Whack-a-Mole

Kaanthi Pandhigunta and Cora Payne

Final Project Report
E155
December 10, 2021

Abstract

For our final project, we built a digital version of the arcade game Whack-a-Mole (WaM). Our setup consists of four major components: the STM32 microcontroller, the MAX1000 FPGA, a WaM board that we built, and a monitor. The FPGA drives a 640 x 480 pixel display on the monitor over VGA to display the moles. The WaM board consists of 3 platforms that are attached on top of snap action microswitches. The STM32 microcontroller controls gameplay by taking in inputs from the WaM board, processing them, and sending control signals to the FPGA via the shared GPIO pins on the breakout board.

Introduction

Motivation

For this project, we wanted to implement a fully functioning arcade game, so we brainstormed quite a few. We settled on Whack-a-Mole because we thought being violent towards some moles would allow us to blow off some steam while debugging. Furthermore, this game seemed possible to implement with our limited hardware and time.

Block Diagram

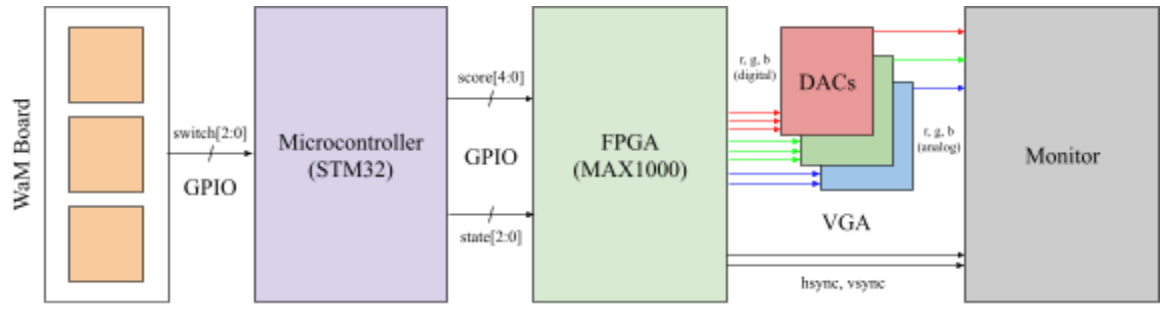


Figure 1. Overall system block diagram

Overview

For our version of the game, rather than making the moles rise mechanically to be whacked, we chose to have them appear on a screen for a newer, more digital approach. The moles appear at one of three locations on the screen in a random sequence and the user must whack the corresponding platform on the WaM board within a certain time limit. The user gets one point for each mole that is whacked, and the score is displayed at the top of the screen. The user is not penalized for hitting the wrong button or pressing the correct button multiple times, but only gets one point per mole if they hit the correct platform. As the game progresses, the moles appear in faster succession, requiring more concentration and speed from the user. Players win by reaching

the maximum score of 32 and lose by missing hitting a mole before time runs out. Either way, the end of the game takes the player back to the start screen, where they can choose to play again.

New Hardware

VGA Display

The VGA Display is the most significant piece of new hardware that we implemented for this project. We are driving the monitor at a resolution of 640 x 480 active pixels (800 x 525 pixels including porches) which requires a 25.175 MHz clock in order to have a frame refresh rate of approximately 60 Hz, a common threshold value for any flickering to be imperceptible. [1] Because the FPGA clock runs at 12 MHz, we needed to use a PLL to get the proper clock speed.

To interface with the monitor, we soldered wires to all the pins on a VGA connector and connected all the signals to the appropriate pins on the breadboard. We used a male-to-male VGA cable to wire the connector to the monitor.

The horizontal and vertical timing signals, hsync and vsync, were generated by the FPGA and outputted directly to the VGA display. We settled on traditional 8-bit color, which is 3 bits for red, 3 bits for green, and 2 bits for blue. The digital data for the color channels is converted to analog voltages of 0-0.7V using 2 and 3-bit resistor-ladder DACs. When we modeled the DACs as resistor-ladders, we included the 75Ω VGA termination resistance and chose resistor values so that the output voltage for each color channel would be approximately linearly proportional to its 3-bit or 2-bit digital control signal. The DACs and all resistor values are shown in full in the schematic in Appendix A.

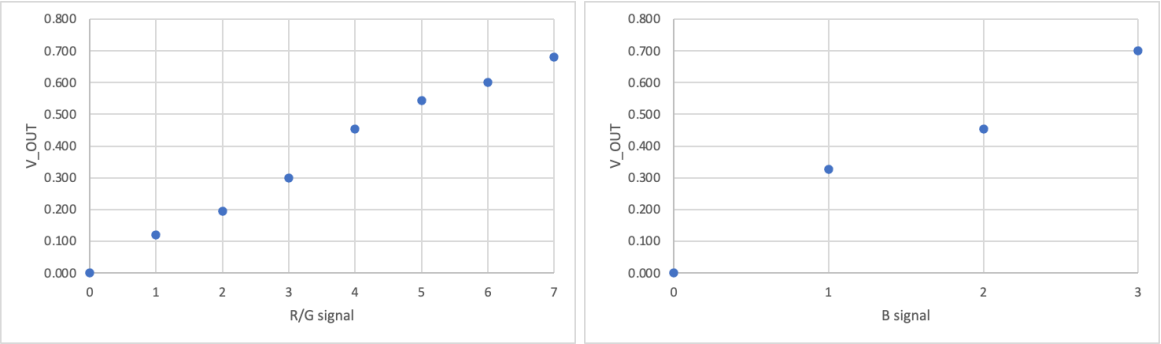


Figure 2. Expected resistor-ladder DAC voltage outputs for all possible color channel values.

Snap Action Microswitches

We decided to use snap action microswitches to collect inputs from the user because they are reliable for long-term use, and we knew we needed hardware that was robust enough to withstand repeated whacking. Microswitches have three connection points: common (C), normally open (NO), and normally closed (NC). When the switch is not being pressed, power flows from C to NC. When the switch is being pressed, power flows from C to NO. Because we want an input to be recorded only when the button is pressed, the NO terminal is most appropriate for this application. To accomplish this, we powered NC, wired C to an MCU GPIO pin, and grounded NC, so whenever the lever is pressed, an input is recorded. We used 3 microswitches to correspond to the 3 mole positions.

Microswitches are often used because they require low operating force. However, we found that the required operating force was too low for our application, because an input would be recorded if the user just tapped the lever rather than whacking it. To remedy this, we attached springs under the platforms and hovered them over the switches, so the user has to apply significant force to record an input.

