is made in software, the castling log is updated, and the current move value is reset to prepare for a new move.

void makeMoveSoftware(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose/Description: Updates the game board structure only in software by changing the board array's values. After the move is made, swapTurn() is called to update the turn. Helper function used in makeMove().

void undoMove();

Parameters: `N/A`
Returns: `N/A`

Purpose: Uses the last played move to restore the game's previous state.

Description: If no moves have been played yet, the function outputs a message indicating the absence of any moves. Otherwise, it retrieves the details of the last move played from the "played_moves" list. The function then checks if the last move involved en passant, resetting the en passant indicator accordingly. If the move was an en passant move, it is undone by calling the appropriate function. Next, the function checks if the last move involved castling. If it did, the

function undoes the castling move using the appropriate function. The turn is swapped back to the previous player's turn to ensure consistency. If the last move involved pawn promotion, the function resets the pawn promotion status. The software board is updated to undo the move, and if a piece was captured during the move, it is revived and returned to its previous position on the board. If the move involved pawn promotion, the promoted piece is demoted back to a pawn. The function then updates the castle log to reflect the previous castle rights, ensuring the game state is consistent. If there is no previous castle history, default castle rights are set. Finally, the function signals the move generator to regenerate moves, sets the game state to "IDLE" to prepare for the next move, and returns control.

void undoMoveSoftware(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose/Description: Updates the game board structure only in software by changing the board array's values. After the undo takes place, swapTurn() is called to update the turn. Helper function used in undoMove()

void setPossibleMoves(MOVE* list, uint8_t* end_idx);

Parameters: `MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Scan the whole board for all pseudo-legal moves based on whoever's turn it is to move. Store (append) all these inside a 'MOVE' array.

Description: Iterates through each of the 64 squares on the board, examining the piece type and color of each square. It checks if the color of the piece matches the current player's turn. Depending on the type of the current piece, the function calls specific helper functions to set the possible moves of the piece currently being observed.

void setPawnMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: This function calculates pseudo-legal moves for a pawn based on its position and color. For a white pawn, it considers moving one or two squares forward (initial move),

capturing diagonally left or right, and en passant captures. Edge cases are handled to ensure moves stay within the board boundaries. Positions are calculated using offsets based on the 0-63 board tile positions and addMove() is called to append the move to the “possible_moves” list.

void setKnightMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: The function iterates through each of the eight possible directions a knight can move, considering both the horizontal and vertical offsets. For each direction, it calculates the destination square and checks if it falls within the bounds of the board. If the proposed move doesn’t capture an ally piece or go out of bounds, the move is appended to the list of possible moves.

void setBishopMoves(uint8_t pos, uint8_t piece_type, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, uint8_t piece_type, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: The function explores each diagonal direction from the bishop's position. It iterates along each diagonal until it reaches the edge of the board or encounters a piece. For each empty square or enemy piece encountered while iterating, addMove() adds the move to the list of possible moves. If the square contains a friendly piece, the function stops exploring that diagonal direction.

void setRookMoves(uint8_t pos, uint8_t piece_type, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, uint8_t piece_type, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: The function explores each direction from the rook's position. It iterates along each direction until it reaches the edge of the board or encounters a piece. For each empty square or enemy piece encountered while iterating, addMove() adds the move to the list of possible moves. If the square contains a friendly piece, the function stops exploring that direction.

void setQueenMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: Calls setBishopMoves() and setRookMoves() to cover all possible directions of movement for a queen.

void setKingMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for the piece associated with the passed-in move and board position.

Description: The function explores each of the eight possible directions a king can move, considering both the horizontal, vertical, and diagonal offsets. For each direction, it calculates the destination square and checks if it falls within the bounds of the board. For each empty square or enemy piece encountered while iterating, addMove() adds the move to the list of possible moves. If the square contains a friendly piece, the function skips that direction.

void setCastleMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Helper function to be used in setPossibleMoves() that appends possible (pseudo-legal) moves for castling.

