ECE 385

Fall 2024

Final Project

Minesweeper via System Verilog

Ashley Saju (asaju2) and Daniel Cho (hc55)

Section XJ / Friday 12:45 PM

Bob Jin

Introduction

For the final project of ECE 385, we used System Verilog and the MicroBlaze processor to create a complete Minesweeper game implementation on the Xilinx Spartan-7 Urbana Board FPGA platform. Our design combines hardware-accelerated game logic in System Verilog with keyboard control/handling through the MicroBlaze processor, enabling sophisticated game mechanics and user interaction. The implementation features a dynamic display output via HDMI, a fully interactive 16x16 game board with 40 mines, and real-time user interaction through USB keyboard controls processed by the MicroBlaze CPU.

In System Verilog, the project's architecture mostly uses hardware-based design. The MicroBlaze CPU is solely used to process USB keyboard input via the MAX3421E USB host controller. The game board module, which controls the mine field, a state machine that governs the game logic, and an advanced display controller that renders the game interface via HDMI output are the main game components, all of which are implemented in System Verilog. Carefully crafted System Verilog modules were used to implement the main game logic in hardware, including initialization, cell revealing, mine production, neighbor counting, and win/loss situation detection. To control game operations like cell selection and cursor movement, the MicroBlaze's function is restricted to receiving and processing keyboard inputs. These inputs are then transmitted to the System Verilog modules over the AXI4-Lite interface.

While developing the game, there were particular difficulties with the technological implementation, mainly in the hardware area. We created effective System Verilog algorithms for neighbor counting, mine creation, cascade cell revelation logic, and game state management that are all tailored for FPGA implementation. Designing the cascading revelation system was a particularly challenging task that necessitated careful management of numerous memory accesses and sequential logic in order to preserve game consistency. MicroBlaze's sole software component processed input from USB keyboards and relayed that information to the hardware modules via the AXI4-Lite interface. Complex state machine architecture, effective memory management for game state storage, and synchronization between keyboard input handling and game logic are just a few of the sophisticated digital design ideas that are demonstrated in this application.

This implementation expands on methods covered in other labs, especially the text mode controller from Lab 7 and the HDMI display controller and USB keyboard interface from Lab 6. By adding new difficulties in the design of hardware-based game logic, which necessitated careful consideration of scheduling, resource use, and state management, the project built upon these foundations. Our solution, which uses MicroBlaze simply as a USB input processor, shows the potential of pure hardware implementation in System Verilog rather than depending on software for game logic. The end product is a thorough illustration of how digital design concepts can be used to the development of an interactive entertainment system, demonstrating the effective direct hardware implementation of intricate game logic.

Written Description of Circuit

MicroBlaze Processor

In order to process user input via a USB keyboard, the MicroBlaze soft processor is at the heart of the Minesweeper system. The MicroBlaze processes incoming keypress signals by interacting with the MAX3421E USB host controller. The processor decodes these inputs, including cell selection and cursor movement, and then relays them to the SystemVerilog hardware modules over the AXI4-Lite interface. This interface acts as a conduit for information between the processor-based software and the hardware-implemented game logic. The remainder of the system may concentrate on high-speed game logic execution and visual rendering by separating input handling to the MicroBlaze.

AXI Interconnect and Peripherals

The MicroBlaze CPU is connected to hardware modules and peripherals via the AXI connection, which serves as the main communication hub. GPIO modules for game state signaling and reset control, along with a timer module for synchronization signals, are essential peripherals. While the timer module controls delays for cascade cell revelations and other timing-dependent events, the GPIO peripherals guarantee correct control over user instructions and system resets. When combined, these add-ons facilitate effective communication between the FPGA hardware, the CPU, and the user.

USB Host Controller

Processing input from the attached USB keyboard is the sole responsibility of the MAX3421E USB host controller. Through an SPI link, it exchanges data with the FPGA and transmits keystroke events to the MicroBlaze CPU for decoding. The only function of the USB controller is input conversion, which makes sure that user commands are converted into data that the game logic modules can use.

