Final Project Paper: Pet Rock Game VGA

Giselle Wu, Lijia Yang

Dartmouth College

ENGS 31 / CoSc 56

Abstract

This report elaborates on the overview, design, implementation, and validation of the VGA system of a Pet Rock Game. Inspired by Tamagotchi, the system simulates caring for a virtual pet displayed on a monitor, with user interaction through pushbuttons. The design is divided into two subsystems: a behavioral shell, which uses finite state machines (Life FSM, Mood FSM, Time Manager, and Coin Manager) to manage the pet's states, timers, and coin system, and a graphics shell, which generates VGA signals and displays sprite images stored in BRAM. The system was implemented in VHDL on a Basys3 FPGA and tested through both simulation and hardware. Simulations confirmed correct state transitions, coin handling, and edge-case behavior, while a hardware demo validated full integration on the board. Resource utilization showed that BRAM (75%) was the main limiting factor. Overall, the project successfully integrated RTL design, FSMs, and VGA graphics into a functional interactive game.

1. System Overview

The goal of our VGA Pet Rock Game project was to design and implement an interactive digital system that simulates the experience of caring for a virtual pet. Inspired by Tamagotchi, the design involves a “pet rock” on a VGA display and allows users to interact with it through physical pushbuttons on an FPGA board. Our objective was to create a responsive, visually clear design that integrated the main topics we learned, RTL design, VGA, graphics, and hardware implementation, into a single cohesive project.

The system accepts several inputs: Start, Pet, Feed, Revive, and Quit pushbuttons, as well as a switch that serves both as the video-on control and the reset signal. Based on these inputs, the pet rock moves through behavioral states including Happy, Bored, Mad, Sad, and Dead. Each state is represented by a distinct graphic on the VGA screen, and transitions occur through timed events or player interaction. The design also handles edge cases such as reviving the pet after it has “died” and blanking the display when the video signal is turned off.

The system is composed of two primary subsystems:

- **Behavioral Subsystem**: Implements the pet’s logic using finite state machines and timers to track moods, user interaction, and life cycle.
- **Graphics Subsystem**: Generates VGA signals, stores and retrieves image data, and displays the pet and UI elements according to the current state.

This combination results in an engaging and visually clear interface while reinforcing core digital design principles.

1.1 Top-level Block Diagram

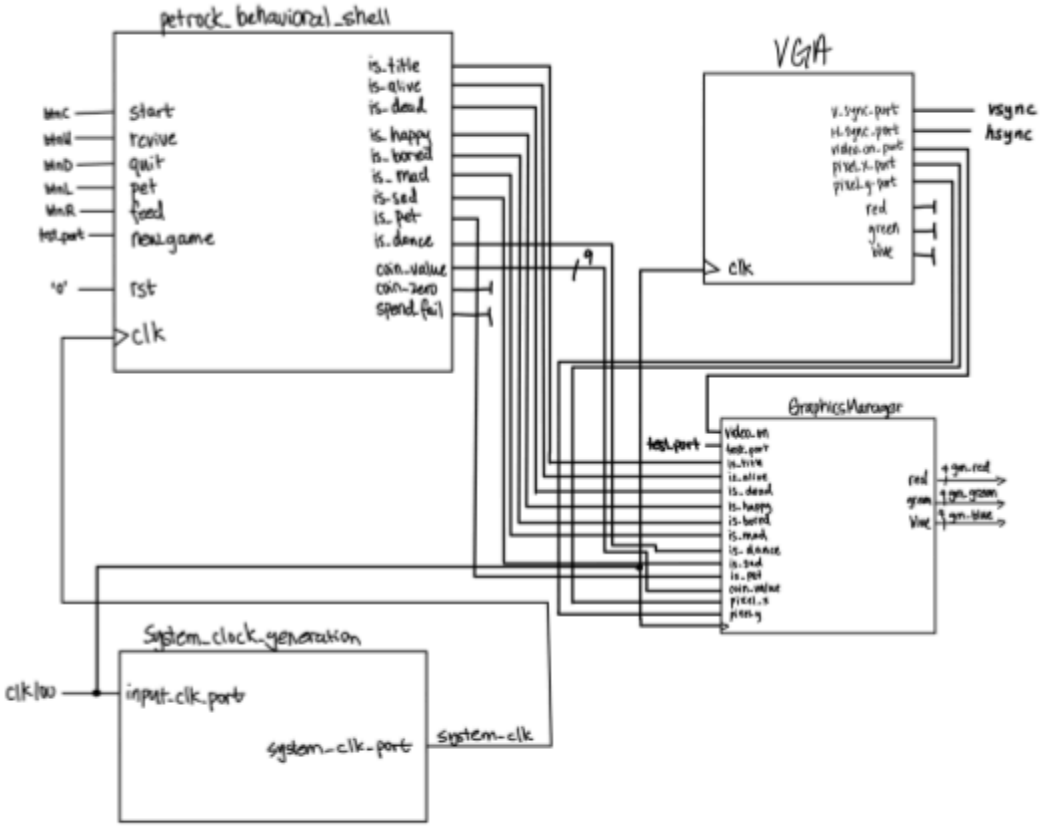


Figure 1: Top-Level Shell

1.2 Description of Ports

1.2.1 top_vga.vhd

1.2.1.1 Inputs

Port	Width	Description
btnC	1	Start. A rising edge starts from Title (or is OR'ed with new_game).
btnU	1	**Revive**. Rising edge revives only when dead (gated in shell).
btnD	1	**Quit.** Rising edge returns to Title from any state.
btnL	1	**Pet.** Rising edge triggers Pet action; mood may return to Happy.
btnR	1	**Feed.** Rising edge spends a coin (if available) and affects mood

		when alive.
test_port	1	Switch that controls the display. **0 = screen blank (black), 1 = video on.** In some versions, a rising edge here at the Title screen also starts a new game. If the game samples this on a slow tick, keep it high for at least one tick so it's detected
clk100	1	100 MHz board clock. Feeds the VGA timing core and the clock divider that makes the slow "game/system" clock. Sampled on the rising edge

1.2.1.2 Outputs

Port	**Width**	**Description**
hsync	1	VGA horizontal sync from the VGA timing block. Wire straight to the VGA connector.
vsync	1	VGA vertical sync. Wire to VGA connector.
red	4	4-bit red color output for the VGA
green	4	4-bit green color output for the VGA
blue	4	4-bit blue color output for the VGA

1.3 Description of Components

1.3.1 petrock_behavioral_shell.vhd

The behavioral shell is where all the game logic lives for the pet rock game. It processes button inputs and coordinates the Life FSM, Mood FSM, timer, and coin logic to decide the pet's current state and coin.

1.3.2 system_clock_generation.vhd

The System Clock Generation block divides the incoming board clock down to a slower system clock that feeds the behavioral portion of this project. It uses a counter and flip-flop to toggle the clock output at the desired frequency and buffers it onto the FPGA. It takes the 100MHz clock

and turns it into a 60Hz, producing a clean, slow system clock with a ~50% duty cycle. This lets the rest of the game logic run at a speed appropriate for user interaction rather than the raw FPGA clock.

1.3.3 GraphicsManager.vhd

GraphicsManager selects a frame based on game state — Title /Happy /Bored /Mad /Sad /Pet /Dance /Dead — and fetches pixels from BRAM. It also drives the VGA RGB outputs, blanking to black when test_port=0 or video_on=0. It maps the 640×480 active area to a 160×120 sprite through 4× scaling, forms the ROM address, expands 3:3:3 RGB to 4-bit channels, and overlays a 2-digit gold coin counter.

1.3.4 VGA.vhd

The VGA module takes the 100 MHz board clock and slows it down to a 25 MHz pixel clock so it can talk to a standard 640×480 display. It keeps track of where the screen is drawing using horizontal and vertical counters, turns on HSYNC and VSYNC at the right times, and only enables video_on during the visible area. It also gives the x and y positions of the current position.

2. Technical Description

This section explains **how the system is built and how the pieces work together**. We start at the top level (Figure 1) and then walk through each instantiated component, describing its role, interfaces, and any important internal submodules.

2.1 petrock_behavioral_shell.vhd

The behavioral shell is where the game logic lives. It reads the buttons and switches, makes one-clock pulses from them, and coordinates the four blocks inside the game:

- **life_fsm** — determines 3 states — Title / Alive / Dead
- **mood_fsm** — determines mood states — Happy / Bored / Mad / Sad / Pet / Dance — and sends state signals to start the timer in time_manager
- **time_manager** — runs state timers and sends "*_done" signals back to the mood_fsm
- **coin_manager** — tracks coins, spending, and earning

2.1.1 Description of Ports

2.1.1.1 Inputs

Port	Width	Description
start	1	Press to start the game from the Title screen. A rising edge starts play.
revive	1	Press to revive the pet when it's dead. Ignored at other times.
quit	1	Rising edge returns to Title from any state.
pet	1	Press to pet the pet when it is happy, bored, or mad. Returns to the previous state at bored and mad, and restart the happy state when in happy.
feed	1	Press to feed the pet using 3 coins when it is alive. Ignored when dead.
new_game	1	Start a fresh session from the Title screen. Resets timers and re-loads starting coins. Implemented on a rising edge at Title.
clk	1	System clock for the shell.
rst	1	Reset. Clears internal registers/edge detectors and returns the game to a known state (Title logic on next start/new_game).

2.1.1.2 Outputs

Port	**Width**	**Description**
is_title	1	1 when the game is on Title screen
is_alive	1	1 when the rock is alive
is_dead	1	1 when the rock is dead
is_happy	1	1 when the rock is happy
is_bored	1	1 when the rock is bored
is_mad	1	1 when the rock is mad
is_sad	1	1 when the rock is sad
is_pet	1	1 during the Petting graphics
is_dance	1	1 during the Dance graphics
coin_value	9	Current coin count. Used by on-screen display and game logic, such as feed and revive
coin_zero	1	1 when coin value is 0
spend_fail	1	1 when try to spend a coin but not enough is available

2.1.2. Register Transfer Level (RTL) Diagram

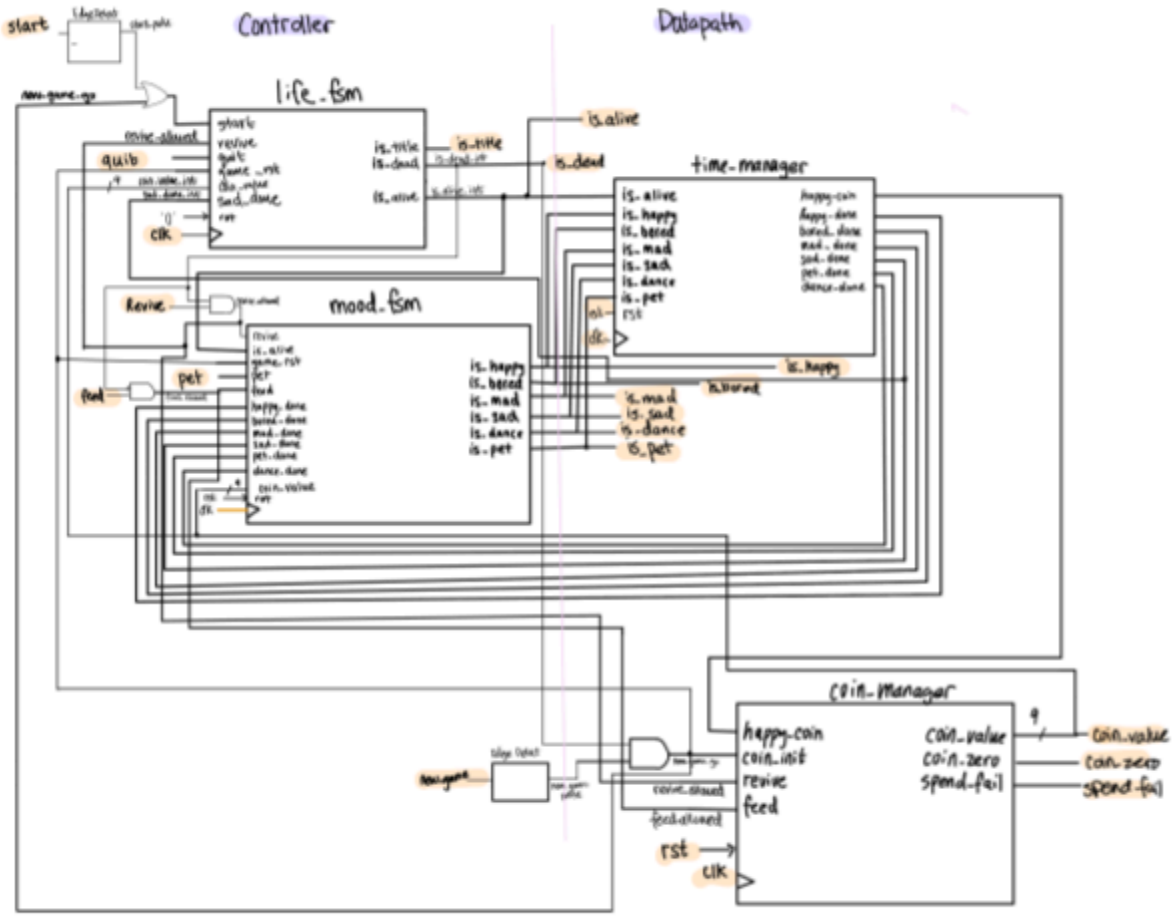


Figure 2: RTL Diagram for petrock_behavioral_shell.vhd

The shell splits cleanly into a Controller (state machines) and a Datapath (timers + coin registers). All inputs are edge-detected into one-clock pulses, then gated by state, such as revive is only allowed when dead. For edge-detection that isn't drawn on this RTL diagram, it is conducted within subcomponents.