Description: Given a castle move, setCastleMoves() calls setKingsideCastleMoves() or setQueensideCastleMoves() depending on the color to play and whether or not the castle rights for the potential move are available or not. If a rook has moved previously, that side’s castle will not be valid and if the king has previously moved, that king will not be able to castle.

void setKingsideCastleMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);
void setQueensideCastleMoves(uint8_t pos, MOVE* list, uint8_t* end_idx);
Parameters: `uint8_t pos, MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose/Description: Given a king-side or queen-side castle move, if the tiles to castle are empty, the move is added to the list of possible moves by calling addMove().

void regenerateMoves();
Parameters: `N/A`
Returns: `N/A`

Purpose: After each board change (i.e. whenever move_made is high), updates the list of possible and valid moves.

Description: If a move has been made, the list of valid moves is reset by calling clearMoveList(). A new valid moves list is generated and “move_made” is reset back to 0.

uint8_t getWKPosition();
uint8_t getBKPosition();
Parameters: `N/A`
Returns: `Black/White king position (0-63)`

Purpose: Getters that return the white and black king position respectively.

Description: Iterate through each tile on the board and return the position value of the index where the white or black king was located.

void setValidMoves();
Parameters: `N/A`
Returns: `N/A`

Purpose: Generates all pseudo-legal moves in a given position, plays out each move, then generates all opponent's pseudo-legal moves to that move. Examines each opponent move to consider if king is under attack. If a move is found that lets your king be captured, it is NOT a valid move.

Description: First, setValidMoves() clears the list of possible moves and generates all possible moves for the current player's pieces using the setPossibleMoves(). Then, it iterates through each possible move and temporarily plays out the move using makeMoveSoftware(). After playing the move, it checks if the opponent's king is in check by generating all possible opponent moves from this move. If the move does not endanger the opponent's king, it is considered valid and added to the list of valid moves. After evaluating all possible moves, it checks if the list of valid moves is empty. If it is, it checks if the current player's king is in checkmate or if the game has resulted in a stalemate. If there are valid moves, it resets the checkmate and stalemate flags.

uint8_t squareUnderAttack(uint8_t pos);

Parameters: `uint8_t pos`
Returns: `1 if attacked 0 else`

Purpose: An 'attack' is defined as an opposing piece that can move a piece to that square on the next turn.

Description: Swaps to opponent's turn using swapTurn() and clears the list of possible moves for the opponent's pieces and generates all possible moves for the opponent using the setPossibleMoves(). After generating the opponent's possible moves, it swaps the turns back to the original player's turn. Next, it iterates through the list of the opponent's possible moves and checks if any of these moves can land on the specified square. If any move has an ending position equal to “pos”, it means the square is under attack by an opponent's piece, and the function returns 1. If no opponent's move can land on the specified square, the function returns 0, indicating that the square is not under attack.

uint8_t inCheck();

Parameters: `N/A`
Returns: `1 if the color in 'to_play' is under attack 0 else`

Purpose: A 'check' is when a side's king is under attack, being threatened to be captured on the next turn. This function returns 1 if the current side's king is attacked.

Description: Depending on the current color to play, this function passes in the white or black king’s position into squareUnderAttack(), which returns a 1 if the king is under attack, indicating a check.

void setPawnPromotion(MOVE* M, uint8_t* flag);

Parameters: `MOVE* M, uint8_t* flag`

Returns: `N/A`

Purpose/Description: A pawn promotion occurs when a pawn of the current color in 'to_play' reaches the back rank of their respective side. Corresponds to row 0 for white and row 7 for black if looking at the board from white's perspective. Examines the given and sets the flag variable to 1 if promotion, 0 else.

void swapTurn();

Parameters: `N/A`
Returns: `N/A`

Purpose/Description: Simple helper to change sides. Updates “to_play” and “enemy” based on the current “to_play” value.

void setEnPassant(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose: Sets the relevant flag variable and position of a potential en passant move when a two advance pawn is recorded. Do not assume the move passed in is a pawn move.

Description: First checks if the moved piece is a pawn. If it is not, en passant is not possible, so it sets “en_passant_possible” to 0 and “en_passant_pos” to 254 (an invalid position) and returns. If the moved piece is a pawn, the function calculates the color of the pawn based on its starting position. It then checks if the move was a two-square pawn advance. If it was, en passant becomes possible, and the en passant position is set to the square behind the moved pawn, either one row above or below, depending on the pawn's color. If the move was not a two-square pawn advance, en passant is not possible, so “en_passant_possible” is set to 0, and “en_passant_pos” is set to 254.

void updateCastleRights(MOVE* M, CASTLE_RIGHTS* rights);

Parameters: `MOVE* M, CASTLE_RIGHTS* rights`
Returns: `N/A`

Purpose: Checks if a move changes the status of castle rights e.g. if a king or rook is moved, the respective right updates to 0.

