ECE 385

Spring 2024

Final Project

Gomoku Game

Ziheng Li (zihengl5)

Xin Yang (xiny9)

Section AL1

Prof. Zuofu Chen

Introduction

Our project is to implement the classic game of Gomoku (also known as Five in a Row) on a Spartan 7 FPGA with additional AI functionality. The game's mechanics mirrors the traditional Gomoku game, where the objective is to have five consecutive chess on a board, either horizontally, vertically, or diagonally. Our game combines keyboard input form Lab 6.2, register memory from Lab 7.1, and color mapper from Lab 7.2. The project leverage the hardware capabilities of the Spartan 7 FPGA and C code to create an interactive and responsive gaming experience.

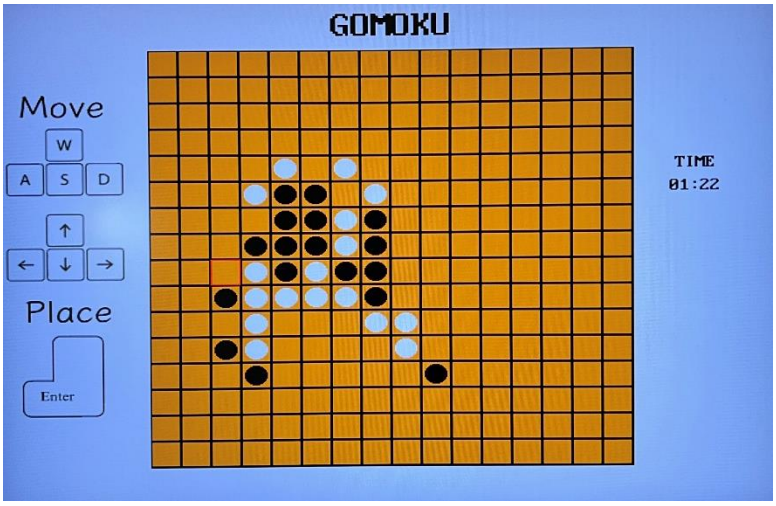

Fig.1 Game Interface

Descriptions for Final Project

i. Game Operation:

*Even though the game support mouse as input, it's not recommended as it have weird issue, see "current issue" for details.

The players can press either P (vs. Human) or I (vs. AI) to enter the game. During gameplay, the operator can navigate the highlighted position using the WASD keys or the arrow keys (↑←↓→) and place their chess piece by pressing the Enter key. Following the traditional rules of the game, the black player always makes the first move. When either the black or white player win, the winning combination of five chess pieces is prominently highlighted on the game board. After a brief celebration period of 10 seconds, the game automatically returns to the main page, allowing the timer to reset and providing players with the opportunity to select their desired game mode again.

ii. Description of the Entire Final Project System:

The overall structure of our Final Project in System Verilog side is pretty much the combination of Lab 6.2 (keyboard) and Lab 7.2 (register memory & VGA drawing). To support the display of the game board and enable smooth sprite drawing, we first optimized the hdmi-text-controller from Lab 7.2. This enhancement allowed us to draw more font and graph than the original font rom. Next, we encapsulated the entire IP block and merge it into the design from Lab 6.2. This integration enabled us to use the USB input from the keyboard. By combining these two essential components, we achieved a seamless fusion of user interaction and visual output.

As for game logic such including chess placement, win condition checks, and AI functionality, we implement it using C# programming language. This high-level approach allowed us to develop more sophisticated algorithms and game mechanics. In the following sections, I will dive deeper into the intricacies of how we achieved these functionalities.

iii. Description of the HDMI Text Mode Controller IP:

The functionality is the same from Lab 7.1. The HDMI Text Mode Controller IP is the core component responsible for generating the graphics output to the HDMI interface. It interfaces with the AXI4-Lite bus to communicate with the MicroBlaze processor. This IP helps manage the registers to store game board data. It also implements the color mapper which draws the correct game board to the screen based on the contents read from VRAM (registers). Then it outputs the signal for HDMI video to draw corresponding graph.

iv. Description of MAX3421E:

Just like in Lab 6.2, we also integrate a USB controller (MAX3421E) communicated via SPI protocol to read user keyboard input.

v. Transaction between System Verilog and C#

The communication between MicroBlaze and C# is achieved by using registers. The game board is 16*16 in size. Similar to Lab 6.1, a 32 bit register could stores 4 grid's information. So in total, we need 16 * 16 / 4 = 64 registers to stores.

vi. Description of Game Logic in C#:

To efficiently manage the game state, we employ a 1D array with a length of 16 × 17. Despite the game board being 16 × 16 in size, we utilize the additional 16 indices to store essential game information, such as game status. This approach allows us to keep all relevant data in a single, easily accessible location. All other game functionalities such as check wining, move transactions, and AI functionality were implemented in C#. I will introduce it later on.

Implementation Details

--- C#

i. Game Board Storing:

In the game of Gomoku, the board consists of three fundamental types of grid cells: empty, occupied by a white piece, and occupied by a black piece. However, we need to develop a mechanism to indicate cursor movement, or in other words, which grid are we selecting. This adds three additional grid types: highlighted empty, highlighted white, and highlighted black, as shown below. This enhancement not only improves the visual experience but also aids in better tracking of player movements and decision-making.