2.1.2.1 Controller (left)

- **life_fsm** — picks the high-level state: Title / Alive / Dead.
    - In: start, new_game, quit, sad_done.
    - Out: is_title, is_alive, is_dead.

- **mood_fsm** — refines Alive into Happy / Bored / Mad / Sad and actions Pet / Dance.
    - In: pet, feed, revive, *_done, coin_value, game_rst.
    - Out: is_happy, is_bored, is_mad, is_sad, is_pet, is_dance.

2.1.2.2 Datapath (right)

- **time_manager** — enables the timer for the active mood, raises one-cycle *_done pulses, generates happy_coin while is_happy=1.
- **coin_manager** — adds on happy_coin, subtracts from feed and revive if it is allowed.
    - Out: coin_value, coin_zero, spend_fail.

2.1.2.3 Important feedback loops

- is_* → timers run → *_done → mood changes.
- is_happy → happy_coin → coins++ → coin_value informs mood rules and displays.
- sad_done → life_fsm → Dead → only revive allowed.

2.1.2.4 Resets

- rst = global reset.
- Allowed new_game creates a one-cycle game_rst and coin_init to re-initialize timers and mood and reload coins.
- start_to_life = start or new_game begins play from Title.

2.1.2.5 Typical flow

- Start → Alive and Happy.
- Timers tick; coins accumulate via happy_coin.
- Inactivity: Happy → Bored → Mad → Sad via *_done.
- Actions: feed/pet can alter mood, spending checks coins (spend_fail if zero).
- sad_done → Dead; revive returns to Alive.

- New Game resets and starts a fresh session.

2.1.3. Finite State Machine (FSM) Diagram

2.1.3.1 Life Finite State Machine

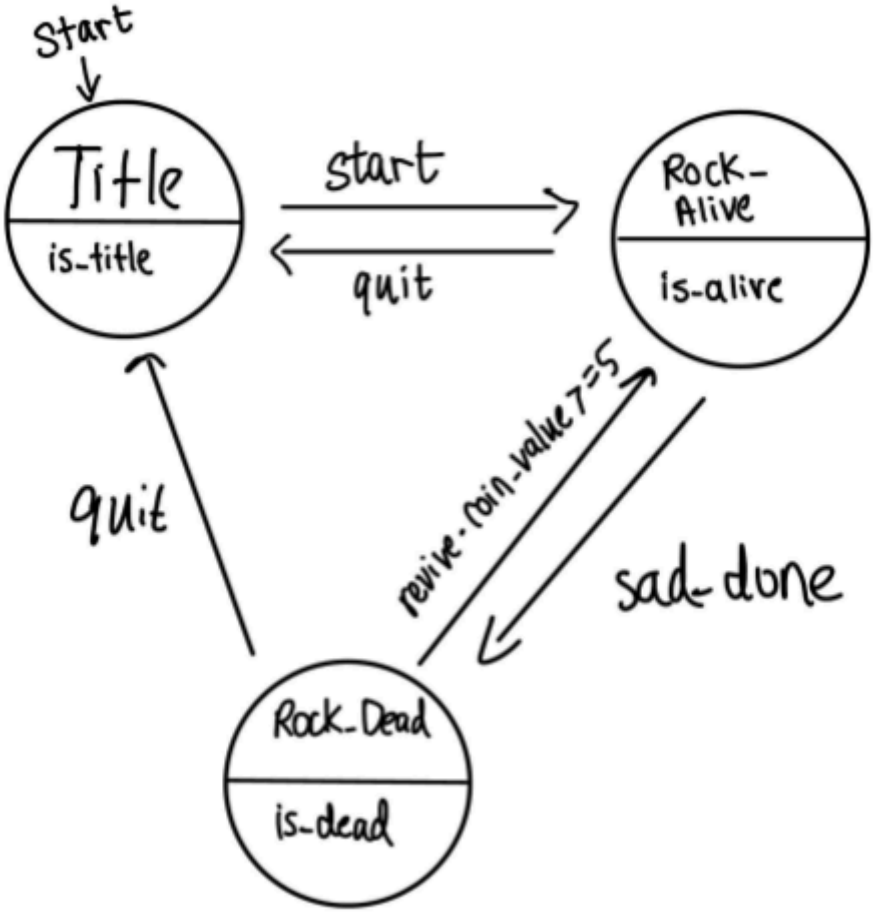


Figure 3: Life Finite State Machine

States & outputs:

- Title → is_title=1
- Rock_Alive → is_alive=1
- Rock_Dead → is_dead=1

Inputs: start, quit, revive, sad_done

Transitions:

- Title → Alive: on start or qualified new_game (also issues game_rst/coin_init).
- Alive → Dead: on sad_done.
- Alive → Title: on quit.
- Dead → Alive: on revive (and coins ≥ cost, if used).
- Dead → Title: on quit.

Priority (if simultaneous): rst/game_rst > quit > revive (in Dead) > start/new_game (in Title) > sad_done.

Role: Drives the high-level lifecycle. The outputs of this FSM gate actions elsewhere and determine what the display shows.

2.1.3.2 Mood Finite State Machine

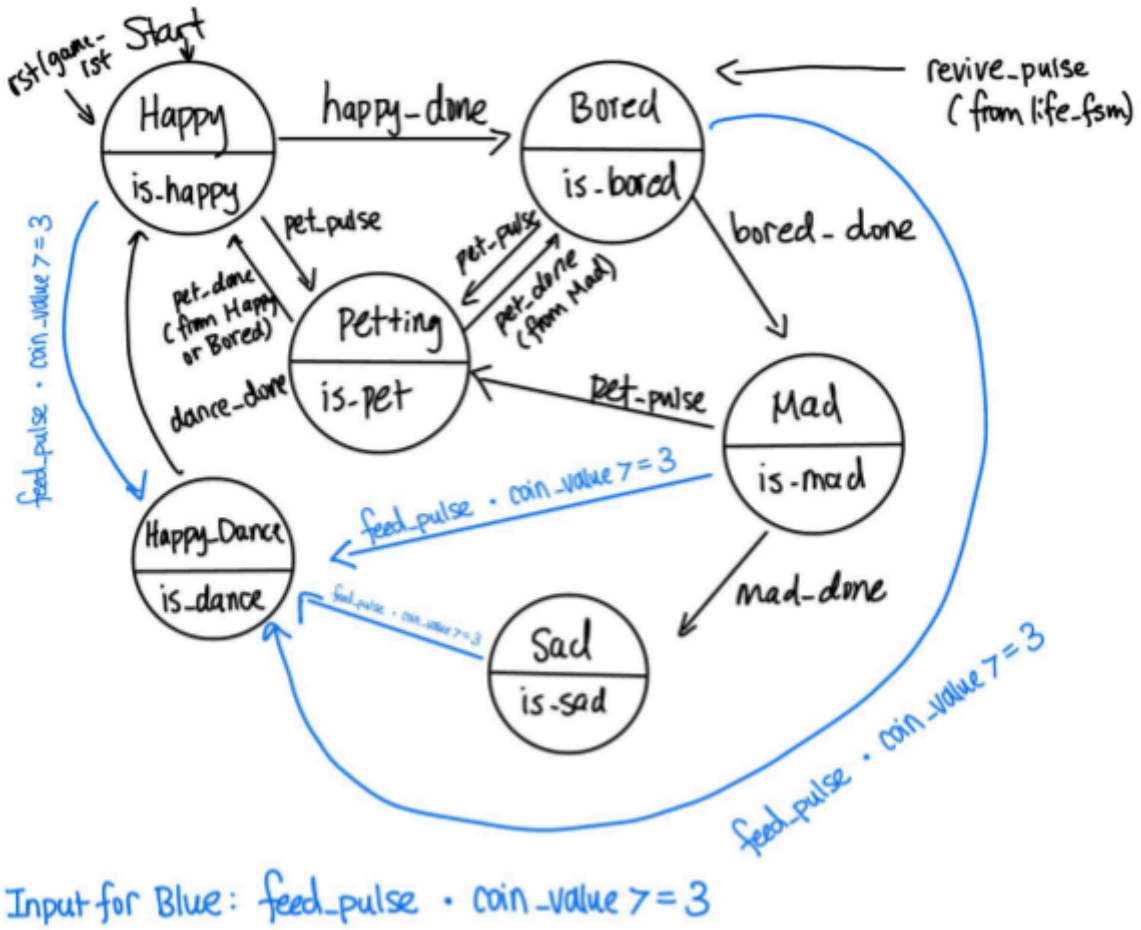


Figure 4: Mood Finite State Machine

States & outputs:

- Happy → is_happy = 1
- Bored → is_bored=1
- Mad→ is_mad=1
- Sad → is_sad=1
- Petting → is_pet=1
- Happy_Dance → is_dance=1

Inputs:

- **Buttons** are edge-detected inside the FSM: pet_pulse, feed_pulse.
- **Timers**: happy_done, bored_done, mad_done, dance_done, pet_done.
- **Other**: revive_pulse (only when Life FSM just revived from Dead), coin_value.

Transitions:

- Highest-priority
    - rst | game_rst → Happy (reset).
    - revive_pulse → Bored (happens only when the game revives from Dead).
    - feed_pulse & coin_value ≥ 3 & state ≠ Petting → Happy_Dance.
- Timed chain
    - Happy --happy_done--> Bored
    - Bored --bored_done--> Mad
    - Mad --mad_done--> Sad
    - Sad: no internal exit (Life FSM handles death via sad_done).
- Pet action with remembered return
    - From Happy or Bored: pet_pulse → Petting, then pet_done → Happy.
    - From Mad: pet_pulse → Petting, then pet_done → Bored.
- Short actions
    - Happy_Dance --dance_done--> Happy.

Priority (if simultaneous): rst/game_rst > quit > revive (in Dead) > start/new_game (in Title) > sad_done.

Role: Drives the high-level lifecycle. The outputs of this FSM gate actions elsewhere and determine what the display shows.