Description: If a white king or a white rook moves, the function updates the white kingside (WKS), white queenside (WQS), or both rights to 0 if necessary. If a black king or a black rook

moves, the function updates the black kingside (BKS), black queenside (BQS), or both rights to 0 if necessary. If a rook is captured, its respective side's castling rights also updates to 0.

uint8_t isCastle(MOVE* M);

Parameters: `MOVE* M`
Returns: `If specified move is a castle attempt, return 2 if queen-side 1 if king-side, else 0.`

Purpose: Determines if a passed-in move is a castle attempt, and which side the castle is happening from.

Description: If the move's piece address is not a white or black king, returns 0. If the move's end position minus the start position is 2, the move is king side. If the move's start position minus the end position is 2, the move is queen side.

void makeCastleMove(MOVE* M, uint8_t* KS);

Parameters: `MOVE* M, uint8_t* KS`
Returns: `N/A`

Purpose: Helper used in makeMove() to perform a castle move. Updates both the piece controller and move data structure. Assumes the move passed in is a castle move.

Description: For a kingside castle, makeCastleMove() moves the king two squares towards the rook and the rook to the square adjacent to the king's final position. It updates both the hardware board and the software board accordingly. For a queenside castle, the function moves the king two squares toward the rook and then moves the rook to the square adjacent to the king's final position. It updates both the hardware board and the software board accordingly. After making the castle move, the function updates the castle rights, pushes the castle rights onto the castle log, updates the move log for undo capabilities, clears the current move, swaps the turn to the opponent, sets the game state to idle, and sets the move_made flag to 1.

void undoCastleMove(MOVE* M, uint8_t* KS);

Parameters: `MOVE* M, uint8_t* KS`
Returns: `N/A`

Purpose: Helper used in undoMove() to undo a castle move. Updates both the piece controller and move data structure. Assumes the move passed in is a castle move.

Description: If the move was a kingside/queenside castle, the function moves the king and the rook back to their original positions. It updates both the hardware board and the software board accordingly. After undoing the castle move, the function pops the castle rights from the castle log and updates the current castle rights accordingly. If there is no previous history of castle rights, it resets all castle rights. Then, it sets the move_made flag to 1, swaps the turn back to the original player, sets the game state to idle, and returns.

uint8_t isEnPassant(MOVE* M);

Parameters: `MOVE* M`
Returns: `If specified move is an en passant attempt, return 1 otherwise 0`

Purpose: Determines if a passed-in move is an en passant.

Description: First, isEnPassant() checks if the moved piece is a pawn and if the captured piece is either a pawn or an empty tile. Then, it calculates the position of the captured piece based on the captured piece's address retrieved from the piece controller. Next, it checks if the move meets the conditions for an en passant capture: The move is a diagonal movement, either to the left or right. The pawn moves to the square adjacent to the captured piece's original position. If these conditions are met, the function returns 1, indicating that the move is an en passant capture. Otherwise, it returns 0.

void makeEnPassantMove(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose: Helper used in makeMove() to perform an en passant pawn move. Updates both the piece controller and move data structure. Assumes the move passed in an en passant pawn move.

Description: First, makeEnPassantMove() calculates the displacement of the move by subtracting the starting position from the ending position. Then, based on the current player's turn (either white or black), it determines whether the en passant capture is to the left or right. After executing the move in both the piece controller and the software board representation, it marks that a move has been made, updates the move log for undo functionality, clears the move structure, swaps the turn, and sets the state to idle.

void undoEnPassantMove(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose: Helper used in makeMove() to perform an en passant pawn move. Updates both the piece controller and move data structure. Assumes the move passed in an en passant pawn move.

