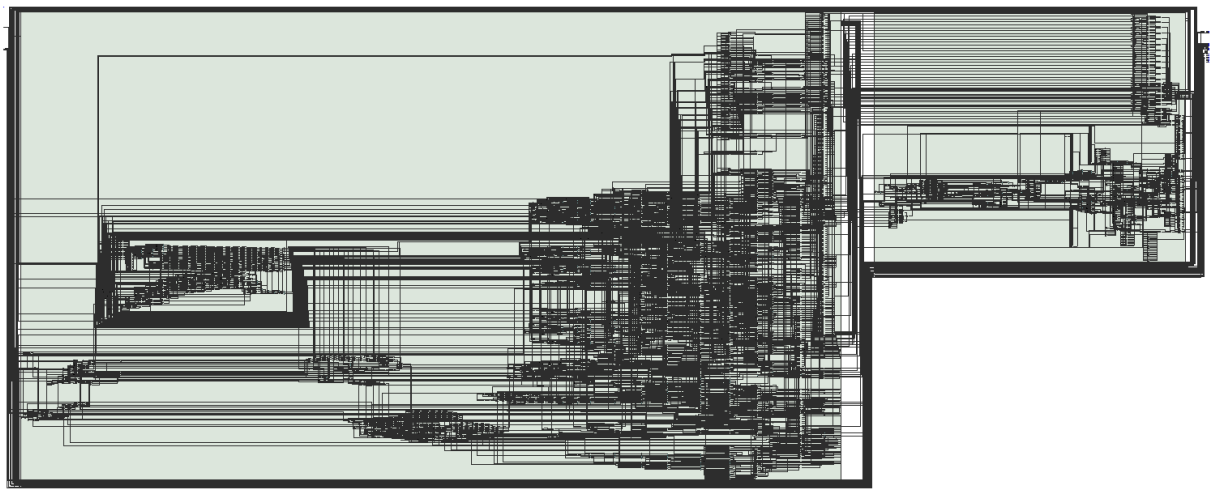
ECE 385
Spring 2022
Final Project



Doodle Jump on FPGA

Gally Huang/Feng Zhao
TA: Abigail Wezelis

Disclaimer:

Many items were reused from lab6.2, such as the VGA controller and VGA output of the game. Additionally, the NIOS II code was reused from Lab 6.2, as well as the ball.sv code. We give credit to online references such as the ones listed on our repo for the code.
https://github.com/Jellyyz/Doodle-Jump

Introduction:

For our final project, we decided on implementing a version of Doodle Jump that was playable through the FPGA. The main idea of the project was to try to allow the user to have the same experience as if they played it on their phones, but in the true spirit of a hardware class - the game was recreated through almost 100% on hardware. The only software parts that were utilized were from Lab 6.2 where the user must take in data from the keyboard/mouse. As far as we know, there is no way around this, unfortunately.

As for the main game, Doodle Jump is a platformer game that follows a character on the screen also known as the doodle as it jumps from platform to platform. The character is known to possess the ability to shoot out of its mouth onto the screen, effectively making it able to defend against monsters. There is no end-game objective to Doodle Jump, as it is an open-world game. Our version of the game utilizes linear feedback shift registers in order to randomize the position of the platforms that are coming in. Additionally, in the original game, there are different types of platforms that respond differently to the doodle's interaction with them. More details are listed below.

There are also additional things that were added in order to boost the completeness of the project, such as the score being displayed on the top left, the main menu screen, a pause screen, and an end game screen. We also added a cannon to the doodle so it can hit monsters at certain score levels, as well as animation for the doodle's jump, and additionally allowing for the difficulty to be chosen for the game so that the platform lengths would change.

Written Description of System:

DoodleCharacter

In our system, the Doodle is defined as a point in space where it must be drawn out. The specific origin of this character is taken from the ball.sv was given in lab 6, where there is a point in space and then there is also space around the ball that must be drawn in order to be displayed to the user. This doodle position is represented as DoodleX and DoodleY. The size of the doodle is represented by DoodleSize and it is the same for both the X and the Y, essentially drawing a box for easier math purposes later when drawing the doodle out. We must also take into consideration the doodle's speed when drawing it out. After all the doodle is the object that is constantly moving across the screen, so we add another modifier (both in the X and Y direction) called DoodleYmotion and DoodleXmotion respectively. These modifiers run off of the main frame_clk that will drive the Doodle across the screen. Every single posedge of the frame_clk, the position of the doodle is increased by an incremental number based on its current motion of it. Of course, this is useful for the X position of the doodle since that is modified either to be motion = 0 (non-moving), 2 (moving right), or -2 (moving left). However, for the movement of the doodle as it is jumping, the whole point of the game, we need to take into some consideration from kinematics class. The speed of an object is always defined to be positive no matter what, but the velocity of an object can be also defined as to be positive or

negative. When the doodle is falling, its velocity is in the positive region since its Y motion is actually increasing- while when it's falling, its velocity is negative. There is an apogee of the doodle when it reaches the "peak" of its motion and needs to fall back down, so therefore in order to cause it to jump up and down, there must be a counter that counts the amount of time the doodle has been in the air before adding to its motion, therefore causing for its negative upward motion to turn into downward motion. We need not worry about when it hits the ground as this is covered by the collision theory discussed in the next section.

Collision Theory

The Doodle possesses the ability to not only move around the screen, but the fundamental portion of the game is its ability to interact with the positions of the platforms on the screen. Inside our code, we see that the doodle needs to be able to "communicate" with the platforms which are all declared inside of colormapper.sv (more details in the next section). Because the frame_clk is so fast, what we decided to do to detect a collision is to only have the doodle collide with the platform not at a specific point, but at a certain range. This range is when the doodle is "inside the platform" and this works because the frame_clk is fast enough to catch this to reverse the motion itself. This collision theory must work with all the different portions of the doodle. The doodle must be able to bounce off each platform, which sends a signal to the doodle physics engine that a platform has been touched, and as discussed later depending on the color this has many drastic effects. The collision also applies with the cannon itself, the cannon must be able to collide with the monster when it detects the "range" inside of the monster it has hit, and by doing so sends out a control signal which will remap the monster to be off the screen and awaiting another respawn.

