Riley Peters

CPE 233 Section 1

Hardware Assignment 9: Snake User Manual

**Introduction:**

My final project implements a recreation of the game “Snake” in RISC-V assembly code. The game is designed to be played on a VGA monitor using the buttons included with the BASYS 3 board. Aside from the monitor and board, no other peripherals are required. The user controls the snake as it moves around a 32x32 plane, steering it with the buttons included on the Basys board. The main objective is to collect as many apples as possible by passing over them. Each apple adds a point to the score and increases the length of the snake. As the game progresses, it becomes more difficult to traverse the board due to its added length. If the snake collides with itself or a wall the game ends. The final score is shown on the seven-segment display, which resets upon the start of a new game. To ensure that the game is responsive to user input, the buttons will register the most recent press every 40 ns. Additionally, the player is given a half-second pause each time the snake moves, providing just enough time for quick decisions.

**Operation Manual:**

**Required Components:**

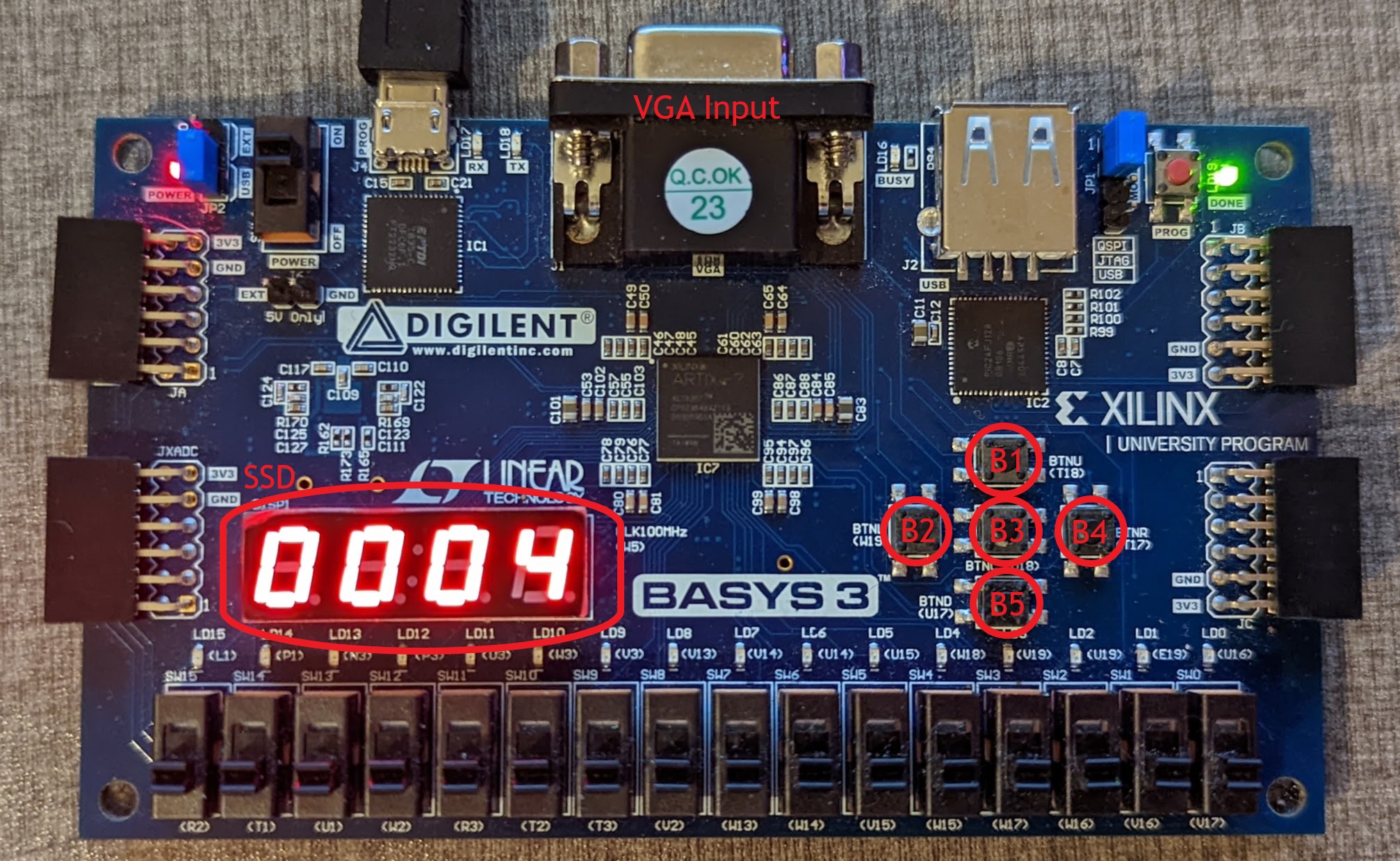
1. Basys 3 FPGA board

2. VGA monitor and cable

3. OTTER MCU SystemVerilog files and Snake.mem

4. Xilinx Vivado

**Figure 1:** Basys Board Locations of Interest



**Setup:**

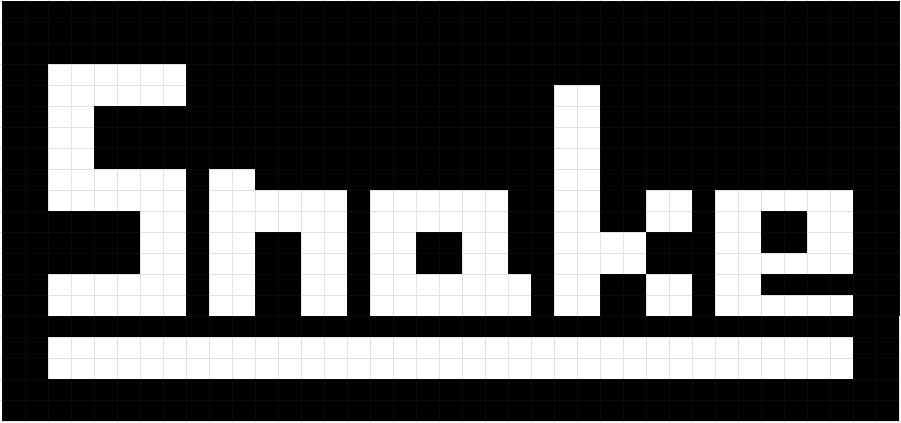
1. Connect the Basys board to a compatible VGA monitor through the included VGA input

2. Open the included SystemVerilog and mem files in Vivado and select “generate bitstream”

3. With the Basys board plugged into the computer, open hardware manager, select auto-connect, and program device

4. Once programmed, the title screen should appear (Figure 2)

**Figure 2:** Snake Title Heading



**How to Play:**

1. The program begins at the title screen (Figure 2)

2. To start the game, press any of the movement buttons (B1 - B4)

3. The snake spawns as a green dot on a 32x32 grid (Figure 3) that traverses using the movement buttons as a directional pad. B1 corresponds to up, B2 to left, B3 to down, and B4 to right. Based on the most recent button press, the dot will move in the specified direction every half-second