Minesweeper Game Logic FSM

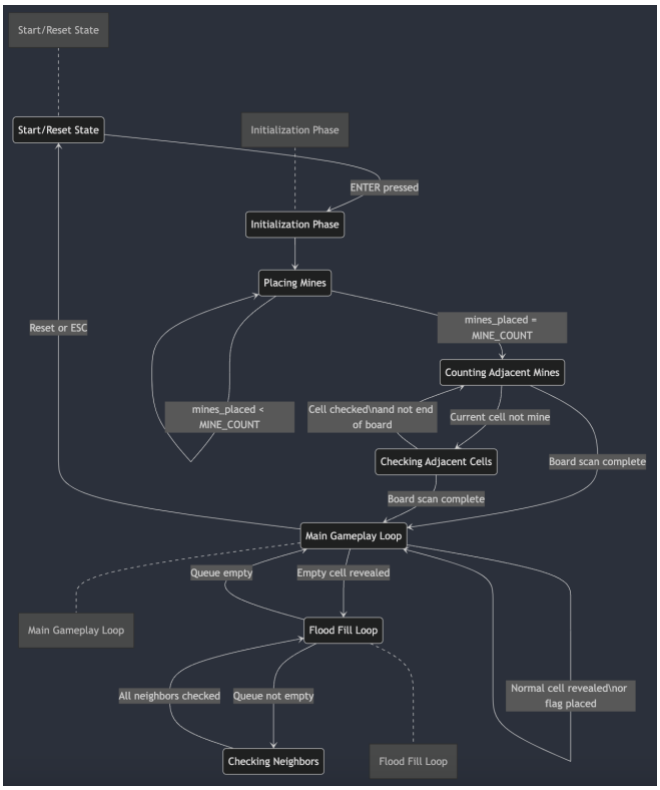

Figure 1. State Diagram of Minesweeper FSM

The state diagram of the FSM used in this project, which coordinates the flow of the Minesweeper game and guarantees smooth transitions between different stages of operation, is shown in Figure 1 above. The ***MENU_STATE (4'h0)***, which is the initial state when the game launches or following a reset, is where the FSM starts. In order to start the game, the system waits for the user to hit the ENTER key. A menu interface is shown while the user waits, giving them a clear place to start. The FSM changes to the ***INIT_CLEAR (4'h1)*** state when you hit ENTER.

The system is ready for mine installation during the brief transitional ***INIT_CLEAR*** state. The game board is prepared for initialization during this setup phase before moving on to the ***PLACE_MINES (4'h2)*** state. Using an LFSR-based randomization method, 40 mines are positioned at random throughout the game board in ***PLACE_MINES***. Two 16-bit LFSRs and a counter are used in the implementation to increase mine placement's randomness. Following the successful placement of all 40 mines, the FSM changes to ***COUNT_MINES (4'h3)***.

The FSM methodically examines every cell on the board when it is in the ***COUNT_MINES*** state. The FSM changes to the ***CHECK_ADJACENT (4'h4)*** state for cells without mines. It proceeds straight to the following cell in the scan if there are mines in that cell. The ***CHECK_ADJACENT*** state counts the number of nearby mines by evaluating each of a non-mine cell's eight adjacent cells. The board_state is then updated with the count using this information. The FSM changes to ***GAME_READY (4'h5)*** after processing the entire board.

The main stage of gameplay, during which the player engages with the board, is ***GAME_READY***. It manages important gaming features including putting flags, disclosing cells, and looking for win conditions. In order to control the flood-fill process, which exposes connected empty cells, the FSM switches to ***PROCESS_EMPTY_CELLS (4'h6)*** when an empty cell is disclosed.

The FSM processes a queue of empty cells that must be revealed when it is in the ***PROCESS_EMPTY_CELLS*** state. It stays in this state until the queue is empty, at which point it goes back to ***GAME_READY***. The FSM switches to the ***CHECK_NEIGHBORS (4'h7)*** state if the queue is not empty. The FSM looks at each of the current empty cell's eight neighbors in CHECK_NEIGHBORS, disclosing numbered cells next to the empty cells and adds any empty neighbors to the queue for additional processing. The FSM goes back to ***PROCESS_EMPTY_CELLS*** to resume the flood-fill procedure after verifying all neighbors.

For efficient operation management, the FSM depends on a number of critical control signals. These include ***empty_queue_front/back***, which controls the ***flood-fill queue***, and ***empty_queue_x/y***, which record the coordinates of cells to be revealed. Coordinate trackers like ***init_x/y, count_x/y***, and ***adj_x/y*** enable systematic board traversal, while ***neighbor_index*** keeps track of which of the eight neighbors is being processed at any given time. When combined, these states and signals allow the FSM to effectively manage the complex Minesweeper game logic, guaranteeing human engagement and real-time performance.