Platforms/Powerups

As discussed briefly in the collision theory section, the platforms have their own respective types that have been discussed in the proposal. There are blue, yellow, brown, and white platforms. These different platforms are controlled by a state machine in conjunction with the RNG hardware discussed below. The state machine allows for the platforms to be changed to different colors whenever it receives a trigger. This trigger only pops up whenever the platforms have left the screen. We shall call this platoffscreen_trigger for future reference. In summary, the blue platform moves in a certain amount of units across the screen, while the yellow platform teleports to a certain location, the white platform disappears after contact, and the brown platform does not have any collision theory behind it at all.

The powerups that were implemented were the spring and the rocket boost. The spring allows for the doodle to move up some amounts of distance, while the rocket also allows for the doodle to move up more, a greater amount of distance. These powerups utilize the same counter mechanic that has been implemented before.

RNG Hardware

RNG is pretty interesting to study. The RNG is implemented in this system by utilizing what is known as a linear feedback shift register. The register essentially takes in a seed key, and then it will bitshift this key while XORing or NXORing some of the inputs. This key will wrap from the last bit of the shift register to the front, but it will allow the use of the result of the XOR and XNOR while it is shifting, in order to preserve the randomness. The module which runs all of this behaves similarly to a shift register, having a reset signal, but also having a seed in and

seed enable signal. There are 16 LFSRs, and they are actually all chained together for each platform on the screen. Whenever the platoffscreen_trigger is tripped on for a specific platform, it will load in a random value of the current LFSR associated with it. This is because the LFSR is running at a super-fast clock, the 50mhZ, in order to preserve the randomness of the situation. We can think of this as having a randomly changing output that changes every 50 MHz, but it is only loaded in from a flip flop depending on a trigger point.

There is also randomness for the plat_color that is triggered offscreen. This is based on the score and has no correlation with the LFSR. This color that is being changed will update on the next frame_clk which is “good enough” and also random enough for the gameplay, as to the user it appears random.

Palette/Bitmap/Animation

We decided to use Rishi’s Helper Tools in order to create many of the sprites for this game. There is a sprite for the crouching of the doodle as well as a sprite for the doodle’s normal state. Thus there are four sprites, one for right/left, and for crouch and normal. These sprites have a background associated with them, and that color is mapped out to magenta. The rest of the sprite is therefore printed onto the screen. The actual process of printing out the sprite is done by reading from the on-chip memory once we are in the region that needs to be drawn. The position of the electron gun must be calibrated in order to match the position of the image that must be drawn. For simplicity, we chose to represent all of these images through a square, since this will then allow for the region to have the same X and Y base region.

Monster/Cannon

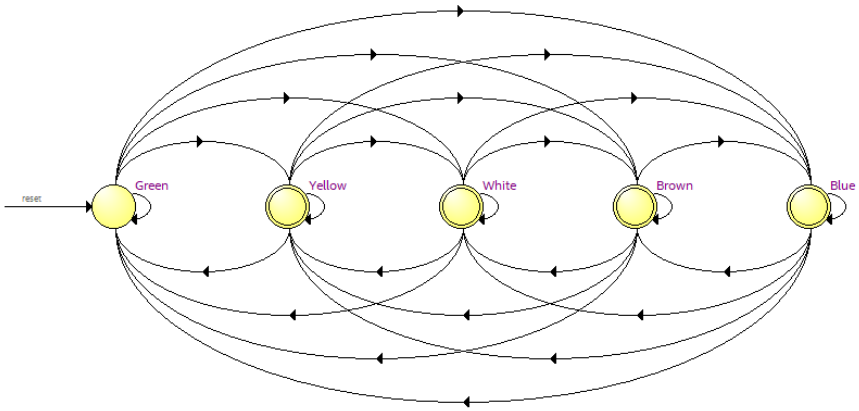
We implemented the monster as an enemy in this game. Monsters are set to appear only when the player reaches a certain score, defaulting to being at the upper left corner of the screen. There are two monsters in the whole process, which appear in two different scene transitions0. Its trajectory is basically set back and forth in the top half of the screen, putting pressure on the player to avoid jumping too high and touching the monster and failing. It's also a kind of difficulty correction for springs and rocket packs, allowing the player to take shortcuts while keeping an eye out for potential dangers. When a doodle is hit by the monster during its jump, the doodle will be defeated and lose the game, which is the same as falling off. Therefore, one way to let the doodle avoid the danger is to carefully jump, making sure the jump path will not encounter the monster. Besides staying away from the monster, there is another way to deal with it, which is more like an active defense system. We give the doodle three cannons that can shoot out cannonballs straight upwards, and the player can use the cannon to shoot down the monster to eliminate the danger beforehand. The monster is destroyed when the scope of the monster model and the shell model overlap.