* Since a 180-degree turn will always lead to a self-collision (ex. switching from moving left to moving right), that form of movement is disabled. Attempting to do so will not change the direction of the snake.

4. Apples are drawn as red pixels on the screen. Slither over them to gain points and grow the snake.

5. The score is provided on the seven-segment display and is equal to the number of apples eaten during the current game

6. If the snake runs into itself or a wall the game will end, returning to the title screen

7. The game can be restarted at any point by pressing the reset button (B5)

| **Figure 3:** Game Objects  Board (Black): 32x32 plane of moveable space where the snake can move and collect apples.  Walls (White): The 34x34 boundary of the board, colliding with it ends the game  Apple (Red): Randomly placed objectives that give points and increase length when collected  Snake (Green): Character controlled by the player as a means of collecting apples |  |
| --- | --- |

**Software:**

**Hold State:**

The hold state is the intermediary between games, active on start-up and between games. This state displays the title screen and enables hardware interrupts. While a single button could be polled repeatedly instead of using an interrupt, implementing “start game” using interrupts allows the use of more than one button as a trigger without polling. As a result, any of the movement buttons (B1 - B4) can be pressed to jump to setup.

| **Figure 4:** Hold State Functions Flowchart |  |
| --- | --- |

**Setup State:**

Setup acts as a transitionary period between the title screen and gameplay. It can be broken down into three primary phases followed by a transition to the action state.

| **Figure 5:**  Setup State  Functions Flowchart |  |
| --- | --- |

**1. Load Board:**

All pixels on the screen are set to black, erasing the title screen. Then, a 34x34 square border is drawn in the top left corner surrounding a 32x32 area. The inner area is used as the game board.

