


The Art Thief

By: Noah Aldridge, Quinn Bell, Mike Plata,
and Eduardo Vargas

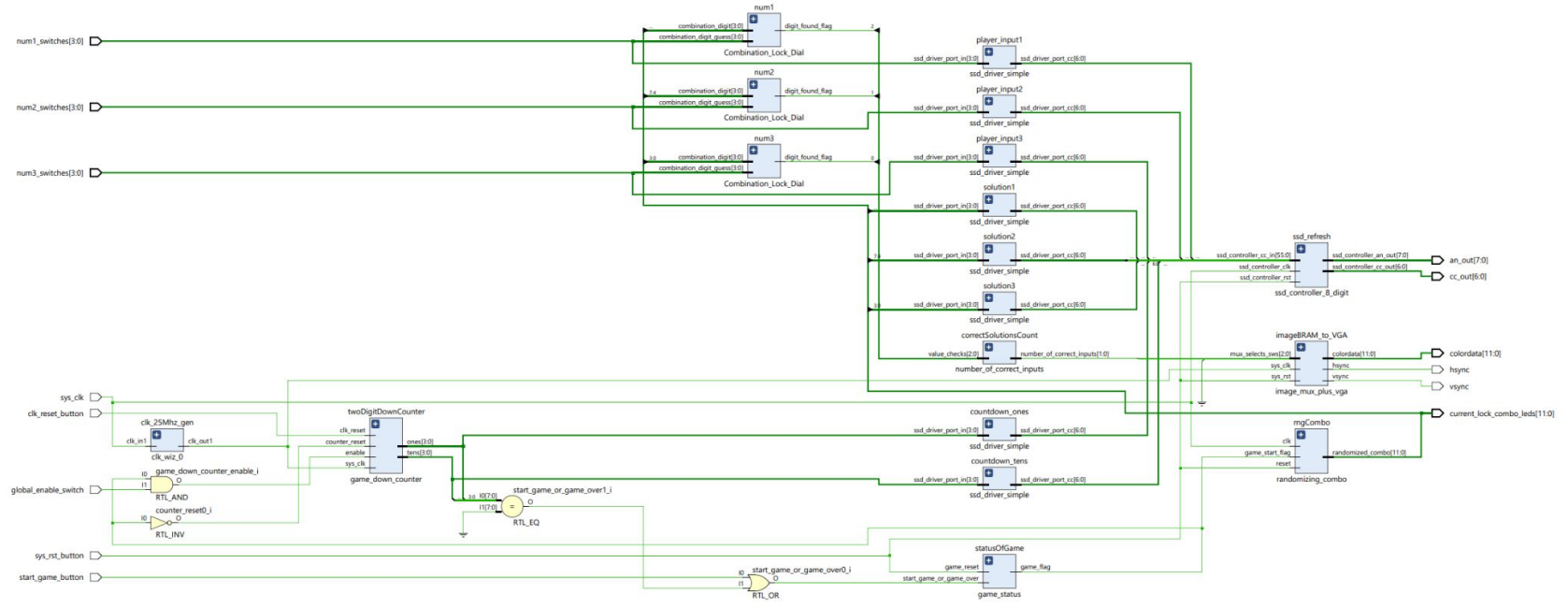
A large, solid dark blue shape that starts from the bottom left corner and extends diagonally upwards towards the right, covering the lower half of the slide.

Objective



In this game titled, “The Art Thief”, players are given 29 seconds to try and crack the 3 digit padlock code that reveals an image. The concept is simple, once the player inputs a 3 digit code an image will pop up with a set level of blurriness. If no digits match the passcode, it will display the blurriest version of the photo. As more digits are set to the correct value, the less blurry the picture will be. Once the player gets the correct randomized code, they will unlock the original photo with no blur and win the game.

