University of Illinois Urbana-Champaign
Department of Electrical and Computer Engineering

ECE 385: Digital Systems Laboratory

Final Project:
Motion Detect Multiplayer Flappy Bird

Team member(s): Nam Hoang Nhat, Andrew Lin

May 15th, 2026

Instructor: Professor Zuofu Cheng
TA: Peidong Yang
Spring 2026

Introduction

For the final project of ECE385 we implemented a duo FPGA motion controlled multiplayer Flappy bird clone by utilizing the skills that we have learned throughout the semester. This project requires two Urbana Boards, with one being the game engine and the other being the computer vision processor (motion detector). The game engine board is set up like lab 6.2, with connections to the keyboard and the HDMI monitor. With keycodes from the keyboard, sprites from the spritesheet, and game logic implemented in Vivado, we created a working Flappy Bird game that can be played and displayed. The motion detector board utilized the ov7670 CMOS color image sensor as source of vision, and utilized an image processing pipe line to gray scale, edge detect, and track the motion of the object in frame, then put a jump pulse signal through the board's PMOD to trigger movement in the game engine. This project aims to achieve two small projects – game engine on hardware and motion detector on hardware– and combine them into one big project. At the end we got the game engine to run mostly on hardware with only the HID device input being processed with microBlaze and the motion detector running purely on hardware, which is satisfying.

Part Description

Game Engine Description

With the same setup from lab 6.2, the game is controlled using the keyboard, displayed on the HDMI monitor, and game logic is implemented on the FPGA board. The entire system is driven by a 100 MHz system clock, but we use a 60Hz clock (the VGA vertical sync signal), ensuring physics updates and rendering are synchronized with the display refresh rate.

The game follows a Model-View-Controller (MVC) architecture in hardware. The Model is the bird(s), pipes, and all the sprites displayed. It maintains the state of the game, including the bird(s) (position, velocity, alive status), the pipes (horizontal position and randomized vertical position), and when to display what (game score, intro screen, game over screen, etc). The View is the color mapper and VGA controller, translating game state into pixel data to display the game. The Controller is the Microblaze-based keyboard, the input synchronization, and later on, the signal from the motion detection FPGA.

Sprite Sheet

To render high quality graphics and have high-speed coordinate translation for the game, we use sprites and have them stored using on-chip memory BRAM. The rendering pipeline can be broken down as follows. First, the hardware calculates the relative coordinate for each object using the pixel's coordinate (DrawX, DrawY) generated by the VGA controller. This is done by subtracting the top left coordinate of the object on the sprite from the DrawX and DrawY. Second, we check if this is in the "bounding box" (larger than or equal to 0 and less than the width (for X) and height (for Y) of the object). Third is getting the ROM address using the relative coordinate. The formula is relative Y times sprite width plus relative X. The address

would get us an index, which we will use to get the RGB value, allowing us to draw objects ("colors") on the monitor through our color mapper. We also tell the color mapper to ignore the pink-ish (sprite background color) to make the graphic look better. That is for one object. With multiple sprites, we need to have a solid Z-order to ensure the game looks good. We set the background and the base in the back (order 0), and everything else in the front (order 1). A good amount of time was dedicated to having the graphic look good and also the desired display.

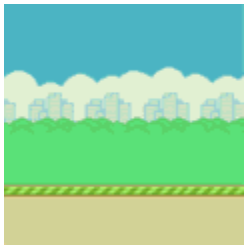


Figure 1 - Background and base Sprite Sheet

Figure 2 - UI Sprite Sheet

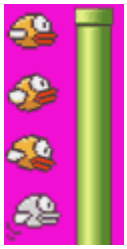


Figure 3 - Bird and Pipe Sprite Sheet

Collision Detection

To detect collisions for the game status update, we use the Axis-Aligned Bounding Box (AABB). We flag a collision if there is any vertical (BirdY < PipeY OR BirdY + Bird_Height > PipeY + Gap_Size) or horizontal (BirdX + Bird_Width > PipeX AND BirdX < PipeX + Pipe_Width) overlaps. There is also a check for collision when the bird hits the ceiling and the base. All of these will update the dead signal, which disables the bird, stops the moving pipes, and displays game over screen.

Game Control and Logic

The physics engine simulates a discrete-time approximation of Newtonian mechanics. To implement gravity as a constant acceleration, in every frame_clk cycle we have $V_y(t+1) = V_y(t) + Gravity$ and $Pos_y(t+1) = Pos_y(t) + V_y(t+1)$. We set V_y to a fixed

negative value to overcome gravity and move the bird up. A key counter-intuitive detail here is that the top left of the screen is (0,0) so to move down, we have to increase and vice versa. We also use flip-flops to ensure that players have to click for the bird to jump, not simply holding it. This allows us to check the previous keycode values and only allow a jump to occur if the player is clicking and releasing, not pressing it. To generate random pipe heights, we use a Linear Feedback Shift Register (LFSR). To solve the common issue of FPGAs generating the same "random" sequence every time they are turned on, we seed the LFSR with a Ring Oscillator. This uses the physical propagation delay of LUTs to create a high-frequency, non-jittering clock that provides a non-deterministic seed.

Vitis

With our pure hatred for Vitis, we simply used what was provided and made in lab 6.2. It somehow still took us 3 hours to debug (deleting and creating a new workspace), and additionally, right before the demo, 25 minutes and exactly 41 seconds to get it to work.

Multiplayer Implementation

With some time left, we added a multiplayer feature for the game. Players can access this by turning on the switches on the FPGA board. This will be a cooperative play where players (2-4) try to survive and score together (any bird passing a pipe will increment one, but then the ones after will not increase the score). Although we could have simply duplicated the single bird module multiple times, we decided to do it more elegantly. Using packed arrays, we allow each bird to have its own physics state. We have 4 instead of 6 or 7 birds because we utilize the 32-bit keycode our interface already had. With each keycode being 8 bits, this fits perfectly for four independent controls (W, I, Space, and Up Arrow).

