ECE 385

Spring 2025

Final Project

Tetris on FPGA with SoC

John Youkhana

ECE 385 Section AL1

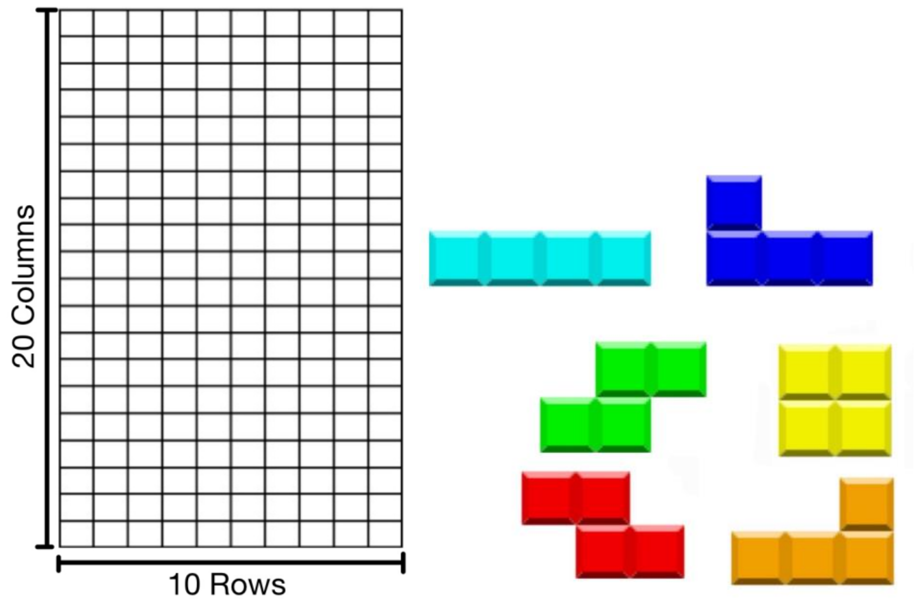
TA: Jinghan Huang

Introduction

For my final project, I decided to recreate one of my favorite childhood games onto the Urbana board. To do this, we used various concepts and modules from lab 6 and lab 7, mainly for USB input and HDMI output. As well as a new concept for audio output, not previously covered in lab. The proper implementation of this game has many moving parts, for display, input, and audio. For the game's combinational logic, we need to ensure that the game functions properly and is as similar to Tetris as possible. This includes block spawning, movement, response to inputs, clearing filled rows, etc. We also need a way to read USB inputs, in this case from a USB keyboard. The USB inputs will be used to control the movement of the Tetris blocks. We also need a way to display our game, we need to create combinational logic that ensures that the playfields, backgrounds, and Tetris blocks are being printed to the correct colors, as well as the on-screen text being printed correctly. This information needs to be converted to HDMI signals that can be sent to the HDMI output of the Urbana board. Finally, we need to send audio signals to the audio out port of the FPGA.

Explanation of game logic

I decided the best way of going about the design of the FPGA Tetris game is to treat the play area as a 10x20 grid, similar to how the real Tetris game is designed. To do this, I used a logic variable [2:0] play_area [10][22], which is a register array of 10 columns by 22 rows, each storing a value that can be up to 3 bits wide. The extra 2 rows are used at the top, and are used for the purposes of drawing blocks, but they don’t appear on screen as part of the play area. This allows us to make it seem like the Tetris blocks are falling from the top of the screen, rather than instantly spawning into the play area. The grid drawing and color is controlled by DrawX and DrawY, which is used to determine if the current pixel being drawn is within the play area. If so, we can choose different RGB values to be printed into that grid, depending on the value in the register. Each register in this array is treated as a grid on screen and can hold a value between 0 and 7. The different values in each grid will determine what color should be printed into that grid. In our game, a value of 0 means that the current grid is empty, and we assign it the color black. for a value of 1, that current grid contains a “settled” or placed block, which is a block that had already been placed down and had not been cleared yet. For a value of 2, this means that the current grid holds a section of the 2x2 square block, which we assigned to be the color yellow. For a value of 3, the current grid holds a section of the line block 4x1 line block, which we chose to be a light blue. A value of 4 means that the grid contains a section of the S block, which we chose to be green. A value of 5 means the grid contains a section of the Z block, which we chose to be red. A value of 6 means the grid contains a section of the L block which we chose to be orange. Finally, a value of 7 means the grid contains a section of the J block, which we chose to be dark blue. The colors of the grid are picked by assigning them RGB values, which can be mixed to pick various colors. For example, we created yellow by assigning a grid with a value of 2, The RGB values of {15,15,0}. The RGB value of the current pixel is sent to the VGA-to-HDMI converter to print the color of the current pixel onto the screen.