Difficulty

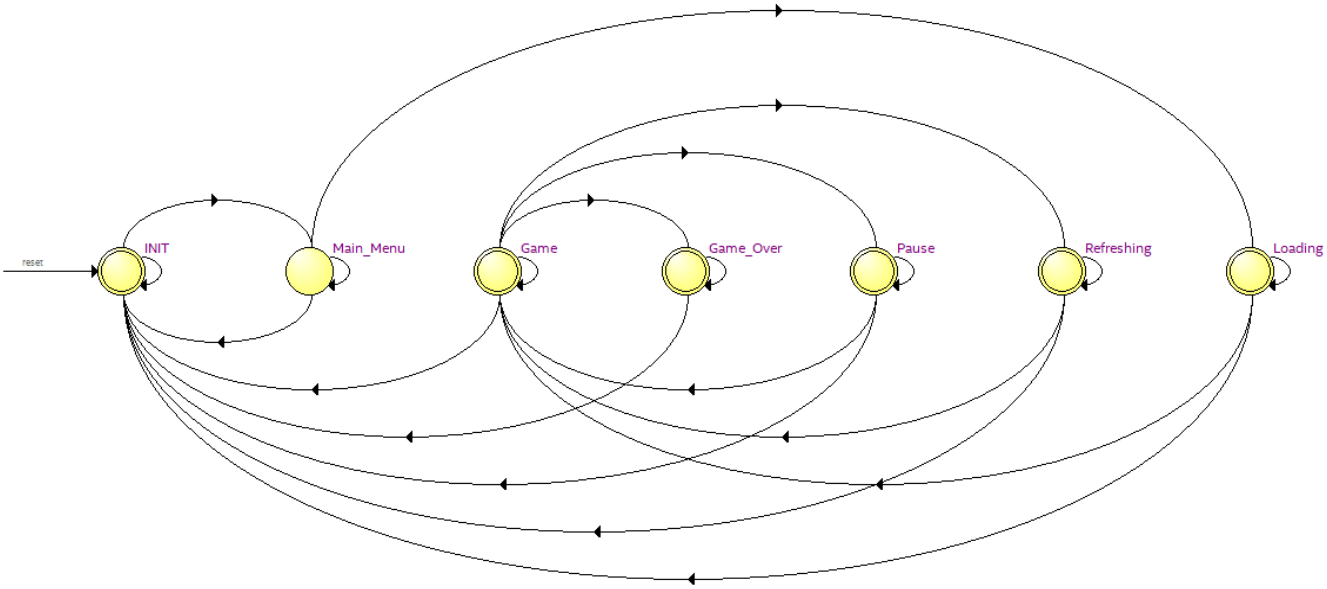
We provide three different difficulty levels for players to choose from: Easy, Medium, and hard. The differences are basically the length and density of the platforms. The shorter and less dense the platforms are, the harder the game is. We set up three difficulty indicators in the shape of platforms on the main menu. Players can select the difficulty they want to play by using the left and right arrow keys. These three indicators correspond to the size of the platform under

each difficulty, which makes it convenient for players to prepare for the difficulty of the level before entering the game.

State Machine:



The first important state machine that needs to be considered is the one that controls the platform's color of the game. This state machine takes in a trigger that will essentially change the color after a single clock cycle. The reasoning behind using a state machine vs something like a bunch of flip flops is that the state machine allows for all possibilities and edge cases to be considered. A common error that we encountered was that color would change quickly to let's say blue from the default green state, but then it would be instantly changed back to the green state since the trigger that changes this color is so short. We need a way to set a "default" holding state that ensures no metastability issues occur here so a state machine was used. There are actually no control signals coming out of this state machine, but rather the colors are read directly from the current state itself.



This is an example of a more complex state machine that actually utilizes control signals is the master state machine that controls the states of the game itself. Each “screen” of the game essentially takes up a state, whether it be the pause screen or the main menu, this allows for an easy overhead that the rest of the logic such as the platform motions can follow easily. The state machine was first devised to control the refreshing of the platforms and allow for scrolling to occur since this quickly turned into having to interface with different screens such as not scrolling when the game was paused, or the screen should not be refreshing constantly at play but only at certain values. Having a trigger during the actual Game state allows for the state machine to diverge into the refreshing state only if a refresh trigger was received. This also meant there must be loading or init stages for the game since we need to declare these triggers to have an initial value or floating values will occur. For more simple tasks such as taking in keyboard values the user will be able to move from screen to screen as follows, the main menu can be jumped to the game state if space is pressed, and the game state can go to the pause state if ESC is pressed and vice versa if ANY button is pressed, and finally the game can go back to the main menu if the ESC key is pressed.

Debugging Strategy:

Quite early on during development, it was realized that the game would be hard to debug using ModelSim. What we used most to develop the game was the HEX displays that could be fed into from any wire anytime there was a bug that appeared. The reasoning for doing this versus something in ModelSim is that Modelsim is good at visualizing things such as specific number outputs of a digital logic block at certain time intervals that would never change, such as an ALU like we did for lab 5.2. However, ModelSim is not that great for visualizing the

exact doodle placement on the screen since this value changes so rapidly, it would be easier to implement a pause function and then check the current doodleX and doodleY position on the HexDrivers itself. You may ask then, wouldn't the hex drivers refresh too fast for us to see any change? The solution is the feed in the frame_clk into a counter that acts as a positive trigger for any flip flop on its 5th bit, essentially causing a 1-second update for any value inside a wire. This solution allowed us to test things such as the doodle's velocity and its ever-changing position also at ease. At other times when the hex drivers were running extremely fast, such as the Doodle's position when it was scrolling the screen and jumping like crazy, we used the slow-motion camera on our phones in order to detect what display was currently showing up on the HexDrivers, as that is one drawback of this method. It was also useful to include things such as the current state, and the current triggers that were active on the LEDs since those are binary values and can be represented easily with just a single LED.

Changes from the Proposal:

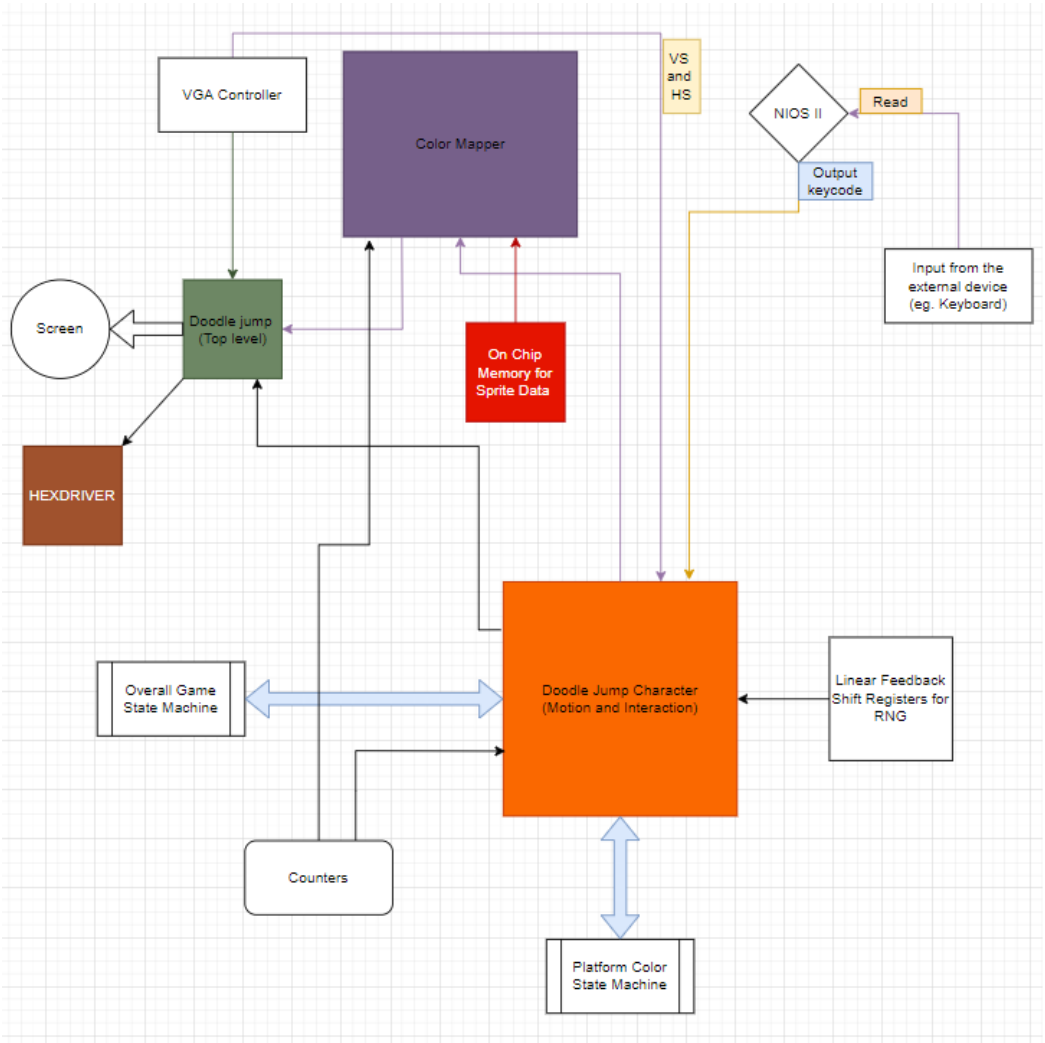
We successfully implemented everything from our proposal for the baseline difficulty (3.5 difficulties) that including the doodle being able to jump and interact with the rest of the platforms on the screen, as well as being able to have platforms that are pre-generated (hardcoded?).

We went beyond this task because we found after implementing the physics engine for the game, it was possible to create other things to fully flesh out our game. This included adding stuff like different colors for our doodle's platforms and adding a start and end screen to increase player immersion. To be short- we basically "changed" the proposal only in the fact that we added more stuff than what was needed for the proposal. We failed to implement all powerups due to time constraints, as then platforms seemed to be more of a priority to increase playability.

Block Diagram:

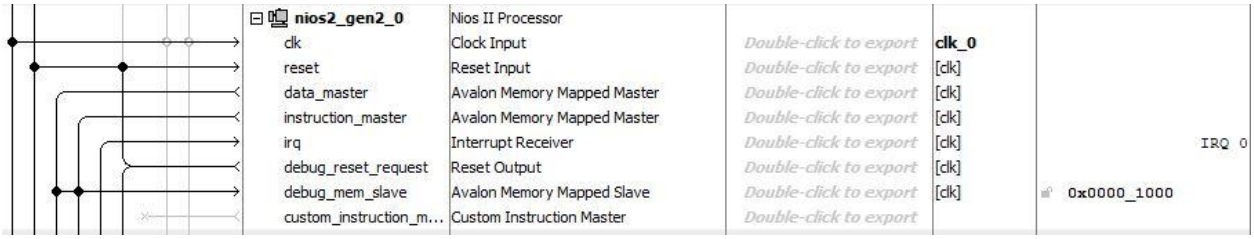
As for the block diagram, we modified it slightly from our initial plan due to more experience with how we should approach the problem. Things such as using a single machine quickly turned into two-state machines since it was just easier to spread the platform color changing and the game state of the game into two different state machines.

For our game, the color mapper serves mainly as the terrain module, as it controls the monster, platforms, and powerups. It interacts mainly with the doodle-jump character module, which controls the actual doodle and the main physics of the doodle itself. With those in mind, we also need two state machines to support the doodle jump character and the color mapper. One will control the platform's color, while the other one will control the physics of the doodle jump character. Additionally, we must have a NIOS II component, since this is the way that we can actually read a keycode into the DoodleJumpChar, and move accordingly. The final step for any signal inside our game would be to be routed into the FPGA through the top level. This is also a crucial block of our game since this allowed for routing into the HexDrivers for debugging, as described above.

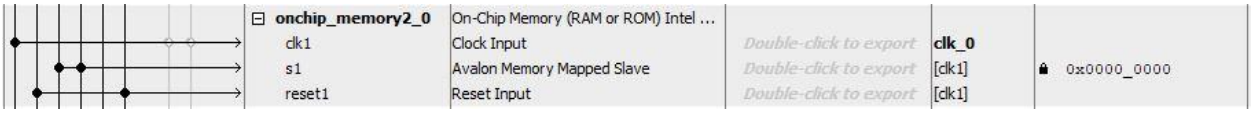


Platform Designer:

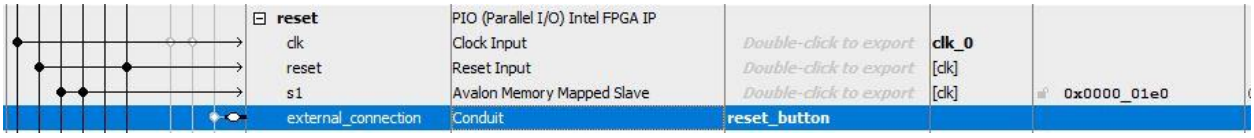
clk_0 is the clock block that generates clocks for all other modules.



nio2_gen2_0 module is the NIOS-II Processor. It has clk, reset, data_master, instruction_master, and instruction_master, irq, bebug_reset, debug_slave, and custom_instruction_master. These are all the I/O ports which allows the processor to interrupt, interface, and execute the instructions. For this lab project, this module is used as the SOC file. Its base address is 0x0000_1000, which allows us to control all signals and interpret instructions. Moreover, it allows the PIOs' interface with other blocks. Its base address is 0x0000_1000.



onchip_memory2_0 is the module that we allocate our on-chip memory. It has the clock input, avalon memory mapped slave, and reset input. It allows the output data from the keyboard or other external device to be read or written into the on-chip memory for later use in the Nios-II processor. This on-chip memory has 16-bit data_width, 1 chip selects, and 4 banks, which it has 32M * 16, which equals 512 MBits. Its base address is 0x0000_0000.