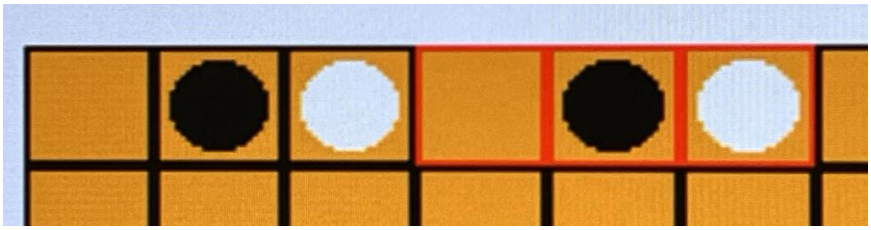

Fig.2 Empty, black, white, highlighted empty, highlighted black, highlighted empty(From left to right)

We use six different numbers to represent these identical grids.

0 --- Empty	1 --- Black	2 --- White
3 --- Highlighted empty	4 --- Highlighted black	5 --- Highlighted white

Hence, inside the 1D array, every grids were represented by one of these six numbers. And color mapper will draw corresponding images to the screen.

ii. Human vs Human Logic:

game_keyboard (uint8_t keycode) is a key component of the chess game implementation. It takes a single parameter keycode, which represents the code of the key pressed by the user. The function maintains the game state using static variables **player** to represent current player, where 1 corresponds to black and 2 corresponds to white, **row** and **col** represent the current position of the cursor on the game board, **prevRow** and **prevCol** store the previous position of the cursor.

The function starts by highlighting the default position on the game board to indicate the cursor position. Based on the keycode received, the function updates the current position of the cursor. If the Enter key is pressed, the function places the chess piece of the current player at cursor location. After placing the chess piece, the function calls the checkWin function to determine if the current move results in a win. If a win is detected, additional actions are performed, such as set the end_flag to 1, and waiting for a certain duration to restart. If the game is not over, the function will switches the current player. Finally, the function removes the highlighting from the previous position and then highlights the new position of the cursor.

iii. Check Win logic:

int checkWin(int row, int col, int player) is responsible for checking if a winning condition is met. It takes three parameters: row and col representing the current position of the placed chess piece, and player representing the current player (1 for black, 2 for white). The function check for four possible directions, horizontal, vertical, diagonal (top-left to bottom-right), and diagonal (top-right to bottom-left). If the contiguous chess count is 5, then we highlight the wining 5 chess and returns 1 to indicate a win. If no winning sequence is found after checking all four directions, the function returns 0 to indicate that there is no winner yet.

iv. Human vs. AI logic:

void game_ai(uint8_t keycode) it handles the game logic for a Human vs AI mode in the Gomoku. It is similar to the game_keyboard function, but after the human player (black) places a chess piece, the AI player (white) automatically makes its move, by using the **ai_move** functions as described in next section.

v. AI moving logic:

int ai_move(int board[BOARD_SIZE][BOARD_SIZE]) is responsible for making the AI player's moves. It takes the current game board as input and returns the best move for the AI player.

The function uses a two-step approach. For the first three rounds (controlled by the count variable), the AI player tries to place white chess pieces close to the black chess pieces placed by the human player. This is because current chess board has too little information to evaluate.

After the first three rounds, it iterates over each empty position on the board and places a white chess piece at that position. If the hypothetical move leads to a win for the AI player, it immediately returns that move and clears the board, resets the count and end_flag, otherwise, it removes the hypothetical chess piece and continues evaluating other positions using the minimax function to get a score for that move.

The function keeps track of the best move and its corresponding score. After evaluating all possible moves, it returns the best move found, which is encoded as row * 16 + col.

vi. Minimax Algorithm and Alpha-Beta Pruning:

*Brief introduction for Minimax algorithm and Alpha-Beta pruning Here

int minimax(int board[BOARD_SIZE][BOARD_SIZE], int depth, int alpha, int beta, int maximizingPlayer) This implements the Minimax algorithm with Alpha-Beta pruning for Gomoku. The Minimax algorithm is a decision-making algorithm commonly used in two-player games, such as chess. It aims to find the best move for the current player by recursively evaluating the game tree and assuming that both players play rationally. Note that as depth is the maximum depth of the game tree to explore, as depth gets larger, AI gets smarter. It also use Alpha-Beta pruning to optimize the algorithm by reducing the number of nodes evaluated in the game tree. It prunes branches that are guaranteed to be worse than the current best move.

vii. Evaluation function:

int evaluate(int board[BOARD_SIZE][BOARD_SIZE]) is another important aspect in Minimax algorithm. This evaluation function is used in conjunction with the minimax algorithm to determine the best move for the AI player. The function takes

the current game board as input and returns a score indicating the utility/rewards of the board position for the AI player (white).

The evaluation function considers six different patterns and assigns different weights to each pattern based on their importance and whether they favor the white or black player. The patterns are:

1. O O O --- open three
2. X O O O / O O O X --- closed three
3. X O O O X --- fully closed three

4. O O O O / O O O --- open four
5. X O O O O / O O O O X --- closed four
6. X O O O O X --- fully closed four

Note that we are not able to detect open four/three patterns with a gap, for exmaple: O O O O. However, with minimax algorithm, if black player is rational, he will place the chess inside the gap to maximize the rewards. And white player (AI) will take corresponding actions.

We also need to consider the weights for those six situations. I assign different weights to those six situations, since AI are playing white chess, anything benefits for white should be +, and anything benefits black is -. Also notice that AI plays after human plays, so black chess have higher priority than white chess. For example, if black chess has a closed four, it clearly has higher priority than an open/closed four for white. Based on that, the following are the priorities.

No.1: Open four (black) Highest
No.2: Closed four (black)
No.3: Open four (white)
No.4: Closed four (white)

No.5: Open three (black)
No.6: Open three (white)
No.7: Closed three (black)
No.8: Closed three (white) Lowest

In addition, fully closed three/four have no threats, in fact, it will highly likely to be a bad move, since it cannot form any useful patterns. So I assign negative weights to prevent we trying to form closed three/four patterns.