2.1.4. Datapath Diagram

2.1.4.1 Time Manager

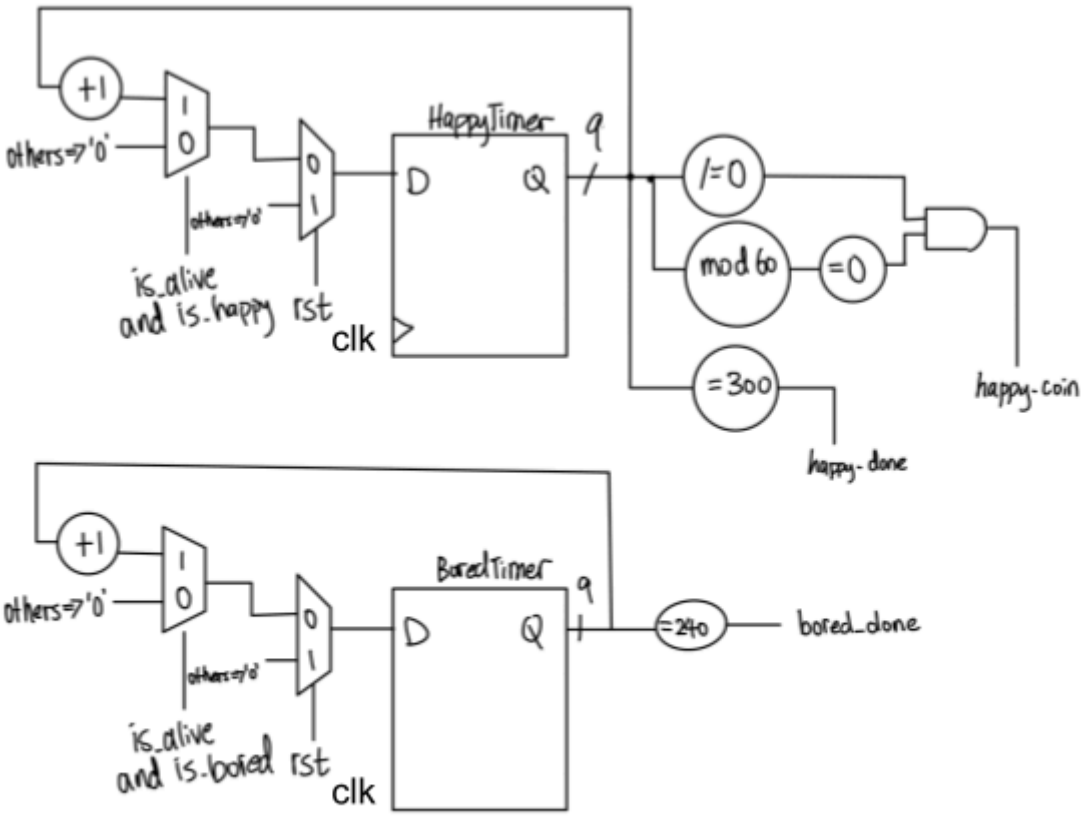


Figure 5(a): Time Manager Datapath for Happy and Bored Timers

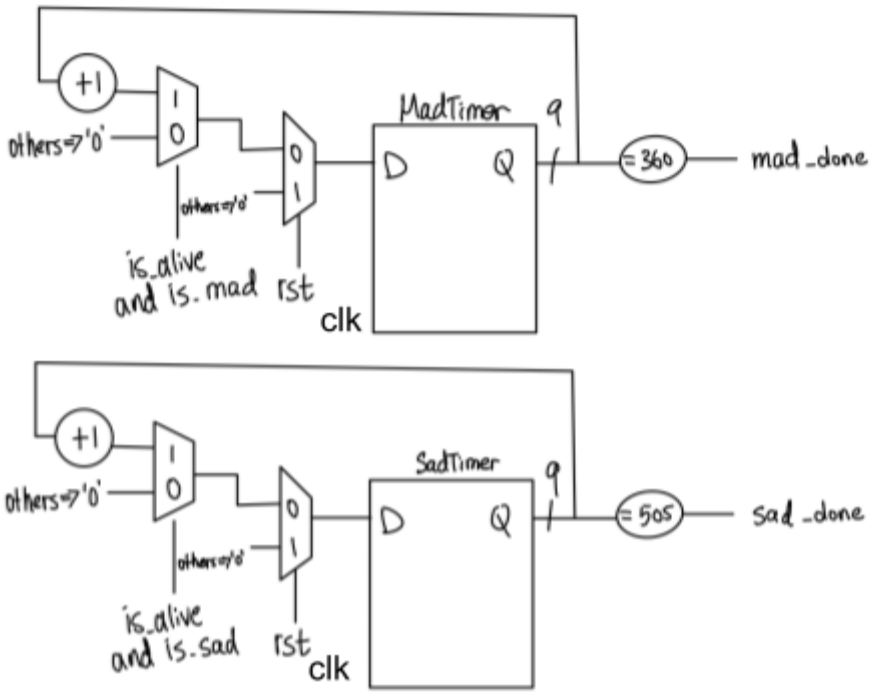


Figure 5(b): Time Manager Datapath for Mad and Sad Timers

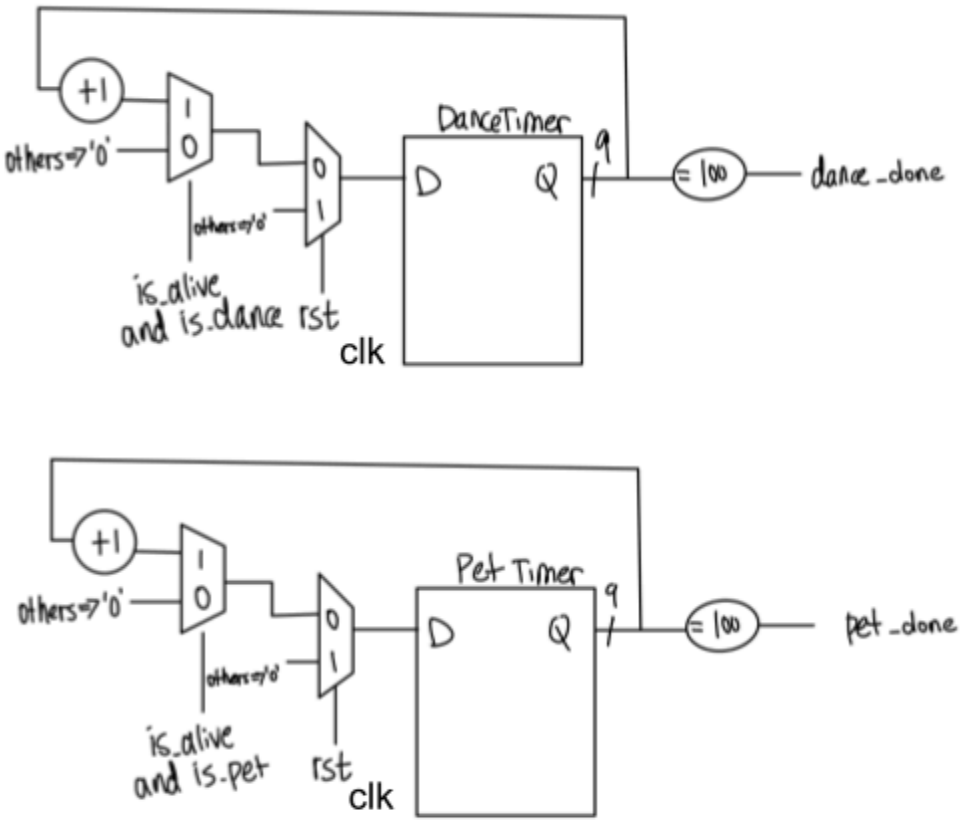


Figure 5(c): Time Manager clk
Datapath for Dance and Pet Timers

The Time Manager consists of 6 timers to determine the transition of states. Each timer is enabled only when is_alive='1' and its corresponding is_* state signal is high. While enabled, the timer counts clock ticks. When it reaches its terminal count, it emits a one-cycle *_done pulse that the FSM uses to request a state transition. All timers synchronously clear on rst and whenever their timer reaches the terminal count. The happy timer additionally generates a periodic happy_coin signal while is_happy remains high.

2.1.4.2 Coin Manager

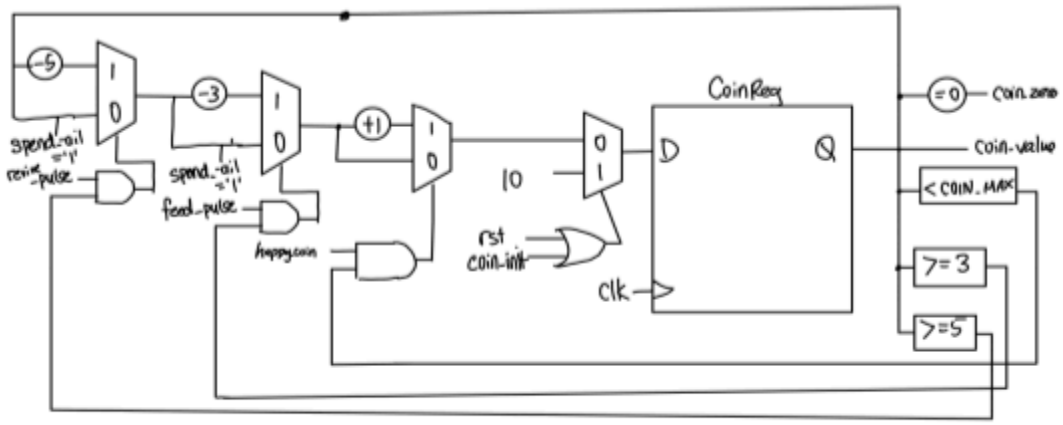


Figure 6: Coin Manager Datapath

This block keeps the game's coin balance in a register and applies exactly one action per clock: revive spends 5, feed spends 3, and happy earns +1. The spend paths are protected—revive only fires if the balance is ≥5 and feed only if it's ≥3; otherwise, the value simply holds, and we raise spend_fail for a cycle. Earning is capped, so the count never exceeds COIN_MAX. If two things happen together, we use a fixed priority (Revive > Feed > Happy) so the result is deterministic. On reset or a new game (coin_init), the register loads START_COINS. We also expose coin_zero (balance = 0) and reuse the ≥3/≥5 checks for gating and UI.

2.2 System_Clock_Generation.VHD

This block takes the 100 MHz board clock and slows it down to a "game" clock. It counts input cycles, flips the output every half-period so it's about a 50/50 square wave, and then runs that

signal through a BUFG so it's distributed cleanly across the FPGA. With the default divide, you get ~60 Hz.

2.2.1. Description of Ports

2.2.1.1 Inputs

Port	**Width**	**Description**
input_clk_port	1	The external source clock (100 MHz on Basys3). The divider counts the rising edges of this clock.

2.2.1.2 Outputs

Port	**Width**	**Description**
system_clk_port	1	The divided "game/system" clock. Near-50% duty cycle, and we run it through a BUFG so the clock reaches the whole FPGA cleanly

2.2.2. Register Transfer Level (RTL) Diagram

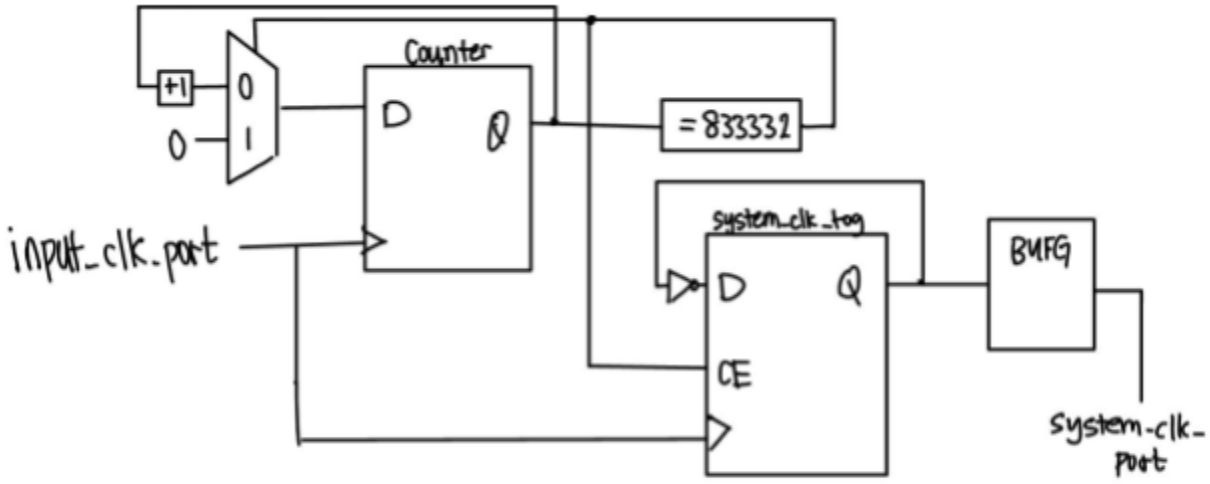


Figure 7: Clock Generation RTL

The counter ticks on each input clock edge; when it hits the terminal count it resets to 0 and flips a toggle flop. That toggled signal goes through a BUFG and becomes system_clk_port.