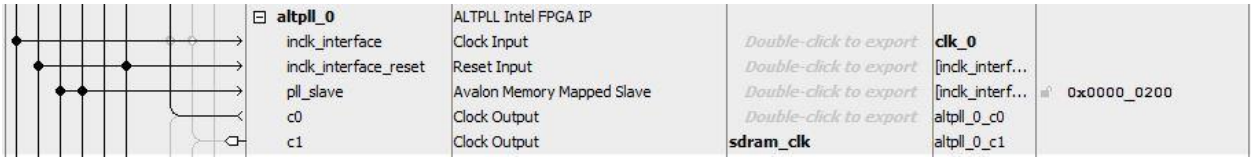
reset is the PIO module for reset key, and it has clk, reset, and s1, and external_connection. At every rising edge of the clock, NIOS_II takes the input value of the reset key. It has the base address at 0x0000_01e0.

sdram is the module used for the interface of LEDs and switches on board(Week 1), and keyboard(Week 2). It is a type of memory that synchronizes itself with the computer's system

clock. Sdram has clk, reset, and avalon memory mapped slave. Its clock is connected to c0, the clock output of sdram_pll to control when to write and read data. Its base address is 0x0800_0000.

keycode is the PIO module for the keycode output It has clk, reset, and s1, and external_connection. At every rising edge of the clock, NIOS_II takes the value of the keycode that matches the pressed key, and uses it as the input of the c coding file. Therefore, the processor can recognize the physical key within the software coding. It has the base address at 0x0000_01d0.

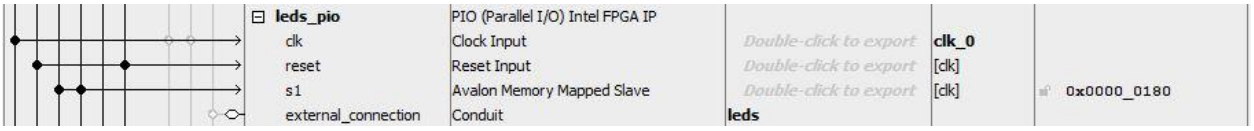
accumulate is the PIO module for the accumulation when pressing the button on the FPGA board. It has clk, reset, s1, and external_connection. We used it to operate the accumulation on the software side at every rising edge of the clock. Its base address is 0x0000_01f0.



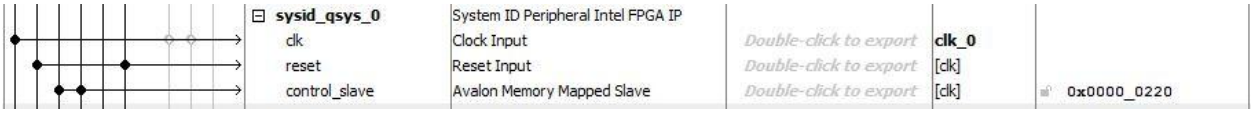
altpll_0 is the module that has two clk outputs. It generates the clock that connects to the sdram to control its writing and reading cycle. This module allows us to add some clock delay for the sdram to get the correct result for reading and writing. It has the base address at 0x0000_0200.

hex_digits_pio is the PIO module for the hex digits that would show on the LEDs. It has clk, reset, and s1, and external_connection. At every rising edge of the clock, NIOS_II takes the input value of the hex digits as an input. It has the base address at 0x0000_0190.

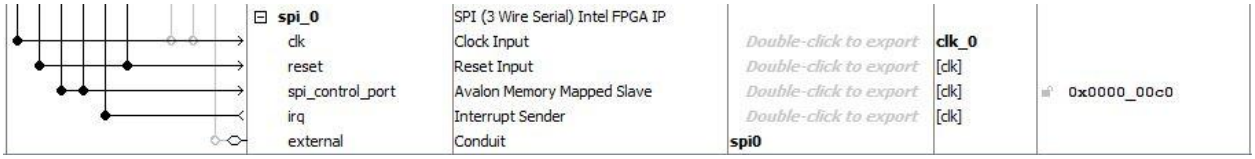
key is the PIO module for the key input on the keyboard that would later be converted to the keycode output. It has clk, reset, and s1, and external_connection. At every rising edge of the clock, NIOS_II takes the input value of the key, and then matches the keycode of this key input. Therefore, the processor can recognize the physical key within the software coding. It has the base address at 0x0000_0170.



leds_pio is the PIO module to control the leds. It has clk, reset, and s1, and external_connection. At every rising edge of the clock, NIOS_II takes the input value of the leds controller to determine the status of leds. It has the base address at 0x0000_0180.

