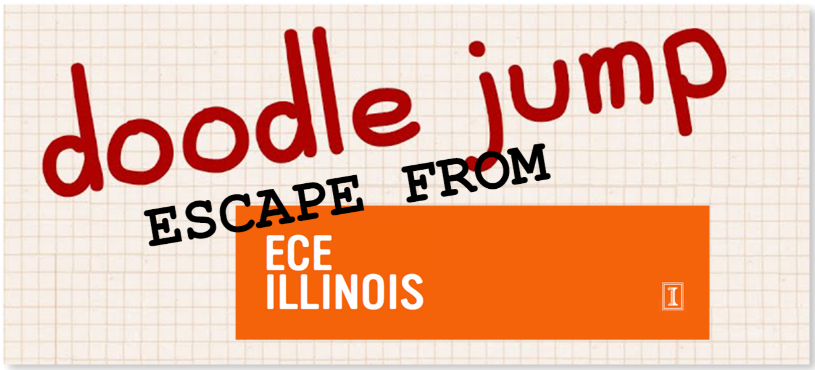
ECE 385

Spring 2019

Final Report

Doodle Jump FPGA & System Verilog

Jack Gronenthal, Sruthi Bhattiprolu
Section ABJ, Friday 2pm - 5pm
David Zhang



Introduction

Doodle Jump is a mobile game popularized in 2013 which features a caricature protagonist who is tasked with navigating a course of platforms littered with monster foes. Coming into contact with said monsters brings upon the Doodle a swift death. To protect the unassuming cartoon character, its been equipped with the ability to shoot and terminate the unsavory fiends. Each time the figure comes in contact with a platform, the Doodle is boosted up wards; however, if the Doodle unsuccessfully maneuvers the course and falls through the platforms, the Doodle meets its untimely demise. To aid the cartoonish protagonist, power-ups can be found occasionally sprinkled throughout the platforms giving the Doodle extra boosts. Score accumulated with each increase in elevation.

Block Diagram

The Doodle Jump system implemented features a bevy of modules all interconnected to produce the desired functionality of the game. *Figure 1* below decomposes said modules into two categories: *Game Utilities* and *System Utilities*. Modules belonging to the subsection of *Game Utilities* enable essential game behavior whereas modules belonging to the subsection of *System Utilities* enable essential system communications of peripherals (ie. The VGA monitor, the keyboard, etc.). Subsequent sections will discuss these modules in depth however, and overview is offered immediately below.

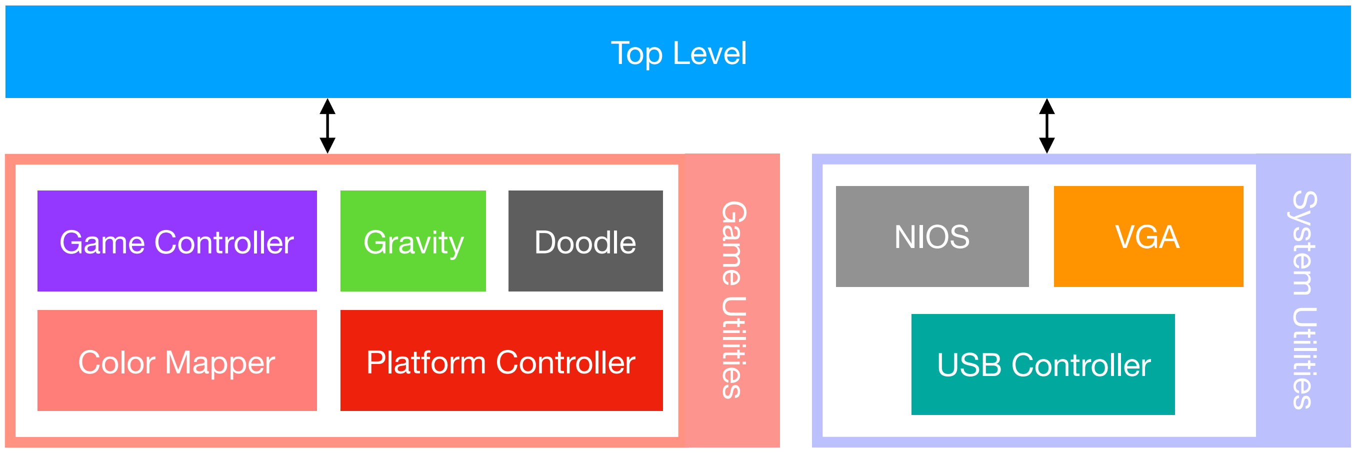


Figure 1
The figure above gives a high level representation of the data movement through the Doodle Jump System. The *Top Level* module acts as a conduit for data to flow between subordinate modules such as *Game Controller* to *Color Mapper* or *USB Controller* to *Doodle*.