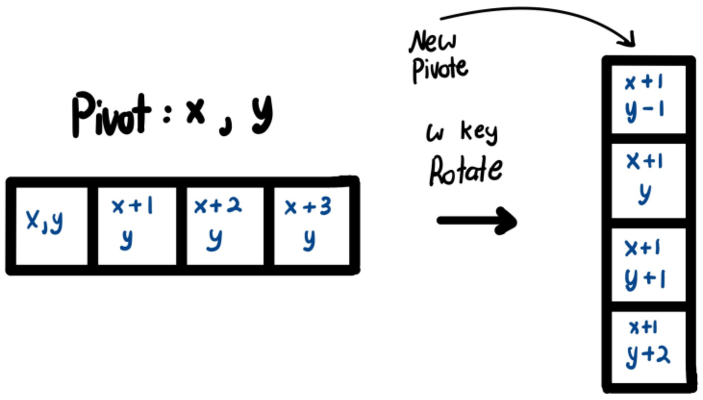


As for on-screen text, we had both static texts, and a changing on-screen score. For static text, we had two different signs. The "Welcome to Tetris" sign, and the "Lines cleared sign. The letters were stored in 7-bit registers, and the amount depended on the length of the phrase. "Welcome to Tetris" for example had 17 characters, so it was stored in a variable as [6:0] welcometext[17]. Each register held the hexadecimal value of a specific ASCII character. welcometext[0] for example stored x57, which is the hexadecimal value of the ASCII character W, which is the first letter of the sign. The hexadecimal value is used to search for the character in font rom. To compute the address for font rom, this hexadecimal value is picked by the algorithm using combinational logic, to determine which letter we are currently drawing. This value is multiplied by 16, because the font data for each character is 16 addresses tall. We then add zero indexed DrawY, to determine which row in the 16 tall character we are currently drawing. For example, ((linestext[(DrawX - 468) >> 3]) << 4) + (DrawY - 224) is used for the address of the lines cleared sign. We subtract 468 from DrawX, because the sign begins at x = 468, and 224 from DrawY, because we begin at y = 224. DrawX is divided by 8, since each character is 8 pixels wide, and multiplied by 16, to account for the 16 rows per character. The score is computed similarly, the "ones" position of the score is computed as ((8'h30 + (score % 10)) << 4) + (DrawY - 240). We add 0x30, so we can start at the ASCII character of zero, score % 10 is used to ensure the score does not exceed 9, this is multiplied by 16 to account for the 16 rows per character, and we add adjusted DrawY to determine which row of the character we are printing. The value of "score" is determined in the block module, and is incremented each time a row is completed, and is set to zero during reset.

Moving on to the block logic, a block to be printed is selected with a random number variable, which determines the current block to be printed. If the random number is 0, we print the square, if 1, then the line block, if 2, then s block, if 3, then z block, if 4, then L block, if 5, then J block. If a certain block is selected, it is printed by assigning certain grids the value of that block. For example, if we select the square block, we assign grid[4][0], grid[4][1], grid[5][0], and grid[5][1] the value of 2, which initiates the square block. We need the Tetris block to only drop a grid vertically, if there is no other block in the way. We have an algorithm that checks if the current block is allowed to drop any lower. We have a nested loop to check every column and row in the play area, if the current grid has a value greater than or equal to 2, that means that grid has part of the current block.