MCU Design

In our design, the microcontroller runs the main game logic and sends game updates to the FPGA over a few control signals. The MCU takes inputs from the WaM board over 3 GPIO pins and dynamically outputs state and score data to the FPGA. We used 3 shared MCU and FPGA pins to transmit 3 state bits and 5 shared pins to transmit 5 score bits to the FPGA. Because we are using 5 score bits, the maximum score that a user can reach is 32, at which point they win and the game ends.

Our game logic consists of 7 states:

- State 3’b000: The first state displays the start screen. Once the user presses the start button, the game moves onto displaying the moles.
- States 3’b100, 3’b010, 3’b001: The mole positions are generated using a time-seeded random number generator in C, and each of the three mole positions corresponds to a different state.
- State 3’b110: We noticed that if the random number generator makes a mole reappear in the same position, it is difficult for the user to tell that they need to whack the same mole again. Thus, we added a state that is just the background with no moles that is displayed for a few milliseconds between mole iterations to distinguish them.

- State 3'b101: If the user does not whack the mole with enough force or within the correct amount of time, they lose, and the game moves into the lose state. The lose state displays a red screen for 3 seconds before automatically returning to the start state.
- State 3'b101: If the user manages to whack all the moles and reaches the maximum score of 32 (a difficult feat!), they win, and the game moves into the win state. The win state displays the win screen for 3 seconds before automatically returning to the start state.

Inside the main function, the outer loop continues infinitely, so the game can be replayed an infinite number of times. The first inner loop remains in the start state until the start button is pressed. The second loop is the most important as it keeps track of whether the user has lost or has reached the maximum score. To keep track of the mole duration, we used one of the MCU timers, TIM4, clocked slowly enough to let us time things on the scale of multiple seconds by using the maximum prescaler value. In the second loop, new moles are generated at increasingly faster speeds (the time window is a decreasing function of the user's current score) and evaluates if the user's input is correct. The user is not penalized for hitting the wrong button or pressing the correct button multiple times, but cannot get multiple points within the same mole iteration. The user loses if the correct input is not recorded within the allotted time and wins if they reach the maximum score of 32. In these cases, the game moves into the appropriate win or lose state for 3 seconds before automatically returning to the start screen.

FPGA Design

The FPGA displays graphics by taking data from the MCU and generating VGA color and control signals. We have multiple modules that can be considered either VGA control modules or pixelmap generation modules.

The VGA control modules instantiate the PLL that generates the pixel clock, generate VGA timing signals (hsync and vsync) based on that pixel clock, and determine red, green, and blue color signals for each pixel. The controller has an x-counter and y-counter. The x-counter increments every clock cycle and resets when it gets to the end of the line. The y-counter increments whenever the x-counter resets and resets when it gets to the bottom of the screen. The controller also drives hsync and vsync signals that control the blanking intervals during which no pixels are drawn. This resolution requires a pixel clock speed of 25.175MHz, so we used a built-in PLL module in Quartus.

The pixelmap generation modules take data from pixelmaps, which are stored in text files, and tell the VGA control modules what colors to display. Each sequence of 8 bits in the pixelmap corresponds to the color of one pixel on the screen. To generate these pixelmaps, we drew pixel art on drawing software and ran it through a MATLAB function we wrote to convert it into the correct format that the FPGA could take into ROM (for code and an example pixelmap, see

Appendix D). Because the M9K memory blocks on the FPGA are limited in the width of the data that they can take in, we chose to make our pixelmap 32 bits per line (which corresponds to four 8-bit pixels) instead of being as wide as the number of pixels on the screen. At first, we tried to create a pixelmap with 8 bits for each pixel on the 640 x 480 pixel screen. However, this generated a huge text file that took very long to compile, so we decreased the resolution of our pixelmaps, using 8 bits to encode either an 8x8 or 16x16 square of pixels on the screen, rather than just one pixel. These differences mean that indexing into the .txt file varies between modules controlling different parts of the game with different resolutions. We created pixelmap generation modules for the start screen, game background, moles, text/numbers, and the win screen.

To overlay pixelmaps on one another (for example, overlaying text on the start screen, or overlaying moles on the background), we used an extra control bit to decide if that particular pixel is a foreground pixel or background pixel. The VGA controllers display a foreground pixel if it exists in that location, but will default to the background pixel color if there is no foreground in that spot.

The FPGA changes what is being displayed based on the state bits that it receives from the MCU. It dynamically updates the score being displayed based on the score bits that it receives from the MCU.

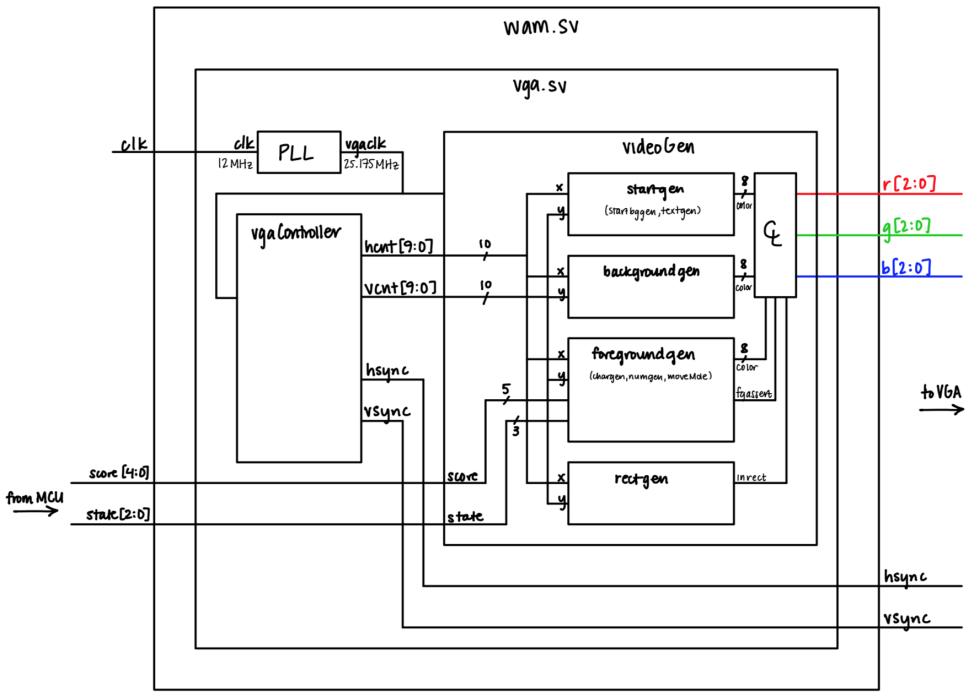


Figure 3. FPGA logic block diagram. Repeated sub-modules for generating components of larger graphics are shown in parentheses.