**2. Load Snake:**

The first snake piece is placed at position 5x5 on the board. Its location address is also written to memory at the beginning of the data segment. Storing the snake segments in memory allows the program to keep track of the entire snake. If the snake extends, the new head position is stored at the next available location in memory.

**3. Load Apple:**

A 32-bit pseudorandom number is generated from a module included in the OTTER MCU wrapper. The lowest 5 bits are used to specify the x position while bits 16-20 are for y. With a random coordinate created, the program searches the location to ensure that it is empty. If open, the apple is placed. Otherwise, the command repeats until a space is found.

**Action State:**

Action is active while the game is running. Given the current user input, the program reacts by moving, extending, or killing the snake depending on its next location. If the snake dies by colliding with itself or a wall, the code returns to the hold state awaiting the next game.

**1. Next Move:**

1. The program decides whether a move is valid or not. Since turning 180 degrees will always kill the snake, the direction of the next move is compared to the current orientation to check if they oppose. If the moves are opposite, the snake maintains its previous course

2. Given a new direction, the program updates the x and y values of the snake’s head

accordingly

**2. Update Position:**

The code explores the new location for obstacles:

**If Empty (Move):**

The snake head is drawn at the new location, and the tail is deleted. As well as that, the current position of each snake segment stored in memory is updated to reflect the new locations.

**If Apple (Eat):**

The new snakehead is drawn over the apple, and its location is appended to the array of snake segment positions in memory. To signify the increase in size, the pointer that locates the top of the array increases. After updating the snake, a new apple is placed at a random position on the board.

**If Snake or Wall (Gameover):**

Upon collision, the program jumps back to the hold state and awaits the next game.

**3. Delay:**

To allow the user enough time to move the snake in their desired direction, the program delays for a half-second between updates. This effect is achieved simply by having a counter compare itself to a constant delay value.

| **Figure 6:**  Action State Functions Flowchart  \*continued  below |  |
| --- | --- |

| **Figure 7:**  VGA Interface Functions  These helper functions allow the code to more easily read from and write to the VGA monitor. |  |
| --- | --- |

