1. Introduction

For the final project I had designed and implemented the Tetris game on the DE10-lite board. This project has the same functionality as the original game where the player must place a Tetris piece in rows using “A” and “D” to move left and right respectively, along with “S” to move the piece down. If a row is filled that row is cleared and the rows above are moved down . The player can also rotate the pieces to fit into spots and lock into place and control the next piece. Some concepts added in are “wall kicks” where when the player attempts to rotate a piece, but the position after rotation is obstructed the game will attempt to "kick" the piece into an alternative position nearby along with a basic scoring system.

2. Written Description

For this project I had utilized the VGA and USB peripherals as visuals and control. Other important elements were added via SoC were the NIOS II CPU for interfacing with the USB and, and SRAM for using the CPU with the hardware. The USB was implemented through software so as to be able handle the inputs from the keyboard, all other components were implemented via SystemVerilog since they are connected to the main system’s bus.

List of Features

The baseline list of features that are included in the final product are the locking of the pieces, deleting and moving of rows, the controlling of the different pieces as they come. Other baseline features included are a basic start screen and end screen asking to play or replay so that there is a way for the game to start and end and, additional features that were added are including a scoreboard and scoreboard text, different colors for each piece, and a progressive speed to make the game harder the better you do.

3. Block Diagram and State Machines

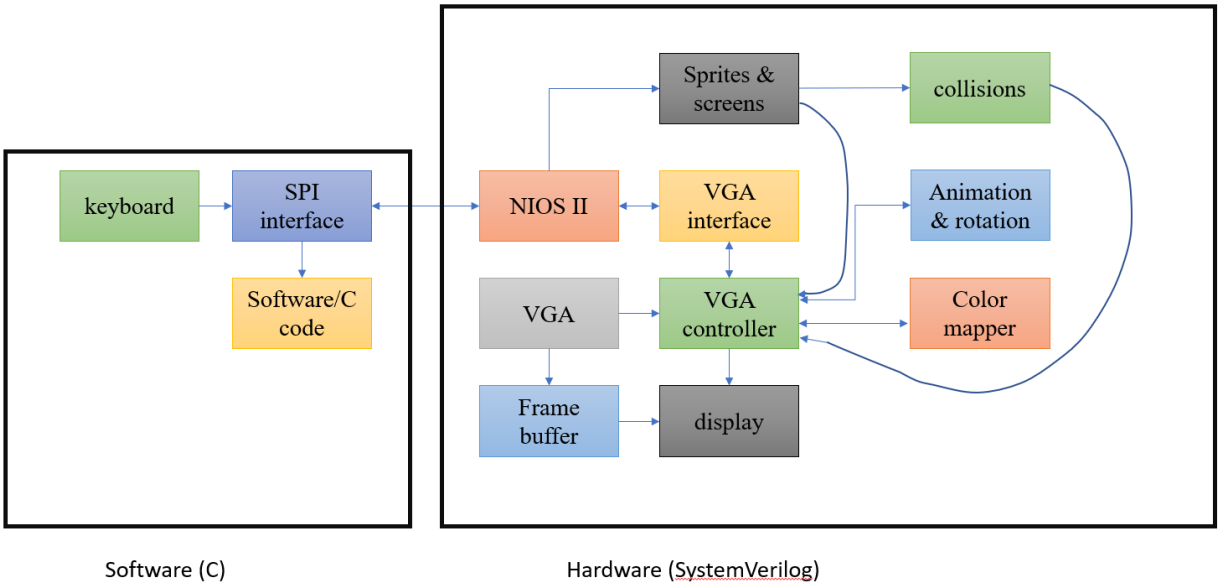


Figure 1: Original Block Diagram Idea

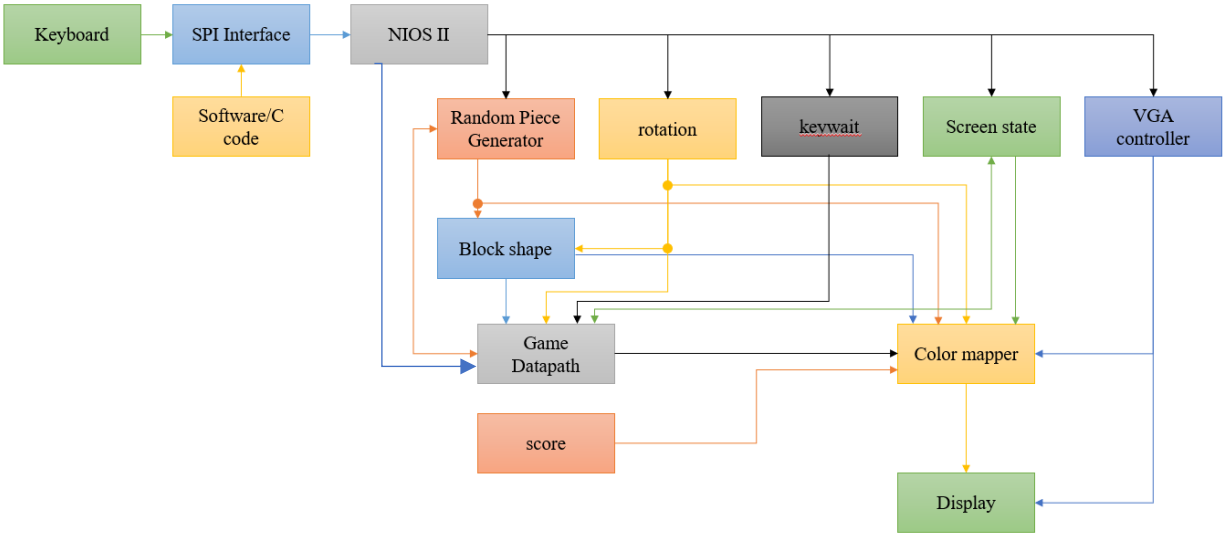


Figure 2: final implemented top level block diagram

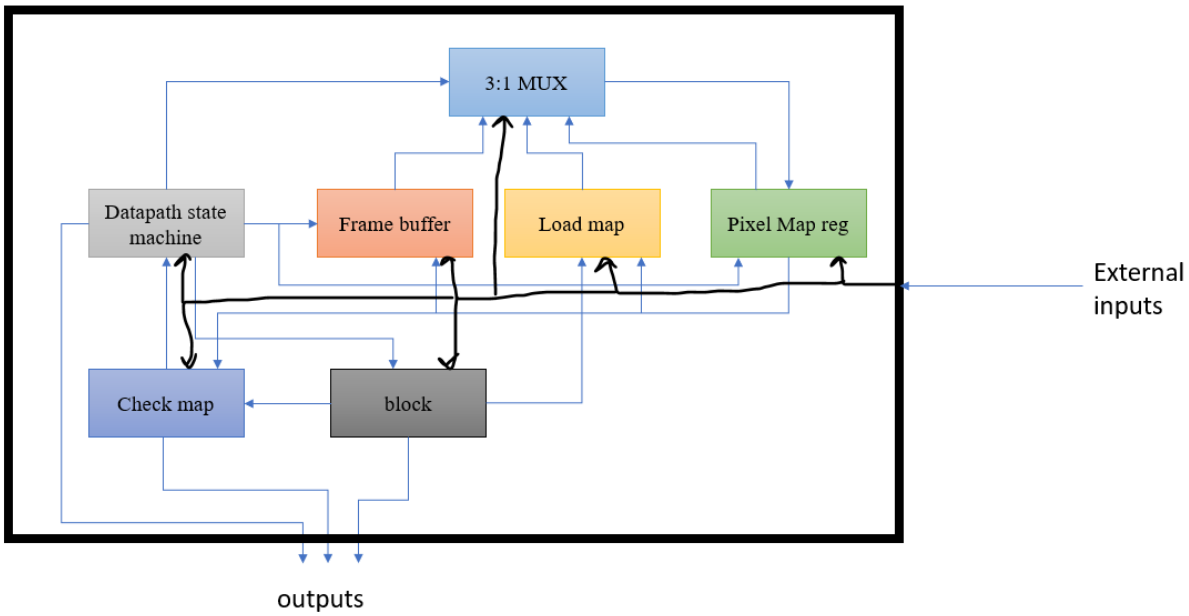


Figure 3: datapath block diagram

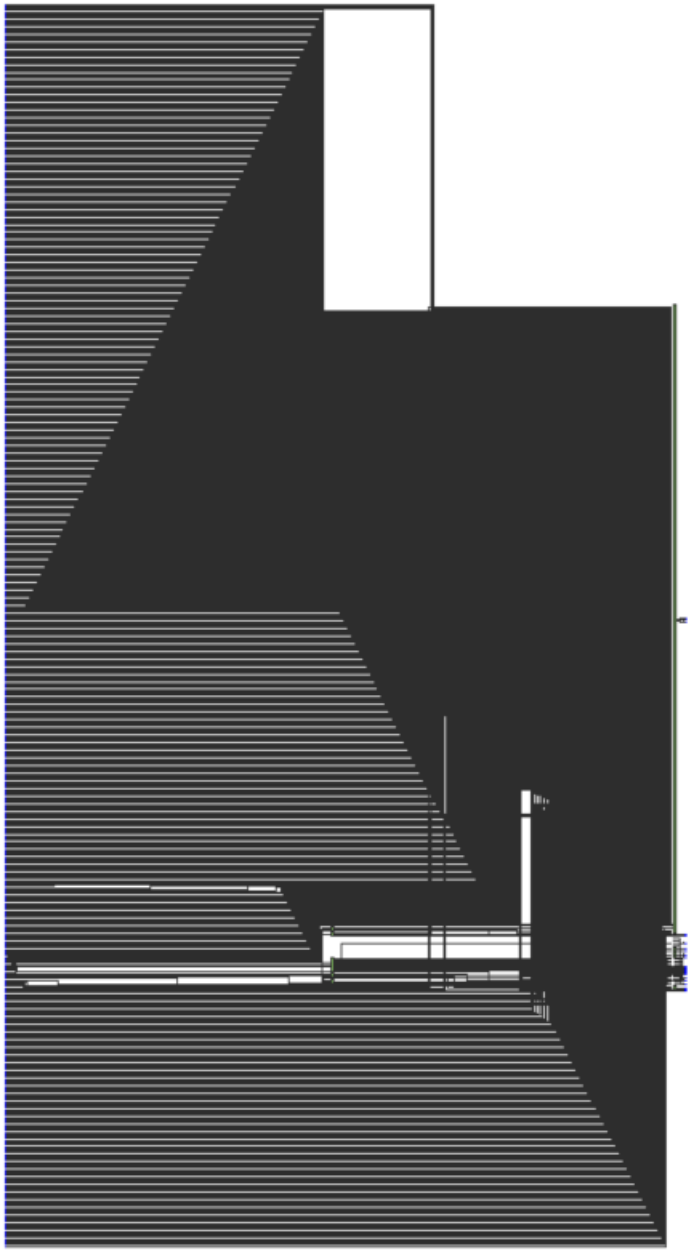


Figure 4: lab62 top-level

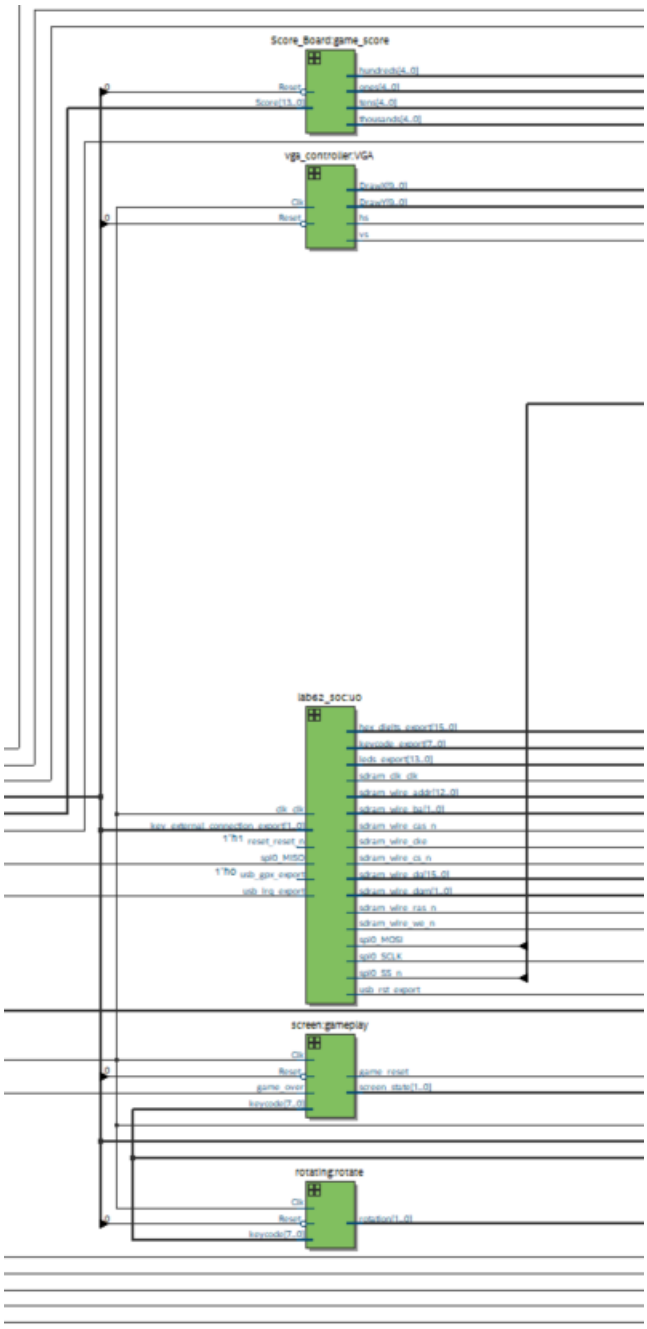

Score_Board:game_score
Reset
Score[13..0]
hundreds[4..0]
tens[4..0]
thousands[4..0]
vga_controller:VGA
Clk
Reset
hs
vs
labez_soc:uo
clk_clk
reset_reset_n
spi0_MISO
usb_gpx_export
usb_irq_export
sdram_wire_addr[12..0]
sdram_wire_ba[1..0]
sdram_wire_cas_n
sdram_wire_cke
sdram_wire_cs_n
sdram_wire_dq[15..0]
sdram_wire_dqm[1..0]
sdram_wire_ras_n
sdram_wire_we_n
spi0_MOSI
spi0_SCLK
spi0_SS_n
usb_rst_export
screen:gameplay
Clk
Reset
game_over
keycode[7..0]
game_reset
screen_state[1..0]
rotating:rotate
Clk
Reset
keycode[7..0]
rotation[1..0]