Counter

- Clock: input_clk_port
- D: Q + 1, else 0 when tc_hit.
- tc_hit comes from Comparator: Q == (TC−1).

Toggle Flip Flop

- Clock: input_clk_port
- D: not Q and tc

BUFG: Drives the toggled clock onto the global net → system_clk_port.

Frequency: Toggle every TC cycles → full period = 2·TC.

- f_out ≈ f_in / CLK_DIVIDER_RATIO

2.3 VGA.VHD

The VGA module is responsible for generating the necessary signals to drive a standard VGA display at 640×480 resolution. It takes the 100 MHz board clock as input and internally divides it down and create a 25 MHz pixel clock. It also manages the horizontal and vertical synchronization timing, keeps track of the current pixel coordinates, and controls the video_on signal to indicate when the display is within the visible area.

2.3.1. Description of Ports

2.3.1.1 Inputs

Port	**Width**	**Description**
clk	1	100 MHz board clock input. Uses this clock to derive the pixel clock (25 MHz)

		and synchronization signals.

2.3.1.1 Outputs

Port	**Width**	**Description**
V_sync_port	1	Vertical synchronization output. Connect to the VGA connector pin.
H_sync_port	1	Horizontal synchronization output. Connect to the VGA connector pin.
video_on_port	1	Indicates when the current pixel is within the visible display 640x480 area. Used by other modules to enable pixel rendering.
pixel_x_port	10	Current horizontal pixel coordinate (0 to 639)
pixel_y_port	9	Current vertical pixel coordinate (0 to 479)
red	4	4-bit red color channel output for VGA
green	4	4-bit green color channel output for VGA
blue	4	4-bit blue color channel output for VGA

2.3.2. Register Transfer Level (RTL) Diagram

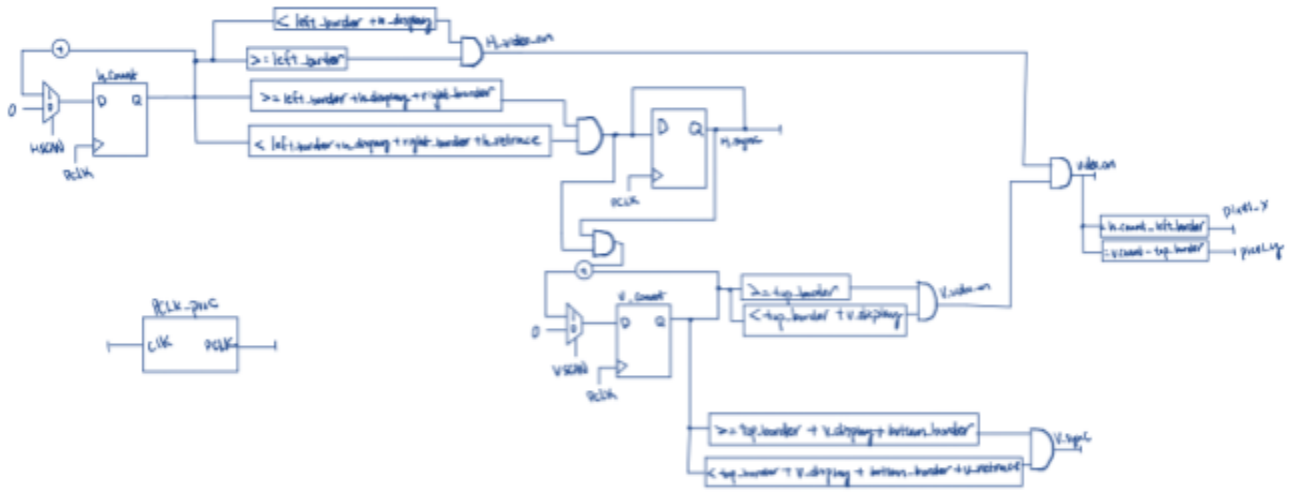


Figure 8: VGA RTL

VGA Horizontal and Vertical Counters

Horizontal Counter

- Clock: pixel clock (PCLK, derived from 100 MHz clk)
- D input: Q + 1 (increment horizontal pixel count)
- Reset: set to 0 when terminal count HSCAN is reached (end of horizontal line)
- Terminal count comparator: h_count == (HSCAN)

Vertical Counter

- Clock: rising edge of horizontal sync (H_sync)
- D input: Q + 1 (increment vertical line count)
- Reset: set to 0 when terminal count VSCAN is reached (end of frame)
- Terminal count comparator: v_count == (VSCAN)

H_sync and V_sync Signal Generation

H_sync is low when h_count is within the horizontal retrace interval

- (h_count >= left_border + h_display + right_border) and (h_count < left_border + h_display + right_border + h_retrace)

V_sync is low when v_count is within the vertical retrace interval

- (v_count >= top_border + v_display + bottom_border) AND (v_count < top_border + v_display + bottom_border + v_retrace)

Video Enable Logic

Horizontal video enables

- H_video_on = (h_count >= left_border) AND (h_count < left_border + h_display)

Vertical video enables

- V_video_on = (v_count >= top_border) AND (v_count < top_border + v_display)

Combined video enables

- video_on = H_video_on AND V_video_on

Pixel Coordinates Calculation

video_on is high

- pixel_x = h_count - left_border
- pixel_y = v_count - top_border

video_on is low

- Pixels = zeros

2.4 GraphicsManager.VHD/ ROM.VHD

The GraphicsManager is the display engine for the game. It takes the current state signals and pixel coordinates and outputs 4-bit VGA RGB. Each frame, it selects a 160×120 sprite from BRAM, scales it by 4 to 640×480, and draws it over a simple two-color background. Zero ROM pixels act as transparent. It also renders the on-screen coin count (0–99) with a small bitmap font. If video_on is low or test_port is off, it blanks the screen to black.

2.4.1. Description of Ports

2.4.1.1 Inputs

Port	**Width**	**Description**
clk	1	Pixel clock for synchronizing ROM and logic
video_on	1	Active video area
test_port	1	Control signal to toggle display output on/off
is_title	1	Title screen state
is_alive	1	Pet is alive
is_dead	1	Pet is dead

is_happy	1	Pet is happy
is_bored	1	Pet is bored
is_mad	1	Pet is mad
is_dance	1	Feeding/coin transition
is_sad	1	Pet is sad
is_pet	1	Petting transition
coin_value	9	Current coin count displayed on screen
pixel_x	10	Horizontal pixel coordinate (0 to 639)
pixel_y	9	Vertical pixel coordinate (0 to 479)

2.4.1.1 Outputs

Port	**Width**	**Description**
red	4	4-bit red color channel output for VGA
green	4	4-bit green color channel output for VGA
blue	4	4-bit blue color channel output for VGA

2.4.2. Register Transfer Level (RTL) Diagram

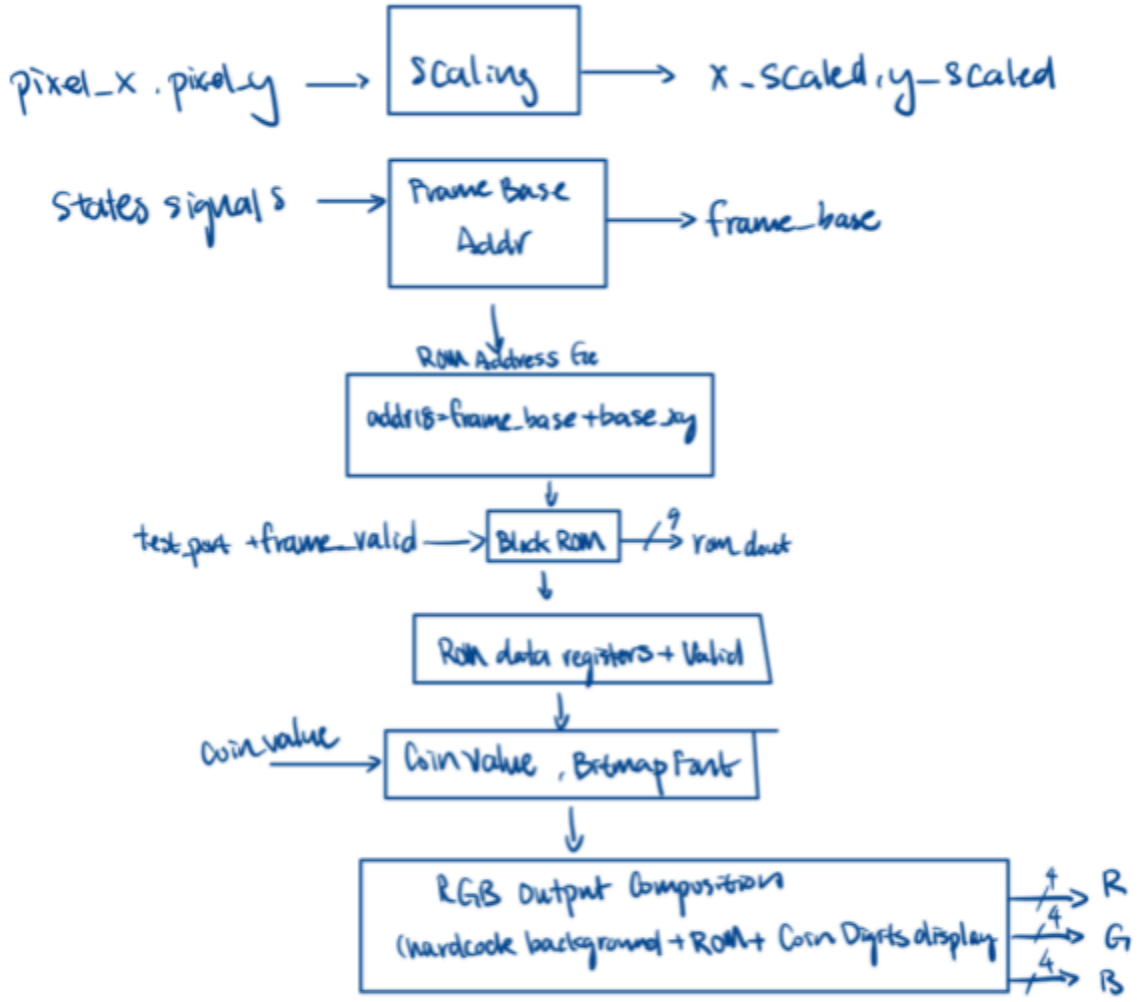


Figure 9: VGA Datapath

2.4.4. Graphics Creation and Description of Memory

To develop the graphics of the Pet Rock game, we initially designed pixel art images at a resolution of 160×120 pixels for each pet state, such as Start, Happy, Bored, Mad, Sad, Petting, Food, and Dead. Each image was exported as a PNG file. After obtaining these PNG files, we converted them into a format suitable for FPGA memory initialization, where we wrote two MATLAB scripts to automate this process:

2.4.4.1 pngToCoe.mlx

It performs the task of converting PNG images into COE files suitable for FPGA ROM initialization. Each PNG image corresponds to a 160×120 pixel sprite representing a pet state, with each pixel storing RGB information. To save memory space on the FPGA, the script reduces the color depth by extracting only the top 3 bits from each RGB channel, resulting in each pixel being represented by 9 bits. Resized the PNG image to ensure it matches the target dimensions (160x120) used in the hardware. The generated COE file can be used directly to initialize block ROMs in the FPGA design.

2.4.4.2 combinedCoe.m

This serves to concatenate multiple COE files into a single combined COE file. This combined COE file is used to initialize the FPGA memory block that stores all sprite images, with each sprite's data located at known offsets within the memory, indexed by the game state.