**Snake RISC-V Assembly Code:**

| .data  .space 3200 #snake memory allocation .text  .eqv DATA, 0x6000  .eqv IO\_ADDR, 0x11000000  .eqv BOARD\_SIZE, 33  .eqv DELAY\_TIME, 0x002FAF08 #1/4 second   #0x005F5E10, 1/2 second    .eqv APPLE, 0xE0  .eqv SNAKE, 0x1C  .eqv BG\_COLOR, 0  .eqv WALL, 0xFF  .eqv VG\_ADDR, 0x11000120  .eqv VG\_COLOR, 0x11000140  # pre-run state of code  # used as an intermediate state between games hold:  li s10, IO\_ADDR #IO address  li sp, 0x10000  call draw\_title #draws the title screen  la t0, main   csrw t0, mtvec #enables interrupt to main method  addi t0, zero, 1  csrw t0, mie STANDBY: beqz zero, STANDBY  # main method: # defines global variables # calls setup methods # enters the game loop main:   li s0, DATA #snake tail data address  add s1, s0, zero #snake head data address  add s2, zero, zero #snake head x location  add s3, zero, zero #snake head y location  add s4, zero, zero #score count  add s5, zero, zero #snake direction  li s11, DELAY\_TIME #load delay  sw s4, 0x40(s10) #update ssd with the initial score  call loadBoard #generates board  call loadSnake #loads inital snake position  call loadApple #places the first apple on the board RUN: call action #run loop  j RUN  # resets the board between games # fill the center square of the board with black # fill the outer border of the board with white loadBoard:  addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  call draw\_background  add a0, zero, zero  add a1, zero, zero  addi a2, zero, BOARD\_SIZE   li a3, WALL  call draw\_horizontal\_line  add a0, zero, zero  add a1, zero, zero  addi a2, zero, BOARD\_SIZE  call draw\_vertical\_line  add a0, zero, zero  addi a1, zero, BOARD\_SIZE  addi a2, zero, BOARD\_SIZE  call draw\_horizontal\_line  addi a0, zero, BOARD\_SIZE  add a1, zero, zero  addi a2, zero, BOARD\_SIZE  call draw\_vertical\_line  lw ra, 0(sp) #pop ra and return  addi sp, sp, 4  ret  # loads snake data in memory # places the snake head on the board  loadSnake:  addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  addi a0, zero, 5  addi a1, zero, 5  add s2, zero, a0 #update current snake x and y position  add s3, zero, a1  slli t0, a1, 7 #save position address in data  add t0, t0, a0  sh t0, 0(s0) #load vga position into data  li a3, SNAKE  call draw\_dot  lw ra, 0(sp) #pop ra and return  addi sp, sp, 4  ret  # generate a pseudorandom coordinate and place an apple object in that location # if the location contains the snake, repeat until an empty space is found  loadApple:  addi sp, sp, -4 #push ra to stack  sw ra, 0(sp) RELOAD: lw t0, 0x60(s10) #load pseudorandom number  andi a0, t0, 0x1F #X: 0 to 31  addi a0, a0, 1 #shift 1 position to avoid wall  srli t0, t0, 16  andi a1, t0, 0x1F #Y: 0 to 31  addi a1, a1, 1 #shift 1 position to avoid wall  call read\_dot  bnez a2, RELOAD #if not empty: retry  addi a3, zero, APPLE #load red color  call draw\_dot  lw ra, 0(sp) #pop ra and return  addi sp, sp, 4  ret #return   # delays for a short period # any button presses during the delay period are registered  delay:  addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  add t0, zero, zero WAIT: addi t0, t0, 1  bne t0, s11, WAIT  lw ra, 0(sp) #return  addi sp, sp, 4  ret  # method loop that begins when the game is played # consists of an input check and two separate case segments: # 1. verify that the next move is legal (not opposite to current motion) # 2. generate next position based on the most recent button press # 3. given next position, determine if the snake moves, eats, or dies action:   addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  lw t0, 0x100(s10) #load next move from button  andi t2, t0, 1 #checks in case previous move contradicts new move  beq t2, zero, ADD #i.e. you cant move left and then right  SUB: addi t2, t0, -1   beq t2, s5, INVAL  j VALID ADD: addi t2, t0, 1  beq t2, s5, INVAL  j VALID INVAL: add t0, s5, zero #if the next move is invalid, continue in previous direction VALID: add s5, t0, zero #set the previous direction to the current direction  addi t1, zero, 3 #comparator (3 = UP, 2 = DOWN, 1 = LEFT, 