

ECE 385

Spring 2025

Final Project

The Better Fruit Ninja



Maya Ashok, Rina Manxhuka

Lab Section RW/Friday

TA Sally Wu

1) Introduction Description of Full Final Project

We designed the Fruit Ninja game on the FPGA as a system-on-chip that uses a VGA screen and mouse as an input device. Fruit Ninja is a game where the goal is to slice each randomized fruit sprite that appears on the screen (three fruits appear at a time on the screen). The player has three lives, and each fruit sprite missed is a life lost. If the player loses three lives or they slice a bomb sprite, then the game is over. If the player manages to reach nine points (nine fruit sprites sliced), they win the game. For the software component of our project, we utilized a SPI interface in the software code between the Microblaze processor and MAX3421E USB Chip in order to read the x and y displacement cursor information from the USB mouse to control the blade sprite. For the hardware component of our project, a VGA controller, VGA to HDMI IP, Cursor Position, Fruit Motion, LFSR, and Color Mapper modules will display a blade that is controlled by the mouse, background dojo image, randomized fruit and bomb sprites.

2a) Module Descriptions

IP Module: Microblaze

Inputs: INTERRUPT, DEBUG, Clk, Reset

Outputs: DLMB, ILMB, M_AXI_DP

Description: This is the Microblaze processor core utilized as a microcontroller.

Purpose: This IP is created in the block diagram that is central to our design.

IP Module: Microblaze Debug Module (MDM)

Outputs: MBDEBUG_0, Debug_SYS_Rst

Description: This is the Microblaze processor core's debug module in the block diagram.

Purpose: This IP is used to enable debugging of the Microblaze processor core.

IP Module: microblaze_0_local_memory

Inputs: DLMB, ILMB, LMB_Clk, SYS_Rst

Description: This is the Microblaze processor core's local memory module in the block diagram.

Purpose: This IP is used to provide a fast, local bus between the Microblaze and BRAM.

IP Module: AXI Interconnect

Inputs: S00_AXI, ACLK, ARESETN, 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

Outputs: M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI

Description: This is the Microblaze processor's AXI Interconnect module in the block diagram.

Purpose: This IP is used to connect master interfaces (such as the Microblaze) to slave interfaces (such as the MAX3421E).

IP Module: AXI Interrupt Controller

Inputs: s_axi, s_axi_aclk, s_axi_aresetn, intr[3:0], processor_clk, processor_rst

Outputs: interrupt

Description: This is the Microblaze processor's AXI Interrupt Controller module in the block diagram.

Purpose: This IP is used to take in multiple interrupt inputs and concentrate them into one interrupt output.

IP Module: Clocking Wizard

Inputs: reset, clk_in1

Outputs: clk_out1, locked

Description: This is the Microblaze processor's Clocking Wizard module in the block diagram.

Purpose: This IP takes in a single-ended 100MHz clock and outputs a primary clock and signal indicating if the output clock is stable.

IP Module: Processor System Reset

Inputs: slowest_sync_clk, ext_reset_in, aux_reset_in, mb_debug_sys_rst, dcm_locked

Outputs: mb_reset, bus_struct_reset[0:0], peripheral_reset[0:0], interconnect_aresetn[0:0], peripheral_aresetn[0:0]

Description: This is the Microblaze's Processor System Reset module in the block diagram.

Purpose: This IP provides customized reset signals for the entire microblaze processor system.

IP Module: AXI Uartlite

Inputs: S_AXI, s_axi_aclk, s_axi_aresetn

Outputs: UART, interrupt

Description: This is the Microblaze's UART peripheral module in the block diagram.

Purpose: This IP provides some basic printf support for debugging.

IP Module: Concat

Inputs: In0[0:0], In1[0:0], In2[0:0], In3[0:0]

Outputs: dout[3:0]

Description: This is the Microblaze's Concat module in the block diagram.

Purpose: This IP combines the interrupt signal from multiple sources together.

