ECE 385

Spring 2025

Final Experiment

Among Us Final Project

Janine Zhu and Sarayu Suresh

XJ, 12:00pm

Xuanbo Jin

Introduction

a. Summarize the basic functionality of the Microblaze processor running on the Spartan 7 FPGA.

The MicroBlaze processor on the Spartan-7 FPGA is a 32-bit RISC CPU implemented using Xilinx IP in Vivado. It serves as the central controller in a System-on-Chip (SoC) design, handling software-executed tasks such as user input/output, while more performance-critical operations are offloaded to custom hardware modules written in SystemVerilog. MicroBlaze interacts with peripherals through memory-mapped I/O using AXI-based components like GPIO for switches, LEDs, and keyboard inputs, UART for serial communication, and SPI for connecting to devices such as the MAX3421E USB controller. Programs for MicroBlaze are written in C using the Vitis SDK, allowing for a high-level development workflow. In the lab, the MicroBlaze is first used to implement a basic system that communicates with a USB keyboard via SPI and displays output on an HDMI monitor, demonstrating its integration with both software and hardware components in an embedded system.

b. Briefly summarize the operation of the overall design

In this project, the MicroBlaze processor is used to communicate with a USB keyboard through the MAX3421E USB host controller using the SPI protocol. The system continuously polls the keyboard for key inputs (W, A, S, D), which are used to control the direction of a moving ball displayed on an external HDMI monitor. Additional keycodes such as numbers and space allow for the user to interact with the map and complete tasks. The map is zoomed in on a small section that moves in all four directions while the character is animated in place in order to simulate the character walking around the full map. The design integrates SPI, timer, and GPIO peripherals in the block diagram, along with custom SystemVerilog modules for VGA signal generation, map movement, color mapping, and multiple FSMs for controlling game logic creating a custom and interactive game of Among Us in the ECE building.

Written Description and Diagrams of SLC-3

a. Written Description of all .sv modules

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This is a positive-edge triggered module that cycles through input in[4] to convert the values to hex values, which are then passed to the hex_grid and the FPGA.
Purpose: This module generates and outputs hex values from the circuit to the FPGA board.

Module: VGA_controller.sv
Inputs: pixel_clk, reset, hs, vs, active_nblank, sync,
Outputs: [9:0] drawX, drawY
Description: This module generates standard 640x480 VGA timing signals using a 25 MHz pixel clock. It outputs horizontal and vertical sync pulses (hs, vs) as well as the current pixel coordinates (drawX,

drawY). It also outputs an active_nblank signal indicating whether the current pixel lies within the visible display region. Internally, it uses horizontal and vertical counters to cycle through pixel positions and produce synchronization pulses according to VGA timing specs. The sync output is unused in this lab and held low.

Module: ball.sv
Inputs: Reset, frame_clk, keycode, vga_clk, DrawX, DrawY, blank
Outputs: MapX, MapY, hit
Description: This module processes keyboard inputs received from the MicroBlaze (via USB keycodes) to update the MapX and MapY coordinates. It also integrates with a boundary-checking module to prevent illegal movement.
Purpose: To control the position of a character sprite on the map. The module updates position in real time and outputs coordinates for use in rendering and collision detection.

Module: Animate.sv
Inputs: Reset, frame_clk, keycode
Outputs: MapX, MapY, MapS, frame
Description: This module implements an FSM to control sprite animation frames based on keyboard input. It cycles through walking and standing frames for both left and right movement.
Purpose: To animate a character sprite based on user input received as USB keycodes from the MicroBlaze. The FSM generates a frame signal indicating which sprite of teh character should be displayed
Module: task_num.sv
Inputs: Reset, frame_clk, keycode, MapX, MapY
Outputs: num, display
Description: Implements an FSM that activates when the sprite is near a specific task location. The module displays a sequence of tiles, prompts the user for a numeric input sequence (via keycodes), and determines success or failure based on correct input. The result is reflected in the display and num outputs, which control visual output on the screen.
Purpose: Creates an interactive numeric task triggered by sprite proximity. This module tracks input sequences, manages display states, and indicates win or lose conditions to the rendering system. It also gives signals to the num_color_mapper, which chooses rgb values based on these signals

Module: num_color_mapper.sv
Inputs: blank, vga_clk, DrawX, DrawY, printer_num, printer_display
Outputs: red, green, blue
Description: This module selects the appropriate RGB pixel color based signals from task_num. It highlights the current tile number during interaction, and rendering win and fail screens using separate ROMs and palettes.
Purpose: Displays the corresponding image to the signals from task_num.