Description: The function undoEnPassant() first swaps the turn to ensure that the undo operation reflects the state before the en passant move. Then, it calculates the displacement of the move by subtracting the ending position from the starting position. Based on the current player's turn (either white or black), it determines whether the en passant capture was to the left or right. After restoring both the piece controller and the software board representation to their previous states, it marks that a move has been made, swaps the turn, and sets the state to idle.

MOVE findRandomMove(MOVE* list, uint8_t* end_idx);

Parameters: `MOVE* list, uint8_t* end_idx`
Returns: `Random move in a given list of moves.`

Purpose: Function that returns a random move in a given list of moves.

Description: Generates a random number between 0 and the index marking the end of the move list (exclusive). This random number determines the index of the move to be selected. Then, it returns the move at the randomly selected index from the provided list of moves.

void setAIMove();

Parameters: `N/A`
Returns: `N/A`

Purpose: Uses generated move to update 'current_move' and 'state'.

Description: Creates a temporary move produced by findRandomMove() using “valid_moves” as the list to randomly choose from. Then, all of current_move’s attributes are updated to the temporary move, the game’s state is set to “VALID”, and “ai_move_generated” is set to 1 to indicate that the AI move has been made.

void promotionAI(MOVE* M);

Parameters: `MOVE* M`
Returns: `N/A`

Purpose: Uses generated move to promote 'current_move' and update 'state'.

Description: Randomly selects a value between 1 and 4 that decides what piece to promote a pawn to. Updates the game's state to "VALID", "promotion_ready" to 1, and "ai_move_generated" to 1.

input.c / input.h:

uint8_t getPosition(uint16_t x, uint16_t y);

Parameters: `uint16_t x, uint16_t y`
Returns: `The respective tile from 0-63`

Purpose: Returns the current tile position based on cursor position (an index into board[] from which you get a word address).

Description: First, getPosition() checks if the input coordinates (x, y) fall within the bounds of the chessboard on the screen. Next, it adjusts the x-coordinate by subtracting 80 to account for the shifted board on the screen. Then, it calculates the tile coordinates by dividing the adjusted x and y coordinates by 60. Since both board_x and board_y range from 0 to 479 (480 pixels for the entire board), dividing by 60 maps them to tile coordinates. Finally, it computes the 1D index of the tile position on the board using row-major addressing, where the row offset is multiplied by 8 (the number of tiles in a row) and added to the column offset. This index represents the position of the tile in the chessboard array.

void setMove(BYTE click, uint16_t x, uint16_t y);

Parameters: `BYTE click, uint16_t x, uint16_t y`
Returns: `N/A`

Purpose: Uses input to update 'current_move' and 'state'. If the middle mouse button is pressed, an undo is performed. Adds piece selection by highlighting its background.

Description: The setMove() function responds to mouse clicks and updates the game state accordingly. In the IDLE state, a left-click on a tile occupied by the player's piece records the piece type, address, and position as the current move, transitioning to the SELECT state and highlighting the selected piece. Subsequent left-clicks on valid tiles record the end position and any captured piece, transitioning to the COMPLETE state. If an invalid move is attempted by left-clicking on a tile occupied by the player's own piece in the COMPLETE state, it returns to SELECT and highlights the new selection. Confirmation of a valid move transitions to the VALID state, removing the highlight. Middle mouse clicks in the IDLE state initiate undo requests, with confirmation triggering the undo operation in the SELECT state.

void addMove(uint8_t piece_type, uint8_t piece_addr, uint8_t start_pos, uint8_t end_pos, uint8_t captured_piece_addr, MOVE* list, uint8_t* end_idx);

Parameters: `uint8_t piece_type, uint8_t piece_addr, uint8_t start_pos, uint8_t end_pos, uint8_t captured_piece_addr, MOVE* list, uint8_t* end_idx`

Returns: `N/A`

Purpose: Takes in attributes to instantiate a move, a list of possible moves, and a pointer to the end index of the list. Instantiates a move and appends the move to a given list.

Description: The addMove() function creates a temporary move and assigns its attributes to those that were passed into the function. It then appends this move to the end of the passed-in list and increases the value of the ending index of the passed-in list by 1.

uint8_t getType(uint8_t cursor_position);

Parameters: `uint8_t cursor_position`

Returns: `The type of piece`

Purpose: Returns the type of piece at a tile position.