The function then will iterates over the game board in four directions: rows, columns, diagonals (top-left to bottom-right), and diagonals (top-right to bottom-left). For each direction, it checks for the presence of the six patterns for both white and black pieces. After iterating over all the patterns in all directions, the function returns the final score, which represents the favorability of the board position for the AI player (white). A higher positive score indicates a more favorable position for white, while a higher negative score indicates a more favorable position for black.

Implementation Details
--- System Verilog

i. Game Process:

In our game process, as outlined in the SV files, we begin by assigning the status_reg to mem_reg[65], which serves as the storage for the game's status. It could be 0, 1, or 2, while 0 represent the game is not started, 1 represent we are in Human vs. Human mode, and 2 represents Human vs. AI mode. When the status_reg is 0, the CM_Control module selects the RGB output from the color_mapper_menu module, meaning the current screen drawing is the menu. Once the status_reg is changed to a non-zero value, indicating that the game is in progress, the CM_Control switches to outputting the RGB values from the color_mapper module, which is the game board.

ii. Timer:

Within the game frame, we have incorporated a timer which displays the time starts from entering the game. To ensure accurate timekeeping, a 1Hz signal is generated based on the existing 125MHz clock signal. When the game is at the menu page, the timer displays "00:00," indicating that no time has passed. Once the user initiates the game by pressing either “I” or “P”, the timer starts to count the ongoing duration of the chess match.

iii. color_mapper_module:

When user enter the game, we display the menu as two frame alternating, as shown below.

Fig.3 Frame 1(left), Frame2 (right)

The frame data is stored in text files and loaded into the frameRAM.sv module during runtime. The logic is driven by a counter, which increments on every positive edge of a 25MHz clock signal. When the counter reaches 24,999,999, the frames are swapped, effectively transitioning between the two frames every second.

As for the fancier text/graph drawing, we utilize sprite drawing technique. To begin with, we generate the overall color palette for each frame. Additionally, we modified the width of the data in the HDMI encoder from 4-bit to 8-bit, enabling a broader range of colors for enhanced visual appeal. To efficiently convert the PNG files into usable data, we leveraged the provided Python helper tools. These tools allowed us to transform each PNG file into an array stored in a TXT file. The resulting array consists of numbers representing the colors in the palette for each pixel, arranged in a sequence corresponding to the image's width and height. We then implemented a new module to read and store these frame arrays, ensuring smooth access to the visual data during gameplay.

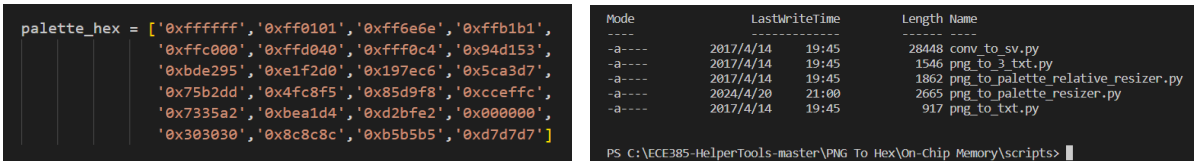

Fig.4 8-bit color palette (left), Sprite drawing helper (right)

iv. color_mapper:

The primary technique employed for displaying content on the screen is basically the same as in Lab 7. First, to highlight individual block fields during gameplay, we took a different approach instead of drawing the entire board as a single frame. We created each block field separately using the font_rom module. The black line surrounding each block field is defined by the outermost pixel. The background color is set to brown (0XF0A000), while the color of the chess pieces (black or white) is determined by the corresponding mem_reg value (16*16 game board). A value of 0 indicates an empty field, 1 represents a black chess piece, 2 signifies a white chess piece, and values 3-5 are used for highlighted versions of the empty field, black piece, and white piece, respectively, as stated in previous section. If the current drawing block field is highlighted version, then the outmost pixels is red instead of black.

To enhance the overall gaming experience, we have incorporated additional visual elements alongside the game board. On the right side of the screen, we have implemented a timer that keeps track of the time during gameplay. This timer functionality is achieved by employing the same technique used in Lab 7.2, utilizing the same font ROM and drawing idea.

Furthermore, we have added an instruction sprite on the left side of the screen, which provides helpful instructions to guide players through the game, as shown below. The displaying logic for this sprite is implemented in a manner similar to the color_mapper_menu module.