Motion Detector Description

The motion detector is composed of an Urbana Board, an OV7670 image sensor, a breadboard, 18 male-to-female (MTF) jumper wires, 4 male-to-male (MTM) jumper wires, and a 4.7kΩ through-hole resistor. The 18 MTF jumpers are used to connect the OV7670 to the Urbana Board's PMODs according to Figure 5, with SDA and SCL going through the breadboard to be pulled up by the 4.7kΩ resistor and connected to the FPGA with MTM jumpers. The other two MTM jumpers are for communication between the two boards; one is connected to GND on both boards to create a common ground, and the signal wire is connected to the D11 pin on the motion detection board.

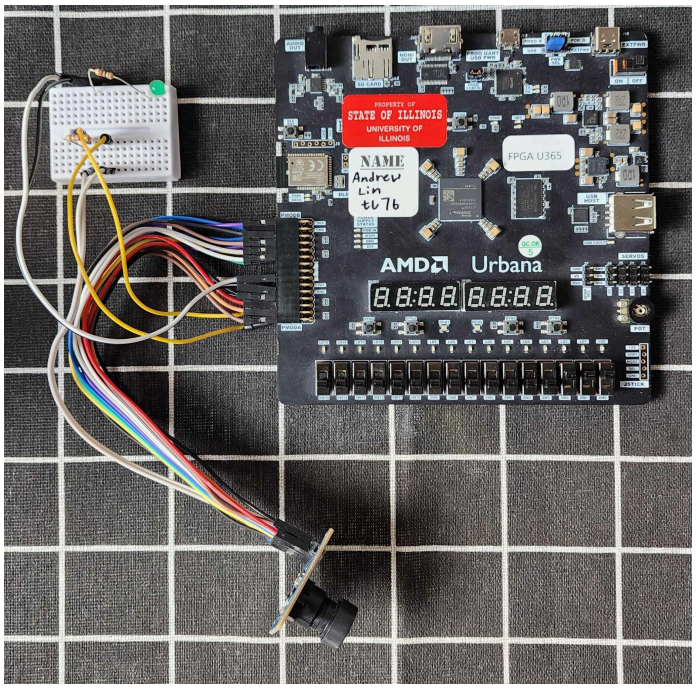
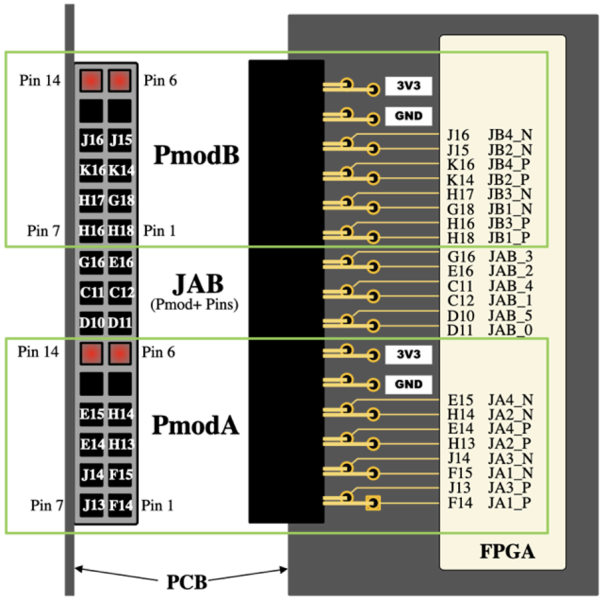


Figure 4 - Motion Detector Unit



OV Pin	PMOD Port		OV Pin
D4	J16	J15	D0
D5	K16	K14	D1
D6	H17	G18	D2
D7	H16	H18	D3
HS	E15	H14	RST
MCLK	E14	H13	PWDN
SDA	J14	F15	PCLK
SCL	J13	F14	VS

Figure 5 - ov7670 pinout (source: https://github.com/souvlakiboi/OV7670_camera)

Inputs and Outputs

The subunit takes pixel data from the OV7670 camera as its primary input: an 8-bit pixel byte, pixel clock, vertical sync, and horizontal reference signals. A 16-bit switch bus provides runtime configuration: sw[5:0] sets the Sobel weak-edge threshold, sw[11:6] sets the motion detection threshold, and sw[15] selects the display mode (0 = edge detection, 1 = grayscale

passthrough). Outputs include an HDMI video stream, a motion indicator LED, and a PMOD signal pin which pulses high for approximately 50ms whenever motion is detected.

Pipeline

The system processes video through the following sequential stages:

1. OV7670 Camera: Captures 640x480 frames and delivers raw 12-bit RGB pixels over the camera parallel bus.

2. grayScaler: Converts each 12-bit RGB pixel to a 3-bit grayscale value using a weighted luminance formula (Y = (R x 77 + G x 150 + B x 29) >> 9), approximating the standard Rec. 601 luma coefficients.

3. GS_buffer (Block RAM 0): Stores the full 640x480 grayscale frame. Port A is written by the camera in the pixel-clock domain; Port B is read at 25 MHz by the edge detector.

4. edgeDetector: Reads the grayscale frame from GS_buffer and applies a Sobel 3x3 spatial filter to each interior pixel, fetching all 9 neighbours sequentially over 10 clock cycles. The resulting gradient magnitude is compared against three thresholds to produce a 2-bit edge strength value per pixel.

5. ED_buffer (Block RAM 1): Stores the full 640x480 edge-detection frame. Port A is written by the edge detector; Port B is read by the centroid tracker.

6. centroidTracker: Sits in series on the ED_buffer read path. It snoops each edge pixel as it passes through and accumulates weighted position sums (sum_x, sum_y, sum_w) across the frame. On each vsync falling edge, a 30-step shift-subtract divider computes the centroid coordinates cx = sum_x / sum_w and cy = sum_y / sum_w, representing the intensity-weighted center of all detected edges. If no edges were seen in a frame, the previous centroid is held.

7. signalRouter: Selects which buffer drives the display and which address bus drives GS_buffer's read port, based on sw[15].