2.4.4.3 ROM

These images are stored in a read-only memory (ROM) block inside the FPGA, instantiated as the blk_mem_gen_2 component in the VHDL

- Width: 9 bits (split into 3 bits each for red, green, and blue color)
- Depth: 153,600 memory locations (8 frames of 19,200 pixels each, corresponding to 160×120 resolution per state)

Control signals for accessing ROM

- clka: Clock input synchronized to the pixel clock
- ena: Enable signal derived from video display enable and test mode status
- addra: 18-bit address input, calculating the offset based on the current frame and pixel coordinates

3. Design Validation

This section will be used to demonstrate the successful implementation of the design. You should use a modular implementation and testing strategy. That means each component should be verified through simulation and hardware testing (as appropriate) before it is combined with other components. This section should have a parallel structure to Section 2. Note that if you need to go more than two levels deep (top level and components) in your testing strategy, then include a subsystem in your testing plan (See Section 3.3)

3.1. petrock_behavioral_shell.vhd

To test the behavioral shell, I used a modular testing approach. Each of the four major submodules (Life FSM, Mood FSM, Time Manager, and Coin Manager) was first tested individually in simulation. The testing of these submodules will be described in more detail in 3.3 and 3.4. Once each submodule's functionality was confirmed, I integrated them into the behavioral shell and tested the overall design.

3.1.1. Behavioral Simulations

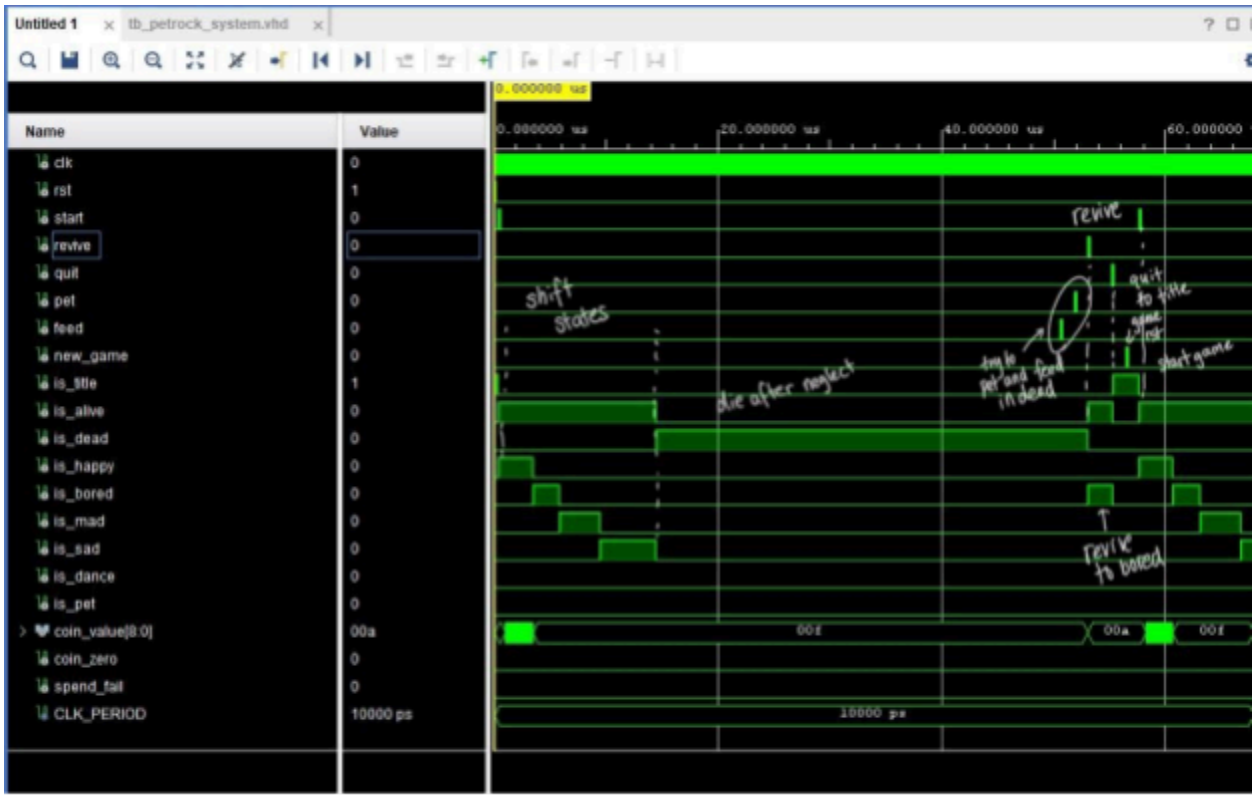


Figure 10: Behavioral Shell Testbench Waveform

To validate the behavioral shell, I made a testbench that tested most game scenarios including edge cases. The waveform in Figure 6 shows the key outputs in response to button inputs.

At the beginning, the FSM starts in the Title state. The testbench's stimulus process is below:

```
stim: process
  begin
    -- Reset
    rst <= '1';            wait for 200 ns;
    rst <= '0';            wait for 200 ns;

    -- START (pulse)
    start <= '1';          wait for 200 ns;
    start <= '0';
```

```
    -- Let timers run all the way to DEAD (with TIMER_WIDTH=3 this is quick).
    -- Use a comfortable fixed wait instead of waits on signals.
    wait for 50 us;

    -- While DEAD: try FEED and PET (should be ignored / no spend / no mood
change)
    feed <= '1';           wait for 200 ns;  feed <= '0';
    wait for 1 us;
    pet  <= '1';           wait for 200 ns;  pet  <= '0';
    wait for 1 us;

    -- REVIVE (pulse) - your shell gates it so it only passes while DEAD
    revive <= '1';         wait for 200 ns;  revive <= '0';

    -- Give time for state registers to update and Bored to assert
    wait for 2 us;

    -- QUIT back to Title
    quit <= '1';           wait for 200 ns;  quit <= '0';
    wait for 1 us;

    -- NEW GAME (refill coins & reset mood/timers), then START again
    new_game <= '1';       wait for 200 ns;  new_game <= '0';
    wait for 1 us;

    start <= '1';          wait for 200 ns;  start <= '0';

    -- Run a bit, then finish
```

```
    wait for 10 us;
    assert false report "Simulation finished" severity failure;
  end process;


end architecture;
```

As we can see from the waveform, the game successfully shifts states and eventually enters the ‘is_dead’ state after the timers expire. The coin was successfully accumulating correctly during the happy state. For edge case testing, we tested petting and feeding from the dead state, and the system ignored these actions as expected, confirming that no unintended mood changes or coin spending occurred. When we tried to revive the rock from the dead state, it successfully returned to the bored state and deducted 5 coins. Lastly, we tested the quit and game reset functionality. The waveform shows that these inputs correctly returned the game to the title screen. Upon pressing start, the system re-entered the happy state with 10 coins, demonstrating that the reset mechanism restored the system to its proper initial conditions.

3.1.2. Hardware Validation

We briefly ran oscilloscope-based validation, but due to time constraints were unable to capture and save annotated images. However, the design was functionally validated through behavioral simulation waveforms, and full system operation was confirmed on hardware via the VGA demo video (see Section 3.4.2).

3.2. VGA.vhd

3.2.1. Behavioral Simulations

To test if VGA is initially working before applying higher logic and design, we focused on first looking at the basic essential signals like clk, V_sync_port, H_sync_port, and RGB color outputs (red, green, blue) transitioning over time.

Figure 11(a): VGA Testbench Waveform

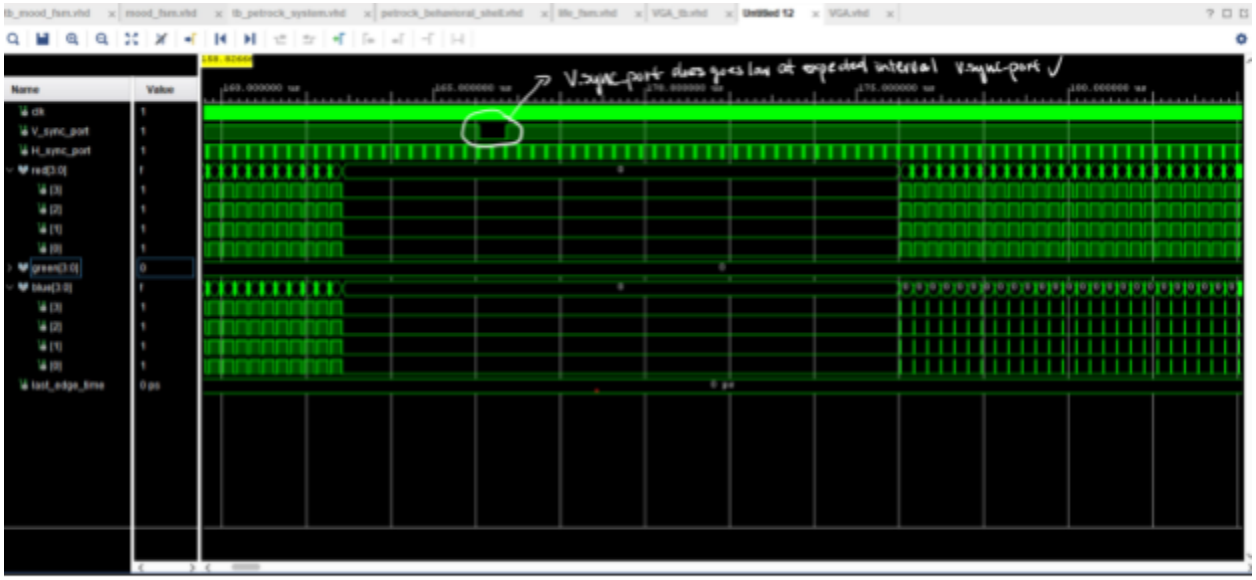


Figure 11(b): VGA Testbench Waveform

As the clock consistently toggles, the H_sync_port and V_sync_port waveforms correctly assert low pulses at the expected intervals, which match the VGA horizontal and vertical retrace timing requirements.

3.2.2. Hardware Validation

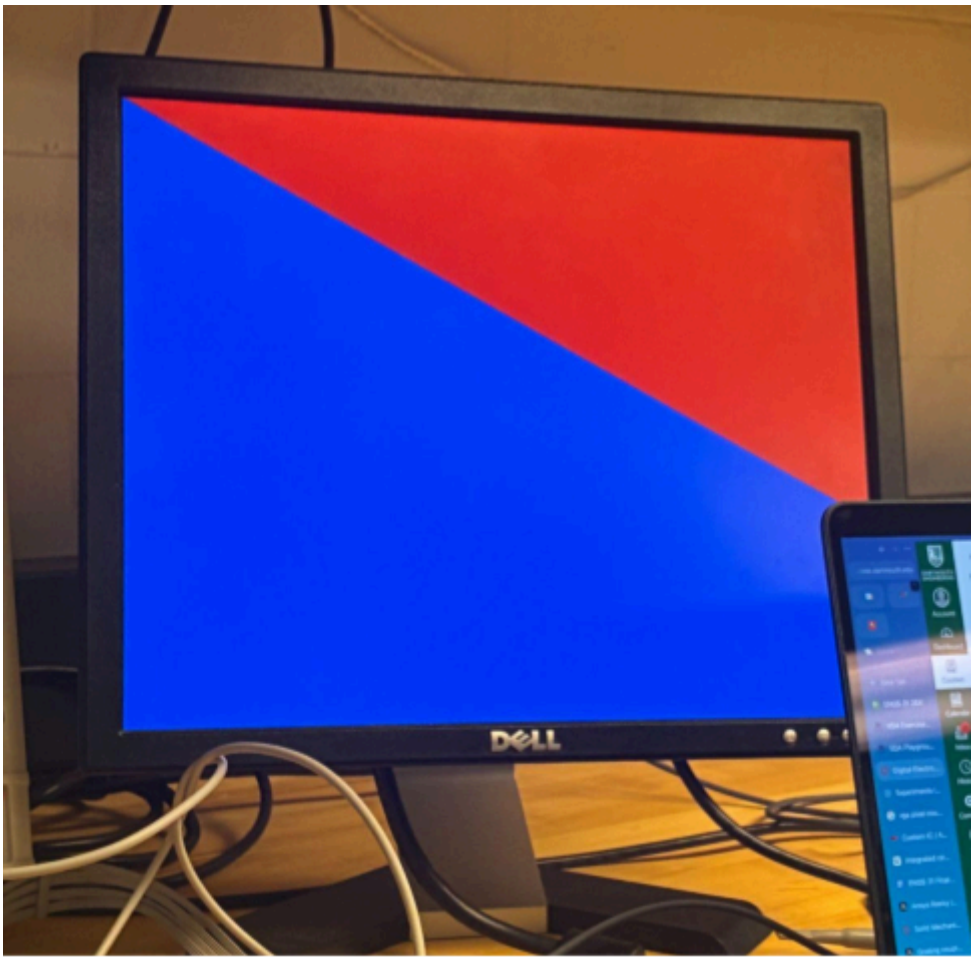


Figure 11(c): VGA Hardware Test

In VGA.vhd I coded the RGB to show a comparison of the current pixel_x with its vertical coordinate pixel_y. If pixel_x > pixel_y, the output color is set to maximum red intensity, and blue is turned off. Otherwise, blue is set to maximum intensity. This logic is to create a simple pattern on the VGA screen that is clear to be viewed. The result is a diagonal line across the screen that separates a red region (above the diagonal) and a blue region (below the diagonal).