Results

Given our hardware and timeline constraints, our project turned out extremely well. We were able to create a fully functioning arcade game with human-machine interaction and VGA graphics of reasonable quality. We had a lot of fun playing it ourselves.

Our final submission had only a few minor differences from our project proposal. We realized that gameplay code was rather difficult to implement on the FPGA, so we decided to move it to the MCU. This way, we also used the MCU more significantly than we originally planned. Also, because the FPGA only needed a few control signals from the MCU, we decided that SPI was unnecessary, since it could more easily be done with shared GPIO pins. Furthermore, instead of using 3-DACs for each color channel, we used 3-bit DACs for the red and green channels and a 2-bit DAC for the blue channel because multiplication and division is simpler with 8 bits of color per pixel rather than 9 bits. Traditional 8-bit color encoding uses this setup because human eyes are less sensitive to blue light.

The timing of the game is crisp and has no perceptible latency between hitting platforms and the game state or score updating on the screen. The game counts the player's score accurately and can handle all inputs that we have tried, as we have been unable to break the game or force it into a failed state.

With our graphics, we ran into the problem of ghosting, where pixel colors bleed into adjacent pixels. We were not able to find the root cause of this, but we suspect that there is an issue with impedance matching based on a previous VGA project report. [2] We added an N14148 diode in series with each VGA color signal, which helped slightly with the ghosting, but did not totally eliminate it. If we had more time, we would have looked into using op-amps to try to resolve this. However, this issue did not significantly affect our game, as the text and major components are still clearly visible.

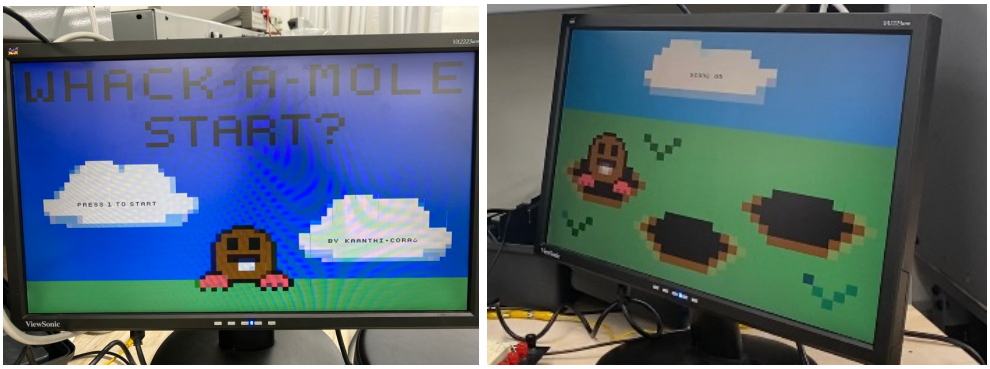


Figure 4. Start screen and gameplay screen.

Our WaM board was a lot more robust than we expected due to the springs we attached at the bottom, making whacking really fun. We laser cut some platforms out of some light, thin wood and engraved numbers and mole icons on them, which significantly contributed to the overall polish of our finished product.

Figure 5. Finished Whack-A-Mole board with laser-etched platforms and spring setup.

References

[1] Sarah L. Harris and David Harris, *Digital Design and Computer Architecture: ARM Edition*, Elsevier Science, 2016.

[2] Reynaldo Farias Zorrilla and Noah Boorstin, Pel Final Report, E155 Fall 2019. http://pages.hmc.edu/harris/class/e155/projects19/Farias_Zorrilla_Boorstin.pdf