8. Crosshair Overlay: Composites a (+) crosshair (arm length +/-20 px, center gap +/-3 px) at (cx, cy) on top of the pixel stream, visible in both display modes.

9. motionIO: On each vsync, computes the Chebyshev-approximated displacement between the current and previous frame centroid. If this exceeds sw[11:6] and a 120ms cooldown

has elapsed, the PMOD output pulses high for 50ms and the on-board LED toggles.

10. vga_top to HDMI: Drives the 640x480 at 60Hz VGA timing and serializes the output via the HDMI TMDS transmitter.

Algorithm

Edge detection uses the Sobel operator. For each interior pixel at (row, col), the 3x3 neighbourhood p0 through p8 is fetched from the grayscale buffer one pixel at a time. The horizontal and vertical gradients are computed as:

Gx = (p2 - p0) + 2 x (p5 - p3) + (p8 - p6), range [-28, +28] Gy = (p6 - p0) + 2 x (p7 - p1) + (p8 - p2), range [-28, +28] magnitude = |Gx| + |Gy|, range [0, 56]

Border pixels are always output as no edge. Centroid tracking uses an intensity-weighted average: every non-zero edge pixel contributes its 2-bit strength value as a weight, accumulating sum_x, sum_y, and sum_w over the full frame before dividing at vsync.

Motion between frames is estimated with the Chebyshev distance approximation applied to the centroid displacement: delta_dist = max(|dx|, |dy|) + min(|dx|, |dy|) / 4.

Thresholds and Adjustments

The edge detector derives three thresholds from the single 6-bit sw[5:0] weak input:

Weak threshold: sw[5:0] — magnitude must meet this to register any edge. Strong threshold: sw[5:0] + 2 — produces a mid-level edge result. Hard threshold: sw[5:0] + 4 — produces a definite edge result.

Increasing sw[5:0] suppresses more gradients, reducing noise but potentially missing faint edges. The 2-bit edge strength is expanded to a 3-bit brightness for display: no edge maps to black, weak to dim, strong to bright, and definite to white.

The motion detection threshold is set by sw[11:6]. Increasing this value requires larger frame-to-frame centroid displacement to trigger a motion event, reducing sensitivity to small or slow movements. A 50ms output pulse and 120ms post-trigger cooldown prevent false re-triggering from the same motion event.