Description: The getType() function first obtains the index of the piece from the board array based on the cursor position. Then, it checks if the index equals 255, indicating an empty square, and returns -1 in such a case. If the square is not empty, it extracts the piece type from the piece control array by shifting the corresponding bits to the right by 16 positions and applying a bitwise AND operation with 0x07 (binary 0000 0111) to extract the 3-bit piece representation.

uint8_t getColor(uint8_t cursor_position);

Parameters: `uint8_t cursor_position`

Returns: `The color (black or white)`

Purpose: Returns the color of a piece at a tile position.

Description: The getColor() function first retrieves the index of the piece from the board array based on the cursor position. If the index equals 255, indicating an empty square, it returns 2 to represent an empty tile. If the square is not empty, it checks the color of the piece by extracting the color bit from the piece control array. It shifts the corresponding bits to the right by 8 positions and applies a bitwise AND operation with 0x01 (binary 0000 0001) to extract the color bit.

uint8_t compareMoves(MOVE* M1, MOVE* M2);

Parameters: `MOVE* M1, MOVE* M2`
Returns: `0 if not the same, 1 if they are same`

Purpose: Checks if two moves are the same.

Description: The compareMoves() function compares two moves represented by the MOVE structures M1 and M2. It returns 1 if all corresponding attributes in both M1 and M2 are equal, indicating that the moves are identical. Otherwise, it returns 0.

uint8_t moveSearch(MOVE* list, uint8_t* end_idx, MOVE* elem);

Parameters: `MOVE* list, uint8_t* end_idx, MOVE* elem`
Returns: `0 if not found, 1 if present`

Purpose: Linearly searches a move array for a specified move.

Description: The moveSearch() function iterates through a list of moves represented by the list array until it finds a move that matches the move represented by the elem pointer. It compares each move in the list with the given move using compareMoves().

void pushMove(MOVE* list, uint8_t* end_idx, MOVE* elem, uint8_t length);

Parameters: `MOVE* list, uint8_t* end_idx, MOVE* elem, uint8_t length`
Returns: `N/A`

Purpose: Adds (pushes) a move to an array, if array is full, clears entire array and adds new element to the start.

Description: The pushMove() function adds a move represented by the elem pointer to a list of moves represented by the list array. It also maintains an “end index” end_idx that points to the next available position in the list. If the list is already full (its length is equal to length), the function clears the entire array and resets the end index to 0 before adding the new move. Finally, it inserts the new move at the position indicated by the end index and increments the end index by 1.

MOVE popMove(MOVE* list, uint8_t* end_idx);

Parameters: `MOVE* list, uint8_t* end_idx`
Returns: `Deleted item from the array`

Purpose: Removes (pops) the topmost move in an array and returns it. If array is empty, returns a MOVE with all fields init to 254.

Description: The popMove() function retrieves and removes the last move from a list of moves represented by the list array. It also maintains an end index that points to the next available position in the list. If the list is empty (its end index is 0), the function returns a default initialized move structure. Otherwise, it retrieves the last move from the list, clears the move at that position, decrements the end index by 1, and returns the retrieved move.

void printMove(MOVE* M);
Parameters: `MOVE* M`
Returns: `N/A`

Purpose/Description: Prints all attributes of a 'MOVE'.

uint8_t leftEdge(uint8_t* pos);
uint8_t topEdge(uint8_t* pos);
uint8_t rightEdge(uint8_t* pos);
uint8_t bottomEdge(uint8_t* pos);
Parameters: `uint8_t* pos`
Returns: `1 if on edge, 0 if not`

Purpose/Description: Checks if a piece is on an edge of the chess board. Each function takes a pointer to the position as input and returns a boolean value indicating whether the position is at the specified edge of the board. The left edge is identified if the position modulo 8 equals 0, the top edge if the position divided by 8 equals 0, the right edge if the position modulo 8 equals 7, and the bottom edge if the position divided by 8 equals 7.

void clearMove(MOVE* M);
Parameters: `MOVE* M`
Returns: `N/A`

Purpose/Description: Clears a MOVE by setting all of the passed in move's attributes to 254.

void clearMoveList(MOVE* list, uint8_t* end_idx);
Parameters: `MOVE* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Clears a MOVE array by setting all elements 0-END-1 (inclusive) to 0.