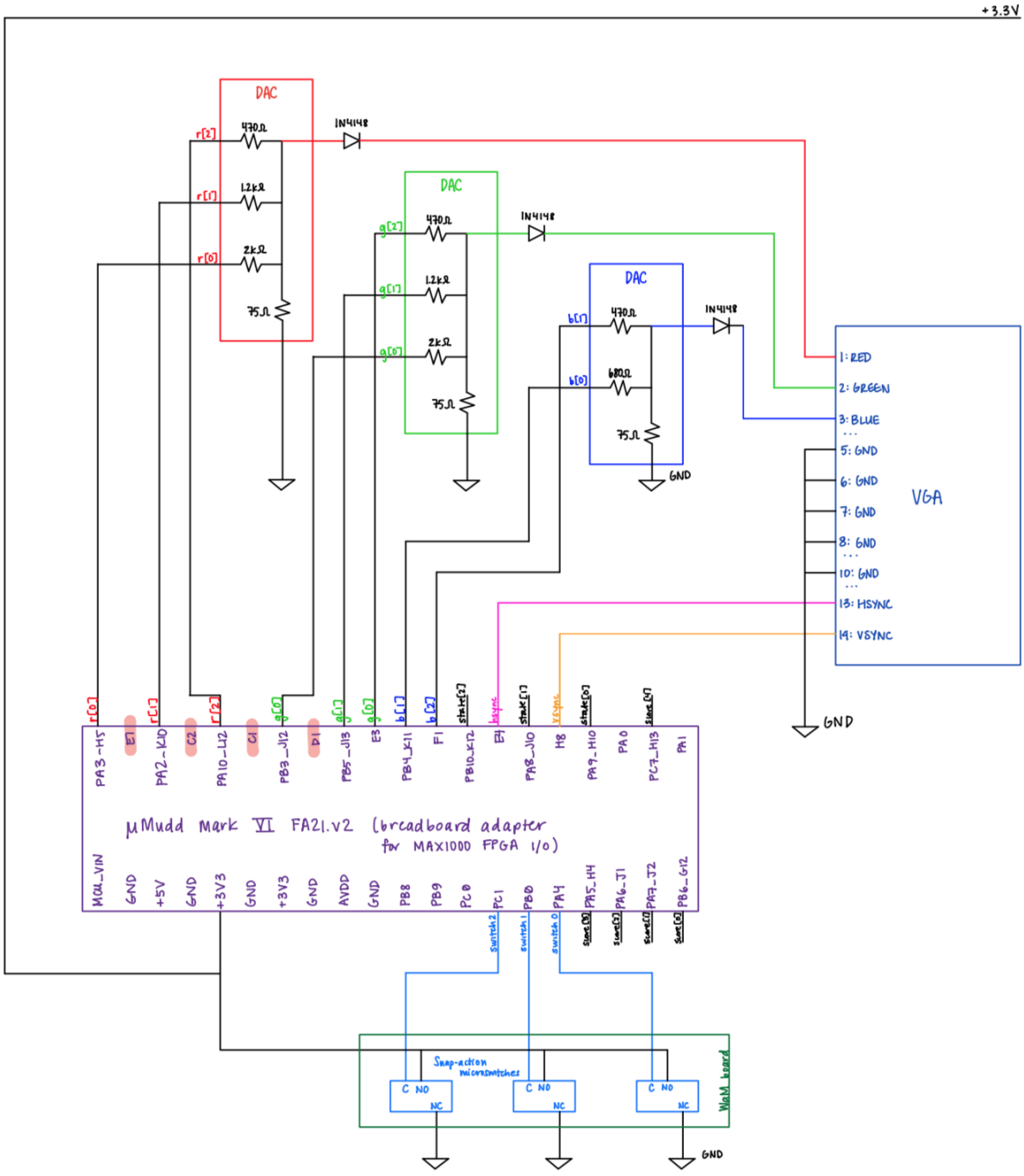
Bill of Materials

Part	Purpose	Part #	Source	Quantity	Budget Cost
Wood	Platforms for WaM board	N/A	Machine Shop	3	
Snap-action switches	WaM board digital input	CYT1073	Amazon (Cylewet)	3	$6.99 (pack of 25)
VGA M/M cable, 3 ft	Interface with monitor	5361	Monoprice	1	
VGA solder connector	Separate VGA signals	N/A	Stock Room	1	
Resistors, 470Ω	DAC components	N/A	Digital Lab	3	
Resistors, 1.2kΩ	DAC components	N/A	Digital Lab	2	
Resistors,	DAC	N/A	Digital Lab	2	

2kΩ	components				
Resistors, 680 Ω	DAC components	N/A	Digital Lab	1	
1N4148 Diodes	DAC components	N/A	Digital Lab	3	
Monitor	Display moles and player score	N/A	Digital Lab	1	
Springs	Support platforms and cushion whacks	N/A	Stock Room	12	
Nucleo STM32 MCU	Interface between switches and FPGA, run gameplay	NUCLEO-F401RE	Lab Kit	1	
MAX10 FPGA	Generate graphics, drive VGA display	TEI0001-03-08-C8	Lab Kit	1	
				Total Budget	$6.99

Appendix A: Schematics

Breadboard Schematic



Appendix B: FPGA SystemVerilog

wam.sv

```
// wam.sv
```

```
// Kaanthi Pandhigunta (kpandhigunta@hmc.edu)
// Cora Payne (cpayne@hmc.edu)
// E155 Final Project: Whack-A-Mole
// 10 December 2021
// Top-level WaM module


module wam(input  logic clk,
                     input  logic[4:0] score,
                     input  logic[2:0] state,
                     output logic hsync, vsync,
                     output logic[2:0] r, g,
                     output logic[1:0] b);

    vga display(clk, score, state, hsync, vsync, r, g, b);

endmodule
```