Schematic



Schematic Summary

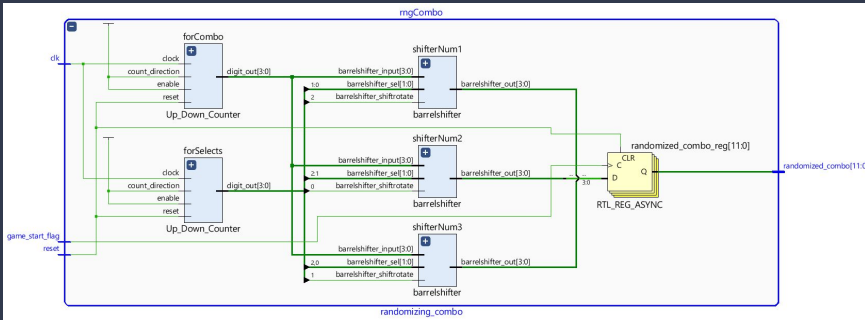
A two-digit down counter outputs to two digits of the seven segment displays which will act as the game timer.

Four switches on the board send the inputted code and display it on the seven segment display while also being value checked against the randomly generated passcode.

The number of correct inputs is then sent to the image mux which will then select which image ROM to read and send to the VGA port.

The VGA driver module generates the necessary parameters for the VGA port to work.

Combination Generator



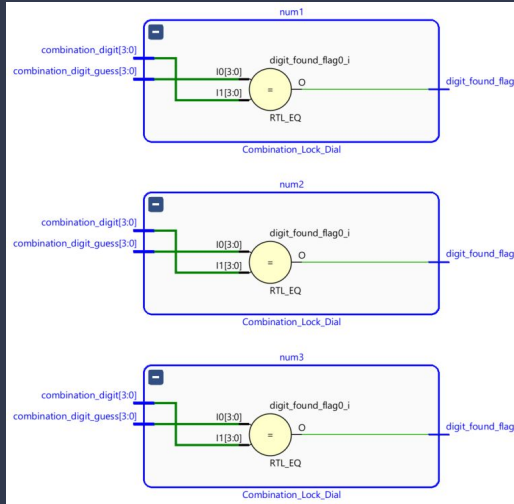
This module contains an always-enabled counter and multiple barrel shifters to create 3 pseudo-random hex numbers for the passcode of the padlock, which is then compared/checked with switch input numbers.

Upon starting the game, the module locks in the current hex numbers for the passcode until the game is restarted.

Its ports include a clock, reset, and game_start_flag.

The clock drives the counter, and the game_start_flag input locks in the passcode upon its positive edge.

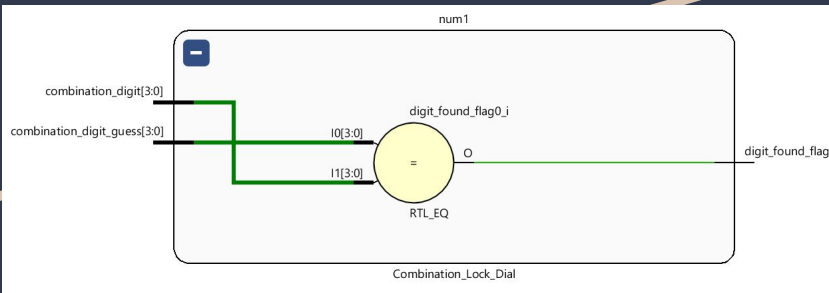
Combination Lock Dial



Instantiations of this module compare the player's inputs to the current passcode.

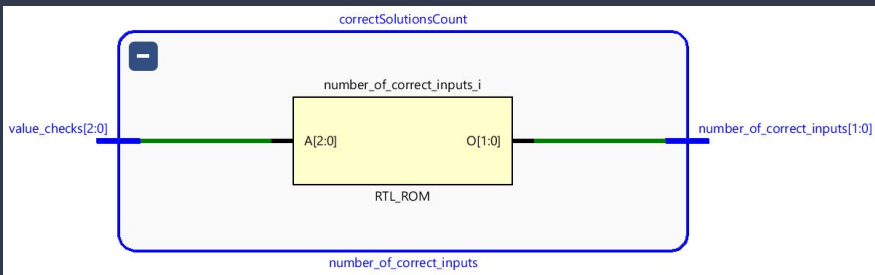
If the switch number equals the random number, it outputs a 1 to the `digit_found_flag`; Otherwise, a 0 is outputted.