Module: task_coffee.sv
Inputs: Reset, frame_clk, keycode, MapX, MapY, screen
Outputs: state
Description: This module implements an FSM that activates when the sprite is near the coffee task location. The module displays a cartoon espresso machine and prompts the user to hold the space key until a cup has been filled to the indicated fill line. It then determines the user's success or failure based on the position of the "liquid" in the filled cup, max_y_coffee. The result is reflected in the state output/coffee display as a fail, success, or overflow.
Purpose: Creates an interactive task by sprite proximity. This module tracks the user input with keycodes, manages display states, and indicates win, lose, or overflow conditions to the rendering system. It also gives signals to the coffee_color_mapper, which chooses rgb values based on these signals.

Module: coffee_color_mapper.sv
Inputs: blank, vga_clk, DrawX, DrawY, coffee_display
Outputs: red, green, blue
Description: This module selects the appropriate RGB pixel color based signals from task_coffee. It displays an espresso machine pouring coffee into a cup and renders win and fail screens using separate ROMs and palettes.
Purpose: Displays the corresponding image to the signals from task_coffee.

Module: task_vials.sv
Inputs: Reset, frame_clk, keycode, task_, MapX, MapY
Outputs: vial_x, display
Description: Implements FSM for a sequential vial-matching task triggered by entering a specific location. The player inputs a series of keycodes to animate a sprite across five vial positions. Correct key sequences progress the animation, while incorrect inputs fail. Final states include win or lose screens, with a timed reset.
Purpose: Outputs control sprite positioning and task display state for vials task.

Module: vials_color_mapper.sv
Inputs: blank, vga_clk, DrawX, DrawY, vial_x, vial_display
Outputs: red, green, blue
Description: This module selects pixel colors for the vial-matching task based on display state. Renders the base vial background, overlays the animated vial sprite at a specified X position, and switches to win or lose screens as needed.
Purpose: To visually represent the state of the vial task, including animations, success, and failure, by dynamically selecting appropriate color values from ROM-based image data and displaying them on the VGA output.

b. Block design component descriptions

Microblaze: This is a processor that executes the program. It's a customizable CPU core provided by Xilinx and handles instruction execution, memory access, and peripheral control.

AXI Uartlite: Used for serial communication, like printing debug messages to a terminal. It connects to the processor over the AXI bus and operates with minimal overhead.

AXI Interrupt Controller: This module manages hardware interrupts from peripheral modules. It collects interrupt signals and sends them to the MicroBlaze.

AXI Interconnect: This module is a routing component that connects the AXI master (the MicroBlaze) to multiple AXI slaves (peripherals like GPIO, UART, etc.). It handles data transfer, address decoding, and arbitration.

MicroBlaze Debug Module: This module enables debugging capabilities like breakpoints and memory inspection. It allows tools like Xilinx SDK/Vitis to interface with the MicroBlaze core for live debugging.

Clocking Wizard: This module is a single ended clock that generates and manages a 100 MHz clock signal required by all components except for the concatenate module in week 2.

Local Memory: This module is a block of BRAM used for fast access to instructions and data. It serves as the primary data for MicroBlaze Processor.

Processor System Reset: This module generates reset signals for all peripheral modules, the MicroBlaze, the Interrupt Controller, and the local memory module.

Gpio_usb_rst: This module is used as an output to reset the USB controller MAX3421E.

gpio_usb_int: This module is an input connected from the USB interrupt signal. When MAX3421E signals an interrupt, the system is notified through this input.

gpio_usb_keycode: This module is an output used to send keycode values from the USB keyboard to the MicroBlaze system.