Block Diagram

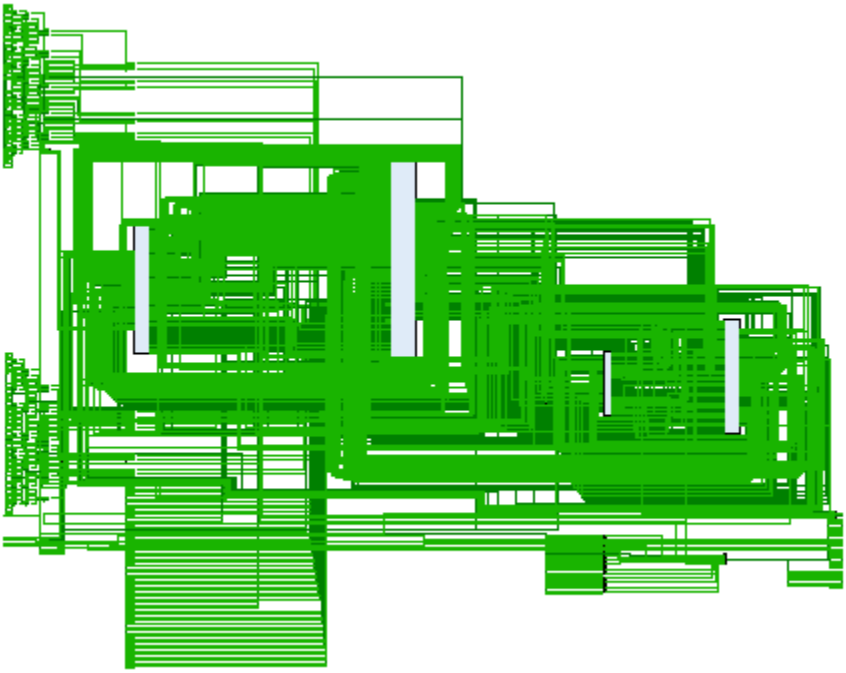


Figure 6 - Game Engine High level RTL

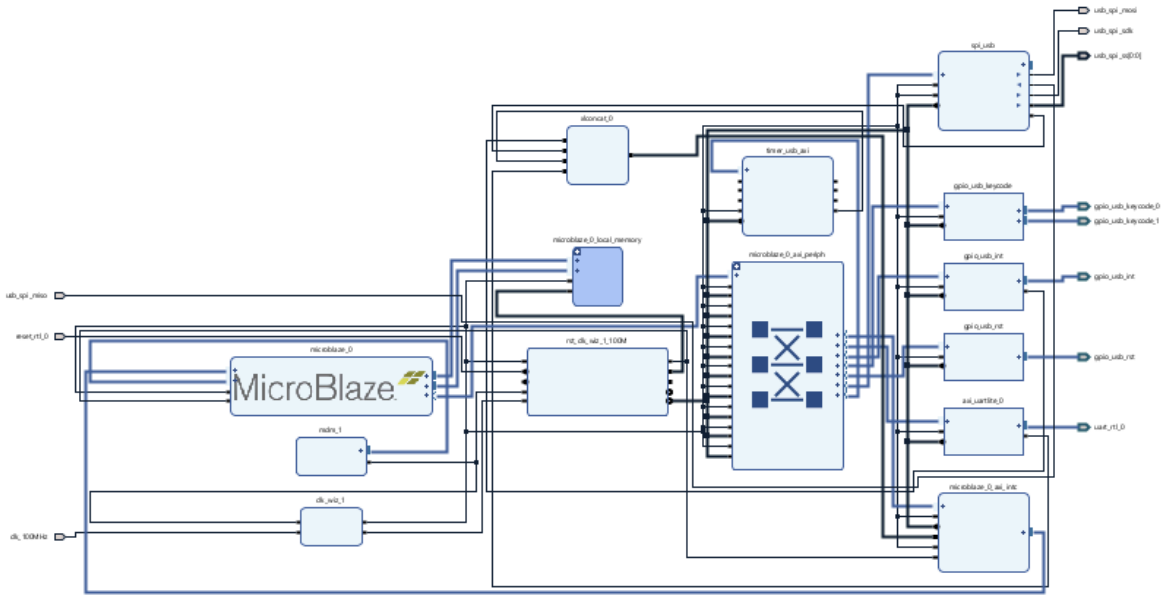


Figure 7 - Game Engine Microblaze Set Up

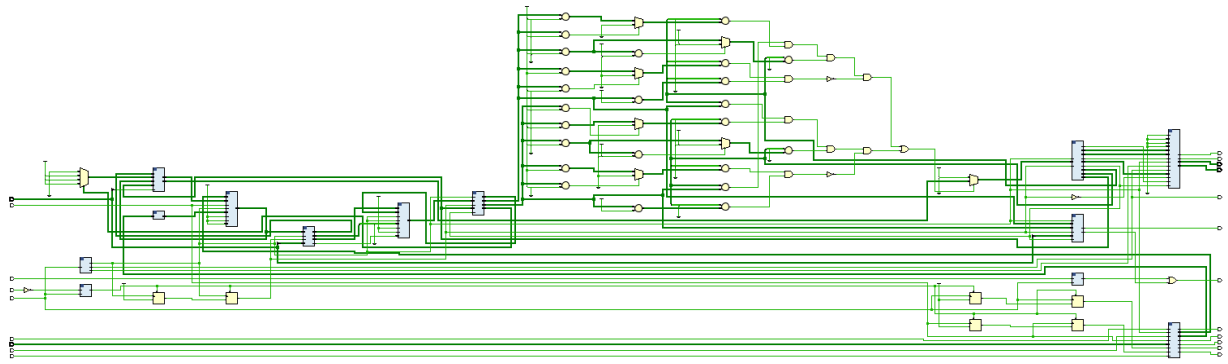


Figure 8 - Motion Detector High level RTL

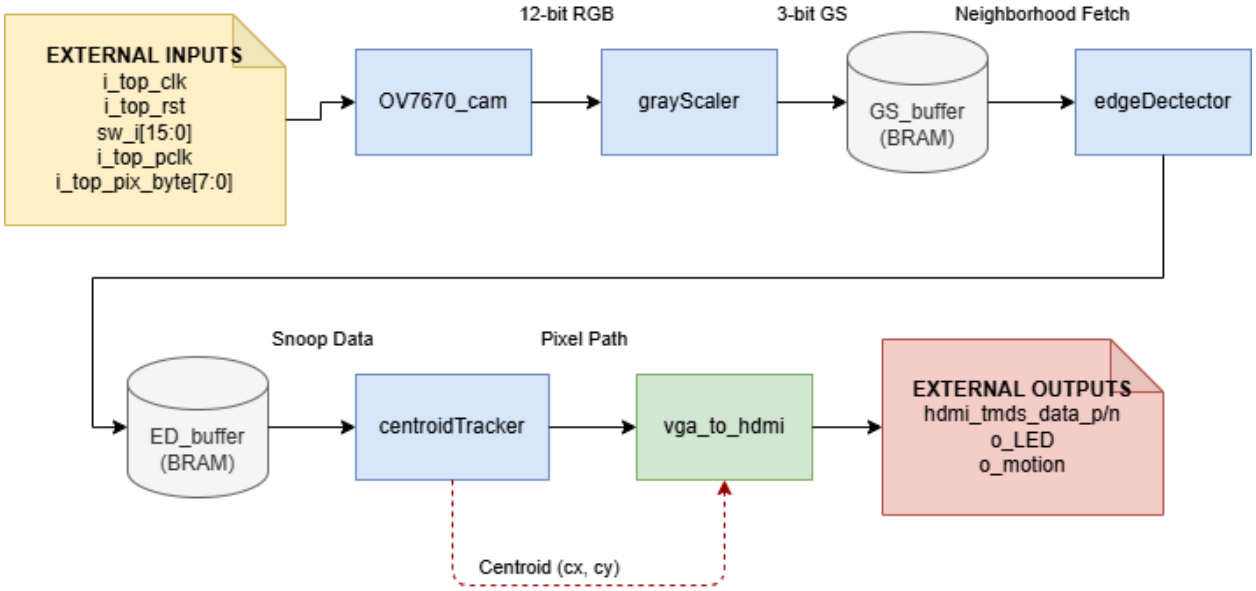


fig 9 -High Level Motion Detector Pipeline

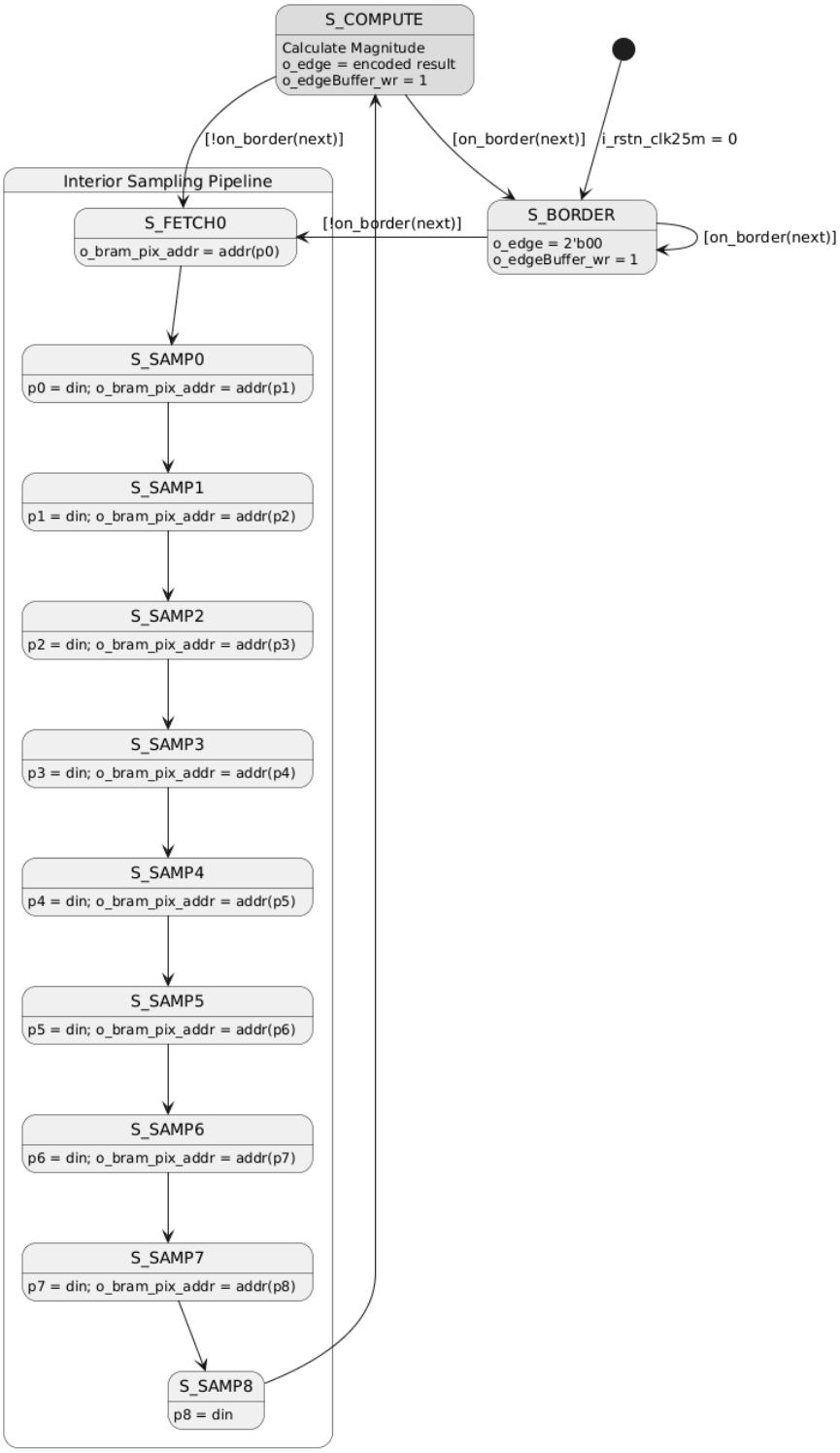


Figure 10 - Edge detector FSM

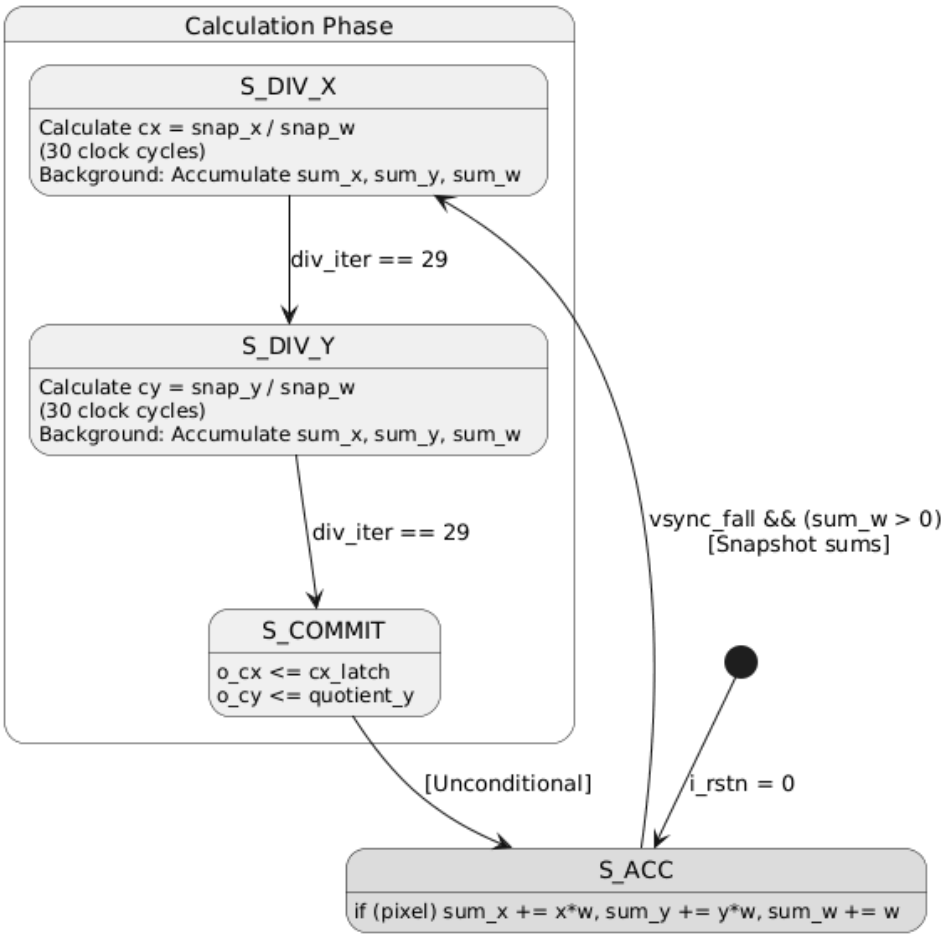


Figure 11 - CenterTracker FSM

Module Description
Game Engine Description

System Top-Level & Control

Module: mb_usb_hdmi_top.sv

> **Inputs:** Clk, reset_rtl_0, [15:0] sw_i, i_motion, [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, hex_segB, [3:0] hex_gridA, hex_gridB
>
> **Description:** This is the system's main top-level module. It integrates a MicroBlaze soft processor with hardware accelerators for game logic and graphics. It handles USB connectivity for peripherals, motion sensor input, HDMI video output, and debug information displayed via Hex displays.
>
> **Purpose:** This module serves as the primary integration point, connecting the processor's software control with the high-speed hardware logic required for real-time game rendering and physics.