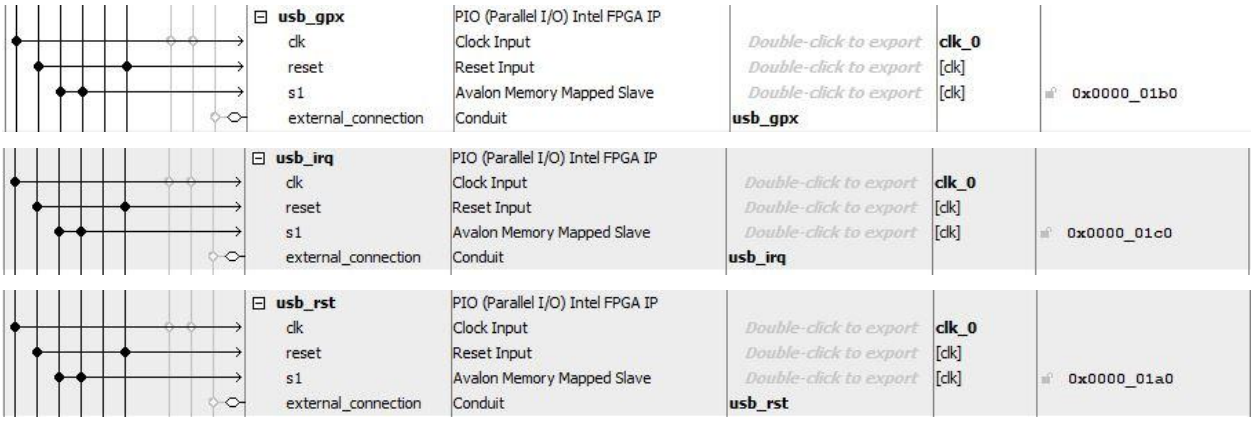


sysid_qsys_0 is the system ID checker, which is designed to prevent any incompatibility between hardware and software. It has the base address at 0x0000_0220.

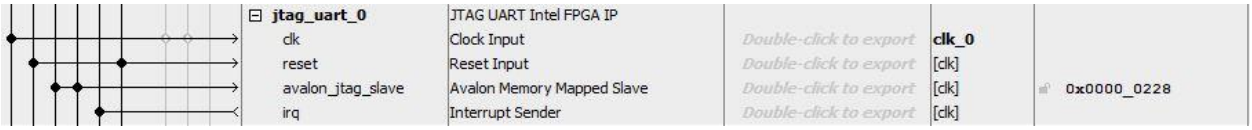


spi_0 is the SPI(3 Wire Serial) module that send data between the master device(Nios-II) to the slave device(MAX3421E). It has a clk, a reset, a Avalon memory mapped slave s1, and a interrupt sender. SPI's full name is synchronous serial bus, so it makes the MAX3421E be synchronous to the system clock. It has the base address at 0x0000_00c0.

timer_0 is the module that is designed for timing for the internal logic inside the hardware system. It has the base address at 0x0000_0080.



These three modules above(usb_gpx, usb_irq, usb_rst) are all used for the usb transmission. The usb_irq is for an interrupt request that allows the Nios-II to ask for data from the USB storage. These three have clk, reset, s1 and external_connection, that allow them to transmit different requests between the Nios-II and the USB. Their base addresses are 0x0000_01b0, 0x0000_01c0, 0x0000_01a0.



jtag_uart_0 is the module that can communicate with the terminal of the host computer with the Nios_II system. Therefore, the computer running the eclipse can transmit data with the processor, which is using the c file to do software programming. It has the base address at 0x0000_0228.

Module Description:

doodlejump.sv:

Input:
[9:0] SW,
[1:0] KEY,
MAX10_CLK1_50

Output:
[12:0] DRAM_ADDR,
[9:0] LEDR,
[7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
[3:0] VGA_R, VGA_G, VGA_B,
[1:0] DRAM_BA,

VGA_HS, VGA_VS, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, DRAM_CLK, DRAM_CKE

Inout:
[15:0]ARDUINO_IO, DRAM_DQ,
ARDUINO_RESET_N

Description:
It's the top level module for the entire project, which basically takes in all the output from other modules. It also instantiates those modules. By connecting all the modules together, it can drive the FPGA to do the work. It also serves as an overall index for all the variables in modules.
Purpose:
It is designed to connect all the modules and use them to drive the FPGA board.

color_mapper.sv:

Input:
[19:0] Score,
[9:0] DoodleX, DoodleY, DrawX, DrawY, Doodle_size, Doodle_Y_Motion, CannonX, CannonY, CannonS, CannonX1, CannonY1, CannonX2, CannonY2,
[8:0] plat_temp_Y,
[7:0] airtime, keycode,

[5:0] outstate,
[1:0] difficulty,
Clk, Reset, frame_clk, Platform_collision, Platform_collision0, Platform_collision,
Platform_collision1, Platform_collision3, Platform_collision4, Platform_collision5,
Platform_collision6, Platform_collision7, Platform_collision8, Platform_collision9,
Platform_collision10, Platform_collision11, Platform_collision12, Platform_collision12,
Platform_collision13, Platform_collision14, Platform_collision15, doodle_down_check, loadplat

Output:
[31:0] countingplat,
[23:0] underwater_BKG_out, soccer_BKG_out, doodle_right_BKG_out, space_BKG_out,
doodle_left_BKG_out, zero_BKG_out, one_BKG_out, two_BKG_out, three_BKG_out,
four_BKG_out, five_BKG_out, six_BKG_out, seven_BKG_out, eight_BKG_out, nine_BKG_out,
A_BKG_out, B_BKG_out, C_BKG_out, D_BKG_out, E_BKG_out, F_BKG_out, G_BKG_out,
H_BKG_out, I_BKG_out, J_BKG_out, K_BKG_out, L_BKG_out, M_BKG_out, N_BKG_out,
O_BKG_out, P_BKG_out, Q_BKG_out, R_BKG_out, S_BKG_out, T_BKG_out, U_BKG_out,
V_BKG_out, W_BKG_out, X_BKG_out, Y_BKG_out, Z_BKG,
[9:0] platX_Motion,
[8:0] springX, springY, springX1, springY1, springX2, springY2, springX3, springY3,
springsizeX, springsizeY, rocketX, rocketY, rocketsizeX, rocketsizeY, platX, platY, plat_sizeX,
plat_sizeY, platX1, platY1, platX2, platY2, platX3, platY3, platX4, platY4, platX5, platY5, platX6,
platY6, platX7, platY7, platX8, platY8, platX9, platY9, platX10, platY10, platX11, platY11,
platX12, platY12, platX13, platY13, platX14, platY14, platX15, platY15, plat_size_easy_X,
plat_size_medium_X, plat_size_hard_X, plat_size_easy_Y, plat_size_medium_Y,
plat_size_hard_Y, readyX, testX, blue_temp_platX, monsterX, monsterY,
[7:0] temp, Red, Green, Blue,
[5:0] outstate,
[4:0] monstersizeX, monstersizeY,
[2:0] plat0_color, plat1_color, plat2_color, plat3_color, plat4_color, plat5_color, plat6_color,
plat7_color, plat8_color, plat9_color, plat10_color, plat11_color, plat12_color, plat13_color,
plat14_color, plat15_color,

trigger, monster_trigger,
plat_enable, plat_reset, test,
doodle_right_BKG_on, doodle_right_BKG_on3_bkg, doodle_left_BKG_on,
doodle_left_BKG_on5_bkg