UI/UX

A sprite sheet with graphics like tiles, numbers, explosives, and flags was used to render the Minesweeper game's visual components on the FPGA. Using Python and the Pillow package, these graphical assets—which were saved in a PNG file—were transformed into a format that worked with the Vivado ROM IP. This procedure made it possible for the sprite sheet to be seamlessly integrated into the FPGA architecture, resulting in effective and superior graphics output.

```
def extract_sprite(img, x, y, width, height):
    """Extract a sprite from the sheet and convert to hex colors"""
    sprite = []
    for j in range(height):
        row = []
        for i in range(width):
            pixel = img.getpixel((x + i, y + j))
            # Convert RGB to 12-bit color (4 bits per channel)
            r = (pixel[0] >> 4) & 0xF
            g = (pixel[1] >> 4) & 0xF
            b = (pixel[2] >> 4) & 0xF
            color = (r << 8) | (g << 4) | b
            row.append(f"{color:03X}")
        sprite.extend(row)
    return sprite

def create_coe_file(sprites, filename="sprite_data.coe"):
    """Create a COE file from sprite data with better formatting"""
    with open(filename, "w") as f:
        f.write("memory_initialization_radix=16;\n")
        f.write("memory_initialization_vector=\n")

        # Join all colors with commas
        all_colors = []
        for sprite in sprites:
            all_colors.extend(sprite)

        # Format in rows of 16 values
        rows = []
        for i in range(0, len(all_colors), 16):
            row = all_colors[i:i+16]
            rows.append(",".join(row))

        # Join rows with comma and newline
        f.write(",\n".join(rows))
        f.write(";")
```

Figure 2. .PNG -> .COE converter script

Extracting individual sprites from the sprite sheet was the first stage in the procedure. A Python script was used to define the position (x, y) and size (width, height) of each sprite. Each sprite's pixel data was retrieved using the extract_sprite function, and each RGB channel was downsized to 4 bits in order to transform it into a 12-bit color representation. While maintaining adequate visual clarity for the Minesweeper game, this reduction reduced memory utilization. The resulting color data, which represented the color of each pixel in the sprite, was saved as hexadecimal numbers.

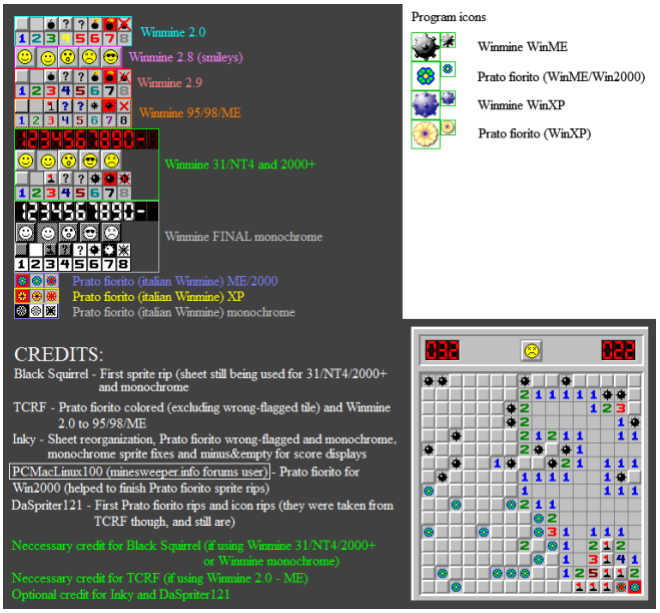

Figure 3. Original PNG spite sheet

Figure 4. PNG of title page created in Photoshop

After extracting the sprites, a .COE (Coefficient) file was created to format the data for use in Vivado's ROM IP core. The create_coe_file function compiled all the hexadecimal color data into a structured format, starting with the memory initialization radix (memory_initialization_radix = 16) followed by rows of 16 values for readability and compatibility. The COE file encapsulated the entire sprite sheet, ensuring that all graphical elements required for the game were ready for integration.