IP Module: AXI GPIO (gpio_usb_rst, gpio_usb_int, gpio_usb_mouse)

Inputs: S_AXI, s_axi_aclk, s_axi_aresetn

Outputs: GPIO, ip2intc_irpt (gpio_usb_int only), GPIO2 (gpio_usb_mouse)

Description: The AXI GPIO module (gpio_usb_rst, gpio_usb_int, gpio_usb_mouse) provides output signals for controlling the MAX3421E USB host chip.

Purpose: This module enables USB mouse control by managing communication with the MAX3421E.

IP Module: AXI Timer

Inputs: S_AXI, capturerrig0, capturerrig1, freeze, s_axi_aclk, s_axi_aresetn

Outputs: generateout0, generateout1, pwm0, interrupt

Description: This is the Microblaze processor's AXI Timer module in the block diagram.

Purpose: This IP allows the Microblaze core to keep track of when 10 milliseconds have passed since the USB has many timeouts that require timekeeping in milliseconds.

IP Module: AXI Quad SPI

Inputs: AXI_LITE, ext_spi_clk, s_axi_aclk, s_axi_aresetn

Outputs: SPI_0, io0_i, io0_o, io0_t, io1_i, io1_o, io1_t, sck_i, sck_o, sck_t, ss_i[0:0], ss_o[0:0], ss_t, ip2intc_irpt

Description: This is the Microblaze processor's SPI peripheral module in the block diagram.

Purpose: This IP is a synchronous serial protocol used to communicate with the MAX3421E.

SV Module: mb_usb_hdmi_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: This top-level module integrates VGA display, USB mouse input, sprite rendering, and game logic to implement a Fruit Ninja game on the FPGA. It connects and coordinates subsystems such as the AXI GPIO for mouse control, sprite modules for rendering fruits, and custom game state logic for tracking score, slicing actions, and game progress.

Purpose: The module serves as the central controller for the Fruit Ninja game, managing input from the USB mouse, updating game state, and driving VGA output to display animated gameplay on a monitor.

SV Module: clk_wiz_0

Inputs: clk_in1, reset

Outputs: clk_out1, clk_out2, locked

Description: This is the clocking wizard module instantiated in the top level/

Purpose: This module is a clocking wizard configured with a 1x (25MHz) and 5x (125MHz) clock for HDMI.

SV Module: vga_controller

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: This is the VGA controller module instantiated in the top level.

Purpose: This module is used as a VGA Sync signal generator.

SV Module: hdmi_tx_0

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, TMDS_DATA_N

Description: This is the hdmi_tx_0 instantiated in the top level that we imported into the project from lab 6.

Purpose: This module is used as a Real Digital VGA to HDMI converter that colors specific pixels on the HDMI monitor given the RGB values.

SV Module: ball

Inputs: Reset, frame_clk, [7:0] keycode, [31:0] mousePos

Outputs: [9:0] BallX, [9:0] BallY, [9:0] BallS

Description: This module tracks the position of the slicing blade based on USB mouse input. It updates the blade's X and Y coordinates using decoded keycodes from the USB mouse.

Purpose: The module enables mouse control of the blade in the Fruit Ninja game, allowing the player to move the blade across the screen to slice falling fruits.

SV Module: hex_driver

Inputs: clk, reset, [3:0] in[4]

Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: The hexadecimal values of the input are displayed on the FPGA's hex drivers.

Purpose: This module is used for debugging purposes. We used it to see how different signals changed during real-time game play.

6.2 SV Module: mb_block_i

Inputs: clk_100MHz, reset_rtl_0, usb_spi_miso

Outputs: uart_rtl_0_rxd, uart_rtl_0_txd, gpio_usb_int_tri_o, gpio_usb_rst_tri_o, gpio_usb_keycode_0_tri_o, gpio_usb_keycode_1_tri_o, gpio_usb_mouse_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss

Description: This is the block diagram module for Lab 6.2.