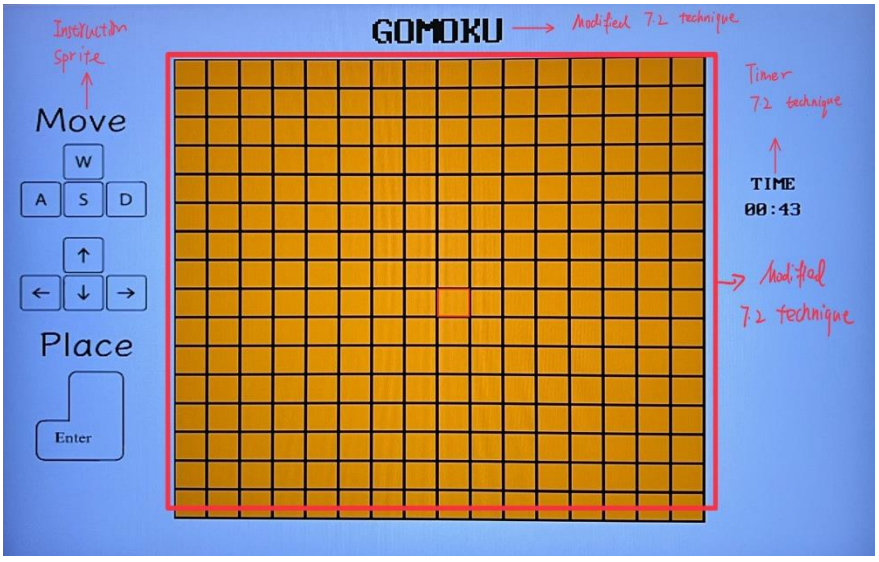

Fig.5 Game board

Block Diagram

High Level Design

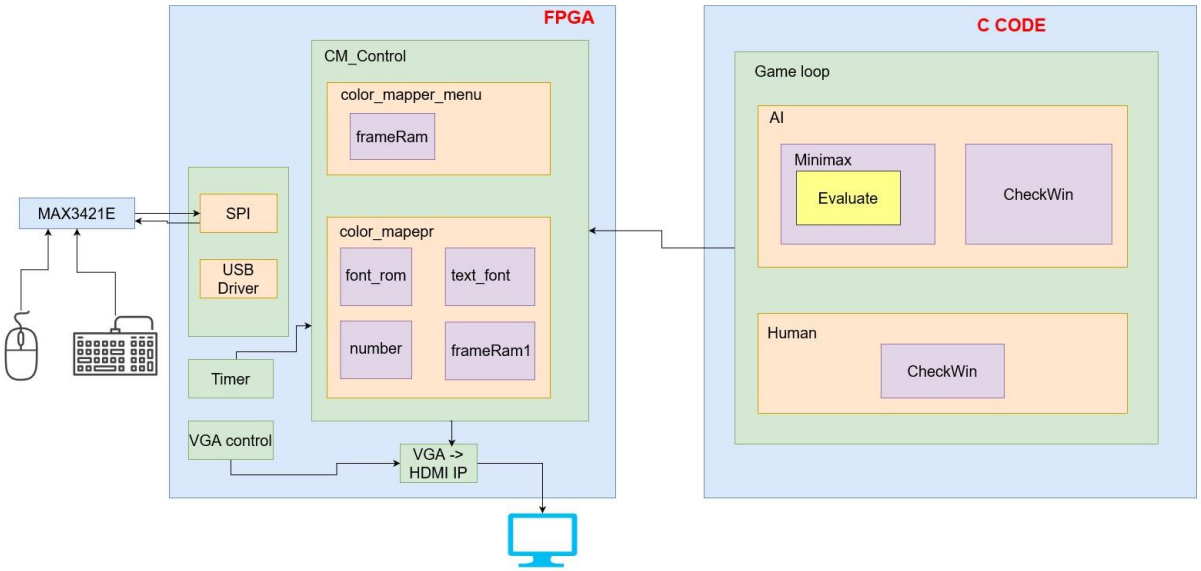

Fig.6 High lever design for Gomoku game

Top level

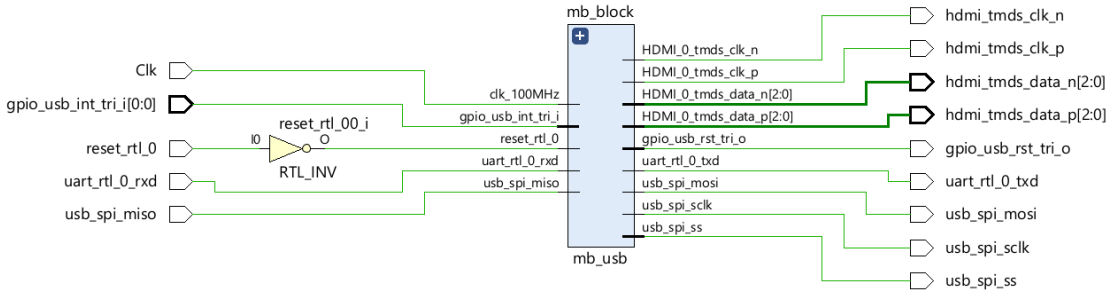

Fig.7 Overall view of top level

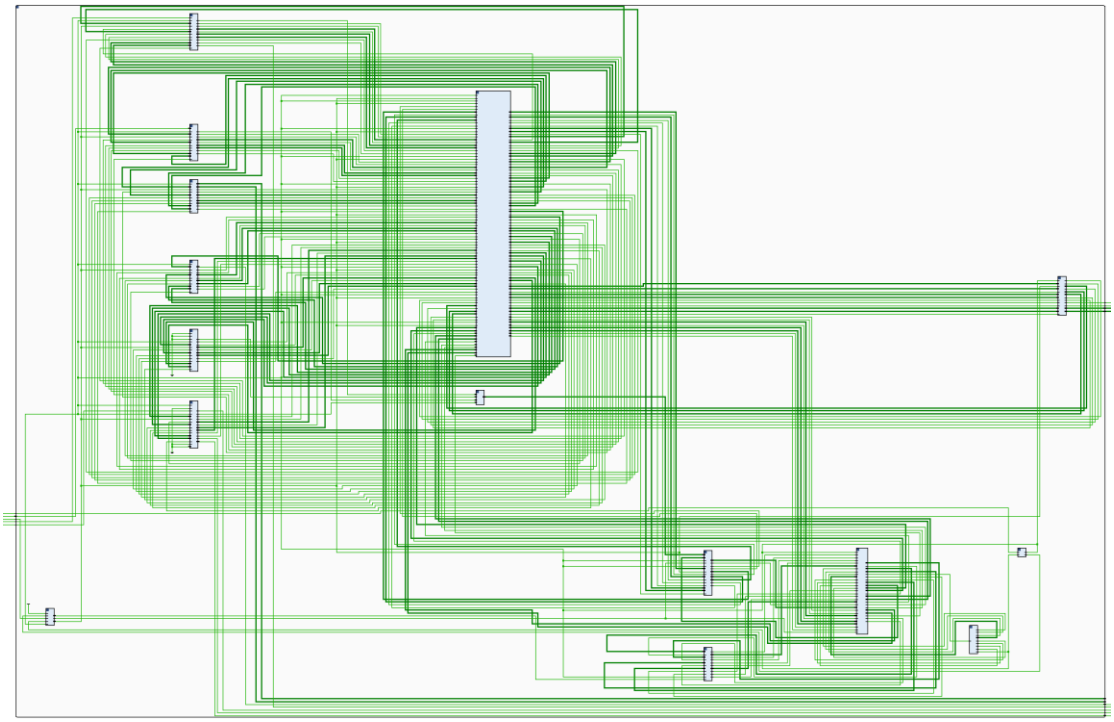

Fig.8 mb_block block diagram

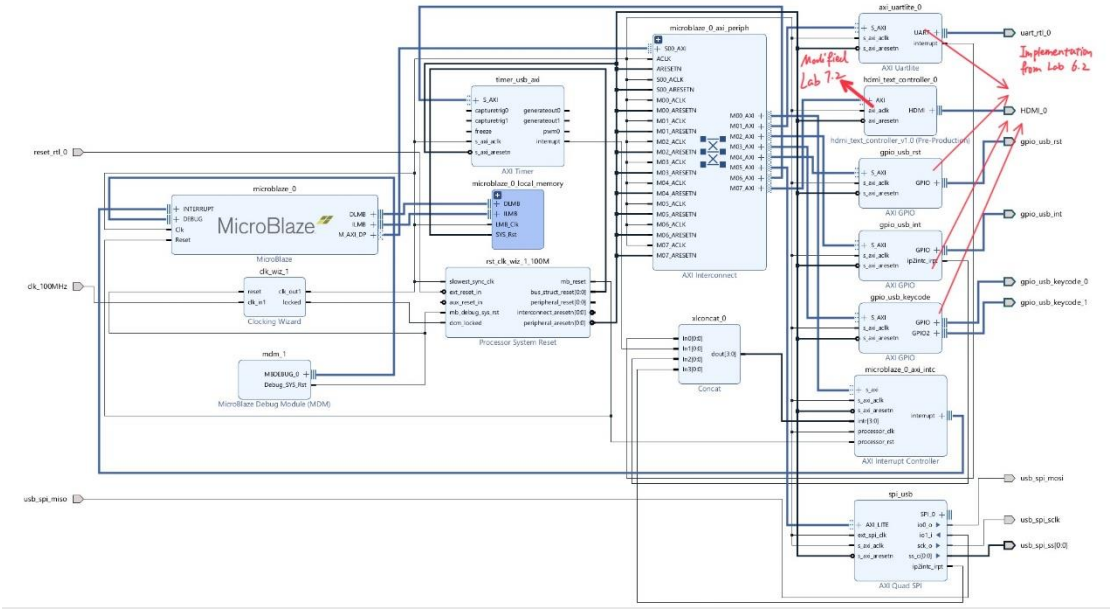

Fig.9 High level diagram

Internal IP

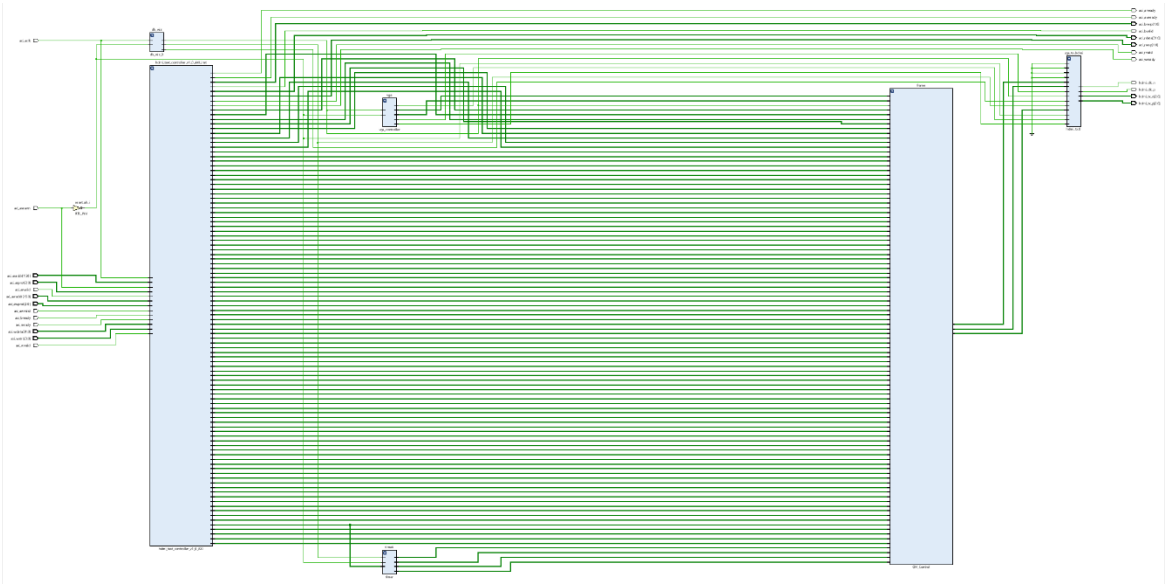

Fig.10 hdmi_to_text block diagram

Color Mapper

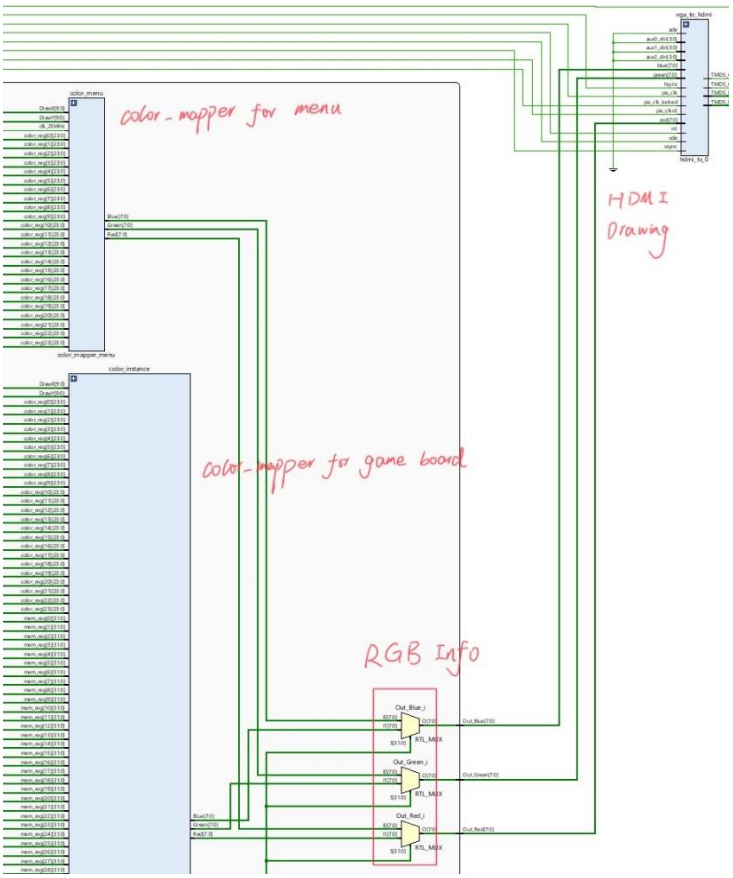

Fig.11 Color mapper block diagram

Module descriptions

Module: mb_usb_hdmi_top.sv
Inputs: Clk, reset_rtl_0, uart_rtl_0_rxd, [0:0] gpio_usb_int_tri_i,usb_spi_miso
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
Description: This module instantiates the test_block module, which is responsible for implementing the MicroBlaze system, UART, USB SPI and HDMI functionalities.
Purpose: The purpose of this module is to provide a top-level interface for the USB and HDMI functionalities implemented in the test_block module.

Module: test_block.v
Inputs: clk_100MHz, [0:0]gpio_usb_int_tri_i, reset_rtl_0, uart_rtl_0_rxd, usb_spi_miso,
Outputs: uart_rtl_0_txd, HDMI_0_tmds_clk_n, HDMI_0_tmds_clk_p, [2:0]HDMI_0_tmds_data_n, [2:0]HDMI_0_tmds_data_p, [31:0]gpio_usb_keycode_0_tri_o, [31:0]gpio_usb_keycode_1_tri_o, [0:0]gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, [0:0]usb_spi_ss