Module: mb_intro_top.sv

> **Inputs:** clk, [3:0] btn, [15:0] sw, uart_txd **Outputs:** [15:0] led, uart_rxd
>
> **Description:** A simplified top-level wrapper used for MicroBlaze initialization and testing. It maps basic I/O like switches, buttons, and LEDs to the processor's GPIO interface.
>
> **Purpose:** This module is used for system bring-up and verifying the communication between the FPGA fabric and the MicroBlaze processor before full game engine deployment.

Game Logic & Physics

Module: ball.sv

> **Inputs:** db, [3:0] collision, Reset, frame_clk, [31:0] keycode, [15:0] SW
>
> **Outputs:** start_game, [9:0] BallX [4], [9:0] BallY [4], [9:0] BallS, [1:0] gamestate

Description: This module handles the physics and state for up to four "bird" players. It calculates vertical motion based on gravity and "jump" inputs (from keycodes or motion triggers), detects ground collisions, and manages the transitions between game states (IDLE, START, GAME OVER).

Purpose: It acts as the physics engine for the player characters, determining their screen positions and survival status during gameplay.

Graphics & Rendering

Module: spriteSheet_example.sv

Inputs: [15:0] SW, [1:0] game_state, vga_clk, [9:0] DrawX, DrawY, blank, [9:0] BirdX [4], [9:0] BirdY [4], [10:0] PipeX [5], [9:0] PipeY [5], [3:0] score_ones, score_tens, score_hundreds **Outputs:** [3:0] red, green, blue

Description: This module is the primary graphics controller. It uses current scan coordinates (DrawX, DrawY) to determine which sprite (bird, pipe, background, or UI numbers) should be rendered. It retrieves pixel indices from ROMs and maps them to 12-bit RGB colors using various palettes.

Purpose: This module generates the actual visual data for the game, handling sprite layering, transparency, and the real-time drawing of the score and environment.

Module: VGA_controller.sv

Inputs: pixel_clk, reset

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

Description: A standard VGA timing generator that creates horizontal and vertical synchronization signals. It provides the current X and Y coordinates of the pixel being drawn to the graphics engine.

Purpose: It establishes the timing framework for the 640x480 video signal, ensuring the monitor remains synchronized with the FPGA's output.

Module: BackGround_Base_palette.sv / bird_pipe_palette.sv / UI_palette.sv

Inputs: [4:0] index

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

Description: These modules act as Color Look-Up Tables (CLUT). They take a 5-bit pixel index from a sprite ROM and output the corresponding 4-bit Red, Green, and Blue values.

Purpose: These modules allow for efficient memory usage by storing small bit-depth indices in ROM while still outputting high-quality colors for the background, characters, and UI.

Peripheral Drivers

Module: hex_driver.sv

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

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

Description: This module converts four 4-bit nibbles into the 7-segment patterns required to display hexadecimal digits. It uses a multiplexing scheme to drive multiple 7-segment displays using a shared bus.

Purpose: It is used to provide real-time hardware debugging and status information, such as displaying game scores or internal state codes on the FPGA's physical displays.

Motion Detector

System Top-Level & Control

Module: top.sv

Inputs: i_top_clk, i_top_rst, i_top_cam_start, [15:0] sw_i, i_test_out, i_top_pclk, [7:0] i_top_pix_byte, i_top_pix_vsync, i_top_pix_href

Outputs: o_top_cam_done, o_LED, o_motion, o_top_reset, o_top_pwdn, o_top_xclk, o_top_siod, o_top_sioc, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p

Description: This is the top-level module that instantiates and connects all sub-modules, including the camera interface, image processing pipeline (grayscale, edge detection, centroid tracking), and HDMI output. It handles clock generation, system resets, and user switch mapping for thresholds and modes.

Purpose: This module serves as the structural glue of the project, defining how data flows from the camera sensor to the final video display and motion indicators.

Module: signalRouter.sv

Inputs: [2:0] GS_signal, [2:0] ED_signal, [18:0] ED_read_addr, [18:0] VGA_read_addr, i_input_switch

Outputs: [18:0] GS_read_addr, [2:0] vga_input

Description: This module is a multiplexer that routes either the raw grayscale stream or the processed edge-detected stream to the VGA output based on a switch input. It also manages memory address routing to ensure the correct buffer is accessed for the selected display mode.

Purpose: This module allows the user to toggle between viewing the standard camera feed and the edge-detection diagnostic view on the monitor.

Module: debounce.sv

Inputs: i_clk, i_btn_in

Outputs: o_btn_db

Description: This module implements a counter-based debouncer for mechanical button inputs. It requires an input signal to remain stable for a specific number of clock cycles (defined by a DELAY parameter) before updating the output.

Purpose: This module prevents noisy mechanical switch transitions from causing multiple false triggers in the system's control logic.

Image Processing Pipeline

Module: gray_scaler.sv

Inputs: [11:0] din

Outputs: [2:0] avg_scale

Description: This module takes 12-bit RGB camera pixel data and converts it into a 3-bit grayscale value. It implements a weighted average calculation using the formula: $(red \times 77 + green \times 150 + blue \times 29) \gg 9$.

Purpose: This module is used to reduce the color depth of the incoming camera stream to a single grayscale intensity, simplifying subsequent edge detection.

Module: edge_detector.sv

Inputs: i_clk25m, i_rstn_clk25m, [5:0] i_thresh_weak, [2:0] din

Outputs: [18:0] o_bram_pix_addr, [18:0] o_edgeBuffer_pix_addr, [1:0] o_edge, o_edgeBuffer_wr

Description: This module implements a Sobel 3x3 edge detection algorithm running at 25 MHz. It fetches a 3x3 neighborhood of grayscale pixels from memory, calculates horizontal and vertical gradients, and determines edge strength based on user-defined thresholds.