The VGA.vhd was then tested to be working by verifying key signals like clock, sync pulses, and RGB outputs behaving correctly over time.

3.3. Controllers

3.3.1. Life FSM

To test the Life FSM, I created a testbench that applied different input sequences, including reset, start, revive, quit, and new game.

3.3.1.1. Behavioral Simulations

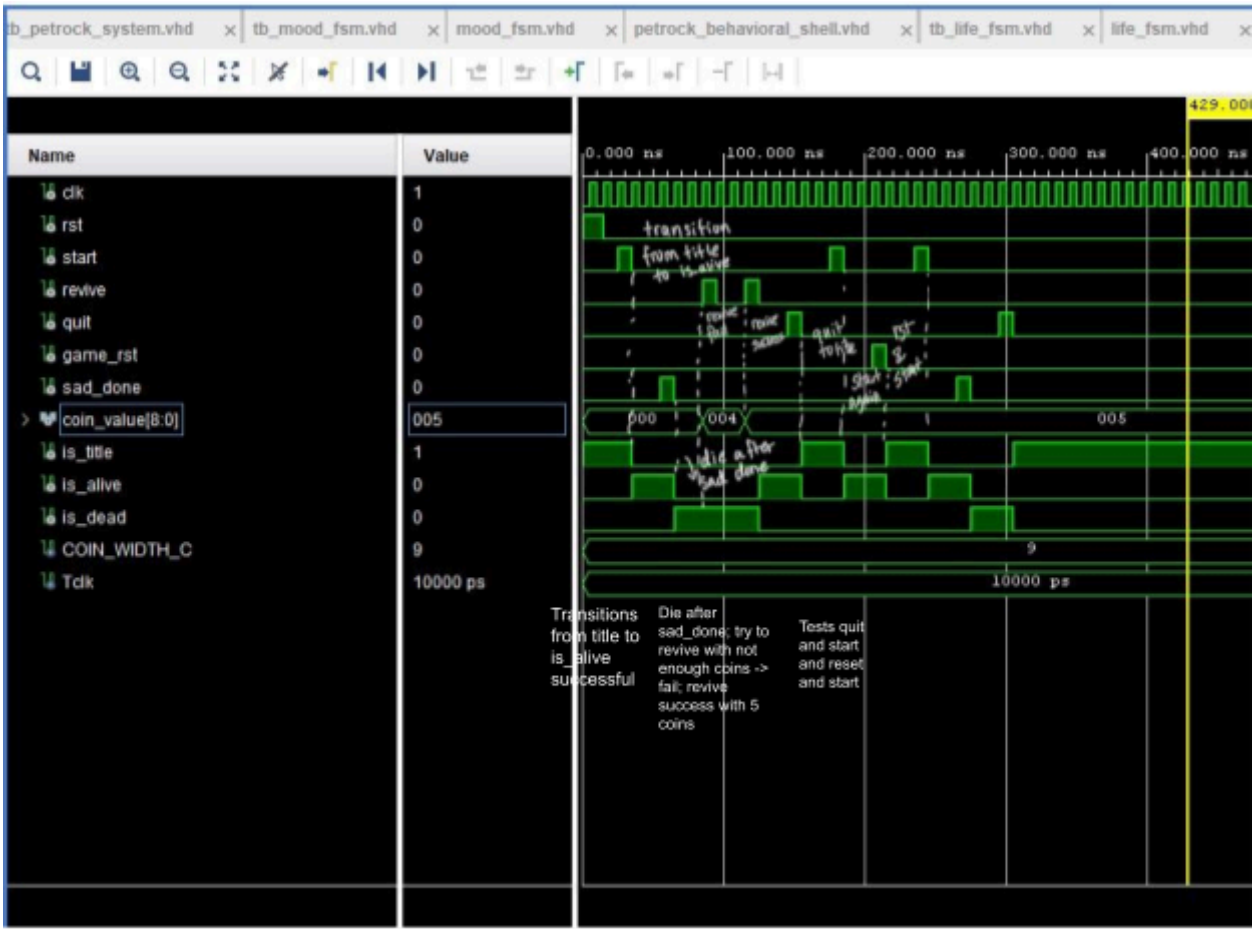


Figure 12: Waveforms of Life Finite State Machine

As seen in Figure 7, the simulation waveform shows that the FSM transitioned correctly between the Title, Alive, and Dead states. Coins were accumulated during the Happy state, inputs were

ignored in the Dead state, and reviving correctly returned the FSM to the Bored state with a coin deduction. The quit and new game functions also behaved as expected, resetting the system to the Title screen and reinitializing coins.

3.3.2. Mood FSM

I wrote a testbench to test a combination of different inputs for the mood FSM.

3.3.2.1. Behavioral Simulations

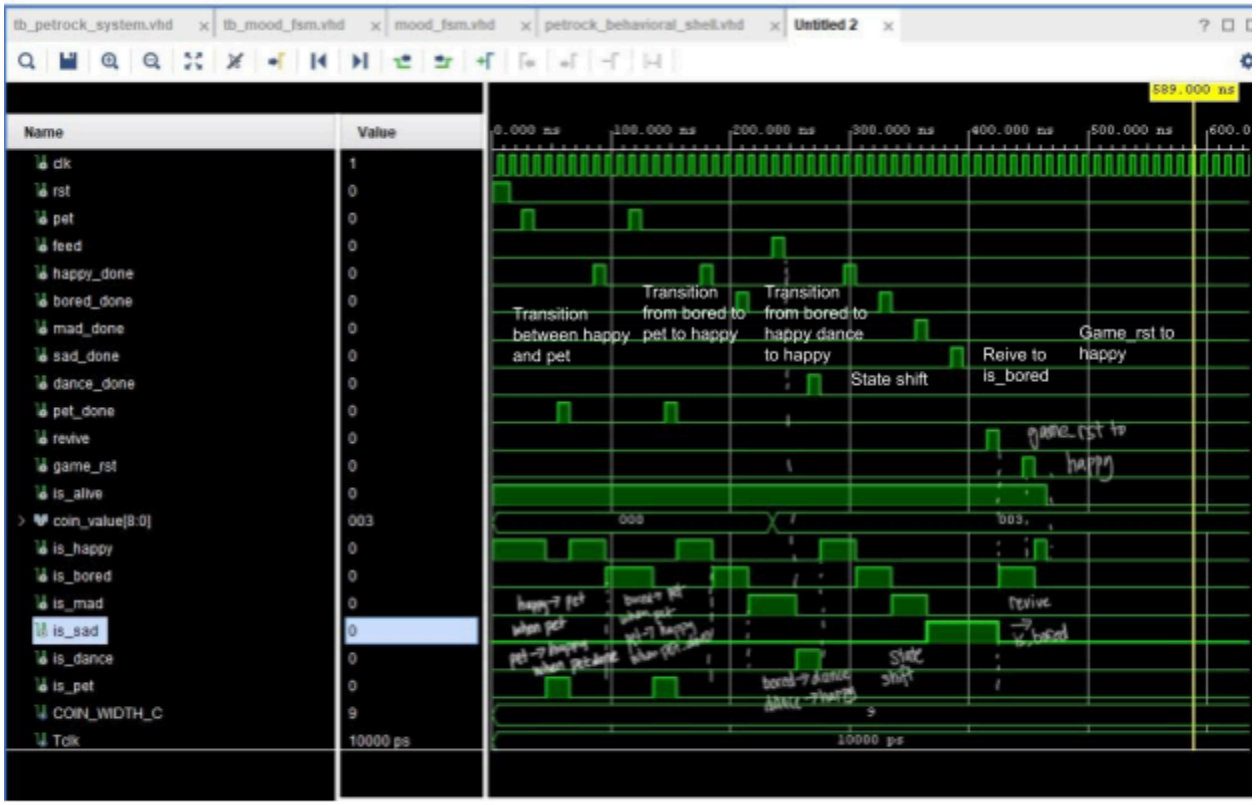


Figure 13: Waveforms for Mood Finite State Machine

The waveform in Figure 9 shows that the FSM transitioned correctly between the Happy, Bored, Mad, Sad, Dance, and Pet states. For example, petting during Bored returned the rock to Happy, while timers advanced the states sequentially toward Sad. The revive input successfully returned the system from Sad back to Bored, and a game reset correctly initialized the FSM to the Happy

state. These results confirm that the Mood FSM responds appropriately to both user inputs and timer expirations.

3.4. Datapath

3.3.1. Time Manager

To test the Time Manager, I created a testbench that monitored the done signals associated with each mood state.

3.3.1.1. Behavioral Simulations

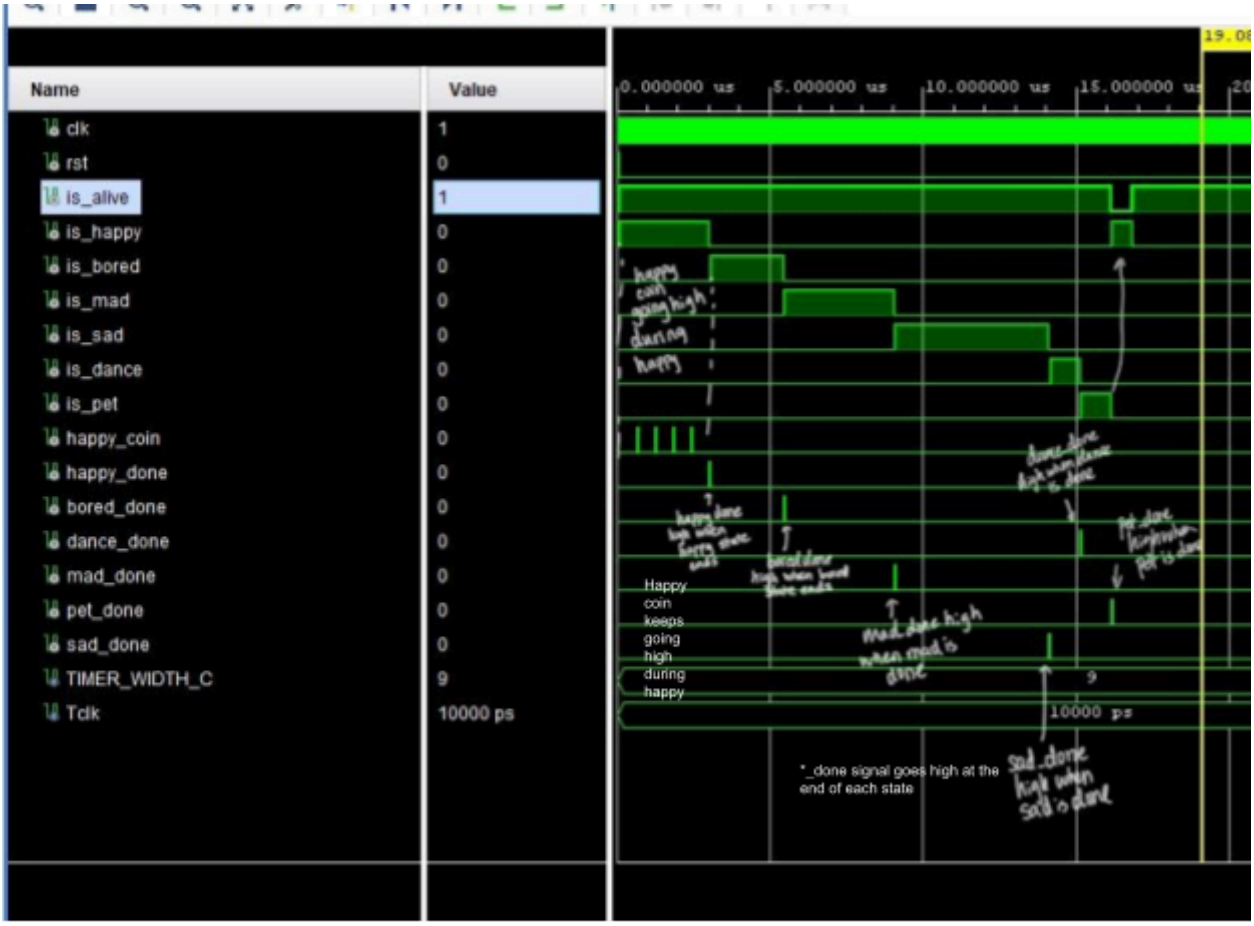


Figure 14: Time Manager Waveforms

As shown in Figure 10, the happy_coin output correctly incremented during the Happy state, and each corresponding _done signal was asserted once its timer expired. The waveform shows that

timers advanced as expected, generating clean one-cycle pulses that synchronized state transitions in the Mood FSM.