Description: This module serves as the top-level block design for the MicroBlaze system. It instantiates and connects the necessary components to provide the HDMI, USB and UART functionalities.
Purpose: The purpose of this module is to integrate the MicroBlaze processor, HDMI, USB and UART components into a complete system-on-chip.

Module: hdmi_text_controller_v1_0.sv
Inputs: axi_aclk, axi_aresetn, axi_awaddr[C_AXI_ADDR_WIDTH-1:0], axi_awprot[2:0], axi_awvalid, axi_wdata[C_AXI_DATA_WIDTH-1:0], axi_wstrb[C_AXI_DATA_WIDTH/8-1:0], axi_wvalid, axi_bready, axi_araddr[C_AXI_ADDR_WIDTH-1:0], axi_arprot[2:0], axi_arvalid, axi_rready
Outputs: axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_arready, axi_rdata[C_AXI_DATA_WIDTH-1:0], axi_rresp[1:0], axi_rvalid, hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0]
Description: This module is the top-level design for an HDMI text controller IP block. It has 6 submodules that implement an AXI4 Slave interface for configuration and control, generates the necessary clocks, handles VGA synchronization, sets up a timer

to generate game time, includes a VGA to HDMI module, and initiates a control module of color mapper for different display frames.
Purpose: The overall purpose of this module is to provide a HDMI text controller IP that can be easily integrated into a larger system-on-chip (SoC) design.

Module: hdmi_text_controller_v1_0_AXI.sv
Inputs: S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_AWPROT[2:0], S_AXI_AWVALID, S_AXI_WDATA[C_S_AXI_DATA_WIDTH-1:0], S_AXI_WSTRB[C_S_AXI_DATA_WIDTH/8-1:0], S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_ARPROT[2:0], S_AXI_ARVALID, S_AXI_RREADY
Outputs: S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP[1:0], S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA[C_S_AXI_DATA_WIDTH-1:0], S_AXI_RRESP[1:0], S_AXI_RVALID, color_reg[23:0][24], mem_reg[31:0][80]
Description: This module is the AXI4-Lite interface for the HDMI text controller IP. It implements the necessary logic to handle the AXI4-Lite read and write transactions, including address decoding, data transfer, and response signaling. The module also provides access to an internal memory registers that can be used to store the board data to display on the screen, and also the status of game. Additionally, it defines a palette which contains 24 colors in 24-bits per register.
Purpose: This module provides a standard AXI4-Lite interface for setting and operating the HDMI text controller IP. It wrapped the AXI4-Lite logic in a separate module which handles communication with the host CPU. It also provide RGB information for the frame buffer.