Purpose: This module identifies boundaries within the camera frames, providing the filtered data necessary for the centroid tracker to locate objects.

Module: centerTracker.sv

Inputs: i_clk25m, i_rstn, i_vsync, [18:0] i_vga_addr, [1:0] i_ed_pixel

Outputs: [18:0] o_bram_addr, [1:0] o_ed_pixel, [9:0] o_cx, [8:0] o_cy

Description: This module calculates the centroid (center of mass) of detected edges in real-time. It snoops the edge pixel stream, accumulates weighted x and y positions, and performs division at the end of each frame to output stable (c_x, c_y) coordinates.

Purpose: This module provides the coordinates of the primary object in the frame, used to draw a crosshair and detect movement between frames.

Module: motionIO.sv

Inputs: i_clk25m, i_rstn, [9:0] i_cx, [8:0] i_cy, [5:0] i_thresh_motion, i_vsync

Outputs: motionLED, motionPMOD

Description: This module compares the current frame's centroid (c_x, c_y) with the previous frame's centroid. If the distance exceeds a threshold, it triggers a pulse-stretched output on an LED and a PMOD pin, including a cooldown period to prevent rapid re-triggering.

Purpose: This module acts as the final output stage, providing physical indicators (LED/PMOD) when significant motion is detected.

Camera Interface & Configuration

Module: cam_top.v

Inputs: i_clk, i_rstn_clk, i_rstn_pclk, i_cam_start, i_pclk, [7:0] i_pix_byte, i_vsync, i_href

Outputs: o_cam_done, o_reset, o_pwdn, o_siod, o_sioc, o_pix_wr, [11:0] o_pix_data, [18:0] o_pix_addr

Description: A top-level wrapper for the camera subsystem. It instantiates the debouncer for the start button, the initialization controller (cam_init), and the pixel capture logic (cam_capture).

Purpose: This module integrates all camera functions, providing the main system with a clean interface for captured data and hardware control.

Module: cam_capture.v

Inputs: i_pclk, i_vsync, i_href, [7:0] i_D, i_cam_done

Outputs: [18:0] o_pix_addr, [11:0] o_pix_data, o_wr

Description: Captures raw pixel data from the camera sensor in the pclk domain. It implements an FSM that waits for vsync, assembles bytes into 12-bit RGB pixels, and generates write addresses for memory.

Purpose: Translates camera hardware timing signals into a structured stream of pixel data for the video buffer.

Module: cam_init.v

Inputs: i_clk, i_rstn, i_cam_init_start

Outputs: o_siod, o_sioc, o_cam_init_done, o_data_sent_done, [7:0] o_SCCB_dout

Description: A structural module that connects cam_rom, cam_config, and sccb_master. It encapsulates the entire camera initialization subsystem.

Purpose: Provides a single interface to trigger camera hardware setup and signals when the sensor is ready to stream.

Module: cam_config.v

Inputs: i_clk, i_rstn, i_i2c_ready, i_config_start, [15:0] i_rom_data

Outputs: [7:0] o_rom_addr, o_i2c_start, [7:0] o_i2c_addr, [7:0] o_i2c_data, o_config_done

Description: Manages the sequencing of camera register configuration. It reads address/data pairs from ROM and coordinates with the SCCB master, including a 10ms delay timer for initialization.

Purpose: Automates the complex initialization process required to set the camera to RGB444 mode and specific resolution.

Module: sccb_master.v

Inputs: i_clk, i_rstn, i_read, i_write, i_start, i_restart, i_stop, [7:0] i_din, [7:0] i_addr

Outputs: [7:0] o_dout, o_ready, o_done, o_ack, o_scl

Inouts: io_sda

Description: Implements the SCCB (Serial Camera Control Bus) protocol. It manages the start, stop, and data transmission phases required to communicate with OmniVision sensors.

Purpose: Used to configure internal registers of the OV7670 camera during system initialization.

Module: cam_rom.v

Inputs: i_clk, i_rstn, [7:0] i_addr

Outputs: [15:0] o_dout

Description: A Look-Up Table (LUT) containing specific register addresses and data values required to configure the OV7670 camera sensor.

Purpose: Stores the fixed configuration settings (reset, clock scaling, color format) for the camera hardware.

Video Display (VGA)

Module: vga_top.v

Inputs: i_clk25m, i_rstn_clk25m, [2:0] i_pix_data

Outputs: [9:0] o_VGA_x, [9:0] o_VGA_y, o_VGA_vsync, o_VGA_hsync, o_VGA_video, o_VGA_red, o_VGA_green, o_VGA_blue, active_nblank, [18:0] o_pix_addr

Description: Top-level controller for the VGA display. It instantiates the driver to generate timing and manages memory addressing to fetch pixel data from BRAM based on current scan coordinates.

Purpose: Coordinates the reading of processed image data from memory and converting it into standard VGA timing and color signals.

Module: vga_driver.v

Inputs: i_clk, i_rstn

Outputs: [9:0] o_x_counter, [9:0] o_y_counter, o_video, o_hsync, o_vsync

Description: Generates standard 640x480 @ 60Hz VGA timing signals. It maintains horizontal and vertical counters and generates the sync pulses (hsync, vsync) and video blanking signals.

Purpose: Provides the master timing reference for the video display, defining exactly where pixels appear on the screen.