Description:
Color mapper converts output data from other modules to VGA signals that display sprites in the proper position on the screen. In general, the method of drawing is to assign an 8 bit RGB hex value for each pixel when a certain “on” signal is active, and make the entire screen be like what we want. By using PNG to HEX conversion helper, we can import those images we need into the module and make them as modifiable sprites and adjust their color and chop off the unneeded areas in the image. Palette is also used in this module to choose what kind color we

want for each sprite. Not only passively receiving outside data to draw, It can also randomly generate platforms inside itself, including green, blue, yellow, brown and white, which have different interactions with the doodle. The randomization is made by implementing Linear Feedback Shift Register to determine the center position of each platform every cycle.
Purpose:
This module is designed to receive the data of the objects positions from other modules, such as jumplogic and the letters A-Z and numbers sprites modules in the Alphabet_sprites.sv, then convert them to RGB values the VGA components can understand.

jumplogic:
Input:
[8:0] plat_sizeX, plat_sizeY,platX, platY, platX1, platY1, platX2, platY2, platX3, platY3, platX4, platY4, platX5, platY5, platX6, platY6, platX7, platY7, platX8, platY8, platX9, platY9, platX10, platY10, platX11, platY11, platX12, platY12, platX13, platY13, platX14, platY14, platX15, platY15,plat_size_easy_X, plat_size_medium_X, plat_size_hard_X, plat_size_easy_Y, plat_size_medium_Y, plat_size_hard_Y, springX, springY, springX1, springY1, springX2, springY2, springX3, springY3, springsizeX, springsizeY, rocketX, rocketY, rocketsizeX, rocketsizeY, monsterX, monsterY,
[7:0] keycode,
[4:0] monstersizeX, monstersizeY,
[2:0] plat0_color, plat1_color, plat2_color, plat3_color, plat4_color, plat5_color, plat6_color, plat7_color, plat8_color, plat9_color, plat10_color, plat11_color, plat12_color, plat13_color, plat14_color, plat15_color,

Reset, frame_clk, Clk, trigger

Output:
[19:0]Score,
[15:0] countingss,
[9:0] DoodleX, DoodleY, DoodleS, CannonX, CannonY, CannonS, CannonX1, CannonY1, CannonX2, CannonY2, Doodle_Y_Motion, Doodle_X_Pos, Doodle_Y_Pos,
[8:0] plat_temp_Y,
[7:0] displacement, airtime,
[5:0] outstate,
[1:0] difficulty,

doodle_down_check, Platform_collision, Platform_collision0, Platform_collision1, Platform_collision2, Platform_collision3, Platform_collision4, Platform_collision5, Platform_collision6, Platform_collision7, Platform_collision8, Platform_collision9, Platform_collision10, Platform_collision11, Platform_collision12, Platform_collision13, Platform_collision14, Platform_collision15, loadplat, refresh_en, Rocket_collision
Description:
It contains all the logic to describe how the doodle should move and how it interacts with other objects on the screen, such as the platforms and monsters. We use DoodleX, DoodleY, and

DoodleS to locate the doodle, and then use Doodle_Y_Motion to determine its motion path. It outputs the result of collision like changed moving pattern and position to the color mapper to let the logical interaction be physical on screen.
Purpose:
It is designed to describe the motion and location of the doodle. The interactions between the doodle and other objects are also included.

BKG_ram:
Input:
[12:0] write_address, read_address,
[4:0] data_In,
we, Clk

Output:
[23:0] data_Out

Description:
This module serves as a ram, reading a text file that includes the palette information of each pixel within an image. Then it will be instantiated in the color mapper to output the VGA signal for that specific image, or the “sprite”. This method is used for all the modules below:
Spring_ram:(*sprite of springs)*,
Right_facing_doodle, Right_crouchdoodle, monstersprite, Left_facing_doodle, Left_crouchdoodle: *(Doodle and monsters sprites in different conditions)*
jetpack_ram, cannon2_sprite, cannon1_sprite, cannon_sprite: *(The additional supportive items)*
greenplat64, greenplat32, greenplat16: *(sprites of green platforms under three different difficulty levels)*
BKG4_ram, BKG2_ram, BKG_ram, Alphabet_sprites: (*The backgrounds and letters & numbers shown on the screen)*
Purpose:
These modules are designed to implement images as sprites into the program. The pictures are more real and vivid, not just simple place blocks. The background also has more variations, and it changes as the score progresses, which is more immersive.

VGA_controller.sv :
Input:
Clk, Reset

Output:
[9:0] DrawX, DrawY,
hs, vs, pixel_clk, blank, sync,

Description:
This module is designed to control the VGA signal on the display screen. It also has logic vc and hc to make sure to go over each column and row. It scans horizontally from left to right, when it

reaches the edge of the screen, it will go to the next row. The hs and vs output logic are the pulses to synchronize with the actual screen. Both of them are set to make sure the pixel does not go out of the screen. Finally, to only display pixels in range, which is between horizontal 0-639 and vertical 0-479, it will only activate the display signal when hc and vc are both within the certain range.
Purpose:
It is designed to control the output of the VGA components and display signals sent to the monitor.

platcolor_on:
Input:
[23:0] greenplat16_out, greenplat32_out, greenplat64_out,
[9:0] DrawX, DrawY,
[8:0] platX, platY, plat_sizeX, plat_sizeY, platX1, platY1, platX2, platY2, platX3, platY3, platX4, platY4, platX5, platY5, platX6, platY6, platX7, platY7, platX8, platY8, platX9, platY9, platX10, platY10, platX11, platY11, platX12, platY12, platX13, platY13, platX14, platY14, platX15, platY15,
[2:0] difficulty, plat0_color, plat1_color, plat2_color, plat3_color, plat4_color, plat5_color, plat6_color, plat7_color, plat8_color, plat9_color, plat10_color, plat11_color, plat12_color, plat13_color, plat14_color, plat15_color