Module: clk_wiz_0.v
Inputs: reset, clk_in1
Outputs: clk_out1, clk_out2, locked
Description: This module is used to create a pixel clock (25MHz) and a 5x TMDS clock (125MHz).
Purpose: The purpose of this module is to provide 2 different clock signals.

Module: hdmi_tx_0.v
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, red, green, blue, hsync, vsync, vde, aux0_din, aux1_din, aux2_din, ade
Outputs: TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P[2:0], TMDS_DATA_N[2:0]
Description: The hdmi_tx_0 module is responsible for converting the input pixel data, synchronization signals, and auxiliary data into TMDS-encoded signals that can be transmitted over an HDMI link.
Purpose: The purpose of this module is to provide a ready-to-use HDMI transmitter solution that can be easily integrated into a larger FPGA-based design.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, frame, sync, [9:0] drawX, [9:0] drawY
Description: This module implements a VGA controller that generates the signals with proper frequency for displaying graphics on a VGA monitor. It generates the horizontal sync (hs) and vertical sync (vs) signals. It outputs the coordinates of the current pixel being drawn (drawX and drawY). Additionally, the module includes a frame signal that pulses once per frame in 60Hz, which can be used for the HDMI module to refresh the frames.
Purpose: The purpose of this module is to generate the timing signals and coordinate information required for displaying on a VGA monitor.

Module: timer.sv
Inputs: reset, clk, status_reg[31:0]
Outputs: s10[3:0], s1[3:0], m1[3:0], m10[3:0]
Description: This module implements a timer that generates the time values for seconds and minutes. It is driven by a 125 MHz clock and counts down to generate a 1 Hz signal for updating the *seconds* counter. The seconds counter increments every second, and the minutes counter is updated based on the seconds value. The module outputs the individual digit values for tens and ones places of seconds and minutes. The timer functionality is controlled by the *status_reg* input, where a non-zero value enables the timer, and zero resets it.
Purpose: The purpose of this module is to generate and display the current time in seconds and minutes, with the ability to reset or pause the timer(when the game is at the menu frame).

Module: CM_Control.sv
Inputs: DrawX[9:0], DrawY[9:0], s10[3:0], s1[3:0], m1[3:0], m10[3:0], clk_25MHz, color_reg[24][23:0], mem_reg[80][31:0]
Outputs: Out_Red[7:0], Out_Green[7:0], Out_Blue[7:0]
Description: This module acts as a control unit for managing the color output based on the status register value stored in mem_reg[65]. It instantiates two separate color_mapper modules: one for the main menu (color_mapper_menu) and another for the game (color_mapper). The main menu color mapper takes the DrawX, DrawY, clk_25MHz, and color_reg inputs and generates the menu_Red, menu_Green, and menu_Blue outputs. The game color mapper takes additional inputs of s10, s1, m1, m10, and mem_reg, and generates the game_Red, game_Green, and game_Blue outputs.
Based on the value of the status_reg_temp (extracted from mem_reg[65]), the module selects either the main menu color outputs or the game color outputs and assigns them to the Out_Red, Out_Green, and Out_Blue outputs.
Purpose: The purpose of this module is to control the color output displayed on the screen by selecting between the main menu colors and the game colors based on the value of the status register.

Module: color_mapper_menu.sv
Inputs: DrawX[9:0], DrawY[9:0], clk_25MHz, color_reg[24][23:0]
Outputs: Red[7:0], Green[7:0], Blue[7:0]
Description: This module generates the color output for the main menu header based on the current pixel coordinates (DrawX, DrawY) and the color register values (color_reg). It uses a frameRAM sub-module to store and retrieve the header data. The header data is read from the frameRAM based on the pixel coordinates, and the corresponding color values are assigned to the Red, Green, and Blue outputs using the color_reg values. The module also includes a 1Hz clock generator to alternate between two sets of color data stored in the frameRAM.
Purpose: The purpose of this module is to generate the color output for displaying the main menu frame on the screen and two frames will switch back and force every 1 second for a flashing effect. It also gives instruction on which keys to press to continue the game (P for players, I for AI).

Module: frameRAM.sv
Inputs: read_address[18:0]
Outputs: data_Out[4:0], data_Out1[4:0]
Description: This module implements a read-only memory (ROM) that stores two sets of frame data for the main menu header. The frame data is stored in two separate arrays,

mem and mem1, each with a width of 5 bits and a total of 166,500 addresses. During initialization, the module reads the frame data from two text files, *sprite_originalsmenu1.txt* and *sprite_originalsmenu2.txt*, and loads them into the mem and mem1 arrays, respectively. When a read address is provided as input, the module outputs the corresponding data values from the mem and mem1 arrays as data_Out and data_Out1, respectively.

Purpose: The purpose of this module is to provide a memory-mapped interface for accessing two sets of frame data for the main menu header. And the value stored in the register ranged from 0-23 to map the color palette.