There are 3 instantiations in the entire project to handle 12 switches representing the player's inputs.



This module's ports include the 4-bit passcode digit input, 4-bit player input, and a `digit_found_flag` output.

Number of correct inputs



```
module number_of_correct_inputs(  
    input [2:0] value_checks,  
    output reg [1:0] number_of_correct_inputs  
);  
  
always@(value_checks)  
begin  
    case(value_checks)  
        3'b000: number_of_correct_inputs <= 2'b00; // No correct inputs  
        3'b001: number_of_correct_inputs <= 2'b01; // 1 correct input  
        3'b010: number_of_correct_inputs <= 2'b01; // 1 correct input  
        3'b011: number_of_correct_inputs <= 2'b10; // 2 correct inputs  
        3'b100: number_of_correct_inputs <= 2'b01; // 1 correct input  
        3'b101: number_of_correct_inputs <= 2'b10; // 2 correct inputs  
        3'b110: number_of_correct_inputs <= 2'b10; // 2 correct inputs  
        3'b111: number_of_correct_inputs <= 2'b11; // 3 correct inputs  
        default: number_of_correct_inputs <= 2'd0; // Standard default  
    endcase  
end
```

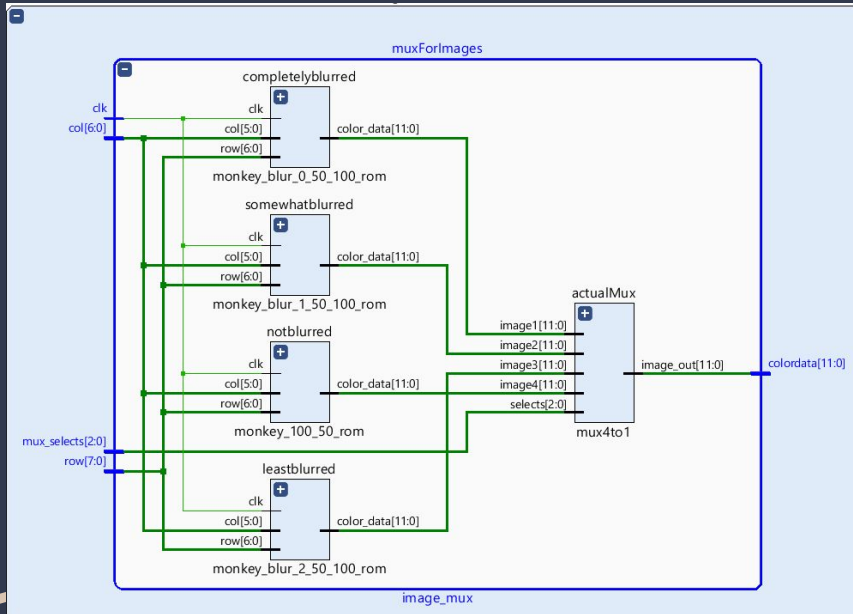
This module lies between the image mux and combination lock dial modules. It takes the 3 bits coming from the output of the 3 lock dials as a bus.

It then outputs a 2-bit number that's equal to the number of 1's in the input bus.

For example, if the input is 001, it will output 2'd1. If the input is 101, it will output 2'd2.

This output is used by the image mux to select an image based on the player's progress.

Image Mux and ROMs



The image multiplexer module initializes four images into the board's block memory (BRAM) as a read-only memory (ROM).

The images are the same image but with four different levels of blur ranging from completely blurry to no blur. The outputs of each memory are sent to the 12-bit wide inputs of a 4-to-1 mux.

It uses the output from the “number of correct inputs” module as the select bits for the mux.

Once it selects an image, it will output the color data for the current pixel of the selected image to the board's VGA port.

This module works in parallel with the VGA Driver module (next).