In the Vivado design environment, the COE file was used to configure a ROM IP core. The ROM was instantiated and initialized with the data from the COE file, which stored the color information for each sprite. This ROM was then connected to the HDMI display controller in the hardware design, allowing the graphical data to be accessed and rendered on the game's display. During gameplay, the ROM provided the required sprite data in real-time, ensuring smooth and accurate rendering of the game board and other graphical elements.

ROMs were created to separately house the tile pictures, score digits, top of the title screen ("Minesweeper" with mines underneath), and second line of text on title screen.

Visual Results

Figure 5 below shows the fully developed game functioning on a monitor. The tiles show correct values, and the game logic functions as expected. *Figure 6* shows the correct title screen when the system first starts or when ESC is pressed at any point into the game.

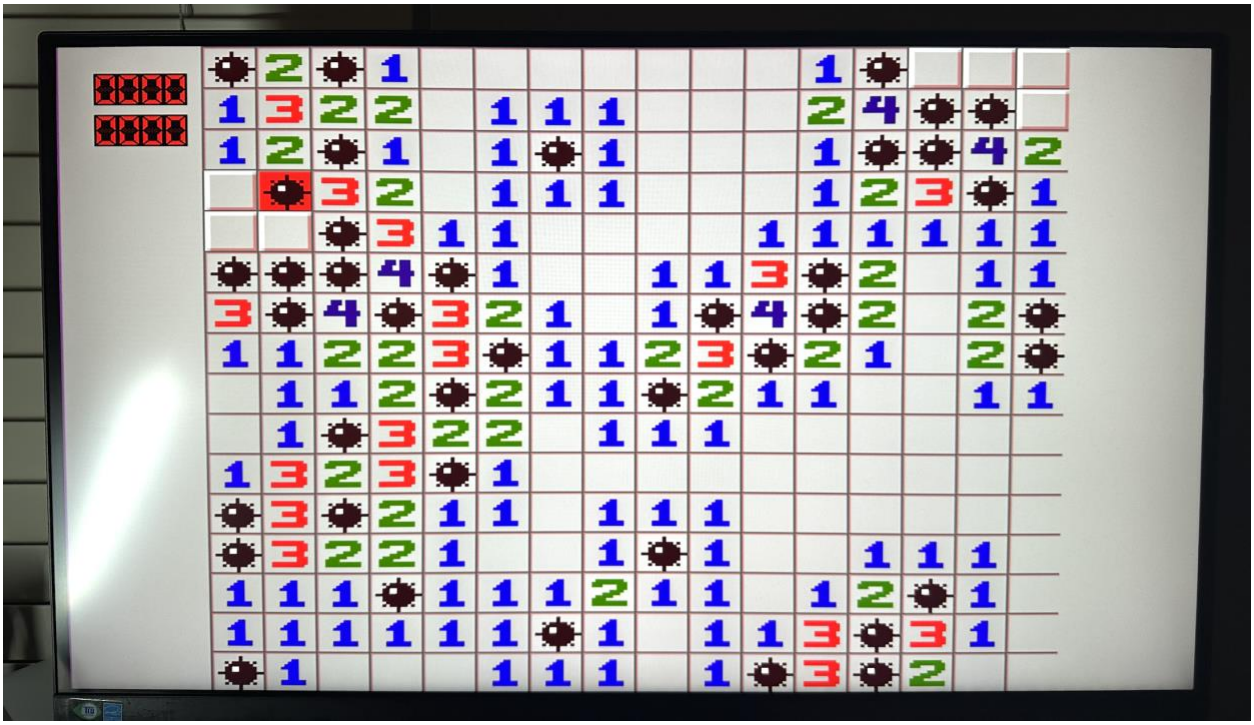

Figure 5. Picture of Minesweeper.

Figure 6. Image of the main menu of game.

Block Diagram (Same as Lab 6)