Description: The clearMoveList() function calls popMove() using the passed in list until the end index of the passed in list reaches 0, indicating an empty list.

void clickPollPosition(BYTE* click, uint16_t* x, uint16_t* y);

Parameters: `BYTE* click, uint16_t* x, uint16_t* y`
Returns: `N/A`

Purpose: Helper to set a position (0-63). Will set 254 if a click transaction is incomplete. In addition, sets promotion_pos to 253 if the middle mouse button is pressed .

Description: If the system is in an idle state and there's no mouse click detected (click is zero), it resets the promotion position to 254 and returns without further action. When the system is idle and detects a left-click (click equals 1), it transitions to the complete state and records the position (pos) where the left-click occurred as the promotion position. In the complete state, if there's no mouse click (click is zero), indicating the completion of the left-click operation, the system returns to the idle state.

void promotionPrompt(BYTE* click, MOVE* M, uint16_t* x, uint16_t* y);

Parameters: `BYTE* click, MOVE* M, uint16_t* x, uint16_t* y`
Returns: `N/A`

Purpose: Prompts the user to select a promoted pawn choice (Q, N, R, B) and sets the associated piece type value (4, 1, 3, 2 respectively). Will set the appropriate flag variable for indicating undo. Assumes move in M is a pawn promotion.

Description: Using the passed-in click position and move, promotionPrompt() sets a display prompt for promotion options based on the destination row of the move. If a promotion position hasn't been selected (promotion_pos is 254), it waits for further input. If the selected position is not a valid promotion option, it cancels the promotion, removes the prompt, and returns to the idle state. Otherwise, it sets the promotion selection based on the cursor position and prepares for promotion.

void pushCastleLog(CASTLE_RIGHTS* right, CASTLE_RIGHTS* list, uint8_t* end_idx);

Parameters: `CASTLE_RIGHTS* right, CASTLE_RIGHTS* list, uint8_t* end_idx`
Returns: `N/A`

Purpose: Adds (pushes) a castle rights log to an array of castle log.

Description: Using the current end index of the passed-in list, pushCastleLog() appends the passed-in castle right to the end of the list and increments the end index.

CASTLE_RIGHTS popCastleLog(CASTLE_RIGHTS* list, uint8_t* end_idx);

Parameters: `CASTLE_RIGHTS* list, uint8_t* end_idx`
Returns: `Castle right popped from list`

Purpose: Removes (pops) a castle rights log from an array of castle log and returns the castle rights. Assumes list is non-empty.

Description: If the log is empty (end index is 0), it returns a default CASTLE_RIGHTS configuration where all castle rights are enabled (kingside and queenside for both white and black). If the log is not empty, it retrieves the last castle rights configuration from the log, decrements the end index, and returns the retrieved configuration.

void updateCheckStatus();

Parameters: `N/A`
Returns: `N/A`

Purpose: Based on the current game state in 'board', periodically updates both of the king check statuses.

Description: For the white king and black king, updateCheckStatus() calls squareUnderAttack() with the white/black king position passed in. If a king is under attack, checkHighlight() is called to enable the red border around a king in check.

void updateGameStatus();

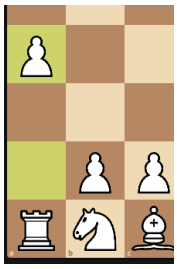
Parameters: `N/A`
Returns: `N/A`

Purpose: Based on the current game state in 'board', periodically updates the game status text whether it is white to move, black to move, checkmate, or draw.

Description: Depending on which color’s turn it is and if the game is in checkmate or stalemate, updateGameStatus() calls the different message functions to display the current status of the game.