Module: color_mapper.sv

Inputs: DrawX[9:0], DrawY[9:0], s10[3:0], s1[3:0], m1[3:0], m10[3:0], color_reg[24][23:0], mem_reg[80][31:0]

Outputs: Red[7:0], Green[7:0], Blue[7:0]

Description: This module generates the color output for the game screen based on the current pixel coordinates (DrawX, DrawY), the time values (s10, s1, m1, m10), the color register values (color_reg), and the memory register values (mem_reg). It consists of several sections:

1. Drawing the game board: The module calculates the column and row indices of the current pixel within the board area and retrieves the corresponding glyph data from the mem_reg. It then maps the glyph data to the appropriate color values (brown, black, or white) based on the font data stored in the font_rom module.
2. Drawing text: The module draws the "GOMOKU" text in a bigger size at the top of the screen and the "TIME" text near the timer area by retrieving the corresponding character data and mapping it to black or white color values based on the font data stored in the text_font module.
3. Drawing the timer: The module draws the timer value by extracting the individual digit values from the time inputs (s10, s1, m1, m10) and mapping them to black or white color values based on the font data stored in the text_font module.
4. Drawing the instruction panel: The module draws a side instruction panel on the left side of the screen(WASD, ↑←↓→, Enter) by retrieving the color data from a separate frameRAM1 module.

Purpose: The purpose of this module is to generate the color output for the game screen, including the game board, text elements, timer, and instruction panel, by mapping the various data sources (memory registers, font data, time values) to the appropriate color values based on the current pixel coordinates.

Module: font_rom.sv
Inputs: addr[4:0]
Outputs: data[23:0]
Description: This module defines a read-only memory (ROM) that stores the bitmap data for the chess in a block field to be displayed on the HDMI output. The module takes a 5-bit address (addr) as input and outputs a 23-bit data value (data) that corresponds to the bitmap.
Purpose: The purpose of this module is to provide the bitmap data that is used by the color_mapepr module.

Module: text_font.sv
Inputs: addr[10:0]
Outputs: data[7:0]
Description: This module defines a read-only memory (ROM) that stores the font bitmap data for the characters to be displayed on the HDMI output. The module takes an 11-bit address (addr) as input and outputs an 8-bit data value (data) that corresponds to the font bitmap for the requested character.
Purpose: The purpose of this module is to provide the font bitmap data that is used by the color_mapepr module.

Module: number.sv
Inputs: number[3:0]
Outputs: index[7:0]
Description: The module takes a 4-bit number (0-9) as input and outputs an 8-bit index value that corresponds to the font bitmap address of the corresponding number.
Purpose: The purpose of this module is to provide the index of text_font address for number 0-9.

Module: frameRAM1.sv
Inputs: read_address[18:0]
Outputs: data_Out[4:0]
Description: Same as frameRAM. During initialization, the module reads the frame data from a text file named sprite_originalswasd.txt and loads it into the mem array. When a read address is provided as input, the module outputs the corresponding data value from the mem array as data_Out.
Purpose: The purpose of this module is to provide a memory-mapped interface for accessing frame data for an instruction panel which can be used in color_mapper.

Design Resources and Statistics

Final Project	
LUT	15832
DSP	5
Memory (BRAM)	32
Flip-Flop	5249
Latches	0
Frequency (MHZ)	76.7
Static Power (W)	0.075
Dynamic Power (W)	0.731
Total Power (W)	0.806

Fig.12 Statistic for final project

Just like Lab 7.1, we use 65 32-bit registers to stores game board, which cause a high usage for flip-flops (for storage) and LUT (for selecting Mux). In addition, the color mapper logic is way more complicated than any of the lab we have done, hence the LUT utilization is also large.

Current Issue

C#

Our project design still has room for improvement in both System Verilog files and C# files. The AI implementation could be more concise. The main issue is that the evaluation function currently has a depth of 0, meaning the AI can only place moves based on the evaluation of the current board state, without considering potential future moves from the opponent. We attempted to increase the depth to 1, allowing the AI to look one move ahead. However, the chip's operating speed was insufficient to support this level of evaluation within a reasonable time frame. With a depth of 1, the AI would take more than two minutes to respond, which is impractical.

Additionally, at the beginning of the game, the game board does not have enough information to be evaluated, hence the AI might place the chess at random places.

To address these issues, we need to optimize the code, potentially by exploring more efficient evaluation function.

System Verilog

Regarding the color mapper, the design of the chessboard could be more concise. We did not utilize an on-chip-memory strategy, which would have been a more efficient approach.

As for potential improvements, we could consider adding an additional font to the font_rom to draw a simple square for highlighting purposes. This would allow us to streamline the code and make it more compact and readable.

In addition to that, we could also design more elegant way for the usage of timer, currently we are generate two separate 1HZ clock for both timer and frame-switching, which is inefficient.

Mouse

During the initial stages of development, we tries to operate the game using a mouse. However, after some tests, we discovered a significant drawback in the mouse's functionality. The mouse could only detect clicking information when its position changed, resulting in an awful gaming experience. Players might click on unintended grid while attempting to make their moves. To address this issue, we might need to modify the code within the mousePoll() function. However, in the interest of simplicity, we just recommend user play the game with keyboard control.

Conclusion

We successfully implemented the functionality outlined in the project proposal. Throughout the design and implementation phases, we applied techniques from Lab 6.2 and 7.2 for color mappers and register storage for frame information. However, we decided not to use block memory, as we handled both read and write operations to registers from the software side. The added AI functionality contribute to a more engaging and user-friendly experience.

By effectively utilizing the learned concepts from previous lab and adhering to modular design principles, we delivered a functional and visually appealing implementation. Looking ahead, incorporating additional features like audio would further elevate the user experience and overall engagement with the game.

The successful completion of this final project represents a significant achievement in our ECE385 journey. It demonstrates our ability to apply theoretical knowledge from all previous lab to practical scenarios. Overall speaking, it is a fun project and ECE385 is a meaningful class to take as it introduce System Verilog programming, this definitely shred lights on our future exploring of FPGAs.