Game Controller	The *Game Controller* behaves as the Finite State Machine of the system. Primary features include determining which “screen mode” the game is currently within (such as *welcome screen, game, death, etc.*)
Gravity	The *Gravity* module determines the Y-Position component for the *Doodle.* After a predetermined set of time, a constant is subtracted from the velocity of the sprite which is ultimately used to update the position of the sprite.
Doodle	The *Doodle* module determines the X-Position component for the *Doodle*. This is then routed into the Color Mapper module to continuously print the updated position of the *Doodle*.
Color Mapper	The *Color Mapper* module combinationally determines the output RGB value for any given pixel on the VGA display through its internal logic which constantly assesses which sprite is to be printed and which are not.

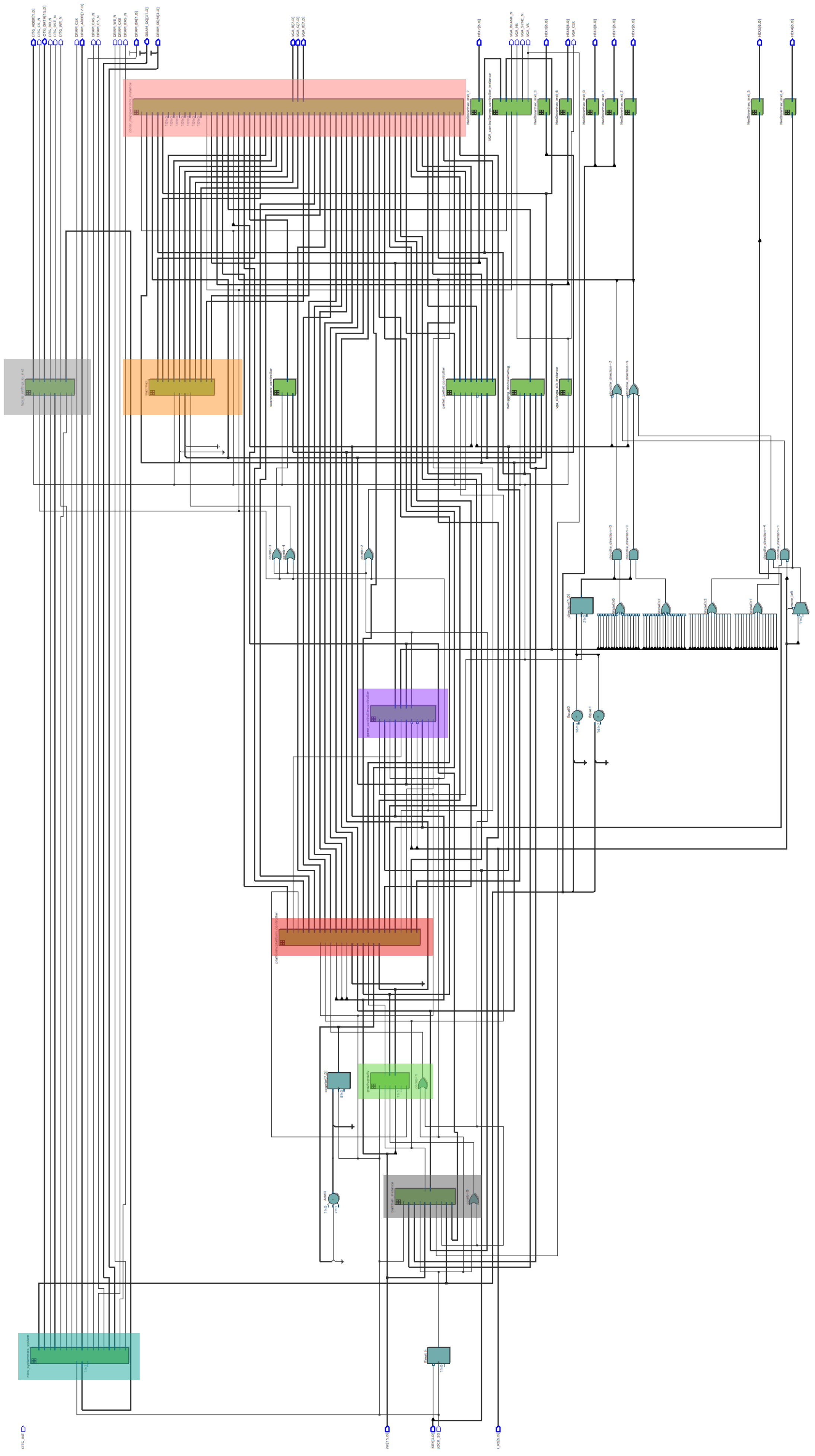
Platform Controller	The *Platform Controller* module handles the logic associated with the motion of platforms, the Doodle intersecting Platforms resulting in a jump, and the randomization of new platform levels.
NIOS	The NIOS II SoC is used to interface with the USB key board via the EZ OTG Cypress protocol. The keyboard is used as one of the control interfaces of the game along with the accelerometer
VGA	The VGA module takes in a variety of signals used for the VGA interface such that the monitor may properly output the data that is passed through the VGA cable.
USB Controller	Handles communication between the Keyboard and the FPGA. This is essential for implementing the Keyboard game mode.