Figure 5: modules in top level

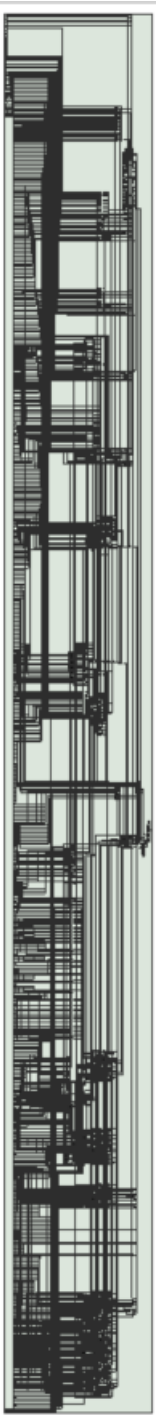


Figure 6: color mapper module

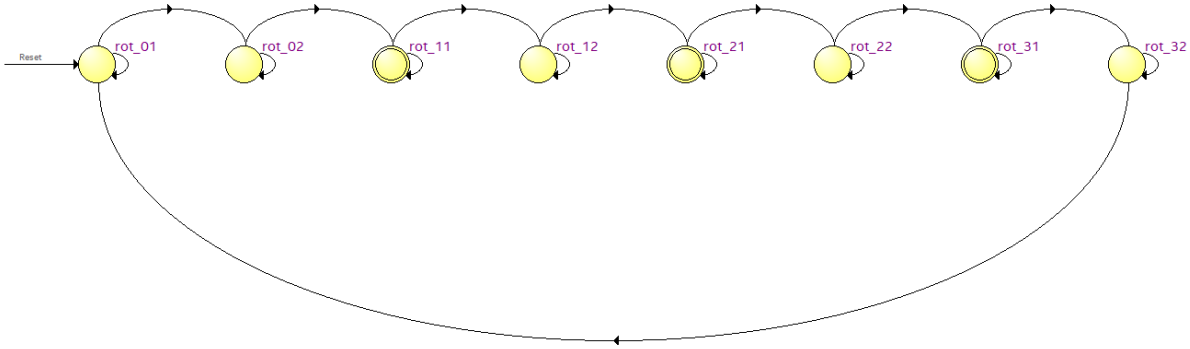


Figure 7: Rotating state machine

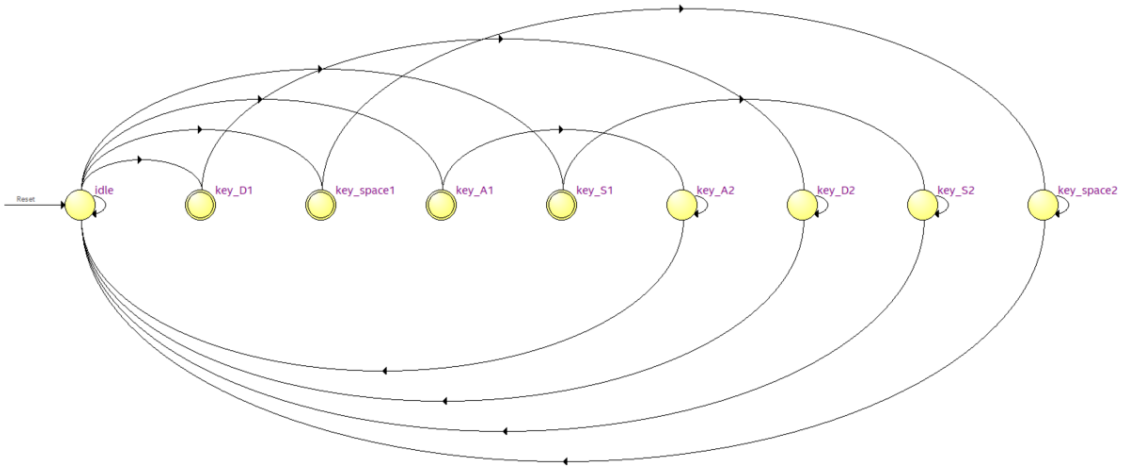


Figure 8: key buffer state machine

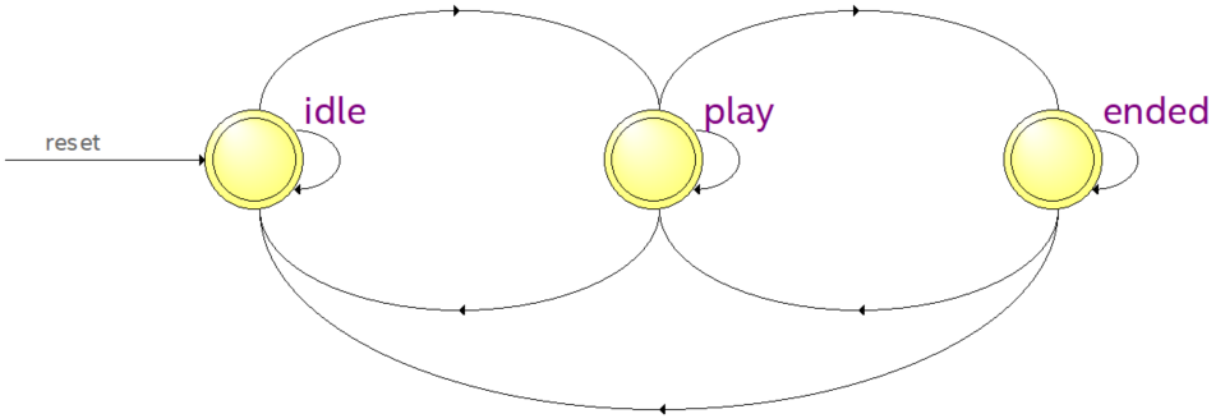


Figure 9: screen state machine

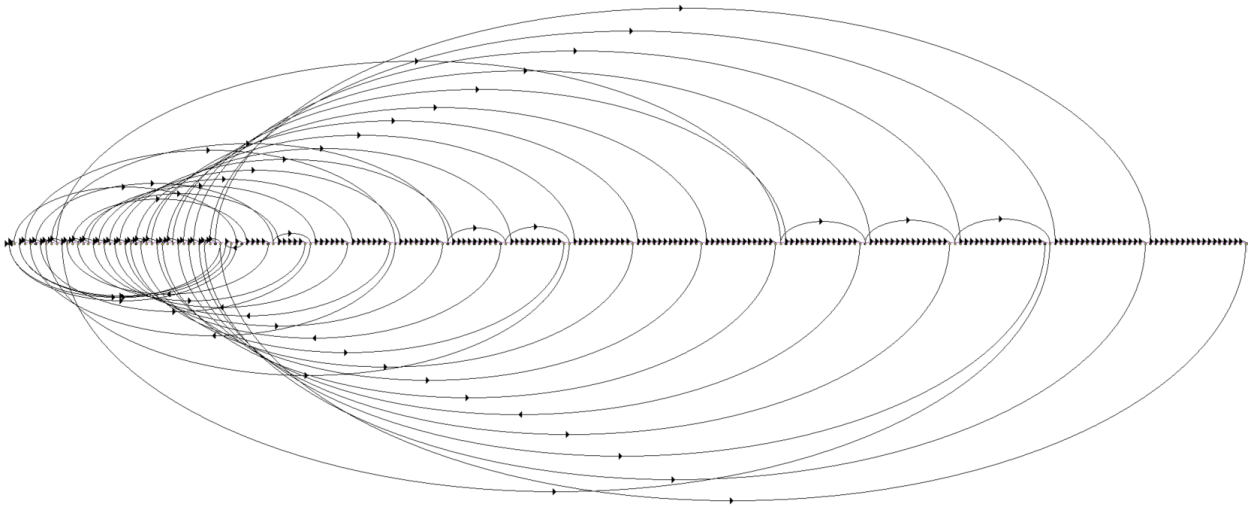


Figure 10: datapath state machine

4. Module Description

Module: **lab62_soc/lab62_soc**

Inputs: clk_clk, [1:0] key_external_connection_export, reset_reset_n, spi0_MISO, usb_gpx_export, usb_irq_export,

Outputs: [15:0] hex_digits_export, [13:0] leds_export, [12:0] sdram_wire_addr, [7:0] keycode_export, [1:0] sdram_wire_dqm, [1:0] sdram_wire_ba, sdram_clk_clk, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export

InOuts: [15:0] sdram_wire_dq

Description: The connected components that were made in the platform designer.

Purpose: The purpose of this module is to connect all the blocks of the SoC from the platform designer so that the NIOS-II processor can interact with the peripheral.

Module: **lab62**

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

Outputs: [9:0] LEDR, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, DRAM_CLK, DRAM_CKE, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B

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

Description: The top-level file that interacts and houses all other game modules.

Purpose: To connect the SoC with the other modules such as the VGA controller, datapath, Color Mapper, screen state, scoreboard, random piece generator to be able to play the game.

Module: **ball**

Inputs: Reset, frame_clk, shape_reset, game_reset, stop_x_left, stop_x_right, [1:0] state, [2:0] keypress, [9:0] shape_size_x, [9:0] shape_size_y, [13:0] Score

Outputs: [9:0] shape_x, [9:0] shape_y, touching

Description: used to draw the Tetris piece as it moves around the screen.

Purpose: As the player moves the piece and rotates it, it will be updated and also uses wall kicks so that the piece stays within bound when rotating. Also, will stop the piece if it is touching the ground.