vga.sv

```
// vga.sv
// Kaanthi Pandhigunta (kpandhigunta@hmc.edu)
// Cora Payne (cpayne@hmc.edu)
// E155 Final Project: Whack-A-Mole
// 10 December 2021
// VGA module to display WaM graphics and text


module vga(input  logic clk,
                     input  logic[4:0] score,
                     input  logic[2:0] state,
                     output logic hsync, vsync,
                     output logic[2:0] r, g,
                     output logic[1:0] b);

    // pixel coordinates. (0,0) is the top left corner
    logic[9:0] x, y;

    // Use a PLL to create the 25.175 MHz VGA pixel clock
    // 25.175 MHz clkperiod = 39.772 ns
    // Screen is 800 clocks wide by 525470 kHz
    // Vsync= 31.474 kHz / 525 = 59.94 Hz (~60 Hz refresh rate)
    PLL vgapll(.inclk0(clk), .c0(vgaclk));

    // generate monitor timing signals
    vgaController vgaCont(vgaclk, hsync, vsync, x, y);

    // user-defined module to determine pixel color at each location
```

```
    videoGen videoGen(x, y, score, state, r, g, b);

endmodule


// Generates VGA timing signals (hsync, vsync) based on pixel clock
module vgaController
#(parameter HACTIVE = 10'd640,    // number of pixels per line
                    HFP = 10'd16,             // horizontal front porch
                    HSYN = 10'd96,            // horizontal sync pulse = 60 to move
electron gun back to left
                    HBP = 10'd48,             // horizontal back porch
                    HMAX = HACTIVE + HFP + HSYN + HBP,    // 48+640+16+96=800:
number of horizontal pixels
                    VBP = 10'd32,             // vertical back porch
                    VACTIVE = 10'd480,    // number of lines
                    VFP = 10'd11,             // vertical front porch
                    VSYN = 10'd2,             // vertical sync pulse = 2 to move
electron gun back to top
                    VMAX = VACTIVE + VFP + VSYN + VBP)    // 32+480+11+2=525: number
of vertical pixels

    (input  logic vgaclk,
     output logic hsync, vsync,
     output logic[9:0] hcnt, vcnt);

    // counters for horizontal and vertical states
    always@(posedge vgaclk) begin
//      if(reset) begin
//            hcnt<= 0;
//            vcnt<= 0;
//      end
        begin
              hcnt++;
              if(hcnt== HMAX) begin
                    hcnt<= 0;
                    vcnt++;
                    if(vcnt== VMAX)
                    vcnt<= 0;
              end
        end
    end

    // compute sync signals (active low)
    assign hsync= ~( (hcnt >= (HACTIVE + HFP)) & (hcnt < (HACTIVE + HFP + HSYN)) );
    assign vsync= ~( (vcnt >= (VACTIVE + VFP)) & (vcnt < (VACTIVE + VFP + VSYN)) );

endmodule
```

```
// Determines color signals (r, g, b) for each pixel based on current game state
module videoGen(input  logic[9:0] x, y,
                            input  logic[4:0] score,
                            input  logic[2:0] state,
                            output logic[2:0] r, g,
                            output logic[1:0] b);

    logic[7:0] startcolor, bgcolor, fgcolor, wcolor, lcolor;
    logic fgassert, inw, inl;

    startgen start(x, y, startcolor);
    backgroundgen background(x, y, bgcolor);
    foregroundgen foreground(x, y, score, state, fgcolor, fgassert);

    rectgen wrect(x, y, 10'd0, 10'd0, 10'd640, 10'd480, inw);
    rectgen lrect(x, y, 10'd0, 10'd0, 10'd640, 10'd480, inl);

    assign c = (state==3'b000) ? startcolor : ((state==3'b111&inw) ? 8'b00111000 :
((state==3'b101&inl) ? 8'b00000111 : (fgassert ? fgcolor : bgcolor)));

    assign r = c[2:0];
    assign g = c[5:3];
    assign b = c[7:6];

endmodule


// Generates a rectance with specified corner locations
module rectgen(input  logic[9:0] x, y, left, top, right, bot,
                           output logic inrect);

    assign inrect = (x >= left & x < right & y >= top & y < bot);

endmodule


// Generates start screen
module startgen(input  logic[9:0] x, y,
                            output logic[7:0] color);

    logic[7:0] bgcolor;
    logic textassert;

    startbggen sgen(x, y, bgcolor);
    textgen tgen(x, y, textassert);
```

```
    assign color = textassert ? 8'h00 : bgcolor;

endmodule


// Generates background image of start screen (8x8px blocks, 80x60-block image)
module startbggen(input  logic[9:0] x, y,
                                output logic[7:0] pxcolor);

    logic[31:0] startrom[1199:0];
    logic[31:0] line;                                       // a line read from the
ROM
    logic[7:0] c;

    // initialize ROM with characters from text file
    initial $readmemb("start.txt", startrom);    // start2

    // index into ROM to find line
    // index: y/8 * 20 + x/8 * 1/4
    // (1 block per 8 px line * 20 txt lines per px line + 1 block per 8 px * 1 txt
line holds 4 blocks)
    assign line = (x >= 0 & x <= 639 & y >= 0 & y <= 479) ? startrom[(y >> 3)*20 + (x
>> 5)] : 32'b0;

    // index: 31 - 8 * (x/8)mod4, get 8 bits (one color) at once
    // (8 because counting by 8's to get a color, 8 because block is 8x8px, mod4
because 32 bits per txt line holds 4 colors)
    assign c = (x >= 0 & x <= 639 & y >= 0 & y <= 479) ? line[(31-8*((x>>3)%4)) -: 8]
: 8'b0;
    assign pxcolor = {c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7]};

endmodule


// Displays pixel-resolution text on start screen
// "PRESS 1 TO START" on the left cloud
// "BY KAANTHI + CORA :)" on the right cloud
module textgen(input  logic[9:0] x, y,
                           output logic textassert);

    logic[7:0] numcolor;

    logic[9:0] left1, top1;
    logic text1assert, p5, p4, p3, p2, p1, p0, numassert, t1, t0, s5, s4, s3, s2, s1,
s0;
    assign left1 = 10'd80;
    assign top1 = 10'd264;
    chargen press4(8'd80, x, y, left1, top1, p5);
```

```
    chargen press3(8'd82, x, y, left1+10'd8, top1, p4);
    chargen press2(8'd69, x, y, left1+10'd16, top1, p3);
    chargen press1(8'd83, x, y, left1+10'd24, top1, p2);
    chargen press0(8'd83, x, y, left1+10'd32, top1, p1);

    numgen n(8'd1, x, y, left1+10'd40+4, top1, numcolor, numassert);

    chargen to1(8'd84, x, y, left1+10'd48+8, top1, t1);
    chargen to0(8'd79, x, y, left1+10'd56+8, top1, t0);

    chargen start4(8'd83, x, y, left1+10'd64+12, top1, s5);
    chargen start3(8'd84, x, y, left1+10'd72+12, top1, s4);
    chargen start2(8'd65, x, y, left1+10'd80+12, top1, s3);
    chargen start1(8'd82, x, y, left1+10'd88+12, top1, s2);
    chargen start0(8'd84, x, y, left1+10'd96+12, top1, s1);

    assign text1assert = (p5|p4|p3|p2|p1|p0)|numassert|(t1|t0)|(s5|s4|s3|s2|s1|s0);

    logic[9:0] left2, top2;
    logic text2assert, b1, b0, k6, k5, k4, k3, k2, k1, k0, p, c3, c2, c1, c0, s;
    assign left2 = 10'd464;
    assign top2 = 10'd340;
    chargen by1(8'd66, x, y, left2-10'd24, top2, b1);
    chargen by0(8'd89, x, y, left2-10'd16, top2, b0);

    chargen kaanthi6(8'd75, x, y, left2, top2, k6);
    chargen kaanthi5(8'd65, x, y, left2+10'd8, top2, k5);
    chargen kaanthi4(8'd65, x, y, left2+10'd16, top2, k4);
    chargen kaanthi3(8'd78, x, y, left2+10'd24, top2, k3);
    chargen kaanthi2(8'd84, x, y, left2+10'd32, top2, k2);
    chargen kaanthi1(8'd72, x, y, left2+10'd40, top2, k1);
    chargen kaanthi0(8'd73, x, y, left2+10'd48, top2, k0);

    chargen plus(8'd93, x, y, left2+10'd56, top2, p);

    chargen cora3(8'd67, x, y, left2+10'd64, top2, c3);
    chargen cora2(8'd79, x, y, left2+10'd72, top2, c2);
    chargen cora1(8'd82, x, y, left2+10'd80, top2, c1);
    chargen cora0(8'd65, x, y, left2+10'd88, top2, c0);