Mechanics

To start the discussion oriented on the mechanics and technicalities of the implementation of the game, it is best to open with the *gravity* module. Just as experienced in the physical world, the doodle constantly is experiencing a gravitational effect. This is done through the use of a module explicitly tasked with controlling the vertical physics of the doodle's movement. To elaborate on the methodology employed to enable gravity, a constant is defined as the gravitational force. Said gravitational force then affects the behavior of the Doodle by constantly subtracting the Doodle's Y-Velocity at a regular interval (such as with the physical world). This updated Y-Velocity is then used to determine the "vertical step"—the number of pixels the sprite moves between screen updates—which then translates into the Doodle's position. All in all, this sequence of arithmetic operations models gravity nicely and permits the Doodle to move smoothly under a gravitational influence.

Furthermore, the Doodle jumps upon contact with any of the green platforms. This is done through constantly polling the X and Y positions on the screen of the Doodle and checking wether or not a platform appears within a window beneath the sprite. Accessing if a platform exists beneath the sprite is done by similarly polling the X and Y positions of every platform. If the Doodle satisfies one of the conditions in question, then a JUMP signal is produced which is then routed into the *gravity* module discussed above. Within, the Y-Velocity of the Doodle is updated in such a way which results in the sprite moving upward momentarily. The method of polling the X and Y component of the sprite is, in large part, how many of the game features operate including contact with monsters and power-ups. In order to ensure that this is regime works at a proper speed, all of these conditions are done combinationally.

On the topic of monsters, the system must be able to determine if the contact that is made with the monster is to result in a killed Doodle or a killed monster. This is done through, as mentioned previously, polling the X and Y components of the Doodle all the while taking into account the vertical direction of the Doodle. If the Doodle is moving down the screen and u


OTG_ADDR[1..0]
OTG_CS_N
OTG_DATA[15..0]
OTG_RD_N
OTG_RST_N
OTG_WR_N
DRAM_CLK
DRAM_ADDR[12..0]
DRAM_CAS_N
DRAM_CS_N
DRAM_WE_N
DRAM_CKE
DRAM_RAS_N
DRAM_BA[1..0]
DRAM_DQ[31..0]
DRAM_DQM[3..0]
VGA_B[7..0]
VGA_G[7..0]
VGA_R[7..0]
HEX7[6..0]
VGA_BLANK_N
VGA_HS
VGA_SYNC_N
VGA_VS
HEX3[6..0]
HEX6[6..0]
VGA_CLK
HEX0[6..0]
HEX1[6..0]
HEX2[6..0]
HEX5[6..0]
HEX4[6..0]
HexDriver:hex_inst_7
vga_controller_instance
HexDriver:hex_inst_3
HexDriver:hex_inst_6
HexDriver:hex_inst_0
HexDriver:hex_inst_1
HexDriver:hex_inst_2
HexDriver:hex_inst_5
HexDriver:hex_inst_4
OTG_INT
KEY[3..0]
Reset_h

makes contact with the monster, the system interprets this contact as killing the monster. If the Doodle, however, makes contact with the monster while moving up the screen then the system interprets this contact as the Doodle kill. It should be mentioned that if the Doodle makes contact with the Doodle under a critical vertical displacement below the Y component of the monster—as is the case when the Doodle makes contact horizontally with the monster—the system interprets this contact as a killed Doodle.

To defend from the monsters, pellets may be shot from the Doodle. These pellets take the X and Y position values from the Doodle once they are produced and move with a constant Y velocity up the screen. If they make contact with the monster, the monster dies; however, if they drift off the screen, the Doodle is free to shoot another pellet. Additionally, a power-up is available in the game, which enables the *Curve Mode* which gives the Doodle another chance at the game before encountering the death screen. The *Curve Mode* may also be invoked at random depending on when the Doodle crosses the bottom of the screen.

Finite State Machine

In order to implement the desired behavior of Doodle Jump, a Finite State Machine has been employed. Within the *Game Controller* module, the FSM included below may be found. It visually demonstrates that the system may only exist in a variety of states: *Welcome_ESCAPE, Welcome_SETTINGS, Game_setup, Settings_INPUT, Settings_COLOR, Death,* and *Game.*

Welcome_ESCAPE: Welcome_ESCAPE is the initial state of the system. Upon the *RESET* the system returns to the Welcome_ESCAPE state. The bifurcation of the welcome screen is done for assistance of the *Color Mapper*. Within the Welcome_ESCAPE state, the ESCAPE button—which when pressed enters the game—is selected. This, as a consequence, means that the INPUT button is unselected which is expressed within the output signals of the Welcome_ESCAPE state.

Welcome_SETTINGS: Welcome_SETTINGS is the state which visually outputs the INPUT button as being selected and the ESCAPE button as unselected. When the ENTER key is pressed, the state transitions into the Settings_INPUT screen.