Module: **Block_pos**

Inputs: [2:0] shape_num

Outputs: [9:0]shape_size_x, [9:0] shape_size_y

Description: determines the size that each shape will

Purpose: From the given current piece, the dimensions of the size are given and are then to be used by the color mapper to properly display them from the given size.

Module: **block_shape**

Input: [2:0] shape_num, [1:0] rotation

Outputs: [9:0] shape_size_x, shape_size_y

Description: determines the size that each shape will have based on its rotation

Purpose: From the current piece and rotation the piece will be properly set int the pixel map and then displayed by the color mapper based on the pixel map

Module: **check_Pixel_Map**

Inputs: Clk, touching, [19:0][9:0][3:0] PixelMap, [9:0] shape_x, [9:0] shape_y, [2:0] rotation, [1:0] shape_num

Outputs: NeedsLoad, stop_x_left, stop_x_right, endgame, NeedsClear, line0, line1, line2, line3, line4, line5m line6, line7, line8, line9, line10, line11, line12, line13, line14, line15, line16, line17, line18, line19

Description: determines if a line needs to be cleared and then loads in the next one

Purpose: if a row is filled(all its cells are filled) then the line is then cleared then loads in the next lines. Also check if a piece is on the edge when doing a rotation. Game over is determined if the there are any block past the top row.

Module: **color_mapper**

Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] shape_size_x, [9:0] shape_size_y, [9:0] shape_x, [9:0] shape_y, [7:0] keycode, [2:0] shape_num, [2:0] next_shape, [2:0] next_shape2, [1:0] game_state, [1:0] rotation, [19:0][9:0][3:0] PixelMapOut, [4:0] thousands,

[4:0] hundred, [4:0] ten, [4:0] ones

Outputs: [7:0] Red, [7:0] Green, [7:0] Blue

Description: Displays the screen based on the current state and piece movements

Purpose: Draws the outputs of everything such as the pieces, the different screens, the text, and score, updating everything based on the "electron gun's" current position.

Module: **datapath**

Inputs: Clk, Reset, game_reset, frame_clk, [9:0] shape_size_x, [9:0] shape_size_y, [2:0] shape_num, [2:0] keypress, [1:0] rotation, [1:0] state

Outputs: shape_reset, touching, game_over, stopxleft, stopxright, [19:0][9:0][3:0] PixelMapOut [13:0] theScore, [9:0] shape_x, [9:0] shape_y

Description: controls what is going in currently in the game

Purpose: controls the logic of the actual Tetris game(pieces and the map of what the game looks like).

Module: **frame_buffer**

Inputs: Clk, reset, LoadAll, line0, line1, line2, line3, line4, line5 line6, line7, line8, line9, line10, line11, line12, line13, line14, line15, line16, line17, line18, line19 [19:0][9:0][3:0] PixelMap

Outputs: [19:0][9:0][3:0] PixelMapOut

Description: top level for all the Row registers and makes 10x20 cell play area

Purpose: using the row registers each it makes the map for the play area and preloads all the rows that are on top of each other so if a row is cleared the proceeding row can be moved down.

Module: **datapath_state_machine**

Inputs: Clk, reset, NeedsClear, NeedsLoad, game_reset, line0, line1, line2, line3, line4, line5 line6, line7, line8, line9, line10, line11, line12, line13, line14, line15, line16, line17, line18, line19

Outputs: [13:0] Score, [1:0] Mux_Select, LoadRealReg, LoadAll, load0, load1, load2, load3, load4, load5, load6, load7, load8, load9, load10, load11, load12, load13, load14, load15, load16, load17, load18, load19, ResetShape

Description: state machine for the actual game that is to be played

Purpose: a state machine that controls the pieces and the pixel map and determines what signals to send when a line is cleared, or game is reset.

Module: **HexDriver**

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Output display to the on-board hex displays.

Purpose: used for debugging by displaying the keycode of the key being pressed.

Module: **keywait**

Inputs: Clk, Reset, [7:0] keycode

Outputs: [2:0] key

Description: asynchronous buffer for keys pressed on the keyboard

Purpose: sets the input from the keyboard to be asynchronous and outputs them as a constant so that only the accepted keys do anything in the game.

Module: **Load_Pixel_Map**

Inputs: Clk, Reset, game_reset, [19:0][9:0][3:0] PixelMapIn, [9:0] shape_x, [9:0] shape_y, [2:0] shape_num, [1:0] rotation

Outputs: [19:0] [9:0][3:0] PixelMapOut

Description: sets each cell in the map to display what is currently going on in the game

Purpose: while the game is active and there are pieces it will properly set each cell in the map to display the piece in that cell as the game dynamically changes.

Module: **PixelMapMux**

Inputs: Reset, sel, [19:0][9:0][3:0] a, [19:0][9:0][3:0] b, [19:0][9:0][3:0] c

Outputs: [19:0][9:0][3:0]out

Description: mux for determine which output map to display

Purpose: Determines which map is to be displayed from between the current, cleared, or updated map.

Module: **Pixel_Map_reg**

Inputs: Clk, Reset, Loadreg, game_reset, [19:0][9:0][3:0] PRegIn,

Outputs: [19:0][9:0][3:0] PRegOut

Description: an array that is used to hold the information of the play area with the Tetris pieces

Purpose: The play area is split as 10x20 array and held as registers so that each cell can be changed independently.

Module: **rotating**

Inputs: Clk, Reset, [7:0] keycode

Outputs: [1:0] rotation

Description: determines the state of the rotation a piece is in based on the user's input.

Purpose: To be used for determining the rotational state of a piece which is then used by the color mapper to figure out how to display the piece in that rotation. Also used by the datapath to send to the maps to be able to do a wall kick.

Module: **Row_Reg**

Inputs: Clk, Reset, LoadAll, LoadRow, [9:0][3:0]PixelMapIn, [9:0][3:0] RowIn

Outputs: [9:0][3:0] Out

Description: used for loading in a row after a line has been cleared.

Purpose: To clear a row that has been filled and move the row above it down one. To be used by the frame buffer

Module: **random_piece**

Inputs: Clk, Reset, shape_reset, game_over, touching

Outputs: [2:0] shape_num, [2:0] next_shape, [2:0] next_shape2

Description: Determines the next pieces that will be displayed randomly using a counter.