    chargen smile(8'd92, x, y, left2+10'd96+4, top2, s);

    assign text2assert = (b1|b0)|(k6|k5|k4|k3|k2|k1|k0)|p|(c3|c2|c1|c0)|s;

    //assign textassert =
(p5|p4|p3|p2|p1|p0)|numassert|(t1|t0)|(s5|s4|s3|s2|s1|s0)|(b1|b0)|(k6|k5|k4|k3|k2|k1|
k0)|p|(c3|c2|c1|c0)|s;
    assign textassert = text1assert | text2assert;
```

```
endmodule


// Generates foreground (overlaid) graphics for game screen: score text, moles
module foregroundgen(input  logic[9:0] x, y,
                                       input  logic[7:0] score,
                                       input  logic[2:0] state,
                                       output logic[7:0] fgcolor,
                                       output logic fgassert);

    // display score
    logic p5, p4, p3, p2, p1, p0;
    chargen score_s(8'd83, x, y, 10'd288, 10'd80, p5);
    chargen score_c(8'd67, x, y, 10'd296, 10'd80, p4);
    chargen score_o(8'd79, x, y, 10'd304, 10'd80, p3);
    chargen score_r(8'd82, x, y, 10'd312, 10'd80, p2);
    chargen score_e(8'd69, x, y, 10'd320, 10'd80, p1);
    chargen colon(8'd91, x, y, 10'd328, 10'd80, p0);

    logic[7:0] score1, score0;
    logic[7:0] s1color, s0color;
    logic s1, s0;
    assign score1 = score/10;
    assign score0 = score - 10*(score/10);
    numgen digit1(score1, x, y, 10'd336, 10'd80, s1color, s1);
    numgen digit0(score0, x, y, 10'd344, 10'd80, s0color, s0);

    logic textassert;
    assign textassert = s1|s0|p5|p4|p3|p2|p1|p0;

    // display mole in correct location
    logic[7:0] molecolor;
    logic moleassert;
    moveMole m1(state, x, y, molecolor, moleassert);

    assign fgcolor = (moleassert ? molecolor : (textassert ? 8'h00 : 8'hFF));
    assign fgassert = moleassert | textassert;

endmodule


// Generates the gameplay background image (16x16px blocks, 40x30-block image)
module backgroundgen(input  logic[9:0] x, y,
                                       output logic[7:0] pxcolor);

    logic[31:0] backgroundrom[299:0];    // background ROM
    logic[31:0] line;                          // a line read from the ROM
```

```
    logic[7:0] c;

    // initialize ROM with characters from text file
    initial $readmemb("background.txt", backgroundrom);    // wam_background4_v2

    // index into ROM to find line
    // index: y/16 * 10 + x/16 * 1/4
    // (1 block per 16 px line * 10 txt lines per px line + 1 block per 16 px * 1 txt
line holds 4 blocks)
    assign line = (x >= 0 & x <= 639 & y >= 0 & y <= 479) ? backgroundrom[(y >> 4)*10
+ (x >> 6)] : 32'b0;

    // index: 31 - 8 * (x/16)mod4, get 8 bits (one color) at once
    // (8 because counting by 8's to get a color, 16 because block is 16x16px, mod4
because 32 bits per txt line holds 4 colors)
    assign c = (x >= 0 & x <= 639 & y >= 0 & y <= 479) ? line[(31-8*((x>>4)%4)) -: 8]
: 8'b0;
    assign pxcolor = {c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7]};

endmodule


// Generates a movable mole sprite (16x16px image)
module molegen(input  logic[9:0] x, y, left, top,
                          output logic[7:0] pxcolor,
                          output logic fgassert);

      logic[31:0] molerom[63:0];              // mole generator ROM
    logic[31:0] line;                         // a line read from the ROM
    logic[9:0] xdiff, ydiff;
    logic[7:0] c;

    // offsets from top left corner of sprite area
    assign ydiff = y-top;
    assign xdiff = x-left;

    // initialize ROM with characters from text file
    initial $readmemb("molesprite.txt", molerom);    // molesprite2

    // index into ROM to find line
    // index: y/8 * 4 + x/8 * 1/4
    // (1 block per 8 px line * 4 txt lines per px line + 1 block per 8 px * 1 txt
line holds 4 blocks)
    assign line = (x >= left & x < (left+8'd128) & y >= top & y < (top+8'd128)) ?
molerom[(ydiff >> 3)*4 + (xdiff >> 5)] : 32'b0;
    assign fgassert = (x >= left & x < (left+8'd128) & y >= top & y < (top+8'd128)) ?
1'b1 : 1'b0;
```

```
    // reverse order of bits
    // index: 31 - 8 * (x)mod4, get 8 bits (one color) at once
    // (8 because counting by 8's to get a color, mod4 because 32 bits per txt line
holds 4 colors)
    assign c = fgassert ? line[(31-8*((xdiff>>3)%4)) -: 8] : 8'b0;
    assign pxcolor = {c[0], c[1], c[2], c[3], c[4], c[5], c[6], c[7]};

endmodule


// Places mole sprite at one of 3 locations (mole holes)
module moveMole(input  logic[2:0] state,
                           input  logic[9:0] x, y,
                           output logic[7:0] color,
                           output logic fgassert);

    logic[7:0] color1, color2, color3;
    logic fgassert1, fgassert2, fgassert3;

    molegen mg1(x, y, 10'd064, 10'd216, color1, fgassert1);    // left
    molegen mg2(x, y, 10'd256, 10'd296, color2, fgassert2);    // middle
    molegen mg3(x, y, 10'd448, 10'd216, color3, fgassert3);    // right

    always_comb
        case(state)
              3'b100:      begin
                                    color = color1;
                                    fgassert = fgassert1;
                                end
              3'b010:      begin
                                    color = color2;
                                    fgassert = fgassert2;
                                end
              3'b001:      begin
                                    color = color3;
                                    fgassert = fgassert3;
                                end
              default:    begin
                                    color = 8'h00;
                                    fgassert = 1'b0;
                                end
        endcase

endmodule


// Generates a character from a bitmap
module chargen(input  logic[7:0] ch,
```

```
                            input  logic[9:0] x, y, left, top,
                            output logic[7:0] c);

        logic[5:0] charrom[231:0];       // character generator ROM
    logic[7:0] line;                     // a line read from the ROM
    logic[9:0] xdiff, ydiff;
    logic pixel;

    // offsets from top left corner of character area
    assign ydiff = y-top;
    assign xdiff = x-left;

    // initialize ROM with characters from text file
    initial $readmemb("charrom.txt", charrom);

    // index into ROM to find line of character
    assign line = (y>=top & y<=top+10'd7) ? charrom[ydiff[2:0]+{ch-8'd65, 3'b000}] :
8'b0;

    // reverse order of bits
    assign pixel = (x>=left & x<=left+10'd7) ? line[3'd7-xdiff[2:0]] : 1'b0;

    // assign text pixels to white
    assign c = pixel ? 8'b111111 : 8'b000000;

endmodule


// Generates a digit from a bitmap
module numgen(input  logic[7:0] num,
                            input  logic[9:0] x, y, left, top,
                            output logic[7:0] c,
                            output logic pixel);

    logic[5:0] numrom[79:0];            // number generator ROM
    logic[7:0] line;                           // a line read from the ROM
    logic[9:0] xdiff, ydiff;

    // offsets from top left corner of character area
    assign ydiff = y-top;
    assign xdiff = x-left;

    // initialize ROM with characters from text file
    initial $readmemb("numrom.txt", numrom);

    // index into ROM to find line of number
    assign line = (y>=top & y<=top+10'd7) ? numrom[ydiff[2:0]+{num, 3'b000}] : 8'b0;
```

```
    // reverse order of bits
    assign pixel = (x>=left & x<=left+10'd7) ? line[3'd7-xdiff[2:0]] : 1'b0;

    // assign text pixels to black
    assign c = pixel ? 8'h00 : 8'hFF;

endmodule
```