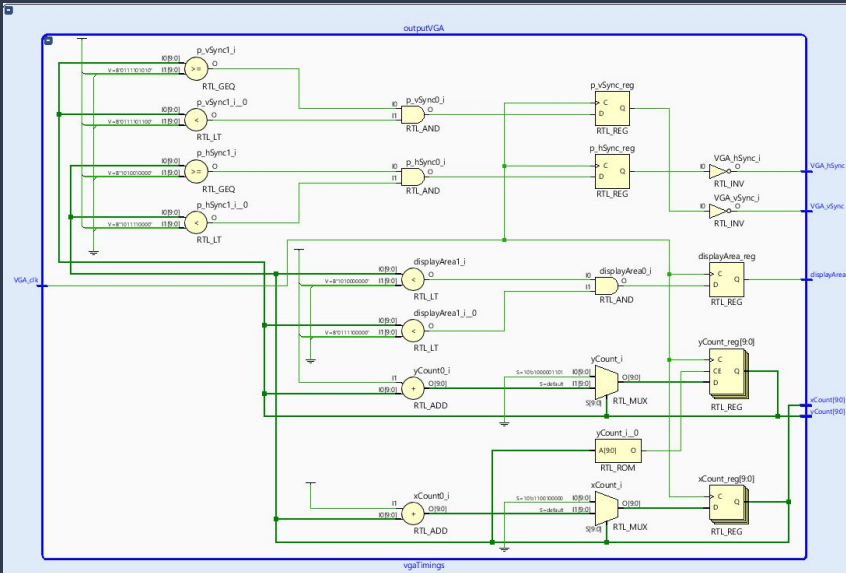
VGA Driver

This module generates the necessary parameters and timings to create a 640x480 resolution image through the board's VGA port.

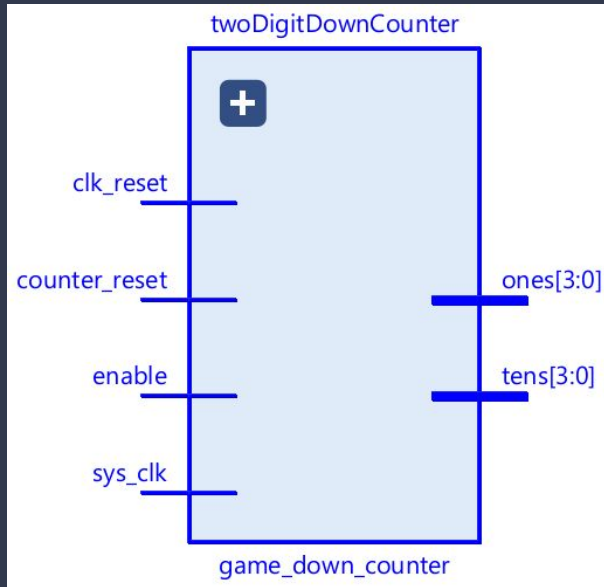
Using the 25MHz clock and various counters, it generates the following signals:

- horizontal sync (Hsync)
- vertical sync (Vsync)
- displayArea (used to check if the pixel being generated is within the monitor's display area)
- xCount (column number of pixel)
- yCount (row number of pixel)

The hsync and vsync signals are sent to the VGA port, while the remaining data is sent to the image multiplexer to determine the pixel color data.



Countdown

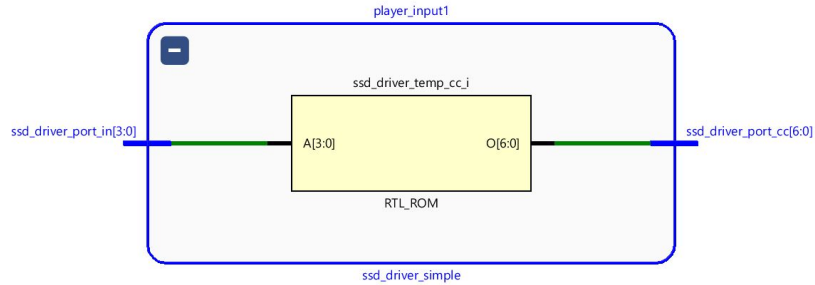


The 2 digit down counter inputs a clock signal and a reset as its only purpose is to act as the players time limit indicator