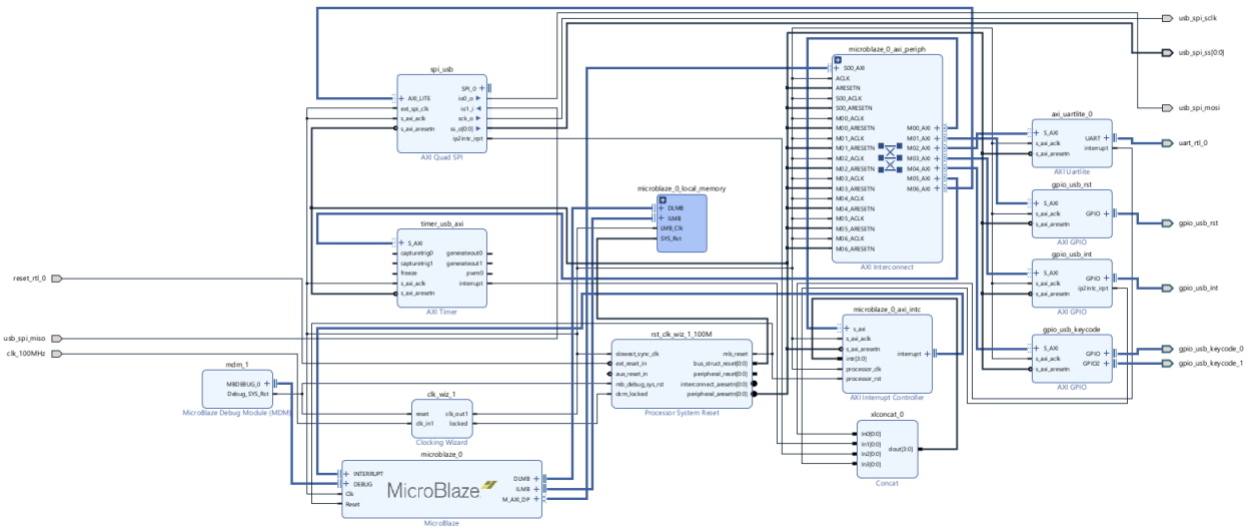

Figure 7. Block diagram of the MicroBlaze circuit

Module Descriptions

minesweeper.sv	
Input	• Reset: Active high reset signal to initialize/restart the game • frame_clk: Main clock signal for game state updates • keycode: 8-bit input from USB keyboard for user controls (WASD movement, Space to reveal, F to flag)
Output	• CursorX, CursorY: 4-bit signals indicating cursor position on game board • board_state[16][16]: 8-bit array storing actual game board state including mine positions • display_state[16][16]: 8-bit array storing visible board state (revealed/hidden/flagged cells) • game_over: Flag indicating game loss condition • victory: Flag indicating win condition • debug_state: 4-bit signal showing current FSM state • in_menu: Flag indicating if game is in menu state • current_score: 16-bit current game score • high_score: 16-bit highest achieved score
Description	Core game module implementing complete Minesweeper functionality. Manages the game board state, user input processing, game logic, scoring system, and implements an 8-state FSM for game control. Features include mine placement using LFSR-based randomization, flood-fill revelation of empty cells, score tracking, and game state management. Supports a 16x16 board with 40 mines.
Purpose	Serves as the main controller for the Minesweeper game, coordinating all game mechanics including board initialization, user interaction, score management, and win/loss conditions. Provides core game logic and state management required for complete Minesweeper gameplay.

Color_mapper.sv	
Input	• clk: Clock signal • DrawX, DrawY: 10-bit coordinates for current pixel • CursorX, CursorY: 4-bit cursor position • board_state[16][16]: 8-bit array of actual game board state • display_state[16][16]: 8-bit array of visible board state • game_over: Game loss condition flag • victory: Win condition flag • in_menu: Menu state flag • current_score, high_score: 16-bit score values
Output	• Red, Green, Blue: 4-bit color channels for HDMI output
Description	Complex display controller that manages all visual aspects of the Minesweeper game. Handles multiple display elements including • Game board rendering with 30x30 pixel cells • Menu interface with title and text • Score display system • Grid lines and cursor highlighting

	• Special effects for game over/victory states Uses multiple ROM interfaces (sprite_controller, title_rom, text_rom, score_controller) to generate appropriate graphics. Implements sophisticated pixel-level calculations for precise positioning and scaling of game elements.
Purpose	Serves as the main graphics pipeline, converting game state information into pixel-level color data for HDMI display. Coordinates all visual elements of the game, handling the transition between menu and gameplay states, and providing visual feedback for player actions and game conditions.