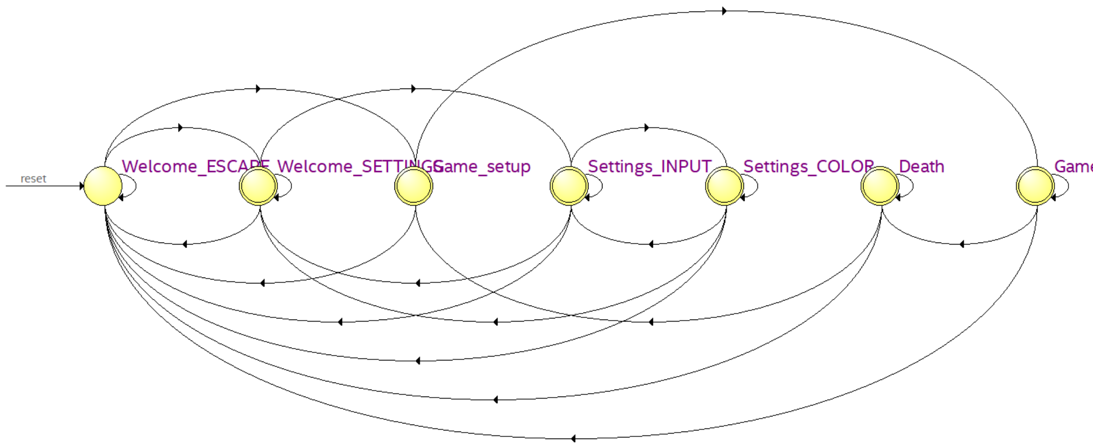
Game_setup: In order to reset the game values though not to reset the global variables that govern the nature of the game—such as the color of the doodle, input mode, etc.—the Game_setup state is created to enable specific resets that target values that need to be restored following every game attempt. This state transitions unconditionally into *Game*.

Settings_INPUT: Settings_INPUT is the initial screen on the settings menu. Upon pressing enter, the input mode changes from the keyboard to the accelerometer. This data is maintained throughout the system such that when the *Game* state is achieved, the data that has been selected from the *Settings_INPUT* state are maintained. Upon pressing the right button, the system transitions into the *Settings_COLOR* state.

Settings_COLOR: Settings_COLOR is the state in which the color of the Doodle may be selected. Upon pressing the ENTER key, the value within the COLOR array is incremented such that when the system enters the *Game* state and the Doodle is printed, the Doodle's appearance may change in color depending on which input is selected. Upon pressing the left button the system transitions back to the *Settings_INPUT* state.

Death: The Death state is used to express to the *Color Mapper* that the death sprite is to be displayed. Upon pressing ENTER, the game restarts.

Game: Within the *Game* state, the system is existing within the game regime. That is, take in keypresses or accelerometer data and move the Doodle accordingly. When the Doodle dies, detected depending on the *Curve Logic*, the system transitions into the *Death* state.



It should be noted that the FSM implemented is a Mealy State Machine such that the outputs of the current state depend on the current state and its inputs. This is done primarily to control the color and input mode through system states.

Video Graphics Array

Video Graphics Array (VGA) is the monitor interface used within the construction of the lab. This is a consequence of the hardware limitations of the DE2 board. To use within the project, two modules have to be instantiated: *VGA_Controller* and *VGA_Clk.* The *VGA_Controller* module is used as the primary engine for the VGA interface. Within, logic is instantiated to output a variety of essential signals with two being of particular importance. *DrawX* and *DrawY,* outputs of said module, behave as X and Y coordinates on the screen, integral in the process of displaying images upon the monitor. These signals sweep across the screen, one row of pixels at a time, pixel by pixel, and denote the current pixel that is available to be colored. This is of great importance to the *Color Mapper* module.

Commenting upon the second module instantiated to ensure the VGA interface is working appropriately, the *VGA_Clk* module produces a distinct clock which has a frequency slower than the clock native to the FPGA. The resultant clock, *VGA_Clk*, is then used to pace the VGA interface as it prints pixel by pixel. Because the nature of the *VGA_Clk* being nearly half the speed of the FPGA clock, the system has ample time to prepare signals that shall be printed upon the *VGA_Clk* rising edge. This ensures that the colors of the desired image are available to be displayed upon *DrawX* and *DrawY* sweeping past a particular coordinate on the monitor.

Accelerometer

The accelerometer used in as a controller in this project is the ADXL345 accelerometer. The ADXL 345 is a versatile accelerometer that can be communicated to using various protocols such as the SPI, I2C, FIFO and interrupt method. The communication protocol used in this project to communicate with the FPGA is the 4-wired SPI protocol. The image below showcases the timing diagram required as per the Analog Devices Datasheet for the ADXL345 accelerometer.

In our project we build an FSM to comply with the timing requirements of the above timing diagram. Given below are screenshots of our FSM utilized in communicating with the accelerometer.