Purpose: This module is the “bag of 7” or random number generator that will randomly choose the next piece that will be used for the game.

Module: **screen**

Inputs: Clk, Reset, game_over, [7:0] keycode

Outputs: game_reset, [1:0] screen_state

Description: determines the state of the screen depending on what the current state and player inputs

Purpose: A state machine used to determine what the screen should be displaying based on its state.

Module: **sprite_table_block**

Output: [3:0][3:0][3:0] I_block_U, [3:0][3:0][3:0] O_block_UB, [3:0][3:0][3:0] J_block_UB, [3:0][3:0][3:0] T_block_UB, [3:0][3:0][3:0] L_block_UB, [3:0][3:0][3:0] S_block_UB [3:0][3:0][3:0] Z_block_UB

Description: used to determine the color of each piece as an individual block

Purpose: since the Tetris piece are made up of 4 blocks in a different combination, a single block is used to determine its color which is used by color mapper to the display the block in that color.

Module: **sprite_table_font**

Outputs: [10:0][24:0][3:0] game_over, [4:0][39:0][3:0] TETRIS_h, [3:0][15:0][3:0] I_block_h, [15:0][3:0][3:0] I_block_v, [7:0][7:0][3:0] O_block, [7:0][11:0][3:0] J_block_0, [11:0][7:0][3:0] J_block_1, [7:0][11:0][3:0] J_block_2, [11:0][7:0][3:0] J_block_3,

[7:0][11:0][3:0] T_block_0, [11:0][7:0][3:0] T_block_1, [7:0][11:0][3:0] T_block_2, [11:0][7:0][3:0] T_block_3, [7:0][11:0][3:0] L_block_0, [11:0][7:0][3:0] L_block_1, [7:0][11:0][3:0] L_block_2, [11:0][7:0][3:0] L_block_3, [7:0][11:0][3:0] S_block_0, [11:0][7:0][3:0] S_block_1, [7:0][11:0][3:0] Z_block_0, [11:0][7:0][3:0] Z_block_1, [4:0][28:0][3:0] SCORE_Letters, [4:0][4:0][3:0] zero, [4:0][4:0][3:0] one, [4:0][4:0][3:0] two, [4:0][4:0][3:0] three, [4:0][4:0][3:0] four, [4:0][4:0][3:0] five, [4:0][4:0][3:0] six, [4:0][4:0][3:0] seven, [4:0][4:0][3:0] eight, [4:0][4:0][3:0] nine

Description: used to determine the shape, size and color of the game text and blocks.

Purpose: game text and blocks are made as arrays. Text, numbers, and every piece orientation is an array of pixels that are then displayed by the color mapper

Module: **VGA_controller**

Inputs: Clk, Reset

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

Description: determines the horizontal and vertical syn to “moves” the “electron gun” to the proper x and y that the pixel is being drawn at and blanks during blank time.

Purpose: To properly position the electron gun for the pixels to be drawn at.