Score_controller.sv	
Input	• clk: Clock signal • sprite_id: 4-bit value selecting which digit (0-9) to display • pixel_x: 4-bit X-coordinate within digit sprite (0-12) • pixel_y: 5-bit Y-coordinate within digit sprite (0-22)
Output	• pixel_color: 12-bit color value for the current pixel
Description	Controller module for rendering numerical digits for score display. Manages sprite selection and pixel-level addressing for a digit display system where each digit is 13x23 pixels. Interfaces with score_rom to retrieve pixel data for each digit sprite. Includes bounds checking for valid pixel coordinates.
Purpose	Handles the rendering of numerical scores in the game interface by converting digit values into pixel-level color data. Acts as an interface between the game's scoring system and the visual display, ensuring proper digit sprite selection and pixel mapping.

Sprite_controller.sv	
Input	• clk: Clock signal • sprite_id: 4-bit value selecting which sprite to display (0-11) • pixel_x: 4-bit X-coordinate within sprite (0-15) • pixel_y: 4-bit Y-coordinate within sprite (0-15)
Output	• pixel_color: 12-bit color value for the current pixel
Description	Controller module for managing game sprites used in Minesweeper (hidden cells, numbered cells 1-8, flags, mines). Each sprite is 16x16 pixels, and the module can handle 12 different sprites. Uses ROM-based storage for sprite data with a calculated addressing scheme to locate specific pixel colors. The address calculation supports 12 sprites with 256 pixels each (16x16).
Purpose	Provides the core sprite rendering functionality for the game board, converting sprite selections into pixel-level color data. Manages all game piece visuals including hidden cells, revealed cells with numbers, flags, and mines. Essential for creating the visual representation of the game board.

IP Descriptions

Title_rom	
Input	• a: 16-bit address input (size [15:0]) for ROM access
Output	• spo: 12-bit data output (size [11:0]) providing color values
Description	Distributed memory generator IP core configured as a ROM for storing text graphics data.
Purpose	Stores and provides access to text graphics data used for menu text and game interface elements. Acts as a lookup table for converting text character positions into pixel-level color information for display. Specifically stores the upper part of the title page test ("Minesweeper" with mines underneath).

Text_rom	
Input	• a: 16-bit address input (size [15:0]) for ROM access
Output	• spo: 12-bit data output (size [11:0]) providing color values
Description	Distributed memory generator IP core configured as a ROM for storing text graphics data.
Purpose	Stores and provides access to text graphics data used for menu text and game interface elements. Acts as a lookup table for converting text character positions into pixel-level color information for display. Specifically stores the "Press ENTER to start" part of the tile page

sprite_rom	
Input	• a: 16-bit address input (size [15:0]) for ROM access
Output	• spo: 12-bit data output (size [11:0]) providing color values
Description	Distributed memory generator IP core configured as a ROM for storing text graphics data.
Purpose	Stores and provides access to text graphics data used for menu text and game interface elements. Acts as a lookup table for converting text character positions into pixel-level color information for display.

score_rom	
Input	• a: 16-bit address input (size [15:0]) for ROM access
Output	• spo: 12-bit data output (size [11:0]) providing color values
Description	Distributed memory generator IP core configured as a ROM for storing text graphics data.
Purpose	Stores and provides access to text graphics data used for menu text and game interface elements. Acts as a lookup table for converting text character positions into pixel-level color information for display.

Lab 6 System Verilog Module Descriptions

VGA_controller.sv	
Input	Logic clk, logic reset, logic [9:0] h_counter, logic [9:0] v_counter
Output	Logic h_sync, logic v_sync, logic video_on
Description	This module generates VGA timing signals, including horizontal sync, vertical sync, and video-on signals. The timing parameters are used to generate the necessary synchronization signals for VGA display. The module takes in horizontal and vertical counters to determine the appropriate states for the sync signals.
Purpose	The VGA_controller module provides the synchronization signals required for VGA display, enabling the display of graphical content on a VGA-compatible monitor.

mb_usb_hdmi_top.sv	
Input	Logic reset, logic clk, logic usb_rx, logic hdmi_in
Output	Logic usb_tx, logic hdmi_out
Description	This module acts as the top-level interface for the MicroBlaze USB and HDMI integration. It connects the MicroBlaze core, USB controller, and HDMI interface to create a complete system that manages USB communication and HDMI output. The module handles data transmission between the USB interface and the display, and manages synchronization between the HDMI and USB peripherals.
Purpose	The mb_usb_hdmi_top module serves as the main integration point for USB and HDMI functionalities in the SoC, enabling the MicroBlaze processor to communicate with USB peripherals and output graphical data via HDMI.

hex_driver.sv	
Input	Logic [15:0] hex_in, logic clk, logic reset
Output	Logic [6:0] hex_display
Description	This module takes a 16-bit input and converts it to a 7-segment display format. The input is used to drive hexadecimal values to the output, which controls a 7-segment display. It operates synchronously with the input clock and can be reset to a default state.
Purpose	The hex_driver module is designed to provide an easy interface for displaying hexadecimal values on a 7-segment display, making it useful for debugging and visual representation of internal states in the SoC.