3.3.2. Coin Manager

To test the coin manager, I wrote a testbench to test whether coins were collected in the happy state and coins were correctly deducted given the different inputs, such as feed and revive. I also tested the coin_init signal and attempted to spend coins when there were insufficient funds.

3.3.2.1. Behavioral Simulations

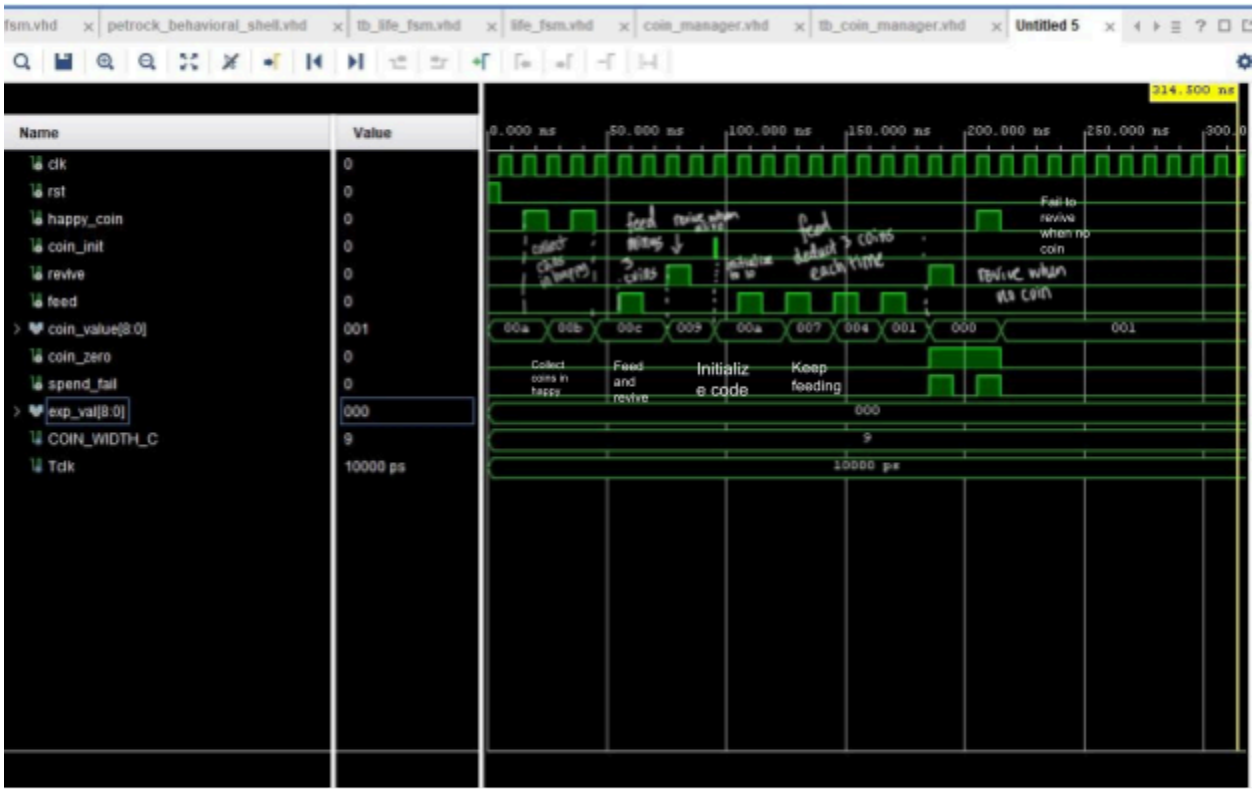


Figure 15: Coin Manager Waveform

As demonstrated in Figure 11, on the happy_coin signal, a coin was collected. Feed and revive each correctly subtracts 3 and 5 coins from the coin value when such an amount is available.

Coin also correctly initializes to 10 when coin_init goes high. Feed and revive when there are not sufficient funds will cause the spend_fail and coin_zero signal to go high.

3.4. Overall Design

Once you demonstrate that each individual component has proper operation, demonstrate the correct operation of the overall design.

3.4.1. Behavioral Simulations

3.4.2. Hardware Validation

We validated the full design on the FPGA with VGA output. The demo confirmed that the rock's states updated correctly in response to button inputs, coins were managed as expected, and the VGA display reflected the system state in real time. A video of the full demo is available here: https://drive.google.com/drive/folders/1JxncKDaPCW5aaGs6EV20lAXuXpMxbY8q?usp=drive_link

4. Analysis of the Design

The project turned out to be fully functional. The pet rock responded correctly to user inputs, displayed the appropriate graphics, and the overall game ran smoothly on the hardware. In this sense, the project was a success.

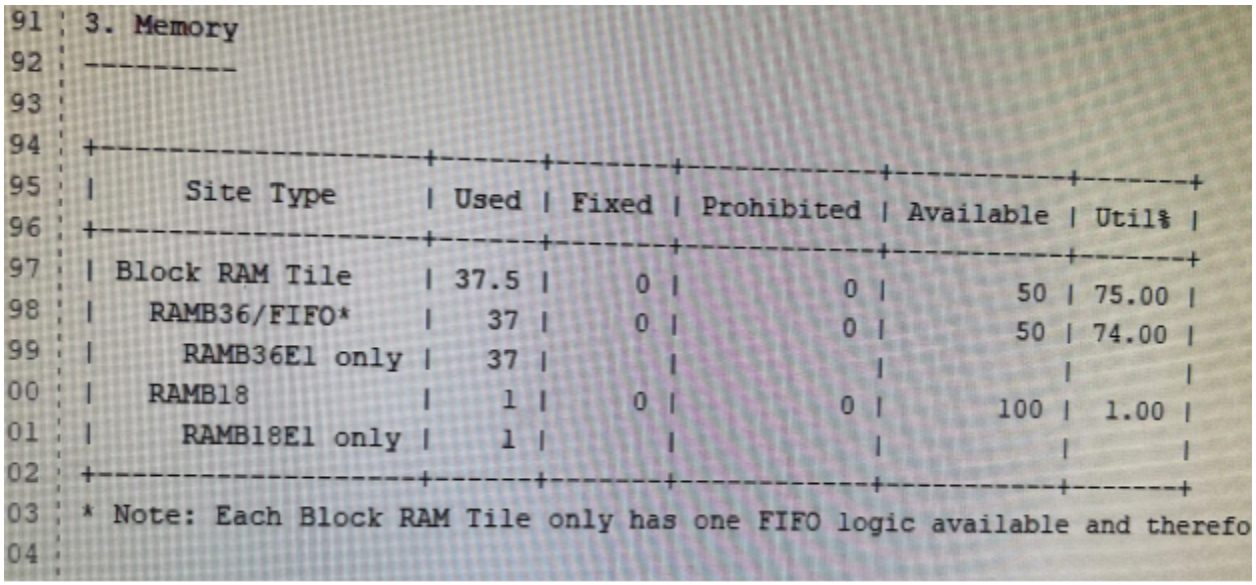
However, several unexpected challenges did happen during the project. One of the biggest challenges was hitting the BRAM limits, which forced us to adjust our .coe files and reduce the number of animation frames to fit within the available memory. Another major challenge was implementing the Pet state. Since petting should return the rock to its previous mood, for

example, from Mad to Pet and then return to Bored, the FSM had to store the prior state and use it for transitions. This made the Mood FSM more complex than expected.

We also learned that making the system ignore invalid inputs (such as petting or feeding when Dead, or reviving when Alive) was just as important as handling valid actions. These “do nothing” cases initially caused bugs that took time to identify. Finally, when combining the behavioral and graphics shells, debugging became more difficult because issues could originate from either logic or display. We often had to cross-check both simulation waveforms and hardware outputs to track down errors.

If we had more time, we would address BRAM constraints by compressing or optimizing sprite storage, which would allow more animation frames. We would also do more testing instead of rushing to finish the project. We would also refine the FSMs to simplify priority logic and reduce the risk of timing violations.

4.1. Resource Utilization

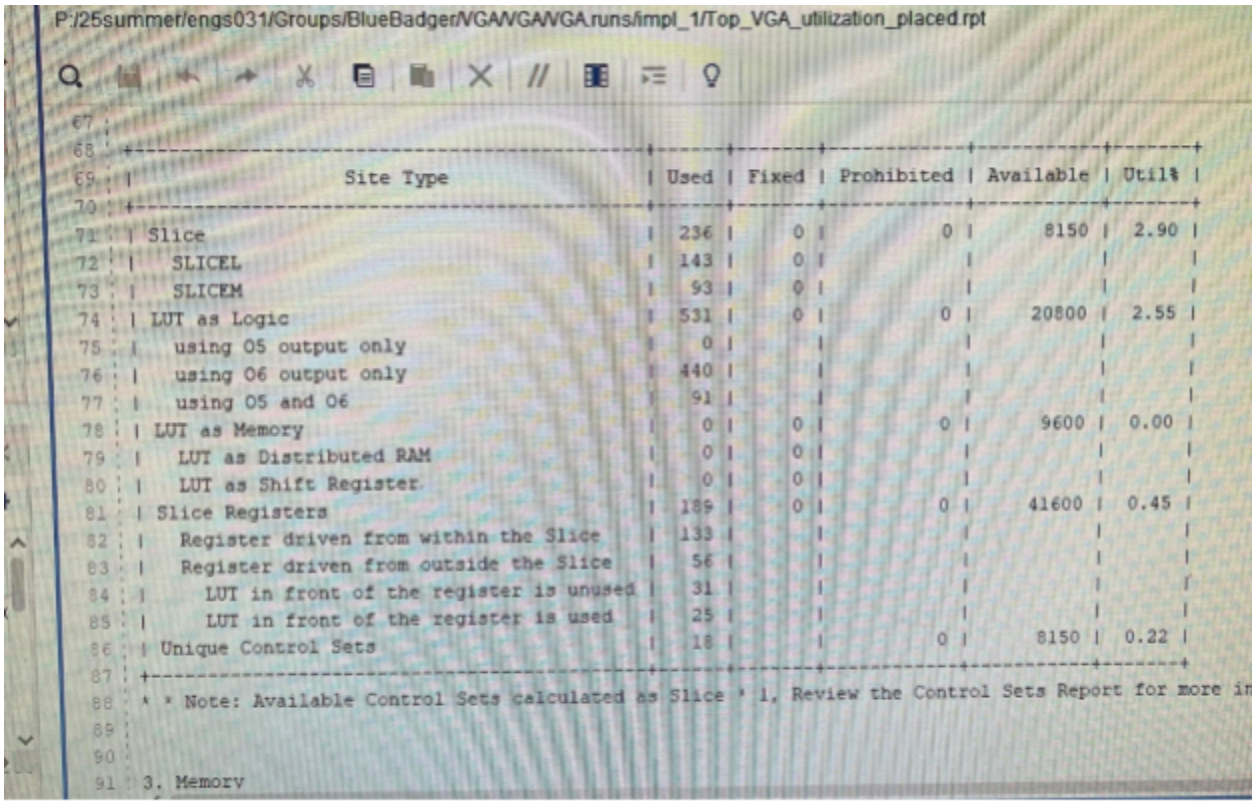


3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	37.5	0	0	50	75.00
RAMB36/FIFO*	37	0	0	50	74.00
RAMB36E1 only	37				
RAMB18	1	0	0	100	1.00
RAMB18E1 only	1				

* Note: Each Block RAM Tile only has one FIFO logic available and therefo

Figure 16(a): FPGA Utilization



P:/25summer/engs031/Groups/BlueBadger/VGA/VGA/VGA.runs/impl_1/Top_VGA_utilization_placed.rpt

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	236	0	0	8150	2.90
SLICEL	143	0			
SLICEM	93	0			
LUT as Logic	531	0	0	20800	2.55
using O5 output only	0				
using O6 output only	440				
using O5 and O6	91				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	189	0	0	41600	0.45
Register driven from within the Slice	133				
Register driven from outside the Slice	56				
LUT in front of the register is unused	31				
LUT in front of the register is used	25				
Unique Control Sets	18		0	8150	0.22

* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more in

3. Memory

Figure 16(b): FPGA Utilization

As seen in Figures 12(a) and 12(b), the design made heavy use of Block RAM (BRAM), which accounted for approximately 75% of the FPGA's available capacity. This was expected, since storing multiple VGA sprite frames required significant memory resources. In comparison, the logic and control resources took up a lot les:s FPGA Utilities: Slices (2.9%), LUTs as Logic (2.55%), Slice Registers (0.45%), and Unique Control Sets (0.22%).