Output:
[8:0] temp_platY, temp_platX,
greenplat16_on, greenplat32_on, greenplat64_on, blueplat16_on, blueplat32_on, blueplat64_on, yellowplat16_on, yellowplat32_on, yellowplat64_on, whiteplat16_on, whiteplat32_on, whiteplat64_on, brownplat16_on, brownplat32_on, brownplat64_on,

Description:
This module controls the on/off status of all the 16 platforms under three different difficulty levels. For each platform, there are 15 types, which are 3 difficulty level * 5 colors. Then we repeat it for 16 times, and let each correspond to different conditions.
Purpose:
This module is designed to control the status of each platform under different conditions.

HexDriver.sv
Input: [3:0] In0

Output: [6:0] Out0

Description:
This module takes in the 4-bit binary numbers that are generated by the actions on switches and convert them into hexadecimal numbers. It has 16 cases, which are corresponding to the 16 possible combinational status of the 4 switches on the FPGA board. A number will be displayed on the LEDs based on the case.

Purpose:
This module is designed to convert the input of the switches to the numbers on the LEDs.

plat_type:
Input:
[2:0] type_trigger,
Clk, Reset

Output:
[2:0] plat_color

Description:
It contains a state machine to switch between each color type of the platforms. Then, by implementing several case statements in the module, we can easily choose a unique case to process on.
Purpose:
It is designed to use a state machine and case statements to make a systematic method for changing platform colors, which makes the process more direct and convenient.

modifiedcounter:
Input:
Reset, Clk

Output:
[4:0] outM

Description:
This is a counter to output outM to the color mapper to control the display of doodle sprite. With the counter, we can adjust when the doodle will be at knee-bent mode, and when be at the normal figure.
Purpose:
This is designed to serve as a counter for the control of the doodle sprite switch.

LFSR:
Input:
[8:0] seed,
Clk, Reset, seed_in

Output:
[8:0] outp, seed_out

Description:
This module creates a random number based on a seed value, and it will do this through XORing specific bits until a random number is generated. Meanwhile, the shift registers allow for

the number to continuously be updated to be random, and it also allows for the number to change depending on the clock. This number also needs to be XNORed if the output is all 0's and vice versa to be XOR if it's all 1's due to the way the boolean algebra works. More information can be provided in the RNG section.
Purpose:
It is designed to generate random numbers for the X position of the platforms.

jumpstate:

Input:
[7:0] Keycode,
Clock, Reset, trigger, frame_clk, monster_collision, refresh_en, game_over_trigger

Output:
[2:0] outstate, loadplat

Description:
This module has a state machine that controls multiple stages of the game, including INIT, Main_Menu, Loading, Game, Pause, Refreshing, and Game_Over. Also we match each stage with a case of outstate, which makes it more convenient when used in other modules, such as assigning certains operations to a stage.
Purpose:
It is designed to make stage switching and operation assigning easier and clearer.

doodle_direction:

Input:
[7:0] Keycode,
Clock, Reset, frame_clk

Output:
direction

Description:
This module contains a small state machine and several case statements. They can make the switching between doodle-facing-left and doodle-facing-right more clear in logic. After receiving certain keycode inputs, the doodle will jump into one of the states so it doesn't return to its original facing direction once the key is released.
Purpose:
It is designed to make the doodle stay in its facing direction after the key is released.

counter:

Input:
Reset, Clk, enable

Output:
[31:0] out

Description:
This is a templated counter module. It can be made into different special counters, which can be enabled by an enable signal and start counting every clock cycle.
Purpose:
This is a templated counter module for multiple uses.

countdown:
Input:
[7:0] seed,
Clk, Reset, enable

Output:
[7:0] temp,
done

Description:
It has the similar function as the counter.sv, but it's counting down instead of counting up, or accumulating. It takes in the seed input from the LFSR.sv, and counts it down by simply subtracting by 1 every clock cycle.
Purpose:
It is designed to count down the value of seeds.

bin2bcd:
Input:
[13:0] bin

Output:
[15:0] bcd

Description & Purpose:
This module is used to convert the hexadecimal numbers to decimal numbers, in order to correctly display the score on screen.
Cited from Internet, please see README file on github for the link.

Power Analysis && Logic Resources Used:

Fmax was found to be 114.2 MHz. Interesting, we only hit the 26% limit for the registers, but we almost maxed out on the on-chip memory, which was

LUT	12480
DSP	0
Memory (BRAM)	1308672
Flip-Flop	3751
Frequency	114.42MHz
Static Power	96.18mW
Dynamic Power	0.66mW
Total Power	106.16mW

Timeline:

The project took place over 4 weeks. By the end of the first 2 weeks, we had the platform randomly generated and were able to utilize the physics engine as promised for the mid checkpoint. Then over the next week, we added the scrolling for the game, as well as the start and end menu, and the last week is when all the final touches went on - such as the colored platforms, sprite backgrounds, and monsters. Overall the last week took the most amount of time, with two all-nighters spent, but everything else was distributed pretty evenly as noted by our commit history on GitHub.

Conclusion:

All in all, we completed the doodle jump game on FPGA successfully, and also went beyond our own expectations of what we decided to implement. We added a lot more things that allow for the game to be quite similar to the original. Had we had more time, we would spend time attempting to make sound work since most modern games have to sound like a feature that adds to the playability of the game. We could also decide to upgrade all the textures, but this would mean that the memory for the sprites has to be moved onto the SDRAM, instead of using the on-chip memory. This is because the OCM is already at 81%. Finally, we would like to say that the final project was very helpful in further improving our System Verilog skills since we got to practice a lot, especially with a VGA screen and actual hardware. It was interesting to understand how each piece of hardware we learned this semester could be connected to each other to create a fun game.