Appendix C: MCU code

main.c

```
// main.c
// Kaanthi Pandhigunta (kpandhigunta@hmc.edu)
// Cora Payne (cpayne@hmc.edu)
// E155 Final Project: Whack-A-Mole
// 10 December 2021
// WaM gameplay and VGA control signals for FPGA graphics

#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_TIMx.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// 3 pins for switch inputs
#define START_PIN   1   // PC1
#define SWITCH2_PIN 1   // PC1  (1. left)
#define SWITCH1_PIN 0   // PB0  (2. middle)
#define SWITCH0_PIN 4   // PA4  (3. right)

// 5 pins for communicating score to FPGA
#define SCORE4_PIN  7   // PC7_H13
#define SCORE3_PIN  5   // PA5_H4
#define SCORE2_PIN  6   // PA6_J1
#define SCORE1_PIN  7   // PA7_J2
#define SCORE0_PIN  6   // PB6_G12

// 3 pins for communicating game state to FPGA
#define STATE2_PIN 10   // PB10_K12
#define STATE1_PIN  8   // PA8_J10
#define STATE0_PIN  9   // PA9_H10

// TIM4 clock speed
```

```
#define CLK4        (int) 84000000/65536      // 84 MHz / (PSC+1) (TIM4->PSC.PSC set
in config function)

// Gameplay constants (times in ms)
#define MAX_SCORE   32     // score[4:0] has 5 pins to send to FPGA --> max = 2^5
#define MAX_TIME    1000   // starting time to hit the mole [ms], decreases as score
increases
#define NOMOLE_TIME  200   // time between moles (none appear on screen)
#define WIN_TIME    2000   // how long to display win screen
#define LOSE_TIME   2000   // how long to display lose screen

// duration calculates the counter value equivalent for a time interval (input in ms,
output in TIM4 clock cycles)
int duration(int d) {
      volatile uint16_t dur;
      dur = (CLK4*d)/1000;
      return dur;
}

// wait causes a delay for a set time interval (input in ms)
void wait(int d) {
      volatile uint16_t dur = duration(d);
      TIM4->CNT.CNT = 0;
      while(TIM4->CNT.CNT <= dur){};
}

// writeScore drives GPIO output pins to communicate the score to the FPGA
void writeScore(uint8_t score) {

      int binaryScore[8] = {0, 0, 0, 0, 0, 0, 0, 0};
      volatile int i = 0;
      while (score > 0) {
            binaryScore[i] = score % 2;
            score = score / 2;
            i++;
      }

      digitalWrite(GPIOC, SCORE4_PIN, binaryScore[4]);
      digitalWrite(GPIOA, SCORE3_PIN, binaryScore[3]);
      digitalWrite(GPIOA, SCORE2_PIN, binaryScore[2]);
      digitalWrite(GPIOA, SCORE1_PIN, binaryScore[1]);
      digitalWrite(GPIOB, SCORE0_PIN, binaryScore[0]);
}

// writeState drives GPIO output pins to communicate the game state to the FPGA
void writeState(uint8_t state) {
      if(state == 0) { // start
            digitalWrite(GPIOB, STATE2_PIN, 0);
```

```
			digitalWrite(GPIOA, STATE1_PIN, 0);
			digitalWrite(GPIOA, STATE0_PIN, 0);
		} else if (state == 1) { // mole 1
			digitalWrite(GPIOB, STATE2_PIN, 1);
			digitalWrite(GPIOA, STATE1_PIN, 0);
			digitalWrite(GPIOA, STATE0_PIN, 0);
		} else if (state == 2) { // mole 2
			digitalWrite(GPIOB, STATE2_PIN, 0);
			digitalWrite(GPIOA, STATE1_PIN, 1);
			digitalWrite(GPIOA, STATE0_PIN, 0);
		} else if (state == 3) { // mole 3
			digitalWrite(GPIOB, STATE2_PIN, 0);
			digitalWrite(GPIOA, STATE1_PIN, 0);
			digitalWrite(GPIOA, STATE0_PIN, 1);
		} else if (state == 4) { // win
			digitalWrite(GPIOB, STATE2_PIN, 1);
			digitalWrite(GPIOA, STATE1_PIN, 1);
			digitalWrite(GPIOA, STATE0_PIN, 1);
		} else if (state == 5) { // lose
			digitalWrite(GPIOB, STATE2_PIN, 1);
			digitalWrite(GPIOA, STATE1_PIN, 0);
			digitalWrite(GPIOA, STATE0_PIN, 1);
		} else if (state == 6) { // blank state
			digitalWrite(GPIOB, STATE2_PIN, 1);
			digitalWrite(GPIOA, STATE1_PIN, 1);
			digitalWrite(GPIOA, STATE0_PIN, 0);
		}
}

// newMole randomly generates a new state (mole position)
int newMole(){
		volatile int mole = (rand() % 3) + 1;
		return mole;
}

// resetMoleTimer restarts the counter (TIM4) to time mole appearances
void resetMoleTimer(){
		TIM4->CNT.CNT = 0;
}

int main(void) {

		// Configure flash latency and set clock to run at 84 MHz
		configureFlash();

		// Enable GPIO clocks
		RCC->AHB1ENR.GPIOAEN = 1;
		RCC->AHB1ENR.GPIOBEN = 1;
```

```
	RCC->AHB1ENR.GPIOCEN = 1;

	// Enable TIM4 for timing the mole's apperances
	RCC->APB1ENR |= (1 << 2);   // TIM4 EN (RM, 119)

	configureClock();
	configureTIM4();          // in standard driver (STM32F401RE_TIMx.c), PSC =
65535

	// Input pins for switches
	pinMode(GPIOC, SWITCH2_PIN, GPIO_INPUT);
	pinMode(GPIOB, SWITCH1_PIN, GPIO_INPUT);
	pinMode(GPIOA, SWITCH0_PIN, GPIO_INPUT);

	// Output pins for score
	pinMode(GPIOC, SCORE4_PIN, GPIO_OUTPUT);
	pinMode(GPIOA, SCORE3_PIN, GPIO_OUTPUT);
	pinMode(GPIOA, SCORE2_PIN, GPIO_OUTPUT);
	pinMode(GPIOA, SCORE1_PIN, GPIO_OUTPUT);
	pinMode(GPIOB, SCORE0_PIN, GPIO_OUTPUT);

	// Output pins for game state
	pinMode(GPIOB, STATE2_PIN, GPIO_OUTPUT);
	pinMode(GPIOA, STATE1_PIN, GPIO_OUTPUT);
	pinMode(GPIOA, STATE0_PIN, GPIO_OUTPUT);

	// Seed random number generator to randomize mole appearances
	srand((unsigned int) time(NULL));

	while (1) {
	volatile int state = 0;
	volatile int start = 0;
	volatile int score = 0;
	volatile int death = 0;
	volatile int moleTime = MAX_TIME;
	writeScore(score);
	writeState(state);

	// Wait for player to start game
	while(!start) {
		if(digitalRead(GPIOC, START_PIN)) {start = 1;}
	}

	// Main gameplay loop
	while (!death && (score < MAX_SCORE)) {
		// Blank state (no mole displayed)
		state = 6;
		writeState(state);
```

```
            wait(NOMOLE_TIME);

            // On to the next random mole
            state = newMole();
            writeState(state);

            volatile int read = 0;
            volatile int scorechanged = 0;
            moleTime = MAX_TIME - 150*sqrt(score);        // Time window to whack
mole decreases
            volatile int moleDuration = duration(moleTime);
            resetMoleTimer();
            while(TIM4->CNT.CNT <= moleDuration){
            // Check if the correct switch is pressed; incorrect hits aren't
penalized
            if (state == 1) {
                  if (digitalRead(GPIOC, SWITCH2_PIN)) {read = 1;}
            } else if (state == 2) {
                  if (digitalRead(GPIOB, SWITCH1_PIN)) {read = 1;}
            } else if (state == 3) {
                  if (digitalRead(GPIOA, SWITCH0_PIN)) {read = 1;}
            }
            // Player scored and it hasn't been counted yet --> increment score only
once
            if (read && !scorechanged) {
                  score += 1;
                  writeScore(score);
                  scorechanged = 1;
            }
            };

            // Didn't hit in time --> player loses
            if (!scorechanged) {
                  death = 1;
                  writeState(5);
                  wait(LOSE_TIME);
                  Break;
            } // Reached max score --> player wins
            else if (score == MAX_SCORE) {
                  writeState(4);
                  wait(WIN_TIME);
                  Break;
            }
      }
      }
}
```