Simulation Waveforms:

Below is a waveform showing a pawn moving from tile a2 to a4:



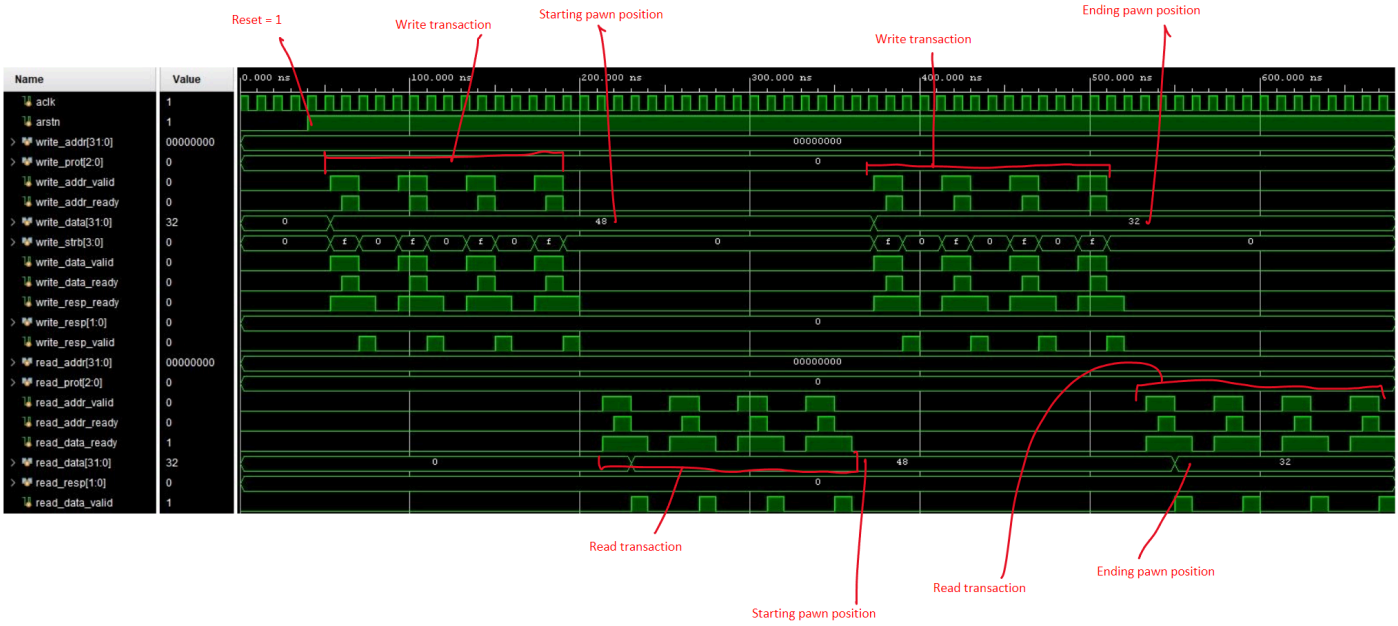


Image with better quality: [waveform]

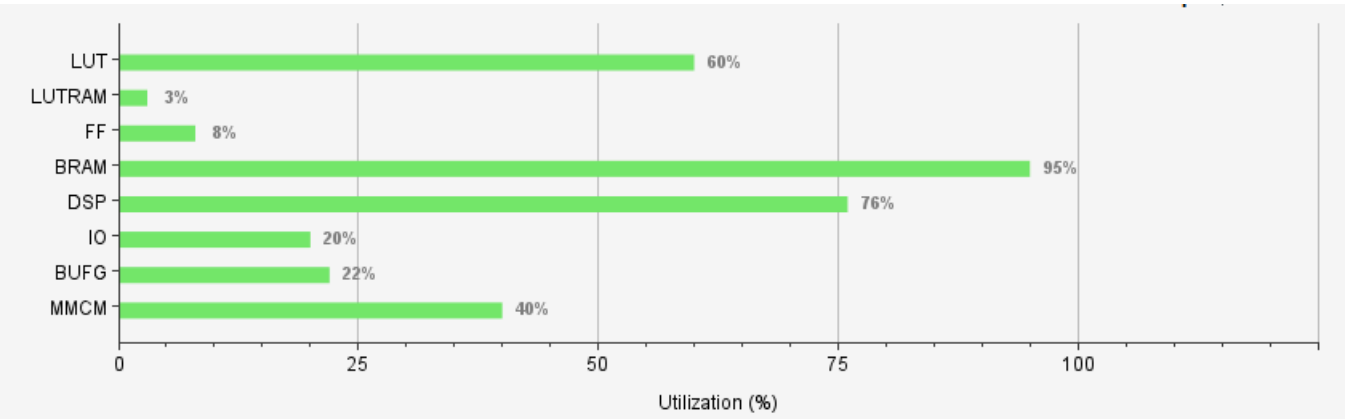
As seen in the waveform, the word (and byte) address that corresponds to the a-pawn is 0. To perform a piece movement, an AXI write transaction place and to read back the position an AXI read happens.