Purpose: This module is the block design that with all the IP modules we created (Microblaze, MDM, Clocking Wizard, Processor System Reset, AXI Interconnect, AXI Interrupt, AXI Uartlite, AXI GPIO, Concat, AXI Timer, AXI Quad SPI).

SV Module: woodBG_example

Inputs: vga_clk, [9:0] DrawX, [9:0] DrawY, [9:0] BallX, [9:0] BallY, [9:0] FruitX, [9:0] FruitY, [9:0] FruitX2, [9:0] FruitY2, [9:0] FruitX3, [9:0] FruitY3, [4:0] choose, [4:0] choose2, [4:0] choose3, collided, collided2, collided 3, start, endWon, endLost, blank, reset, [3:0] score

Outputs: [3:0] red, [3:0] green, [3:0] blue, startGame, endGame

Description: Top-level VGA display controller for rendering game graphics based on pixel location, object positions, and game state inputs.

Purpose: Combines sprite outputs to generate the final RGB pixel data, displaying fruit, blade, score, and start/end screens based on game logic.

SV Module: `fruit_motion, fruitMotion3`

Inputs: `Reset, frame_clk, [9:0] xRand`

Outputs: `[9:0] FruitX, [9:0] FruitY, outside`

Description: The `fruit_motion` module simulates the projectile motion of a fruit in a fruit-slicing game. Based on a random horizontal starting position, it updates the fruit's X and Y coordinates each frame to simulate an arc motion, and indicates when the fruit has exited the screen with the outside signal. There are two instantiations of `fruitmotion`, indicating that there are two fruits moving in the same direction. There is one instantiation of `fruitMotion3` that creates a motion of a fruit in the opposite direction.

Purpose: Control the fruit's movement behavior in the game by implementing a model that moves in an upward and downward motion. It enables time-based animation of the fruit and signals when the fruit leaves the visible area, supporting collision detection and game state updates.

SV Module: `lfsr_5bit, lfsr_5bit2, lfsr_5bit3`

Inputs: `clk, reset, enable`

Outputs: `[4:0] out`

Description: Implements a 5-bit Linear Feedback Shift Register that generates random sequences using taps at bit positions 4 and 2. It shifts the register contents on each positive clock edge when enabled and can be reset to a predefined seed. There are three different modules with three different seeds so there are three different starting positions for three different fruits

Purpose: The LFSR generates random numbers that are used to determine the starting x-positions of fruit motions as well as randomizing which type of fruit appears.

SV Module: `collision`

Inputs: `clk, start, [9:0] BallX, [9:0] BallY, [9:0] FruitX, [9:0] FruitY, Reset, resetFruits`

Outputs: `collided`

Description: Detects collisions between blade and a fruit object based on their x and y coordinates. It sets `collided` high when the ball enters a 64x64 pixel bounding box around the fruit. Collided can be reset with `resetFruits`. There are three different instances of `collision` so the game continuously checks for collisions with three different fruits

Purpose: Determines whether a fruit has been successfully hit by the blade during the game, enabling score incrementation/game logic when a collision occurs.

SV Module: game

Inputs: clk, frame_clk, reset, startMelonCollided, bombCollided, outOfScreen, outOfScreen2, outOfScreen3, collided, collided2, collided3

Outputs: resetFruits, startSprites, endSpriteWon, endSpriteLost, chooseRand, [3:0] points, [2:0] lives

Description: Controls the game state and logic, including updating the score, lives, and game logic. It handles different stages of the game and updates the game based on event triggers such as fruit collisions and screen exits. It also updates the score and lives based on these events and transitions between game states like winning, losing, or resetting

Purpose: Coordinates the main game flow, including the handling of fruit slicing, collision detection, and updating the player's score and lives. Progresses between game states and triggers appropriate events like game over or victory