It will start the count from 29 all the way to 0 as the set time limit for the player to guess the code

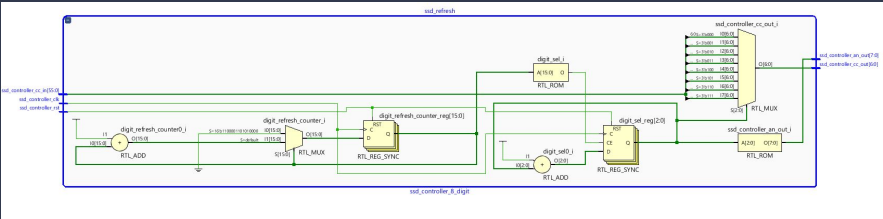
It outputs a 7 segment display code to the digit refresh module as well as outputting a one-bit game over signal to the rest of the modules to properly stop the game.

SSD Driver



The driver takes in a 4-bit number and generates a 7-bit code that lights up the seven segment display with the symbol for the input number.

Its ports are the 4-bit number input and a 7-bit code output.

[illegible]

The controller takes in all seven segment display codes and outputs them to the 8-digit display on the board, which can be seen on the bottom image.

It uses the board's 100MHz clock and a counter to refresh every digit on the display every 2ms, so that all numbers appear to be displayed at the same time.

Num 1, Num 2, and Num 3 are the inputted codes by the player, which are displayed on the 3 far left digits.

The last 2 digits on the 8-digit display are set to the countdown timer in which the player can see how much time left they have to guess the code.

Image Module Generator

One issue was generating images for the project.

- The script is written in **Python**
- Utilizes **PIL**, **OS**, and **CV2** libraries
- The code can be summarized as:
 - Creates a **New File** & names it
 - CV2 functions allow us to **process image**
 - Creates beginning of module
 - Loops through each pixel, outputs **color data** for the pixel inside a case statement
 - End module
- Allows for readily available modules for changes made to images we want to generate.

```
generate.py • monkey_blur_0_50_100_102_bit_rom.v
```

```
> generate > im
35 # make output filename from input
36 file_name = name.split('.')[0] + "_12_bit_rom.v"
37
38 # open file
39 f = open(file_name, 'w')
40
41 # get image dimensions
42 y_max, x_max, z = im.shape
43
44 # get width of row and column case words
45 row_width = math.ceil(math.log(y_max-1,2))
46 col_width = math.ceil(math.log(x_max-1,2))
47
48 # write beginning part of module up to case statements
49 f.write("module " + name.split('.')[0] + "_rom\n\t(\n\t\t\t")
50 f.write("input wire clk,\n\t\t\tinput wire [" + str(row_width-1) + ":0] row,\n\t\t\t")
51 f.write("input wire [" + str(col_width-1) + ":0] col,\n\t\t\t")
52 f.write("output reg [11:0] color_data\n\t);\n\t")
53 f.write("(* rom_style = \"block\" *)\n\t\t\t/n\t;/signal declaration\n\t")
54 f.write("reg [" + str(row_width-1) + ":0] row_reg;\n\t")
55 f.write("reg [" + str(col_width-1) + ":0] col_reg;\n\t\t\t")
56 f.write("always @(posedge clk)\n\t\t\tbegin\n\t\t\trow_reg <= row;\n\t\t\tcol_reg <= col;\n\t\t\tend\n\t\t\t")
57 f.write("always @*\n\t\t\tcase ({row_reg, col_reg})\n\t")
58
59
60 # loops through y rows and x columns
61 for y in range(y_max):
62     for x in range(x_max):
63         # write : color_data =
64         case = format(y, 'b').zfill(row_width) + format(x, 'b').zfill(col_width)
65         f.write("\t\t\t\tstr(row_width + col_width) + \"b\" + case + \": color_data = \" + str(12) + \"b\"\n\t\t\t\t")
66
67         # if mask is set to false, just write color data
68         if(mask == False):
69             f.write(get_color_bits(im, y, x))
70             f.write("; \n\t\t\t\t")
71
```