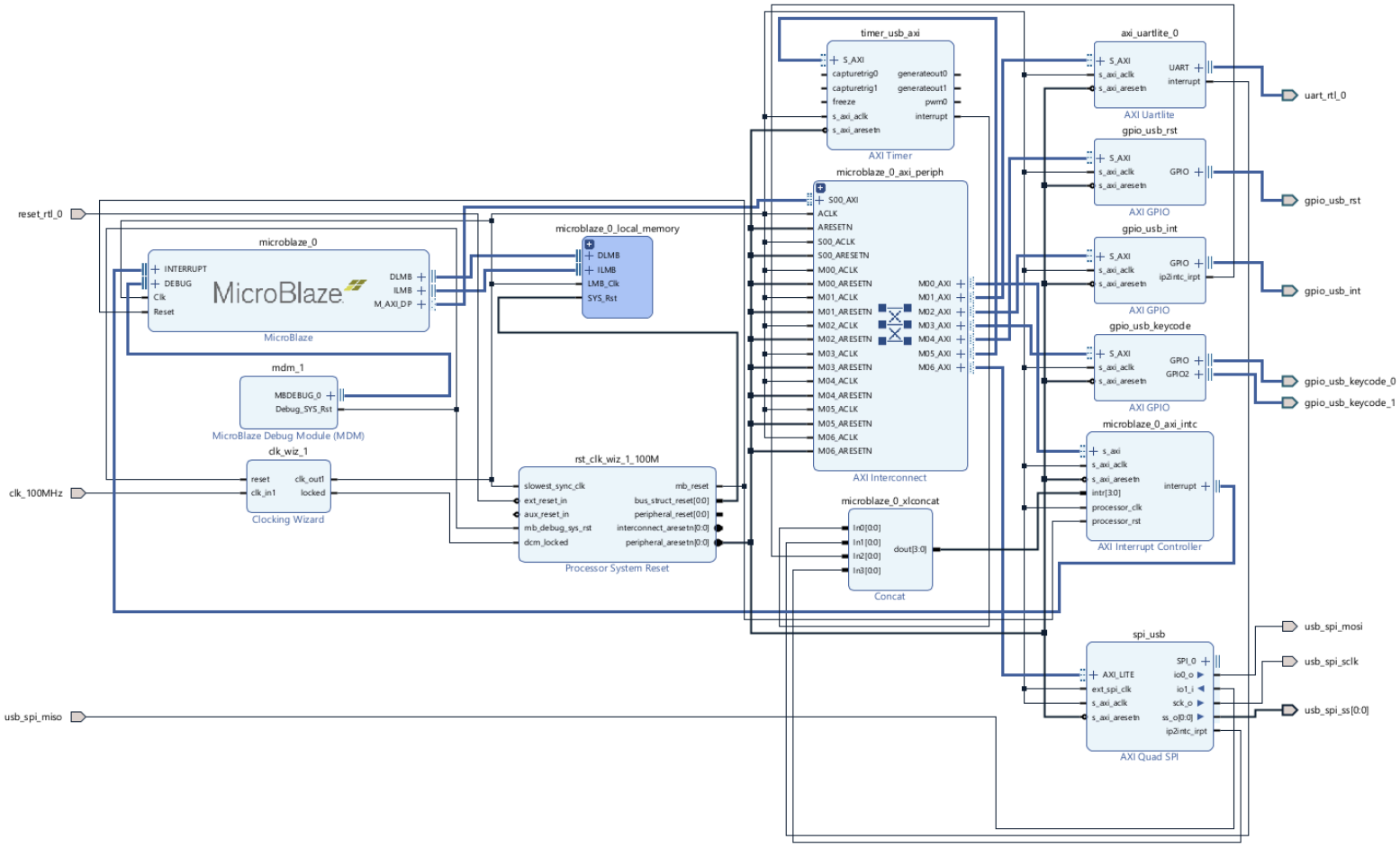
spi_usb: This module is an SPI peripheral used to send and receive data from the MAX3421E. It is a synchronous serial protocol that communicates with the MAX3421E.

concat: This module was deleted in week 1. However, it is used in week 2 to combine multiple interrupt signals into a wider single bus. In this case, the timer_usb_axi, axi_uartlite, spi_usb, and gpio_usb_int interrupts are combined and output as a singular 4-bit signal to the AXI Interrupt Controller.

timer_usb_axi: This module manages the timekeeping for the USB timeouts in milliseconds. The timer allows the MicroBlaze to keep track of the time passed in milliseconds rather than clock cycles.