SV Module: blade_example, wmelon_example, wmelonSliced_example, startName_example, startCircle_example, endWon_example, endLost_example, apple_example, appleSliced_example, berry_example, berrySliced_example, orange_example, orangeSliced_example, pineapple_example, pineappleSliced_example, starfruit_example, starfruitSliced_example, pepper_example, pepperSliced_example, pomegranate_example, pomegranateSliced_example, bomb_example

Inputs: vga_clk, [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY, blank

Outputs: [3:0] red, [3:0] green, [3:0] blue, [spriteName]_visible

Description: Renders game sprites on a VGA display. They use inputs such as clock signals, ball positions, draw positions, and a blank signal to control sprite rendering. Each module outputs RGB values for the color of the sprite and a visibility signal to indicate whether the sprite is currently drawn on the screen. Each module also has an associate palette module that an index input to an rgb value.

Purpose: Manages the display of game sprites, determining which sprites are visible and ensuring that the correct colors are rendered based on the game state. Controls sprite visibility and updates the display based on the ball's interaction with the game environment.

SV Module: num0_example, num1_example, num2_example, num3_example, num4_example, num5_example, num6_example, num7_example, num8_example, num9_example

Inputs: vga_clk, digit, [9:0] DrawX, [9:0] DrawY, blank

Outputs: [3:0] red, [3:0] green, [3:0] blue, num[current number]_visible

Description: Render individual numeric digits (0-9) as sprites for display

Purpose: Generates sprite graphics corresponding to the digits 0 through 9 for use in score display. The visibility of the digits depends on the current score.

SV Module: `alive_example`, `dead_example`

Inputs: `vga_clk`, `[1:0] number`, `[9:0] DrawX`, `[9:0] DrawY`, `blank`

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

Description: Render two distinct "X" sprites on the VGA display: an empty "X" for the `alive_example` and a red "X" for the `dead_example`, representing if a player missed slicing a fruit

Purpose: To have a visual representation of the game state. If the player misses three fruits, i.e. the three empty x's are now three red x's, the player loses.

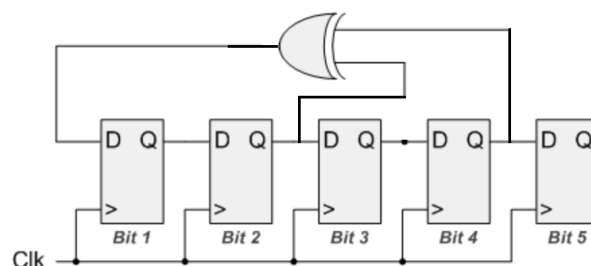
2b) Description of MicroBlaze, MAX3421E USB Chip Interaction, Blade Components

The FPGA's MAX3421E USB Chip communicates with the FPGA's Microblaze processor using a SPI Interface implemented in Lab 6.2 that allows the Microblaze to act as a master and interact with the USB peripherals, specifically a mouse. We created one new mouse position GPIO in our Microblaze processor to store the mouse x and y cursor position data. SPI enables the Microblaze processor to read or write to the mouse position GPIO from one of the MAX3421E's 32 1-byte/8-bit registers.

The x displacement is located in bits 8-15 and y displacement is located in bits 16-23 of the `mousePosition` input. Our mouse position data from the block diagram GPIO is then fed as an input into our ball module that outputs the center x and y coordinates of our blade sprite controlled by the cursor. Our ball module contains logic that updates the blade sprite's center x and y coordinates when the mouse moves as well as makes sure it stays within the bounds of the screen.

2c) Description of Randomization Logic (LFSR), Fruit Motion, and Fruit Collision

To randomize the initial starting x position for each of the three fruit sprites on the screen as well as randomly choose which fruit sprite is visible, we utilized three 5-bit linear finite shift registers. A LFSR works by initializing the register first to a random seed and then left shifting the register on an enable signal where the new LSB or feedback bit is the XOR of the 2nd and 4th bit. The utilization of the XOR for the feedback bit ensures pseudo-random sequences. Each of our three LFSR uses a different seed value to ensure they give different random 5-bit outputs.