Lab 6 IP Modules and Block Diagram Component Descriptions

mdm_1 (MicroBlaze Debug Module (MDM))	
Input	
Output	MBDEBUG_0, Debug_SYS_Rst

Description	This module provides a wide range of debugging capabilities for the MicroBlaze processor such as debugging line by line, setting breakpoints, accessing registers and memory, processor and system information, and connection to Xilinx's debug tools.
Purpose	Gives a wide range of debugging capabilities for the developer.

spi_usb (AXI Quad SPI)	
Input	AXI_LITE, ext_spi_clk, s_axi_aclk, s_axi_aresetn
Output	SPI_0, io0_o, io1_i, sck_o, ss_0[0:0], ip2intc_irpt
Description	The AXI Quad SPI is an IP block provided by Vivado which communicates to the MAX3421E USB host controller over SPI.
Purpose	Used by the MicroBlaze CPU such that it can communicate to the MAX3421E USB host controller over SPI.

timer_usb_axi (AXI Timer)	
Input	S_AXI, capturetrig0, capturetrig1, freeze, s_axi_aclk, a_axi_aresetn
Output	Generateout0, generateout1, pwm0, interrupt
Description	This module allows the MicroBlaze to keep track of when the polling period has passed. This is needed since the USB protocol has many timeouts, which require timekeeping in milliseconds like how some USB mouse might have a polling period of 10 ms.
Purpose	This timer module allows the MicroBlaze core to keep track of when the polling period has passed.

clk_wiz_1 (Clocking Wizard)	
Input	reset, clk_in1
Output	clk_out1, locked
Description	This module takes in a 100 MHz clock signal from the top level and outputs a 100 MHz clock signal for the MicroBlaze CPU.
Purpose	Generates 100 MHz clock signal for the MicroBlaze CPU.

microblaze_0 (MicroBlaze)	
Input	INTERRUPT, DEBUG, Clk, Reset
Output	DLMB, ILMB, M_AXI_DP
Description	MicroBlaze CPU that runs our C code which reads the keycode from a USB keyboard through SPI using the MAX3421E USB host controller.
Purpose	To run a C program that reads the USB keycodes.

microblaze_0_axi_intc (AXI Interrupt Controller)	
Input	s_axi, s_axi_aclk, s_axi_aresetn, intr[3:0], processor_clk, processor_rst
Output	interrupt
Description	This block facilitates the interrupts. It can handle multiple interrupts and manage them efficiently.
Purpose	The purpose of this module is to handle all the interrupts correctly.

microblaze_0_axi_periph (AXI Interconnect)	
Input	S00_AXI, ACLK, ARESETN, S00_ACLK, S00_ARESETN, M00_ACLK, M00_ARESETN, M01_ACLK, M01_ARESETN, M02_ACLK, M02_ARESETN, M03_ACLK, M03_ARESETN, M04_ACLK, M04_ARESETN, M05_ACLK, M05_ARESETN, M06_ACLK, M06_ARESETN, M07_ACLK, M07_ARESETN
Output	M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI, M07_AXI
Description	This is a module that facilitates communication between the MicroBlaze processor and various AXI peripherals.
Purpose	This block allows MicroBlaze to communicate to AXI peripherals.

microblaze_0_local_memory	
Input	DLMB, ILMB, LMB_Clk, SYS_Rst
Output	
Description	The primary purpose of the local memory is to provide fast access storage for the MicroBlaze processor. It stores the executable code that the MicroBlaze runs and the data it processes.
Purpose	This memory is essential for the performance and efficiency of the processor in executing applications.

xlconcat_0 (Concat)	
Input	In0[0:0], In1[0:0], In2[0:0], In3[0:0]
Output	dout[3:0]
Description	This module combines multiple signal lines into a single, wider signal line. In this case it is used to combine the interrupt signals into one.
Purpose	It combines each interrupt bit into a 4-bit signal which goes to microblaze_0_axi_intc (AXI Interrupt Controller).