Design Resources and Statistics:

Below is the Design Resources and Statistics table for our final project. Compared to previous labs, all of the table values are generally higher since this project requires a lot of resources. The LUT number is higher because this project uses registers to store piece and other status information as well as distributed memory for the on-screen promotion prompt, which utilizes lookup tables. Most of our memory-mapped components like the piece registers and piece highlighting are stored on BRAM, which is why we have a high BRAM utilization. DSP is also high because we make multiple shifts and other operations in our code. The number of flip-flops is not as high compared to something like lab 7.1 because of our use of BRAM and not strictly registers through direct access. Additionally, the total power usage was higher than our previous labs because of the increased use of FPGA resources.

LUT	19581
DSP	91
Memory (BRAM)	71.5
Flip-Flop	4908
Latches	0
Frequency	0.07813 GHz
Static Power	0.080 W
Dynamic Power	0.568 W
Total Power	0.649 W

Design Statistics Table



Visualization of utilization

Conclusion:

In conclusion, our final project successfully achieved its aim of implementing Chess on an FPGA using a System-on-chip (SoC) setup and hardware-software co-design methodology. We utilized SystemVerilog to create graphics in hardware while managing game logic through C programming. Integration of a mouse allowed intuitive piece selection and movement, while the addition of an artificial opponent expanded gameplay possibilities beyond human players. Incorporating features like undo functionality and game status messages enhanced the user experience.

Chess, a classic board game, presented unique challenges in adapting to FPGA architecture. Our design leveraged memory-mapped HDMI controllers and SPI protocols, demonstrating proficiency in interfacing with peripherals and handling HID devices like mice. Through this project, we deepened our understanding of SoC principles and honed skills in software-peripheral interaction. While our AI opponent currently makes random legal moves, this project lays a foundation for future exploration into more sophisticated algorithms. Overall, this endeavor enriched our knowledge of FPGA development and system design.

Throughout the implementation and design process of our final project, we encountered multiple bugs that required solving. One visual bug was the dimensions of the chess board being the same as the dimensions of the screen, which caused the board to stretch slightly. This became an issue when we added the square pieces onto the stretched rectangular-shaped tiles, which left no space above and below the piece in the square but left space on the left and right of the piece. To resolve this visual issue, we reduced the size of the chessboard to 480x480 pixels while keeping the screen size 640x480. Another visual bug was that the pieces were originally showing on the screen as a fully black square with white accents where they were supposed to be instead of having a transparent background. This was because we tried using images with a transparent background, but this was not possible as we needed to filter the background out. We filtered the background out by making the background a hot pink color, which is not used anywhere else in the project. The next bug was occurring when deleting pieces (capturing). When attempting to capture a piece, the mirrored piece would delete from the board instead of the one we intended to capture. This was happening because of the endianness of unpacked arrays versus packed arrays as we messed up the ordering of the two during assignment. Another bug was an issue with the squareUnderAttack() function not working properly. The issue was that the “enemy” value was not being updated inside the function, which was causing the position being checked to not get properly analyzed. Another issue was with the getBKPosition() and getWKPosition() functions. Originally, these positions were being fetched directly from the piece controller but this caused problems as when our engine was playing out the moves it updates the board locally within the software and not the piece controller. Another bug we encountered was stalemate not working correctly. We ran into a case where the game should’ve been in stalemate, but the white queen was able to move diagonally from the right edge of the board to the left edge of the board. This was happening because in the setBishopMoves() function, the portion that was checking the bottom right movement had !leftedge instead of !rightedge, which led to an invalid move being passed through in this particular case.

References:

Image to COE tool - made by CA Ian Dailis, adapted for Vivado by CA Arnav
https://github.com/amsheth/Image_to_COE

HID Mouse Documentation
https://wiki.osdev.org/USB_Human_Interface_Devices#USB_mouse

Eddie Sharick - Creating Chess in Python
*https://www.youtube.com/playlist?list=PLBwF487qi8MGU81nDGaeNE1EnNEPYWKY_*