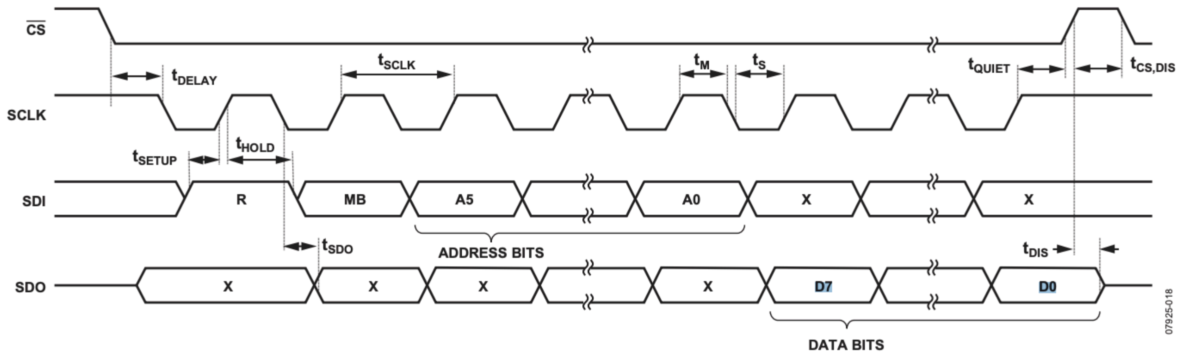
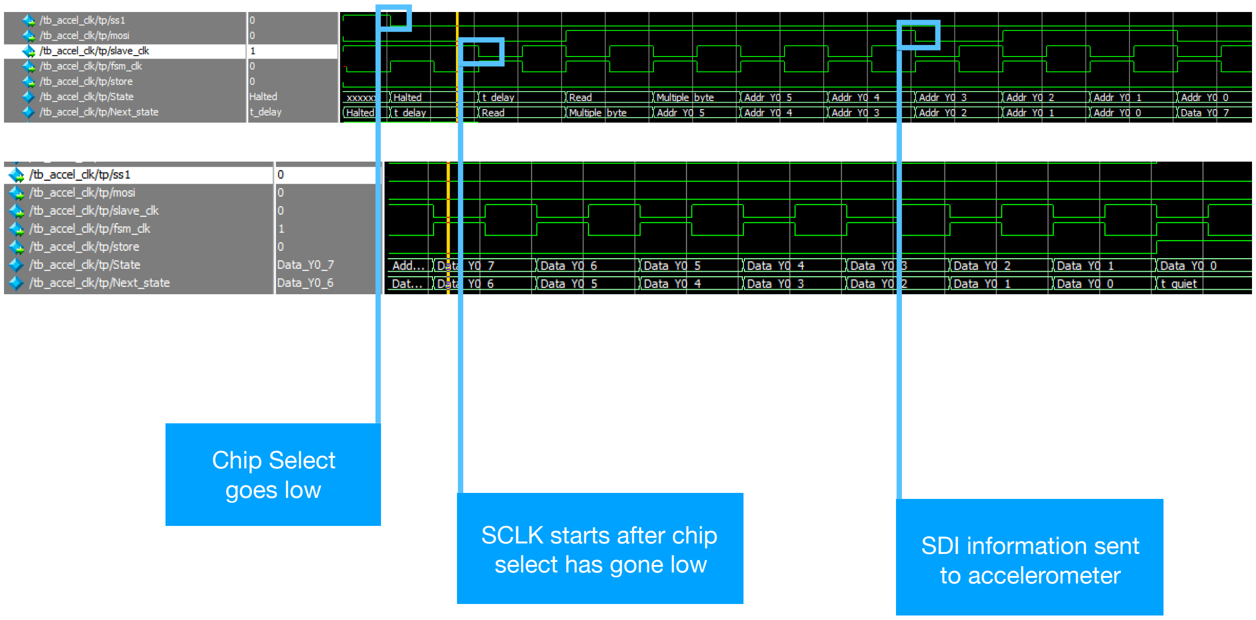


Figure 38. SPI 4-Wire Read


```
/tb_accel_clk/tp/ss1         0
/tb_accel_clk/tp/mosi        0
/tb_accel_clk/tp/slave_clk   1
/tb_accel_clk/tp/fsm_clk     0
/tb_accel_clk/tp/store       0
/tb_accel_clk/tp/State       Halted      xxxxx | Halted | t_delay | Read | Multiple_byte | Addr_Y0_5 | Addr_Y0_4 | Addr_Y0_3 | Addr_Y0_2 | Addr_Y0_1 | Addr_Y0_0
/tb_accel_clk/tp/Next_state  t_delay     Halted | t_delay | Read | Multiple_byte | Addr_Y0_5 | Addr_Y0_4 | Addr_Y0_3 | Addr_Y0_2 | Addr_Y0_1 | Addr_Y0_0 | Data_Y0_7

/tb_accel_clk/tp/ss1         0
/tb_accel_clk/tp/mosi        0
/tb_accel_clk/tp/slave_clk   0
/tb_accel_clk/tp/fsm_clk     1
/tb_accel_clk/tp/store       0
/tb_accel_clk/tp/State       Data_Y0_7   Add... | Data_Y0_7 | Data_Y0_6 | Data_Y0_5 | Data_Y0_4 | Data_Y0_3 | Data_Y0_2 | Data_Y0_1 | Data_Y0_0
/tb_accel_clk/tp/Next_state  Data_Y0_6   Dat... | Data_Y0_6 | Data_Y0_5 | Data_Y0_4 | Data_Y0_3 | Data_Y0_2 | Data_Y0_1 | Data_Y0_0 | t_quiet
```
Chip Select goes low
SCLK starts after chip select has gone low
SDI information sent to accelerometer