rst_clk_wiz_1_100M (Processor System Reset)	
Input	slowest_sync_clk, ext_reset_in, aux_reset_in, mb_debug_sys_rst, dcm_locked
Output	mb_reset, bus_struct_reset[0:0], peripheral_reset[0:0], interconnect_aresetn[0:0], peripheral_aresetn[0:0]
Description	The Processor System Reset module is designed to manage the reset signals. It generates reset signals and manages input reset signals.
Purpose	This module manages multiple reset signals to ensure the system resets normally.

gpio_usb_int (AXI GPIO)	
Input	S_AXI, s_axi_aclk, s_axi_aresetn
Output	GPIO, ip2intc_irpt
Description	This block is used to handle the interrupt signal for the USB controller. Note that this is not the interrupt used for the USB keyboard since we poll data from the USB keyboard. This interrupt is used for the MAX3421E USB host chip.
Purpose	To have interrupt signals for the MAX3421E USB host chip.

gpio_usb_keycode (AXI GPIO)	
Input	S_AXI, s_axi_aclk, s_axi_aresetn
Output	GPIO, ip2intc_irpt
Description	This block is used to send the USB keycode data read from the MAX3421E USB host chip to the top level so that it can be used by the ball module to assign the direction of motion.
Purpose	To send the USB keycode data to the top level.

gpio_usb_rst (AXI GPIO)	
Input	S_AXI, s_axi_aclk, s_axi_aresetn
Output	GPIO, ip2intc_irpt
Description	The gpio_usb_rst is an AXI GPIO IP block module that is used to reset the MAX3421E USB host controller.
Purpose	AXI GPIO IP Block that is used to reset MAX3421E.

hdmi_tx_0 (HDMI/DVI Encoder)	
Input	ade, aux0_din[3:0], aux1_din[3:0], aux2_din[3:0], blue[3:0], green[3:0], hsync, pix_clk, pix_clk_locked, pix_clkx5, red[3:0], rst, vde, vsync
Output	TMDS_CLK_N, TMDS_CLK_P, TMDS_DATA_N[2:0], TMDS_DATA_P[2:0]
Description	This module converts the VGA signal generated by our modules into an HDMI signal.
Purpose	To convert the VGA signal to HDMI so that it can be displayed on a modern monitor.

clk_wiz_0 (Clocking Wizard)	
Input	reset, clk_in1
Output	clk_out1, clk_out2, locked
Description	This module generates two clock signals necessary for the VGA controller (pixel clock) and the HDMI TX block. It generates two clock signals: clk_out1 generating 25 MHz for the VGA controller and clk_out2 generating 125 MHz for both the VGA controller and HDMI TX. The locked signal goes to the HDMI TX block. The two clock signals are used to implement transition minimized differential signaling (TMDS) for the HDMI signal.
Purpose	Generates necessary clock signals for the HDMI TX block and the VGA controller.

Design Resources and Statistics

LUT	16695
DSP	**7**
Flip-Flop	8962
Memory	8
Latches	0
Frequency	115MHz
Static Power	0.076W
Dynamic Power	0.482W
Total Power	0.557W

Conclusion

Building Minesweeper on the Spartan-7 FPGA allowed us to explore the balance between hardware-accelerated logic and MicroBlaze driven software. By handling most of the game's core mechanics in System Verilog, like mine placement, cascading cell reveals, and state management, we pushed the limits of what hardware can do in real-time. At the same time, the MicroBlaze processor made keyboard input seamless, acting as the perfect bridge between user commands and game logic.

This wasn't without challenges. Cascading cell revelations, efficient memory usage, and ensuring synchronization between modules were tricky, but solving these problems taught us the importance of systematic debugging and creative thinking. Using HDMI for graphics and USB for input added complexity but also elevated the final product to feel polished and professional. Translating sprite data into FPGA-compatible ROMs with Python scripts brought the visuals to life, adding a layer of immersion that truly completed the project.

Ultimately, this project wasn't just about making a game, it was about understanding how to bring software and hardware together to create something tangible and interactive. It's satisfying to see everything come together on-screen, from a fully functional Minesweeper board to smooth user interaction and dynamic visuals. This experience has been a fantastic blend of learning, problem-solving, and creativity, leaving us excited to tackle even more ambitious hardware designs in the future.