Our three LFSRs are used to get a random initial Fruit X(1/2/3) position within the screen x coordinates 110-420 (by doing LFSR output *10 + 100). Our three LFSRs are also used to get a random choose(1/2/3) value by doing LFSR output % 3 + 1 for choose and choose3 and LFSR output % 4 + 1 for choose2. We had to utilize a register to update the choose values on the 1-cycle pulse of the random enable (chooseRand output of the game FSM) signal or else it would be continuously updating.

The fruit_motion module is used to create vertical and horizontal movement for fruit to simulate a projectile motion. First, the fruit starts at a random horizontal position that it gets as an input from the top level, which is mapped from the output of the LFSR. There is a fixed vertical starting position, which is at the bottom of the screen. The motion is implemented using a state machine. There are four states. The first state is IDLE which forces the game to wait one frame before starting the game. There is an UP state that causes the fruit to move upwards with a constant vertical velocity until it reaches a certain y position, which is near the center of the screen. The next state is DOWN which forces the fruit to move with a constant vertical velocity downwards until it reaches a certain y position, which is near the bottom of the screen. Finally, there is a STOP state that forces the fruit to stop moving once the fruit is outside of the screen. The module resets the fruit's position once the reset is triggered.

The collision module (which we have three instantiations of for each fruit on the screen) is used to signal whether or not we have sliced the fruit. It takes in the BallX and BallY cursor positions as well as FruitX(1/2/3) and FruitY(1/2/3) positions as inputs. It sets the collided output to 1 signaling a sliced fruit if BallX and BallY are within the 64x64 square of the FruitX(1/2/3) and FruitY(1/2/3) center position. The collided output is also set to 1 if we are in the start state and the blade cursor is within the 64x64 square that the start melon is located at (430, 330). Collided is set to 0 which means unsliced otherwise.

2d) Description of VGA Operation, Color Mapper, and VGA Controller Interaction

The Video Graphics Array or VGA consists of a screen organized as a matrix of pixels with dimensions of 640 horizontal pixels by 480 vertical lines. The VGA works by having an electron beam paint or color each pixel from left to right in each row and color each row from top to bottom. The VGA Controller module is instantiated in the top level and outputs sync signals as well as the current drawX and drawY coordinates that are painted.

Using the Image to COE tool provided by the course assistants (https://github.com/amsheth/Image_to_COE), we created a background color mapper module, woodBG_example, instantiated in the top level that takes in the three fruit x and y positions, the blade cursor x and y position (ball module output), three choose inputs (to choose which fruit sprites is displayed for each of the three fruits on the screen), three collided logic variables (indicating if each fruit has been sliced), a start variable (indicating if we are in the start game

FSM state), an endWon variable (indicating if we are in the endWon game FSM state), an endLost variable (indicating if we are in the endLost game FSM state), a scores variable, and a lives variable as inputs. It outputs a startGame signal, which is the the startMelonCollided input of the game FSM. It also outputs an endGame signal, which is the bombCollided input of the game FSM.