Appendix D: MATLAB code and Pixelmap Example

make8bitmap.m

```
function make8bitmap(filename, width, height, b)
% converts a .png image into an 8-bit color pixelmap for vga.sv

if nargin<4
	b = 1;		% block size, default is full resolution
end

im = imread([filename '.png']);
tic
str = '';
for i=1:b:height
	for j=1:b:width
	color = '';
	for k=1:2
		rg = median(im(i:i+(b-1),j:j+(b-1),k));
		if rg < 32
		channel = '000';
		elseif rg < 64
		channel = '001';
		elseif rg < 96
		channel = '010';
		elseif rg < 128
		channel = '011';
		elseif rg < 160
		channel = '100';
		elseif rg < 192
		channel = '101';
		elseif rg < 224
		channel = '110';
		else
		channel = '111';
		end
		color = strcat(color, channel);
	end
	bl = median(im(i:i+(b-1),j:j+(b-1),3));
	if bl < 64
		channel = '00';
	elseif bl < 127
		channel = '01';
	elseif bl < 191
		channel = '10';
	else
		channel = '11';
	end
```

```
        color = strcat(color, channel);

        str = strcat(str, color);       % add block's color to running str

        if ~mod(j+(b-1),4*b)            % 32 bits/line --> 4 colors/line
                str = strcat(str, '\n');
        end
        end
end
toc
fprintf(fopen([filename '.txt'],'w'), str);   % save to a .txt file

end
```

molesprite.txt

```
// Encodes 16 x 16 px mole sprite:
// 32 bits/txt line * 1 px/8 bits = 4 px/txt line
// 16 image lines * 16 px/image line * 1 txt line/4 px = 64 txt lines
01110101011101010111010101110101
01110101011101010000000000000000
00000000000000000111010101110101
01110101011101010111010101110101
01110101011101010111010101110101
00000000010001001000110010001100
10001100100011000100010000000000
01110101011101010111010101110101
01110101011101010111010101000100
10001100100011001000110010001100
10001100100011001000110010001100
01000100011101010111010101110101
01110101011101010111010101000100
10001100000000000000000010001100
10001100000000000000000010001100
01000100011101010111010101110101
01110101011101010100010010001100
10001100000000000000000010001100
10001100000000000000000010001100
10001100010001000111010101110101
01110101011101010100010010001100
10001100000000000000000010001100
10001100000000000000000010001100
10001100010001000111010101110101
01110101011101010100010010001100
10001100100011001000110010001100
10001100100011001000110010001100
10001100010001000111010101110101
01110101011101010100010010001100
```

```
10001100010001000100010001000100
01000100010001000100010010001100
10001100010001000111010101110101
00000000000000000100010010001100
10001100010001000100100101101101
01101101010010010100010010001100
10001100010001000000000000000000
00000000000000000100010010001100
10001100100011001101011011111111
11111111110101101000110010001100
10001100010001000000000000000000
00000000000000000100010010001100
10001100100011001011001011011011
11011011101100101000110010001100
10001100010001000000000000000000
00000000000000000000000000000000
10001100100011001000110010001100
10001100100011001000110010001100
01000100000000000000000000000000
00100000111011011110110111101101
10101001100011001000110010001100
10001100100011001000110001000100
11101101111011011110110101100100
11101101111011011110110111101101
11101101000000000000000000000000
00000000000000000000000011101101
11101101111011011110110111101101
11101101101010011110110110101001
11101101000000000000000000000000
00000000000000000000000011101101
10101001111011011010100111101101
10001100100011000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000001000110010001100
```