0 = RIGHT) UP: bne t0, t1, DOWN #if next move is UP:  addi s3, s3, -1 #decrement y position  j OBST DOWN: addi t1, zero, 2   bne t0, t1, LEFT #if next move is DOWN:  addi s3, s3, 1 #increment y position  j OBST LEFT: addi t1, zero, 1  bne t0, t1, RIGHT #if next move is LEFT:  addi s2, s2, -1 #decrement x position  j OBST RIGHT: addi s2, s2, 1 #else, increment x position OBST: add a0, zero, s2 #use a0 and a1 temporaries for helper methods  add a1, zero, s3  call read\_dot #explore next space EMPTY: bnez a2, HUNGRY #if the next space is empty:  call move #move to the next space  j NEXT HUNGRY: li t3, APPLE   bne a2, t3, DEATH #if the next space contains an apple:  call eat #eat the apple  j NEXT DEATH: j hold #otherwise, gameover NEXT: call delay #delay between moves  lw ra, 0(sp) #return  addi sp, sp, 4  ret    # moves the snake a single space in the direction it is traveling: # draws the head at the next position, clears the tail location, # and refreshes the VGA addresses stored in memory move:  addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  li a3, SNAKE #load snake color  add a0, zero, s2 #load snake position into arguments  add a1, zero s3  call draw\_dot #draw snake head position  lhu t1, 0(s0) #load the address at the snake's tail  li a3, BG\_COLOR   srli a1, t1, 7 #convert the address to xy coord  andi a0, t1, 0x3F  call draw\_dot #clear the tail of the snake from the board  slli t2, s3, 7  add t2, t2, s2 #turn x, y coord into address  add t3, s1, zero #head address temporary UPDATE: blt t3, s0, MOVED #While there are more segments to visit  lhu t4, 0(t3) #load the address stored in that location  sh t2, 0(t3) #replace the address with the one stored in the next segment  add t2, t4, zero #set loaded address to be stored in the next segment  addi t3, t3, -2 #move to the next spot in memory  j UPDATE MOVED: lw ra, 0(sp) #return  addi sp, sp, 4  ret  # generates a new snake piece and increments the score when an apple is encountered # makes an internal call to loadApple so that a new apple is available eat:   addi sp, sp, -4 #push ra to stack  sw ra, 0(sp)  li a3, SNAKE #load snake color  addi s1, s1, 2 #add an extra snake piece  slli t0, s3, 7  add t0, t0, s2 #generate location address  sh t0, 0(s1)  add a0, zero, s2  add a1, zero, s3  call draw\_dot #draw new snake head  addi s4, s4, 1 #add 1 to the score  sw s4, 0x40(s10) #update ssd with the new score  call loadApple #generate a new apple  lw ra, 0(sp) #return  addi sp, sp, 4  ret  # draws a horizontal line from (a0,a1) to (a2,a1) using color in a3 # Modifies (directly or indirectly): t0, t1, a0, a2 draw\_horizontal\_line:  addi sp,sp,-4  sw ra, 0(sp)  addi a2,a2,1 #go from a0 to a2 inclusive draw\_horiz1:  call draw\_dot # must not modify: a0, a1, a2, a3  addi a0,a0,1  bne a0,a2, draw\_horiz1  lw ra, 0(sp)  addi sp,sp,4  ret  # draws a vertical line from (a0,a1) to (a0,a2) using color in a3 # Modifies (directly or indirectly): t0, t1, a1, a2 draw\_vertical\_line:  addi sp,sp,-4  sw ra, 0(sp)  addi a2,a2,1 draw\_vert1:  call draw\_dot # must not modify: a0, a1, a2, a3  addi a1,a1,1  bne a1,a2,draw\_vert1  lw ra, 0(sp)  addi sp,sp,4  ret  # Fills the 60x80 grid with one color using successive calls to draw\_horizontal\_line # Modifies (directly or indirectly): t0, t1, t4, a0, a1, a2, a3 draw\_background:  addi sp,sp,-4  sw ra, 0(sp)  li a3, BG\_COLOR #use default color  li a1, 0 #a1= row\_counter  li t4, 60 #max rows Start: li a0, 0  li a2, 79 #total number of columns  call draw\_horizontal\_line # must not modify: t4, a1, a3  addi a1,a1, 1  bne t4,a1, start #branch to draw more rows  lw ra, 0(sp)  addi sp,sp,4  ret  # draws a dot on the display at the given coordinates: # (X,Y) = (a0,a1) with a color stored in a3 # (col, row) = (a0,a1) # Modifies (directly or indirectly): t0, t1 draw\_dot:  andi t0,a0,0x7F # select bottom 7 bits (col)  andi t1,a1,0x3F # select bottom 6 bits (row)  slli t1,t1,7 # {a1[5:0],a0[6:0]}   or t0,t1,t0 # 13-bit address  sw t0, 0x120(s10) # write 13 address bits to register  sw a3, 0x140(s10) # write color data to frame buffer  ret  # reads the color data at the given coordinates: # (X,Y) = (a0,a1) # (col, row) = (a0,a1) # a2 = color data # Modifies (directly or indirectly): t0, t1  read\_dot:  andi t0,a0,0x7F # select bottom 7 bits (col)  andi t1,a1,0x3F # select bottom 6 bits (row)  slli t1,t1,7 # {a1[5:0],a0[6:0]}   or t0,t1,t0 # 13-bit address  sw t0, 0x120(s10) # write 13 address bits to register  lw a2, 0x160(s10) # read color data from frame buffer  ret  # draws the title screen visible before the program runs  draw\_title:  addi sp,sp,-4  sw ra, 0(sp)  call draw\_background  li a3, WALL  call draw\_header  lw ra, 0(sp)  addi sp,sp,4  ret  # long series of draw\_line calls that spell out "Snake" # used for the title screen draw\_header:  addi sp,sp,-4  sw ra, 0(sp) S: addi a0, zero, 22  addi a1, zero, 19  addi a2, zero, 25  call draw\_vertical\_line  addi a0, zero, 23  addi a1, zero, 19  addi a2, zero, 25  call draw\_vertical\_line  addi a0, zero, 24  addi a1, zero, 19  addi a2, zero, 27  call draw\_horizontal\_line  addi a0, zero, 24  addi a1, zero, 20  addi a2, zero, 27  call draw\_horizontal\_line  addi a0, zero, 24  addi a1, zero, 24  addi a2, zero, 27  call draw\_horizontal\_line  addi a0, zero, 24  addi a1, zero, 25  addi a2, zero, 27  call draw\_horizontal\_line  addi a0, zero, 26  addi a1, zero, 26  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 27  addi a1, zero, 26  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 22  addi a1, zero, 29  addi a2, zero, 25  call draw\_horizontal\_line  addi a0, zero, 22  addi a1, zero, 30  addi a2, zero, 25  call draw\_horizontal\_line n:   addi a0, zero, 29  addi a1, zero, 24  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 30  addi a1, zero, 24  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 31  addi a1, zero, 25  addi a2, zero, 34  call draw\_horizontal\_line  addi a0, zero, 31  addi a1, zero, 26  addi a2, zero, 34  call draw\_horizontal\_line  addi a0, zero, 33  addi a1, zero, 27  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 34  addi a1, zero, 27  addi a2, zero, 30  call draw\_vertical\_line a:  addi a0, zero, 36  addi a1, zero, 25  addi a2, zero, 41  call draw\_horizontal\_line  addi a0, zero, 36  addi a1, zero, 26  addi a2, zero, 41  call draw\_horizontal\_line  addi a0, zero, 36  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 37  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 40  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 41  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 36  addi a1, zero, 29  addi a2, zero, 42  call draw\_horizontal\_line  addi a0, zero, 36  addi a1, zero, 30  addi a2, zero, 42  call draw\_horizontal\_line k:  addi a0, zero, 44  addi a1, zero, 20  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 45  addi a1, zero, 20  addi a2, zero, 30  call draw\_vertical\_line   addi a0, zero, 46  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 47  addi a1, zero, 27  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 48  addi a1, zero, 25  addi a2, zero, 26  call draw\_vertical\_line  addi a0, zero, 49  addi a1, zero, 25  addi a2, zero, 26  call draw\_vertical\_line  addi a0, zero, 48  addi a1, zero, 29  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 49  addi a1, zero, 29  addi a2, zero, 30  call draw\_vertical\_line e:   addi a0, zero, 51  addi a1, zero, 25  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 52  addi a1, zero, 25  addi a2, zero, 30  call draw\_vertical\_line  addi a0, zero, 55  addi a1, zero, 25  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 56  addi a1, zero, 25  addi a2, zero, 28  call draw\_vertical\_line  addi a0, zero, 53  addi a1, zero, 25  addi a2, zero, 54  call draw\_horizontal\_line  addi a0, zero, 53  addi a1, zero, 28  addi a2, zero, 54  call draw\_horizontal\_line  addi a0, zero, 53  addi a1, zero, 30  addi a2, zero, 56  call draw\_horizontal\_line underline:  addi a0, zero, 22  addi a1, zero, 32  addi a2, zero, 56  call draw\_horizontal\_line  addi a0, zero, 22  addi a1, zero, 33  addi a2, zero, 56  call draw\_horizontal\_line  lw ra, 0(sp)  addi sp,sp,4  ret |
| --- |