The MicroBlaze interacts with the MAX3421E USB chip via =the AXI Quad SPI peripheral. It sends and receives data over the SPI bus using software functions that read from and write to the MAX3421E's internal registers. These functions handle USB enumeration and polling to retrieve keycodes from a connected USB keyboard. GPIO peripherals are also used to control and monitor the MAX3421E's reset and interrupt lines.
The VGA-HDMI IP block serves as a bridge between the VGA signal generated by the hardware logic and the HDMI interface required by modern displays. It takes in the VGA signals—RGB pixel data, horizontal and vertical sync signals, and the pixel clock—and converts them into a digital HDMI output using Transition-Minimized Differential Signaling (TMDS). While VGA is an analog standard and HDMI is digital, both use similar timing principles based on horizontal and vertical synchronization. However, HDMI offers higher resolution support, better signal integrity, and can carry audio in addition to video, unlike VGA. The IP block ensures compatibility by encoding the VGA-style signals into the HDMI format, allowing the FPGA to display graphics on an HDMI monitor using familiar VGA timing and logic.

Top Level Block Diagram



The SPI (Serial Peripheral Interface) protocol is a synchronous serial communication method used for short-distance, high-speed data exchange between a master and one or more slave devices. It operates using four main signals: SCLK (serial clock), MOSI (master out, slave in), MISO (master in, slave out), and SS (slave select). The master generates the clock and initiates all data transfers. Data is transmitted in full duplex, meaning both master and slave can send data simultaneously, one bit per clock cycle.

In the context of the MAX3421E USB host controller, the MicroBlaze processor acts as the SPI master and communicates with the MAX3421E (the slave) using an AXI Quad SPI peripheral. To write to a register on the MAX3421E, the MicroBlaze sends the register address followed by the data byte on the MOSI line. To read from a register, the MicroBlaze first sends the register address (with the read bit set), then reads the returned data byte from the MISO line. The SS line is pulled low to activate the MAX3421E during each transfer and pulled high to end the transaction.

The SPI protocol is essential for sending USB commands, reading responses, and polling input data (like keycodes) from a connected USB keyboard. The MicroBlaze uses C functions (like MAXreg_wr() and

MAXreg_rd()) to issue these SPI commands and control the behavior of the USB controller via its internal registers.

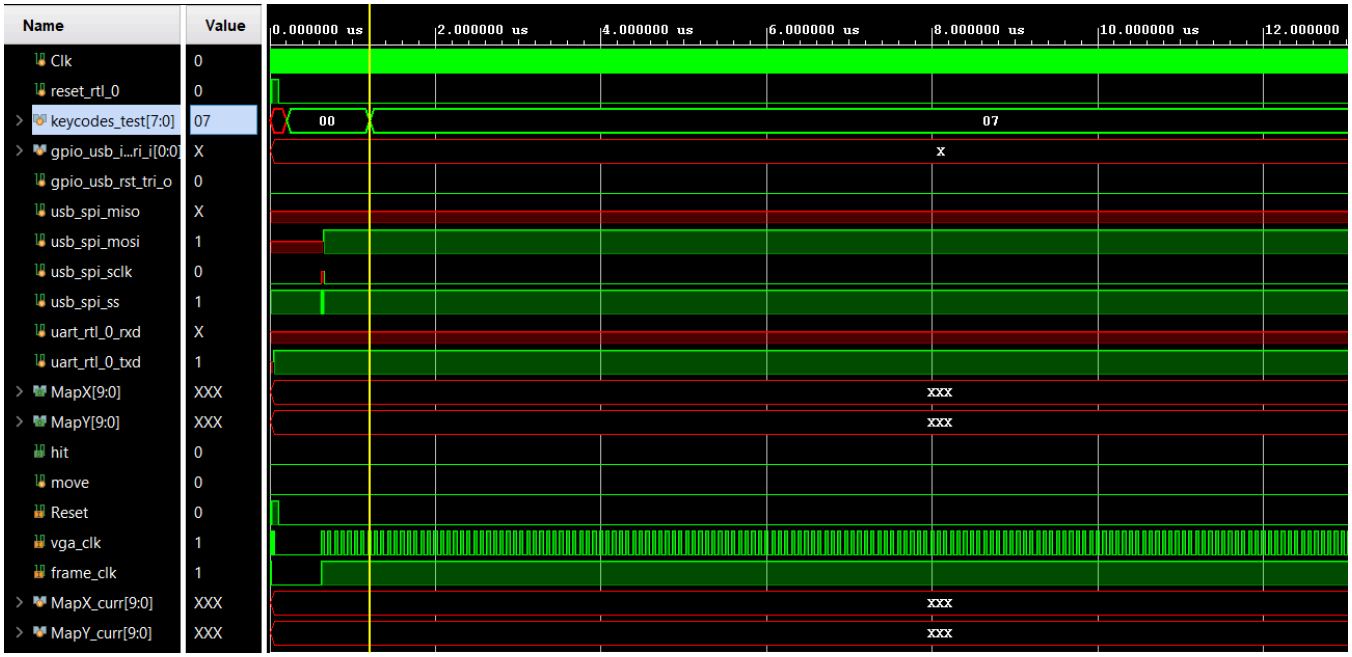
MAXreg_wr(BYTE reg, BYTE val): This function writes a single byte val to the MAX3421E register specified by reg. It formats the register address with the write command, asserts the slave select line, sends the address and data over SPI, then deasserts the slave select to complete the transaction.