Working Result

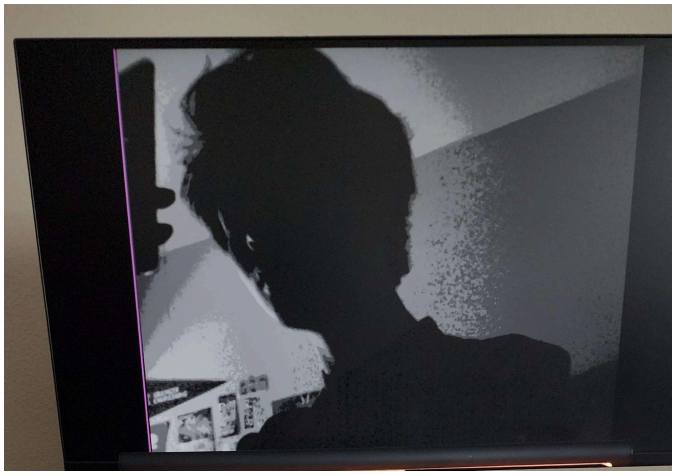


Figure 12 - Gray Scale Output

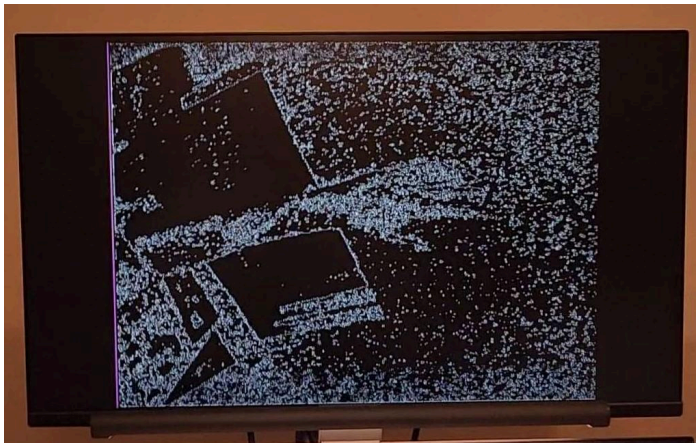


Figure 13 - Edge Detection

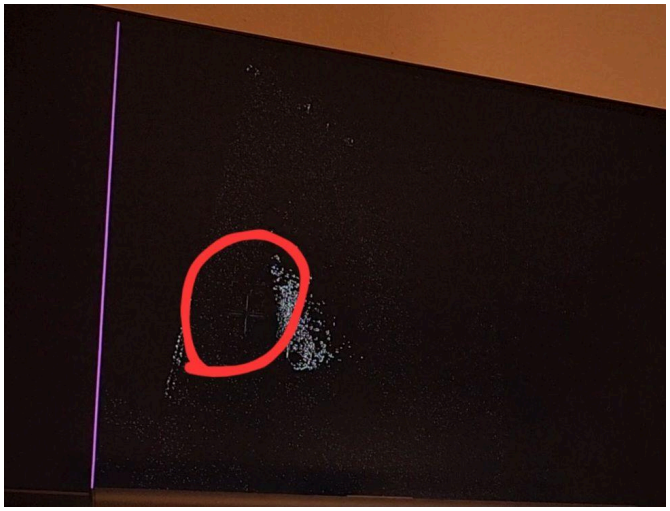


Figure 14 - Tracking Cursor Overlay

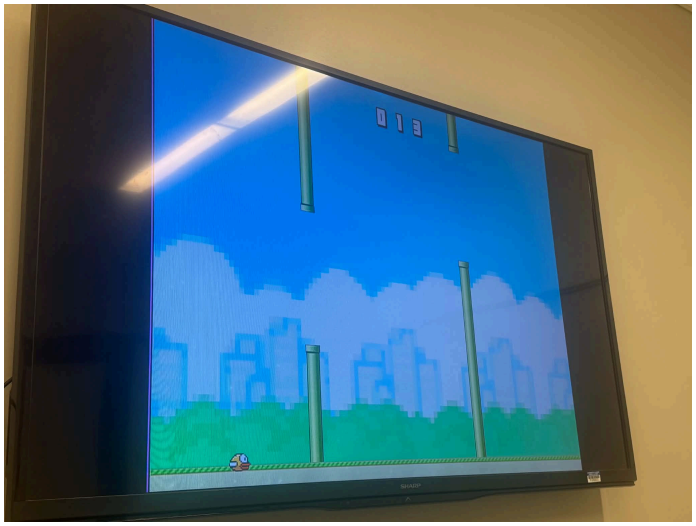


Figure 15 - Gameplay

*Due to the mass amount of physical inputs that are required and the inherently visual property of our project, most of the debugging was done visually by observing the behavior of either the game engine or the visual output as shown in the pictures above, no test bench was used for the two units.

Metrics

Table 1 - Game Engine Metrics

DSP	22
Memory (BRAM)	22.5
Flip-Flop	3002
Frequency	116.13 MHz
Static Power	0.076 W
Dynamic Power	0.409 W
Total Power	0.485 W

Table 2 - Motion Detector Metrics

DSP	3
Memory (BRAM)	48
Flip-Flop	840

Frequency	74.62Mhz
Static Power	0.075
Dynamic Power	0.265w
Total Power	0.34w

Conclusion

In conclusion, we have demonstrated and applied our knowledge from this semester to create a working project. With the base being the result of lab 6.2, we have added features to it to have a working Flappy Bird game, with an option for multiple players and cross FPGA motion control. What makes our project stand out is the motion tracking control option, where players can move with the bird. While our personal goal of reaching the final showcase was not fulfilled, it is undeniable that this semester has been amazing and memorable. Shoutout to Professor Cheng, our Teaching Assistant Peidong “PY” Yang, the Course Assistants (Sam, Sreeram, and SpiderDerp) for teaching and helping us, it would not be possible without all of your help and we are grateful for it.

Credits

- ECE 385 Lab6.2 based code
- COE file generation
    - https://github.com/amsheth/Image_to_COE
- Flappy bird sprite
    - https://www.spriters-resource.com/mobile/flappybird/asset/59894/
- Motion control PMODs pinout
    - https://github.com/souvlakiboi/OV7670_camera
- Basic camera function verilog code
    - https://github.com/amsacks/OV7670-camera
- Motion control reference
    - https://ieeexplore.ieee.org/document/9792984
- Pipeline reference
    - https://www.youtube.com/watch?v=iIuA-HQGytk