As we can see in the simulations above. To communicate with the accelerometer (which is the slave), first chip select (ss1) must go low. After which the SCLK for the accelerometer should start running. This should start exchange of information between the accelerometer and the master (FPGA).

Modules

Module Name: lab8.sv (top - level module)
Inputs: CLOCK_50, [3:0] KEY, [15:0] SW, [15:0] OTG_DATA,
Outputs: [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, [7:0] VGA_R, VGA_G, VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,   [1:0] OTG_ADDR , OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_INT, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N,
Inout: [15:0] OTG_DATA, [31:0] DRAM_DQ, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK
Description: The lab8.sv file is taken from the lab8 offered in the ECE 385 course. It is the top level file that acts as the central file which connects different modules of the game. Lab8.sv connects the game logic, to spite motion and color mapper files to bring to life different elements of the game.
Purpose: PIO communications with the FPGA board are done through this top level module. This involves interfacing the Keyboard and the accelerometer controls to the game. The Keyboard is interfaced through the NIOS II SoC and accelerometer through SPI communications enabled through the 14 pin IO available on the DE2 board.

Module Name: Color_Mapper.sv
Inputs: is_ball, [9:0] DrawX, DrawY, [9:0] DoodleX_right, DoodleY_right, DoodleX_left, DoodleY_left, [1:0] doodle_direction, monster_data, [7:0] debug_x0_coor, [2:0] graphics_control, Clk, [3:0] Color, pellet_data, [9:0] Platform_Y, [15:0] Settings_INPUT, Settings_COLOR, [10:0][0:31] map_y_data, Reset, [7:0] col_data, [9:0] position_y_out0, position_x_out0, position_y_out1, position_x_out1, position_y_out2, position_x_out2, position_y_out3, position_x_out3, position_y_out4, position_x_out4, position_y_out5, position_x_out5, position_y_out6, position_x_out6, [10:0] xpellet0, xpellet1, xpellet2, xpellet3, ypellet0, ypellet1, ypellet2, ypellet3, [5:0] mag, [9:0] E0, E1, E2, E3, E4, E5, [3:0] score2, score1, score0, [9:0] monster1x, monster1y
Outputs: [7:0] VGA_R, VGA_G, VGA_B, [3:0] weird_platforms,
Description: The color mapper file converts various sprite information into colors that can be printed by notes VGA screen.
Purpose: Using several always_comb blocks and sprite rom instantiations, color_mapper chooses between a different array o colors to represent different images on the VGA display. Modules like ball.sv can be developed for different required sprites I order to associate motion with those visuals.

Module Name: doodle_right
Inputs: [5:0] doodle_x, doodle_y,
Outputs: [2:0] doodle_color_out
Description: doodle_right is the sprite used to represent the the doodle character when he is facing the right direction
Purpose: The contents of the array can be accessed to create the right facing doodle on the VGA screen.

Module Name: platform
Module Inputs: [5:0] platform_x, platform_y,
Outputs: [2:0] platform_color_out
Description: This module is used several times in Color_Mapper to develop the map of the game.

Purpose: The platforms are the elements of the game that the doodle can jump on not increase his score. When the doodle comes in contact with the platform from the top, he experiences an upwards motion.

Module Name: background
Inputs: [5:0] back_x, back_y,
Outputs: [2:0] back_color_out
Description: This sprite is responsible for printing a 40x40 dimensions square representing the background.
Purpose: By using modulus with the DrawX and DrawY values, we print multiple blocks of the background sprite to cover the entire screen with the graphical background to resemble that of the original game

Module Name: welcome_screen
Inputs: [8:0] welcome_screen_x, welcome_screen_y
Outputs: [2:0] welcome_screen_out
Description: The welcome screen module contains the text spanning the display in the welcome screen of the game
Purpose: The module is used to develop the graphic for the image responsible for making the welcome screen more immersing and creative

Module Name: pellet
Inputs: [8:0] pellet_x, pellet_y
Outputs: [2:0] pellet_out
Description: Contains the array featuring the palette for the pellets that the doodle uses to shoot with.
Purpose: Tea pellets are a weapon instrument for the doodle to utilize to eliminate its enemies within the game.