MAXreg_rd(BYTE reg): This function reads a single byte from a specified MAX3421E register. It sends the register address with the read bit set, then reads and returns the data byte received from the MAX3421E over the MISO line.

MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data): This function writes multiple bytes (nbytes) from a data array to a register in the MAX3421E. It starts with the register address, then sends the entire data block sequentially using the SPI interface.

MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) This function reads multiple bytes from the MAX3421E register into a data array. After sending the register address, it reads nbytes from the SPI and stores them in the provided buffer.

Simulation Waveforms and Images



Annotation:
In order to directly test keycode inputs the simulation uses a separate logic variable and the inputs to modules are changed from gpio_keycodes to the simulation keycode inputs. At the start of the simulation Reset is toggled high and low again and the keycode is set as 00. At 210 ns the keycodes are changed to 07 and this is supposed to change the MapX and MapY. However, as the simulation shows,

MapX and MapY are always don't cares. This is due to certain variables not being initialized properly when reset is pressed.

Post-Lab Questions

1. Design Resources and Statistics table.

LUT	5207
DSP	40
Memory (BRAM)	72.5
Flip-Flop	2899
Latches	0
Frequency	1/(10-(-4.517)) = 0.0688
Static Power	0.079 W
Dynamic Power	0.415 W
Total Power	0.494 W

Conclusion

a. **Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.**

The overall functionality of the design was very successful in designing and implementing an interactive and custom game of Among Us. The MicroBlaze processor was able to communicate with the MAX3421E USB controller over SPI, properly enumerate a USB keyboard, and detect keypresses. The keycodes correctly moved the map in the background of the character in order to simulate the character moving around the map. The keycodes were also able to allow the user to complete tasks such as entering a code, pour coffee to an indicated level, and select the vials in the correct order. The character's movements are successfully contained within the walls of the map and obstacles in each room. The printer code task properly highlights the correct code and determines whether or not the user's input is correct. The coffee task also properly animates the coffee cup being filled and determines if the cup was filled enough. The vials task correctly animates the selection nozzle above the selected vials and determines whether or not the user input is correct or not. Each task also successfully displays when the character is within a certain range and continually allows the user another attempt if they fail the task. The user is also unable to redo a task if it has already been completed. All components—SPI, GPIO, VGA controller, and custom logic—interacted as intended. One of the biggest challenges we faced was implementing edge detection. We originally wanted to use a separate map with two colors and index into the map at the character's location in order to determine whether or not the next move is

possible. However, indexing into the other map was challenging since there were timing issues with how the map got drawn and how we were indexing the character's position relative to the map. We ended up resolving this issue by hard coding the area that the character was allowed to move in and if the next movement of the character was out of bounds we would set the position of the character in reverence to the map to be the same. Another challenge we encountered was properly displaying the map as a zoomed in section of the map. The indexing was a little difficult to determine, but we resolved this by scaling DrawX and DrawY by the amount we wanted to zoom in on the map by and offsetting those numbers by the character's position in relation to the map. Another minor issue we encountered was certain keyboards would not be recognized by the FPGA despite working a few days beforehand. This was solved by switching to other keyboards. If any other minor bugs had occurred during development, they were resolved through careful testing and validation of each module.