**Modified OTTER Wrapper:**

\*adds random number generation and directional buttons to Wrapper provided in the OTTER\_VGA download

| module OTTER\_Wrapper(  input CLK,  input BTNL,  input BTNC,  input BTNR,  input BTNU,  input BTND,  input [15:0] SWITCHES,  output logic [15:0] LEDS,  output [7:0] CATHODES,  output [3:0] ANODES,  output [7:0] VGA\_RGB,  output VGA\_HS,  output VGA\_VS  );    *// INPUT PORT IDS ////////////////////////////////////////////////////////*  *// Right now, the only possible inputs are the switches*  *// In future labs you can add more MMIO, and you'll have*  *// to add constants here for the mux below*  localparam SWITCHES\_AD = 32'h11000000;  localparam RANDOM\_AD = 32'h11000060;  localparam BUTTONS\_AD = 32'h11000100;  localparam VGA\_READ\_AD = 32'h11000160;    *// OUTPUT PORT IDS ///////////////////////////////////////////////////////*  *// In future labs you can add more MMIO*  localparam LEDS\_AD = 32'h11000020;  localparam SSEG\_AD = 32'h11000040;  localparam VGA\_ADDR\_AD = 32'h11000120;  localparam VGA\_COLOR\_AD = 32'h11000140;     *// Signals for connecting OTTER\_MCU to OTTER\_wrapper /////////////////////////*  logic s\_reset, s\_interrupt;  logic sclk = 1'b0;   logic [31:0] IOBUS\_out,IOBUS\_in,IOBUS\_addr;  logic IOBUS\_wr;    *// Signals for connecting VGA Framebuffer Driver*  logic r\_vga\_we; *// write enable*  logic [12:0] r\_vga\_wa; *// address of framebuffer to read and write*  logic [7:0] r\_vga\_wd; *// pixel color data to write to framebuffer*  logic [7:0] r\_vga\_rd; *// pixel color data read from framebuffer*    logic [15:0] r\_SSEG;    *// Connect Signals ////////////////////////////////////////////////////////////*  assign s\_interrupt = btn\_int;  assign s\_reset = BTND;    *// Declare OTTER\_CPU ///////////////////////////////////////////////////////*  OTTERMCU MCU (.CPU\_RST(s\_reset),.CPU\_INTR(s\_interrupt), .CPU\_CLK(sclk),   .CPU\_IOBUS\_OUT(IOBUS\_out),.CPU\_IOBUS\_IN(IOBUS\_in),  .CPU\_IOBUS\_ADDR(IOBUS\_addr),.CPU\_IOBUS\_WR(IOBUS\_wr));   *// Declare Seven Segment Display /////////////////////////////////////////*  SevSegDisp SSG\_DISP (.DATA\_IN(r\_SSEG), .CLK(CLK), .MODE(1'b0),  .CATHODES(CATHODES), .ANODES(ANODES));  // Configure Directional Pad and Interrupts //////////////////////////////  assign btn\_int = (BTNL | BTNR | BTNC | BTNU);  logic [1:0] button\_pressed = 2'd0;      always\_ff @ (posedge sclk) begin  if (BTNR == 1'b1)  button\_pressed <= 2'd0;  if (BTNL == 1'b1)  button\_pressed <= 2'd1;  if (BTNC == 1'b1)  button\_pressed <= 2'd2;  if (BTNU == 1'b1)  button\_pressed <= 2'd3;  end    *// Declare Pseudorandom Number Generator /////////////////////////////////*  logic [31:0] pseudoRand;  RandGen rg (.CLK(CLK), .RST(1'b0), .RANDOM(pseudoRand));    *// Declare VGA Frame Buffer //////////////////////////////////////////////*  vga\_fb\_driver\_80x60 VGA(.CLK\_50MHz(sclk), .WA(r\_vga\_wa), .WD(r\_vga\_wd),  .WE(r\_vga\_we), .RD(r\_vga\_rd), .ROUT(VGA\_RGB[7:5]),  .GOUT(VGA\_RGB[4:2]), .BOUT(VGA\_RGB[1:0]),  .HS(VGA\_HS), .VS(VGA\_VS));     *// Clock Divider to create 50 MHz Clock /////////////////////////////////*  always\_ff @(posedge CLK) begin  sclk <= ~sclk;  end   *// Connect Board peripherals (Memory Mapped IO devices) to IOBUS /////////////////////////////////////////*  always\_ff @ (posedge sclk) begin  r\_vga\_we<=0;   if(IOBUS\_wr)  case(IOBUS\_addr)  LEDS\_AD: LEDS <= IOBUS\_out[15:0];   SSEG\_AD: r\_SSEG <= IOBUS\_out[15:0];  VGA\_ADDR\_AD: r\_vga\_wa <= IOBUS\_out[12:0];  VGA\_COLOR\_AD: begin   r\_vga\_wd <= IOBUS\_out[7:0];  r\_vga\_we <= 1;   end   endcase  end    always\_comb begin  case(IOBUS\_addr)  SWITCHES\_AD: IOBUS\_in = {16'b0, SWITCHES};  RANDOM\_AD: IOBUS\_in = pseudoRand;  BUTTONS\_AD: IOBUS\_in = {30'b0, button\_pressed};   VGA\_READ\_AD: IOBUS\_in = {24'b0, r\_vga\_rd};  default: IOBUS\_in = 32'b0;  endcase  end  endmodule |
| --- |