Module Name: settings_unselected
Inputs: [8:0] settings_un_x, settings_un_y
Outputs: [2:0] settings_un_out
Description: This sprite is the settings button in the main screen without a black border around it.
Purpose: Having a version of this sprite with and without the black border around it allows the user to differentiate between which menu item that they are choosing.

Module Name: escape_selected
Inputs: [8:0] settings_x, settings_y
Outputs: [2:0] settings_out
Description: This sprite is the settings button in the main screen with a black border around it.
Purpose: Having a version of this sprite with and without the black border around it allows the user to differentiate between which menu item that they are choosing.

Module Name: settings_unselected
Inputs: [8:0] settings_un_x, settings_un_y
Outputs: [2:0] settings_un_out
Description: This sprite is the settings button in the main screen without a black border around it.

Purpose: Having a version of this sprite with and without the black border around it allows the user to differentiate between which menu item that they are choosing.

Module Name: escape_selected
Inputs: [8:0] escape_s1_x, escape_s1_y
Outputs: [2:0] escaper_s1_out
Description: This sprite is the play button in the main screen with a black border around it.
Purpose: Having a version of this sprite with and without the black border around it allows the user to differentiate between which menu item that they are choosing.

Module Name: escape_selected
Inputs: [8:0] escape_s1_x, escape_s1_y
Outputs: [2:0] escaper_s1_out
Description: This sprite is the play button in the main screen without a black border around it.
Purpose: Having a version of this sprite with and without the black border around it allows the user to differentiate between which menu item that they are choosing.

Module Name: frame_rom_COLOR_un
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for the Color selection button without any black border around it.
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_INPUT_un
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for the Input mode selection button without any black border around it. (showing that it is unselected)
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_INPUT_sl
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for the Input mode selection button with a black border around it. (showing that it is selected)
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_COLOR_sl
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out

Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for the Doodle Color selection button with a black border around it. (showing that it is selected)
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_ZERO
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of zero for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_ONE
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of one for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_TWO
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of two for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_THREE
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of three for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module_Name: frame_rom_FOUR
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of four for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_FIVE
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of five for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_SIX
Inputs: [8:0] : [13:0] read_address,
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of six for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_SEVEN
Inputs: [8:0] : [13:0] read_address
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of seven for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: frame_rom_EIGHT
Inputs: [8:0] : [13:0] read_address
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of eight for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module_Name: frame_rom_NINE
Inputs: [13:0] read_address
Outputs: [3:0] data_out
Description: This is a ROM instantiation that utilizes that SRAM space available on the board to store information about palette arrays for an image of nine for the score board
Purpose: Since the array nets used to earlier for graphics was based on the usage of registers available on the FPGA board, it made it inefficient to compile the entire program. Therefore, storing images on the SRAM in text files simplifies their access time and reduces access time.

Module Name: ball
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, DrawY, [2:0] direction, [7:0] keycode, [15:0] SW, game_reset,]2:0] welcome_mode,

Outputs: [9:0] x_position, y_position, [1:0] exported_direction, [31:0] _seconds, [15:0] jumping_status_out, 9:0] Ball_Y_Step_, Frame_Clk
Description: The ball.sv file is a basic file that helps provide motion to any visual object that we choose to control in the game
Purpose: This module was primarily used to debug motion for different objects as we created them to be motion capable.
Purpose:

Module Name: debugging_module(
Inputs: [9:0] x_coordinate, y_coordinate, DrawX, DrawY, Clk,
Outputs: [7:0] x_data_out_2, x_data_out_1, x_data_out_0
Description: The debugging module when activity is used to track the coordinated of various sprites on the screen in motion
Purpose: This module was greatly helpful in debugging issues with motions that we experienced as we programmed characteristic movements to many sprites essential to our game.

Module Name: doodle_motion(
Inputs: Clk, Reset, frame_clk, [9:0] DrawX, DrawY, [7:0] keycode, key
Outputs: is_doodle
Description: This module is modified version of the ball.sv module described above. It has been tailored to program motion specifically for the doodle in the game.
Purpose: The module performs fo recalculations for the doodle's motion. It coordinated with other modules such as gravity to incorporate acceleration.

Module Name: game_controller(
Inputs: Clk, Reset, [3:0] KEY, [15:0] SW, [9:0] doodle_y
Outputs: [2:0] color_mapper, reset_game, [15:0] Settings_INPUT_out_, Settings_COLOR_out_
Description: The FSM that tales scare of the different screens of the game, such as the welcome screen, gameplay screen and the death screener present in this module.
Purpose: Signals produced by the FSM are produced in the Color_Mapper model to control visual elements associated with each screen or phase of the game.

Module Name: gravity
Inputs: Clk, Frsme_Clk, Reser, SW, jump, activate_gravity,
Outputs: [9:0] position_y_out, y_velocity, [15:0] Counter_, move_platforms_down, [10:0] score
Description: The gravity for the doodle is implemented using this module. His module works in tandem with doodle_motion.
Purpose: Without acceleration, the doodle will traverse the game with constant velocity which is not how the original game is implemented. The module is used to create acceleration which defines the vertical axis automatic motion of the doodle.