Overall, the design fits well within the FPGA device capacity, with memory usage the primary limiting factor.

4.2. Residual Warnings

There were no remaining critical warnings after synthesis and implementation. There were minor informational warnings, but these did not impact functionality or hardware operation. Overall, the design compiled cleanly and was verified to run correctly on the FPGA.

4.3. Division of Labor

We divided the project into two main parts: the behavioral shell and the graphics shell. Giselle focused on the behavioral side, designing and creating the Life FSM, Mood FSM, Time Manager, and Coin Manager modules, as well as creating and running the corresponding testbenches and integrating the design at the top level. Lijia focused on the graphics side, implementing the VGA controller, drawing the rock sprites, and converting the images into .coe files for memory initialization. We collaborated on the Graphics Manager, where much of the behavioral and graphical logic intersected, and Giselle wrote the top-level file connecting all components. Together, these efforts produced a functioning Pet Rock game with integrated behavioral logic and VGA graphics.

4.4. Future Work

If we had more time, the next steps would focus on enhancing both the visuals and the gameplay. On the visual side, the pet rock could be made more animated by adding new sprite frames. Since BRAM already accounted for about 75% of FPGA utilization, one improvement would be to optimize or compress sprite files to reduce memory usage, allowing more frames to be stored. On the gameplay side, new parameters such as a “heart” or affection value and additional user actions (e.g., play, sleep) could be added to increase interactivity and make the game more engaging. Additional features could include allowing the user to assign and store a name for their pet rock, as well as enabling multiple pet rocks to interact or “make friends,” further expanding the sense of personalization and play.

5. Acknowledgements

This project is for Will—without his guidance and support, we would not have been able to make it happen. Biggest thanks to him for his assistance throughout the project, even when we are away. We also thank all the other TAs. Special thanks to Tad for providing insightful feedback and advice on how to approach our initial design, incorporate VGA, and how to connect what we learn in class to real-life applications. We also appreciate the entire ENGS 31 class this term, who collaborated with us and shared helpful suggestions in identifying errors and bugs in our code. For all of our codes, we extended from examples provided in labs and class exercises. We also used ChatGPT to help identify and resolve bugs when we encountered difficult problems and for insights.

6. Conclusions

In conclusion, we would love to choose this project again because it provided great hands-on experience combining theory with practical FPGA implementation. However, we wish we had more time to avoid having to rush and leave early, which limited our ability to add more features and polish our game. Our advice to future students is to start early, plan thoroughly, and allocate enough time for integration and testing so you can fully explore enhancements and debug effectively.

Appendix A: VHDL Source Code

petrock_behavioral_shell.vhd
This file contains the main behavioral shell of the Pet Rock game. It handles the game logic by coordinating several finite state machines that manage the pet's life cycle, moods, timers, and coin system. Detailed information can be found in Section 2.1.
coin_fsm.vhd
The file contains the coin system's finite state machine. It manages the control logic for coin-related actions in the Pet Rock game. It transitions between five states: Initial_Coin, Earn, Spend, Zero, and Check, based on input signals representing coin events and game interactions. Detailed information can be found in Section 2.1.
life_fsm.vhd
The file contains the life system's finite state machine. It controls the high-level life cycle states of the Pet Rock game. It has three states: Title, Rock_Alive, and Rock_Dead, representing the different life phases of the virtual pet. Detailed information can be found in Section 2.1.
mood_fsm.vhd
The file contains the mood system's finite state machine. It manages the pet's mood states throughout the game. It has six states: Happy_Reset, Happy, Bored, Mad, Sad, and handles transitions based on inputs such as timers expiring, user interactions, and game resets. Detailed information can be found in Section 2.1.
petrock_datapath.vhd
This file is responsible for managing the core data operations related to the Pet Rock game's timers and coin handling. It consists of three major parts: the coin register, the happy timer, and the other timer (tracking bored, mad, sad, etc.). Detailed information can be found in 2.1.
time_manager.vhd
This file implements multiple timers that track the duration the pet spends in various mood state. It includes six independent timers corresponding to the moods: Happy, Bored, Mad, Sad, Dance, and Pet. Detailed information can be found in 2.1.
coin_manager.vhd
This file is responsible for managing the player's coin balance in the Pet Rock game. It tracks coin increments and decrements, detects spending attempts, and signals error conditions such as insufficient coins for spending. Detailed information can be found in 2.1.
system_clock_generation.vhd
This file is responsible for dividing the 100 MHz input clock down to a slower system or game clock at approximately 60 Hz. The design uses a counter and a toggle flip-flop, followed by a BUFG for proper clock distribution inside the FPGA. Detailed information can be found in Section 2.2.
top_vga.vhd
This file serves as the top-level integration point for the Pet Rock game system on the FPGA. It connects various subsystems, including VGA timing, game logic, graphics generation, and

hardware inputs, to produce the interactive VGA display and respond to user inputs. Detailed information can be found in Section 1.2.1

VGA.vhd

The VGA file generates synchronization signals for a standard 640×480 VGA display. It divides the 100 MHz clock to produce a 25 MHz pixel clock, manages the horizontal and vertical timing counters and sync, and outputs the pixel coordinates along with a simple test pattern in the RGB outputs. Detailed information can be found in Section 2.3.

GraphicsManager.vhd/ ROM.vhd

The GraphicsManager file is the display of the game. It selects and draws sprite frames from Block ROM according to the current pet state and scales the 160×120 sprites to the full 640×480 VGA display area. It also handles background colors, transparency, and overlays a coin count using a small bitmap font. Detailed information can be found in Section 2.4.

Appendix B: VHDL Testbenches

tb_petrock_behavioral_shell.vhd

This testbench verifies the overall behavioral shell of the Pet Rock game. Individual submodules such as the Life FSM, Mood FSM, Time Manager, and Coin Manager were first tested separately in simulation. After confirming their correct operation, this testbench applies overall stimulus to validate typical game scenarios and edge cases. The waveform analysis demonstrates correct state transitions, including handling of feeding, petting, reviving actions, and proper coin accumulation. Detailed information can be found in Sections 3.1 and 3.3.

VGA_tb.vhd

This testbench verifies the VGA module's basic functionality by stimulating the clock and observing key outputs such as H_sync, V_sync, and RGB color values. The behavioral simulation confirms that VGA timing signals are asserted at expected intervals and that the RGB outputs produce the intended color pattern display. Detailed information can be found in Section 3.2.

tb_coin_fsm.vhd

This testbench is designed to verify the correct functionality of the coin FSM module in the Pet Rock game. It provides clock generation, stimulus signals, and monitors the outputs to confirm that the FSM transitions and coin control signals behave as expected under various scenarios.

tb_life_fsm.vhd

A testbench for the Life FSM validates its correct management of top-level states (Title, Alive, Dead). Through crafted input sequences including reset, start, revive, quit, and new game, the testbench demonstrates expected state transitions. Detailed information can be found in Section 3.3.1.

tb_mood_fsm.vhd

This testbench verifies the Mood FSM's response to user inputs and timer events. It checks transitions of Happy, Bored, Mad, Sad, Petting, and Dancing states and verifies proper handling

of edge conditions, such as petting in various mood states and reset functionality. Detailed information can be found in Section 3.3.2.

tb_time_manager.vhd

The Time Manager testbench monitors timer signals associated with each mood state. The simulation confirms that timers count correctly, generate done pulses, and trigger mood transitions reliably. Detailed information can be found in Section 3.4.1.

tb_coin_manager.vhd

This testbench tests the Coin Manager module by applying different inputs for coin earning, spending, and initialization. It verifies that coins increment in the Happy state, that spending is properly gated by available coins, and that error signals assert when coins are not enough coins. Detailed information can be found in Section 3.4.2.

Appendix C: MATLAB and COE

pngToCoe.mlx

```
% Read and resize image
filename = 'sad.png'; % PNG image file
output_coe = 'sad.coe'; % Output COE
target_width = 160;
target_height = 120;

img = imread(filename);
img = imresize(img, [target_height, target_width]);

% Ensure uint8
if ~isa(img, 'uint8')
   img = im2uint8(img);
end

% RGB channels to 4-bit each (keep top 4 bits)
%r = bitshift(img(:,:,1), -4);  % values 0..15
%g = bitshift(img(:,:,2), -4);
%b = bitshift(img(:,:,3), -4);
r = bitshift(img(:,:,1), -5);  % values 0..15
g = bitshift(img(:,:,2), -5);
b = bitshift(img(:,:,3), -5);

% Pack into 12-bit value: R in bits 11:8, G in 7:4, B in 3:0
% pixels = uint16(bitshift(uint16(r), 8)) + uint16(bitshift(uint16(g), 4)) + uint16(uint16(b));
pixels = uint16(r) * 64 + uint16(g) * 8 + uint16(b);
```

```
% Open output COE file
fid = fopen(output_coe, 'wt');
if fid == -1
   error('Cannot open output file: %s', output_coe);
end

% Write COE header
fprintf(fid, 'memory_initialization_radix=16;\n');
fprintf(fid, 'memory_initialization_vector=\n');

% Write pixel data row-wise as hex
for row = 1:target_height
   for col = 1:target_width
      val = pixels(row, col);
      if row == target_height && col == target_width
         % Last value ends with semicolon
         fprintf(fid, '%03X;\n', val);
      else
         fprintf(fid, '%03X,\n', val);
      end
   end
end

fclose(fid);
disp(['COE file generated: ', output_coe]);
```

Detailed information can be found in Section 2.4.4.1.

combinedCoe.m

```
% all .coe filenames
files = {'start.coe', 'bored.coe', 'happy.coe', 'sad.coe', 'mad.coe', 'dead.coe', 'petting.coe',
         'food.coe'};
output_file = 'combined.coe';

fid_out = fopen(output_file, 'wt');
fprintf(fid_out, 'memory_initialization_radix=16;\n');
fprintf(fid_out, 'memory_initialization_vector=\n');

for i = 1:length(files)
```

```
    fid = fopen(files{i}, 'rt');
    % Skip the first 2 lines (header)
    fgetl(fid);
    fgetl(fid);

    while true
        tline = fgetl(fid);
        if ~ischar(tline)
            break;
        end
        tline = strtrim(tline);

        % Remove trailing semicolon or comma to handle concatenation properly
        tline = regexprep(tline, '[,;]$', '');

        if i == length(files) && feof(fid)
            % Last line from last file: add semicolon
            fprintf(fid_out, '%s;', tline);
        else
            fprintf(fid_out, '%s,', tline);
        end
        fprintf(fid_out, '\n');
    end

    fclose(fid);
end

fclose(fid_out);
disp(['COE file created: ', output_file]);
```

Detailed information can be found in Section 2.4.4.2.

combined.coe

The file was created using the MATLAB script combinedCoe.m and represents a memory initialization file designed for the FPGA's BRAM. This COE file has a width of 9 bits and a depth of 153,600 entries. Detailed information can be found in Section 2.4.4.3.