Module: **Score_board**

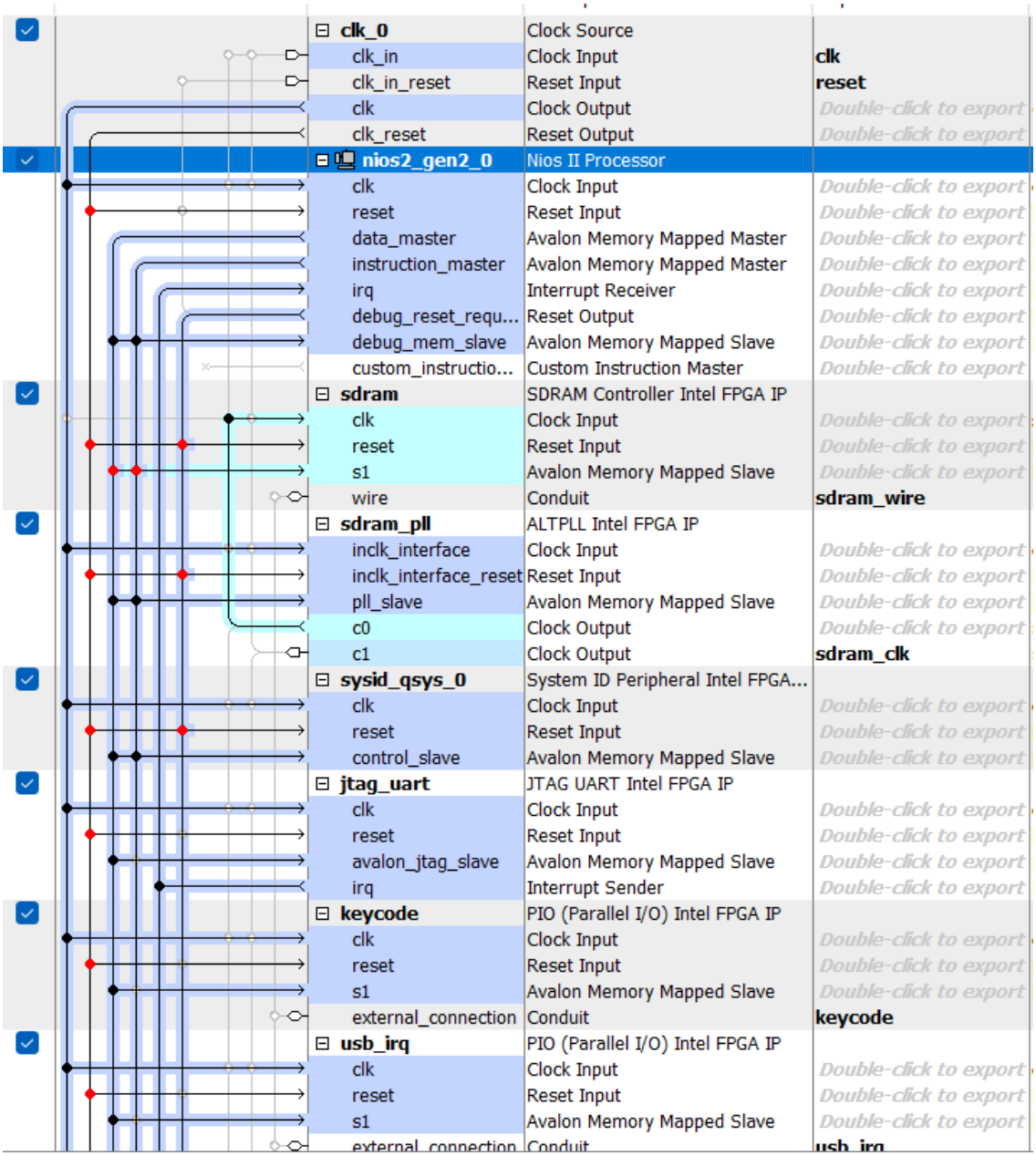
Inputs: Reset, [13:0] Score

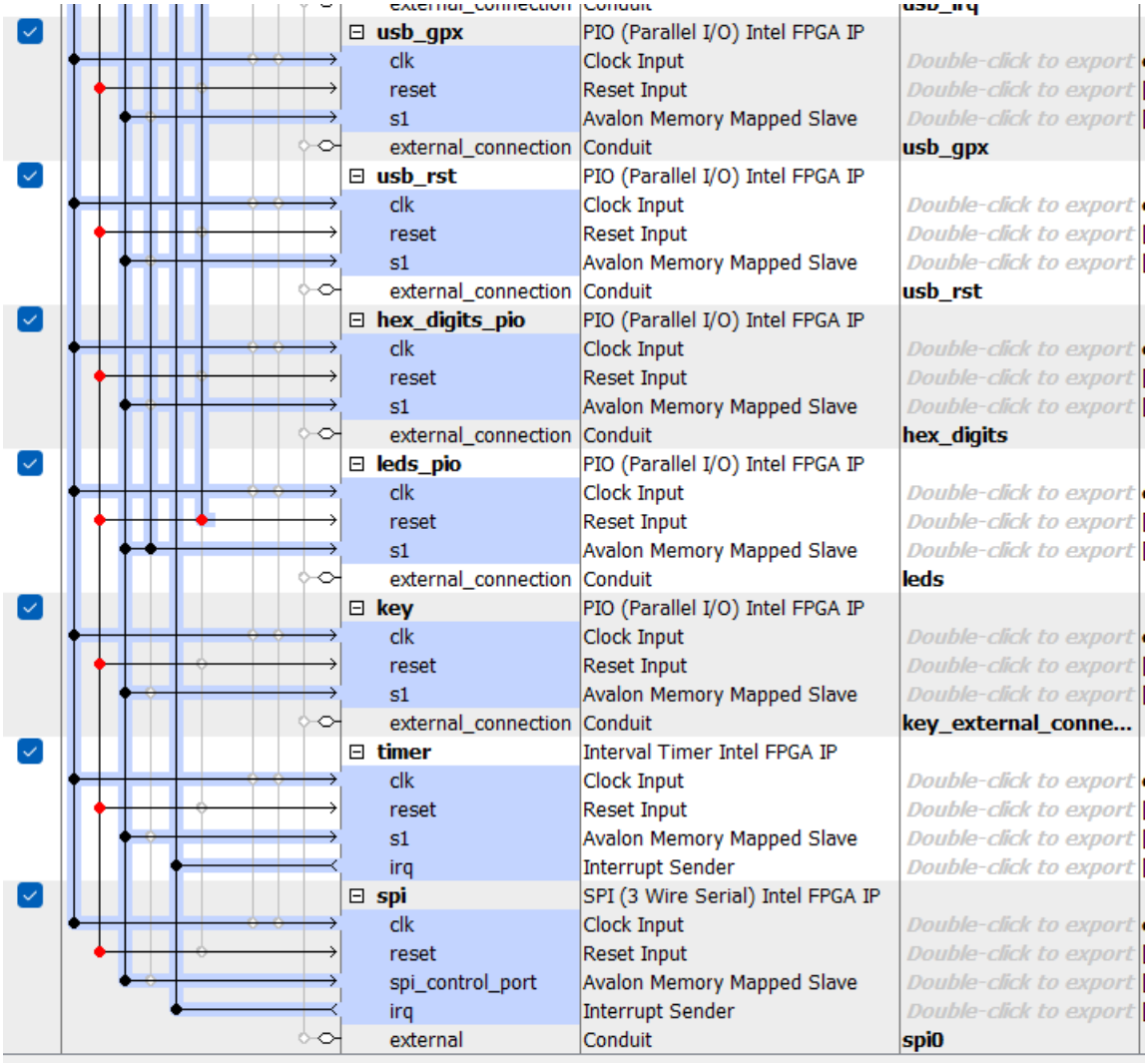
Outputs: [4:0] thousands, [4:0] hundreds, [4:0] tens, [4:0] ones

Description: Given an input of a score it outputs each digit of the score.

Purpose: To be used to determine what to draw on the screen on the scoreboard for the current score

5. Platform designer


clk_0 Clock Source
clk_in Clock Input clk
clk_in_reset Reset Input reset
clk Clock Output Double-click to export
clk_reset Reset Output Double-click to export
nios2_gen2_0 Nios II Processor
clk Clock Input Double-click to export
reset Reset Input Double-click to export
data_master Avalon Memory Mapped Master Double-click to export
instruction_master Avalon Memory Mapped Master Double-click to export
irq Interrupt Receiver Double-click to export
debug_reset_requ... Reset Output Double-click to export
debug_mem_slave Avalon Memory Mapped Slave Double-click to export
custom_instructio... Custom Instruction Master Double-click to export
sdram SDRAM Controller Intel FPGA IP
clk Clock Input Double-click to export
reset Reset Input Double-click to export
s1 Avalon Memory Mapped Slave Double-click to export
wire Conduit sdram_wire
sdram_pll ALTPLL Intel FPGA IP
inclk_interface Clock Input Double-click to export
inclk_interface_reset Reset Input Double-click to export
pll_slave Avalon Memory Mapped Slave Double-click to export
c0 Clock Output Double-click to export
c1 Clock Output sdram_clk
sysid_qsys_0 System ID Peripheral Intel FPGA...
clk Clock Input Double-click to export
reset Reset Input Double-click to export
control_slave Avalon Memory Mapped Slave Double-click to export
jtag_uart JTAG UART Intel FPGA IP
clk Clock Input Double-click to export
reset Reset Input Double-click to export
avalon_jtag_slave Avalon Memory Mapped Slave Double-click to export
irq Interrupt Sender Double-click to export
keycode PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click to export
reset Reset Input Double-click to export
s1 Avalon Memory Mapped Slave Double-click to export
external_connection Conduit keycode
usb_irq PIO (Parallel I/O) Intel FPGA IP
clk Clock Input Double-click to export
reset Reset Input Double-click to export
s1 Avalon Memory Mapped Slave Double-click to export
external_connection Conduit usb_irq