Module Name: Green_Platforms
Inputs: Clk, Reset, [9:0] doodle_x, [9:0] Platform_Y, input logic Frame_Clk
Outputs: [9:0] Platform_Y
Description:Green_Platforms is the ball.sv model but for the platform motion instead.
Purpose: The green platform has to be accelerated down every time that doodle goes higher. This module helps accomplish that motion.

Module Name: hexDriver.sv
Inputs: [3:0] In
Outputs: [6:0] Out
Description: Accepts input data as a decimal value and outputs to the corresponding hexadecimal number through the use of a decoder.
Purpose: Used to map input to the hexadecimal displays on the DE2 board through converting to hexadecimal.

Module: hpi_io_intf.sv
Inputs: Clk, Reset, [1:0] from_sw_address, [15:0] from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset
Outputs: [15:0] from_sw_address, [1:0] OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N
InOut: [15:0] OTG_ADDR
Description: Various relevant information that has been exported from the NIOS II are sequentially driven into the EZ_OTG chip. When the system is reset, various values are hardcoded into the EZ_OTG chip signal paths so as to prevent information leaks. Some of the data going into the EZ_OTG includes signals like read, write, chipselect, address, etc.
Purpose: The program acts as the interface between NIOS II and EZ-OTG chip. The chip also uses a tri-state buffer to account for when NIOS is not writing data onto the chip.

Module: randomizer
Inputs: Clk, Reset
Outputs: [4:0] data
Description: The randomizer is a module used to generate generate random numbers based on the increments of the clock.
Purpose: The random numbers are crucial to creating random platforms and increasing the difficulty of the game with elevation.

Module: register
Inputs: Clk, Reset, Load, [15:0] D
Outputs: [15:0] Q
Description: The register is a module that helps store information and accepts information synchronously.
Purpose: The registers are used to reset the state of the game after the death screen has occurred.

Module: score
Inputs: Clk, Reset, Load, [10:0] score
Outputs: [5:0] magnitude , [9:0] E0, E1, E2, E3, E4, E5
Description: The core module keeps track of score of the player based on the elevation of the doodle.
Purpose: The score is obtained in hex from the game and needs to be converted into a decimal formats that it can be displayed on the screen. This module helps convert the hexadecimal score into decimal values that can then be printed on the screen.

Module: VGA_controller
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, DrawY

Description: The model is used to generate DrawX and DrawY signals which tell the programmer which part of the screen is being printed by the VGA at the moment.
Purpose: To position different sprites on the screen, setting if conditionals with the DrawX and DrawY signals will help setup signals that indicate when different sprites have to be printed.

Resources

The following table highlights the resource consumption for the implementation of the system described above.

LUT	12435
DSP	N/A
Memory (BRAM)	1262852 Bits
Flip-Flops	2402
Frequency	124.22 MHz
Static Power	105.79 mW
Dynamic Power	.75 mW
Total Power	194.43 mW

Conclusion

Throughout the implementation of this lab, many difficulties were encountered. For one, an executive decision was made to store many of the sprites as data written into an array rather than using the **$readmemh** protocol. This decision was made to simplify the implementation of the sprites; however, the capacity of the FPGA itself limited the amount of sprites that could stored using this method—and if the limit was not by storage capacity, it was the compilation time. Faced with a 40 plus minute compilation, the largest sprites had been put into On-Chip memory and the lab continued with out error with a reduced compilation time south of 4 minutes.

While these structural limitations proved to be challenging, other aspects of the game logic proved to be unruly as well. Score keeping, generating platforms (randomly, at that), reading from the accelerometer, iterating upon the color mapper—and then some—all resulted in unique digital design considerations throughout the process of the implementation of this lab. For one, in the context of the accelerometer, an Arduino was introduced to handle the production of the requisite clock so that the accelerometer may even operate in the proper regime for the FPGA to read from. Furthermore, in the case of the platforms, the decision to let the platforms generate randomly without using a data structure such as an array enabled a simplified implementation though more verbose code. Nonetheless, these decisions ultimately lead to the successful implementation of the game.

In total, implementing the Doodle Jump system posed many challenges that were ultimately overcome by clever design and team development of ideas. That said, many of the topics discussed within ECE 385 are prominently featured throughout the digital design of the system.

Work Cited

Lab 8 Files, ECE 385, Spring 2019

Analog Devices Data Sheet, *ADXL345*

Note To David

Thank you for the very interesting semester. ECE 385 has taught me a lot about the space and I am glad I got to do it as you as my TA. You made the environment welcoming, helpful, and enjoyable. I shall train this summer and become the ultimate Doodle Jump Champion!
I wish you well.
—Jack

Note To David

Thank you for looking out for us in labs and helping us get familiar with the very interesting domain of Embedded Systems. I hope you have a great summer and that your pursuit of a PhD in quantum computing goes well.
—Sruthi