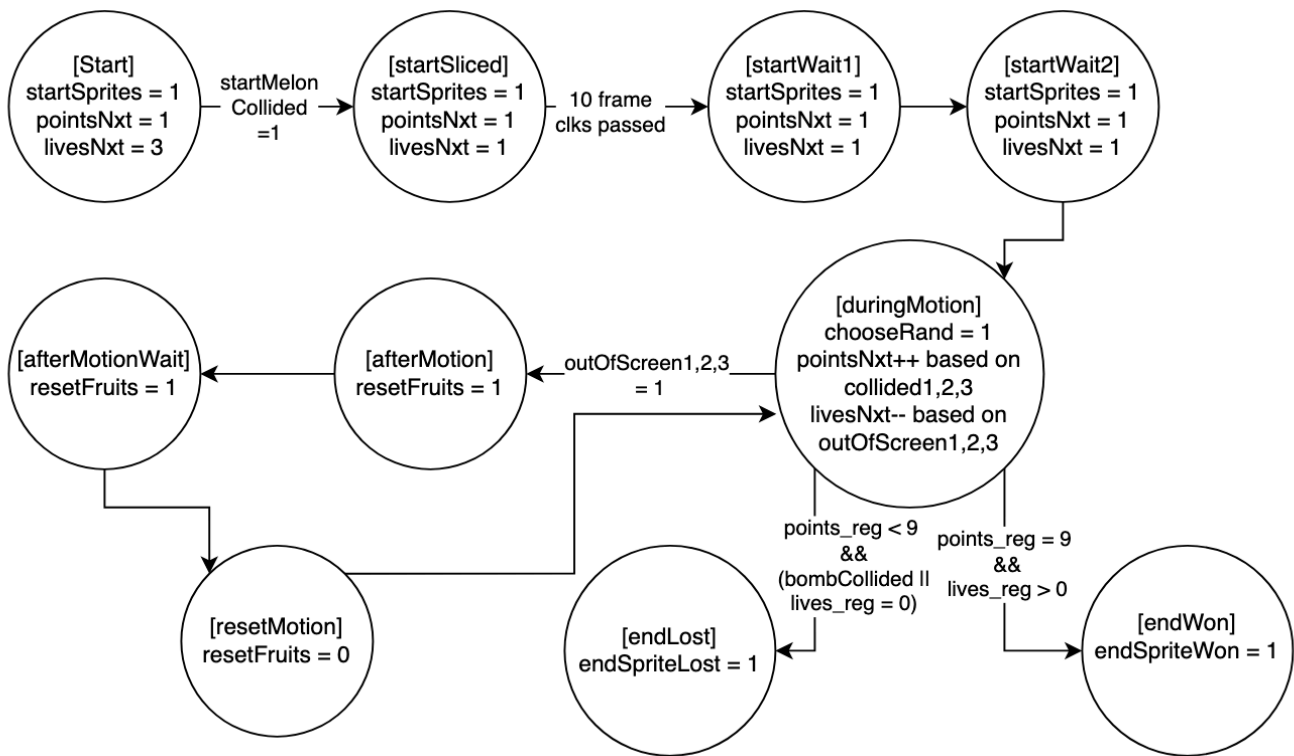
Along with the woodBG_palette instantiation given by the github tool in woodBG_example, we instantiated all the sprites (also created by the github tool) utilized in our project as well. Each of these sprite instantiations have the necessary inputs (either BallX & BallY for the blade cursor, FruitX(1/2/3) & FruitY(1/2/3) for the three fruits, and choose(1/2/3) for the three fruits, start signal for the start screen sprites, endWon/Lost signal for the end lost/won screen sprites) that are utilized to output a [sprite]Visible logic variable indicating if that sprite should be visible at that DrawX and DrawY position. We then have logic in the form of multiple if statements that set red, green, and blue to [sprite]Red, [sprite]Green, and [sprite]Blue based on its [sprite]Visible value and other necessary logic. For example, the fruit sprites depend on collided being 0, sliced fruit sprites depend on collided being 1, the numbers sprites depend on the current score, and lives sprites depend on the current number of lives. Here is a list of all of our sprites instantiated in the woodBG_example color mapper module (if these visibility conditions aren't met the wood background is visible):