clk_0 – The main clock driving for the components and SoC on the FPGA at 50MHz with a clk reset signal.

nios2_gen2_0 – The CPU and centerpiece of the SoC sending data and instructions out to other blocks.

sdram – Off-chip memory used for storing the software and allows for interfacing with external PIO's. The sdram takes in data from the CPU and takes in a shifted clock from sdram_pll.

sdram_pll – This block outputs a shifted clock for the sdram so that the sdram is able receive stable and precise timing for read and write.

sysid_qsys – Ensures compatibility between hardware and software by checking for incompatibility.

key – PIO input from the keys from on-board button on the FPGA. Only the reset button was established to reset the program and hardware.

led – PIO 8-bit output to the LEDs on the FPGA used for debugging the shape and rotation.

jtag_uart – Allows for debugging by giving the host computer a way to communicate with the NIOS-II(print and scan statement in C).

keycode – Used for determining which key is pressed on the keyboard at any given moment.

usb_irq/usb_gpx /usb_rst – Connection for the NIOS-II to be able to use the USB on the MAX3421E chipset.

hex_digit_pio – Output to the on-board display showing the keycode of the latest key pressed.

timer – Keeps track of the timing for the NIOS-II processor.

spi – Allows for communication between the USB and NIOS-II.

6. Design Process

Design

There are many different design choices made along the way for this project in the way that it was initially set up. First off is that this lab was originally built using the same logic as lab 6 using the USB I/O and SPI protocol to be able to use a keyboard with all the hardware. Since only one key is ever really used at any given time no real modification was needed, except for the modifications in my lab6 software so that I had proper communication with the keyboard.

Next, there was the choice of breaking the screen into blocks of size 16x16 and use those blocks as cells to be able to control them individually from one another. This choice in in of itself I had decided to make as registers as opposed to using ram since I wanted this to be a purely hardware implementation of the game and was not very good with determining the usage of the ram as software. With the use of registers and dividing the stage into cells I could make an 3D array for the map of the using the width and height as 2 dimensions and the size that each cell is as the third(the array of [19:0][9:0][3:0] denotes the 20 blocks high x 10 blocks wide, where each block is 16 pixels)

For text the sprites were made by making a character be 5 bits x 5 bits and making an array with other characters so that once on the proper spot, the consecutive bits would be displayed. The array had been made using decimal number so that each number in the array denoted the color of that should be displayed for the given text or sprite.

Another design choice that had been made was using state machines, this was done for different for the various places that it was used. The three places a state machine was used were the rotation of a piece, they keypress on the keyboard, and the datapath. For the pressing of a key, it was used for both making key presses asynchronous and determining if the key pressed was a key that we had actually cared for to determine any game logic by using wait states(keys such as “W”,”A”,”S”,”D”, ENTER, and SPACE BAR). For rotation it was so the information of the transition would only happen when they key was pressed, and the piece had not done one to many rotations. As for the datapath, it was used more for controlling the bus that was internal to the datapath since the datapath controlled the map of the stage and there needed to be proper control of what was going on and what to do when. The state machine of the datapath was similar to that of lab 5’s ISDU and datapath, only here it was more to do with the loading and clearing of lines as the game went on.

Bugs

While this project works well there are a few bugs that were not able to be fixed due to time constraint, one of these bugs was that with the next randomly generated pieces. The bug was after the first piece had been set and locked the second piece would be “glitchy” and would be hard to determine what the actual piece was due to all the sprites mixing together when displaying. This made it so that the game could not be perfectly playable, and the workaround to ensure that everything else still worked was to hardwire the next shapes to a specific piece to be able to properly test.

Another notable bug is in implementing the locking hard drop. The idea behind the hard drop was that when the player had pressed space bar the piece would drop all the way down and lock into place. While hard dropping a piece and then placing another has no problem the issue comes when you hard drop a piece as soon as it appears and then hard drop another piece at the same spot that the first piece was hard dropped. This bug comes from how the stage is set up where the pixel map of the 10x20 play area does not perfectly update, so the map detects that the two pieces are on top of each other even thought it displays normally. This issue only occurs if you hard drop near the top and the process for checking for a loss believes that there is a piece out of bounds and leads the game to a game over state.

7. Design Resource stat

LUT	40004
DSP	0
Memory (BRAM)	11264
Flip-Flop	5176
Frequency	39.42 MHz
Static Power	97.13 mW
Dynamic Power	177.85 mW
Total Power	309.22 mW

8. Conclusion

While this project was mostly properly functional with the exception of the drawing of the next piece, this project was successful in implementing what I had learned over the course of this class. This project was a standard Tetris game that had lots of its own challenges, namely being that it was worked on as a solo project with many errors and bugs before any work had even gotten started. It was fun to see how it all came together using the hardware and what can be done on the FPGA DE10-lite board.

In its final state the Tetris game there are many features that could be added or fixed beyond this point. Such as fixing the next piece after a piece is locked and the hard drop, along with sound for a better experience and using ram instead of registers for a faster compile time. The design and work of this project was hard but met in time for the deadline to be functioning well enough to be shown in what was done and is capable of doing.

Overall, this project was a valuable part of the ECE 385 course. Making me think about the creativity of what was to be implemented and how to go about implementing that in hardware to be successful in working properly and correctly.