Visibility Depends on BallX, BallY (cursor position)	Visibility Depends on FruitX, FruitY, Choose <i>([sprite]Sliced Visibility depends on collided)</i>	Visibility Depends on FruitX2, FruitY2, Choose2 <i>([sprite]Sliced Visibility depends also on collided2)</i>	Visibility Depends on FruitX3, FruitY3, Choose3 <i>([sprite]Sliced Visibility depends also on collided3)</i>
Blade	Wmelon (choose = 1), Pepper (choose = 3), Starfruit (choose = 2)	Apple (choose2 = 1), Banana (choose2 = 4), Orange (choose2 = 3), Pineapple (choose2 = 2)	Berry (choose3 = 2), Pomegranate (choose3 = 3), Bomb (choose3 = 1)
Visibility Depends on Start Game State	Visibility Depends on End Won/Lost Game State	Visibility Depends on !Start Game State and Score	Visibility Depends on !Start Game State and Lives
StartName, StartCircle, Wmelon	EndWon, EndLost	Num0, Num1, Num2, Num3, Num4, Num5, Num6, Num7, Num8, Num9	Alive (3 instantiations), Dead (3 instantiations)

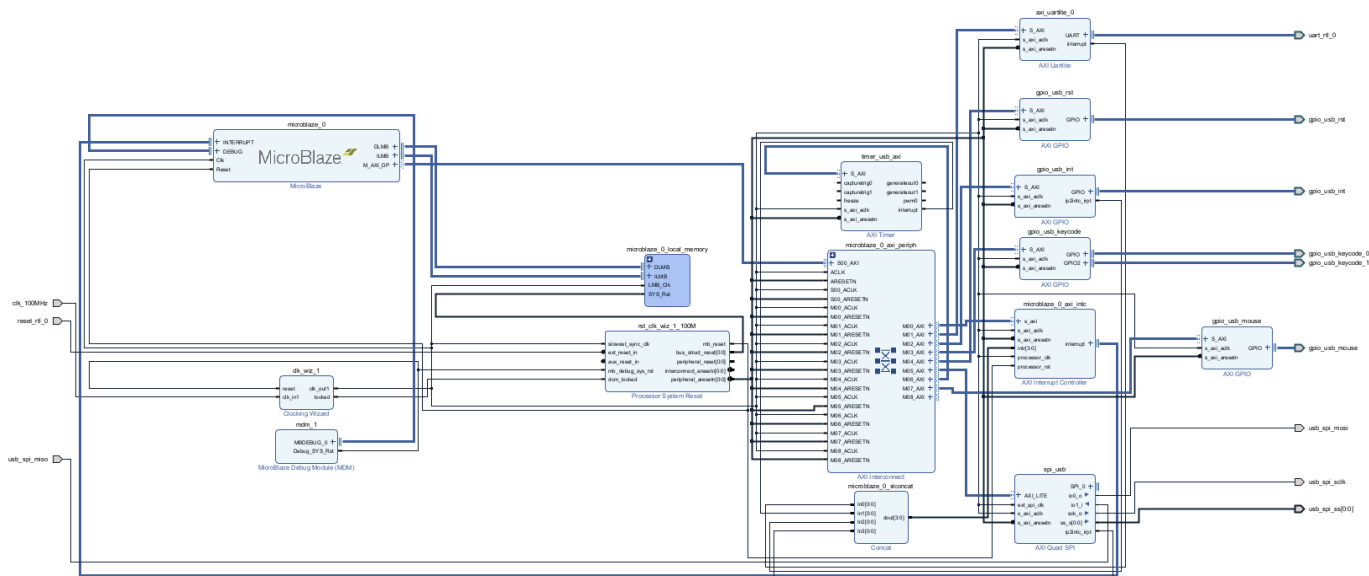
2e) State Diagram of Game Finite State Machine Unit

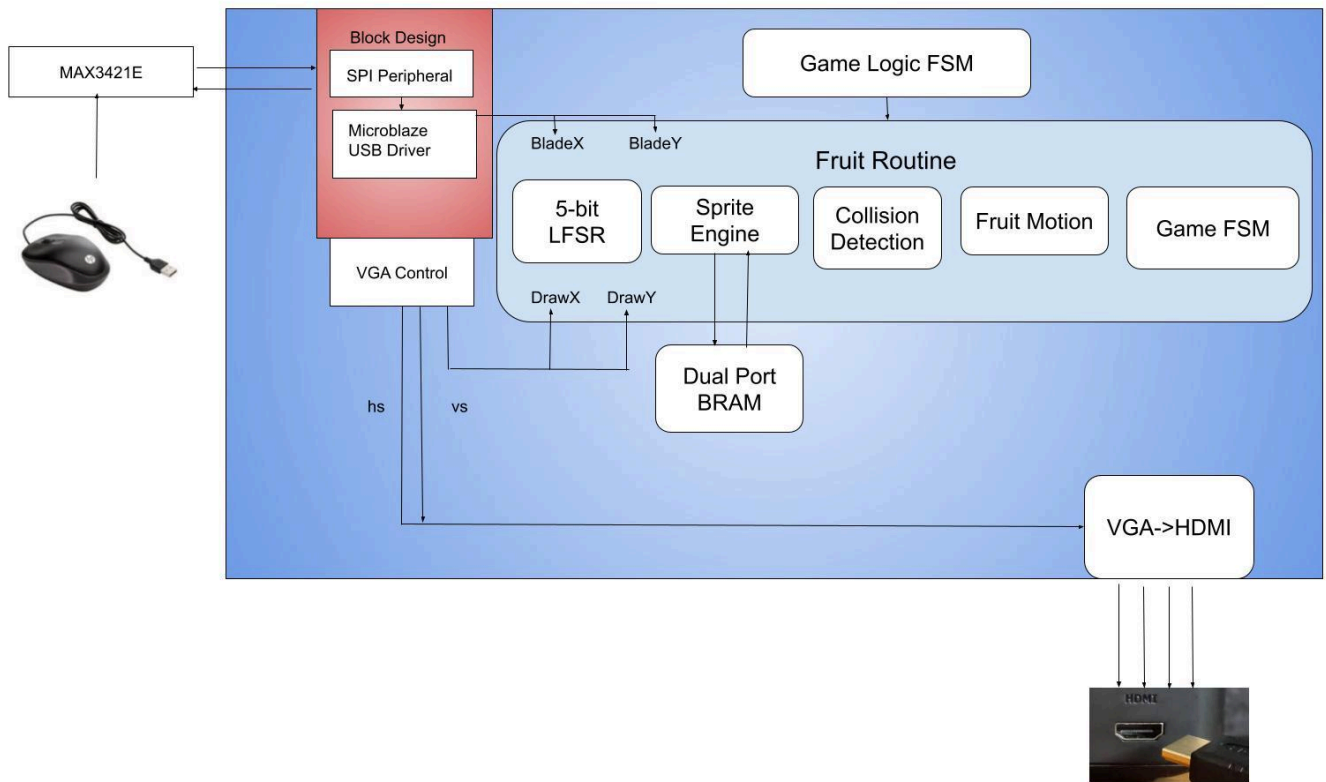
We utilized a points register and lives register in our game module to store the number of points and lives left the player has. The logic for incrementing the points is based on the 1-cycle clock pulse of collided, collided2, and collide3. This was necessary because without it, we found while

debugging using the hex drivers that the points register was continuously updating its value even when it was not supposed to.



3) Top Level Block Diagram





4) Statistics

LUT	6136
DSP	37
Memory (BRAM)	55
Flip-Flop	3454
Latches*	0
Frequency	0.10257 Hz
Static Power	0.078 W
Dynamic Power	0.420 W
Total Power	0.498 W

5) Conclusion

For our final project, we accomplished having a background image, implementing a mouse user input, utilizing randomness to select fruit initial positions and which fruits are chosen, outputting the score counter on the screen, outputting the lives counter on the screen, color mapping multiple fruit sprites and a blade sprite, and implementing a start screen, game won, and game lost screen using sprites. To debug our final project, we utilized the hex drivers so we did not include any simulations as we did not use them in our design and testing process. Through this design process, we utilized the skills we learned from the previous labs as well as our own research to create a project that encompasses our knowledge of FPGA design.