To check if the block is good to move, we use grid[i][j+1] != 0 && grid[i][j+1] != grid[i][j]. grid[i][j+1] != 0 checks if something is at the next row, in this case, 1 would be a placed block, and grid[i][j+1] != grid[i][j] checks if that something is NOT part of the current block. If these are true, that means our Tetris block is no longer able to move, and we set a "validtodrop" signal to zero which flags for this block to be placed. For our movement algorithm, we first check if the block is valid to move, using the signal from earlier. If we are okay to drop, we scan the whole grid, if a grid is occupied by the block, we lower the block by assigning grid[i][j] to grid[i][j-1]. If a block is no longer able to move, If grid[i][j] is currently occupied by a block, we assign grid[i][j] to equal 1, setting it to a "settled" block. a "generate" signal is then used to generate a new block and start this same cycle over again. Next, we have an algorithm for checking for completed rows, if we find that every column in a row contains the value 1, meaning its full of settled blocks, we set the RowComplete signal to 1. With this signal high, we increment the score, and we lower every grid on top of that cleared row, down by one row, We do this by incrementing through every column of only the rows we have checked already, and assigning each grid to grid[k][l-1], effectively giving it the value of the previous row. We also increase the speed at which the blocks move down.

The next section of block logic is used to determine what happens depending on certain keycode inputs. We first deal with the logic necessary to ensure that the block rotates when the w key is pressed. To simplify this process, I assign each block "pivot points". For example, the pivot point of the square block is grid[4][0] which is the top left of the square block, while the pivot point of the line block is grid[3][1] which is the bottom of the line. For this example, we will use the line block, which only has one rotation, before reverting back to the default. Line block starts with grid positions [3][1], [4][1], [5][1], [6][1]. With [3][1] being the pivot, we can express [3][1] as (x,y), [4][1] as (x+1,y), [5][1] as (x+2,y), and [6][1] as (x+3,y). This makes it easier to rotate this block. When flipping the block, we decided that (x,y), (x+2,y), and (x+3,y) will be set to zero, leaving just (x+1,y). Then (x+1,y-1), (x+1,y+1), and (x+1,y+2) can be set to 3, effectively flipping the block, we will also pick a new pivot point as (x+1,y-1), which will set this to the new (x,y), next time we try to flip the block. A handwritten illustration is shown below.



A and D keys function in a similar manner. The A and D keys are used to move the block left and right respectively. We first need to check if the block is valid to move in the chosen direction. We do this in a similar manner that we had done previously for the falling block logic, except this time we check adjacent grids, to see if we have room to move either left or right. If we are good to move in that direction, we initiate the movement. For example, let's say we want to move our line block to the right, we initially start with pivots (x,y), (x+1,y), (x+2,y), and (x+3,y). For a right movement, we set (x,y) to zero, leaving the rest of the blocks, and setting (x+4,y) to 3, making that now part of the block, the new pivot will be set to (x+1,y). this effectively shifts the line block, one grid to the right. The final input used is the s key, which we use to force the block to drop all the way to the bottom of the grid. Pressing the s key forces a signal called "DropBlock" to high, this signal is used in an algorithm similar to the one previously mentioned, which controls the block dropping logic, The difference being, the block dropping logic mentioned previously uses a timer , so the block drops by one grid every half second. The DropBlock signal forces the block to continue to drop rows until the block is no longer valid to move.

USB Inputs

We modified the MicroBlaze microcontroller preset to support USB input from a simple USB keyboard. We used a USB host controller "MAX3421E" which allows our USB input to communicate with the MicroBlaze. The MAX3421E communicates with our MicroBlaze processor with an SPI protocol. With the USB input being set up through the MicroBlaze, The MicroBlaze sends keycodes depending on which USB key is pressed, and this keycode is read by our combinational logic to determine what happens in our game.

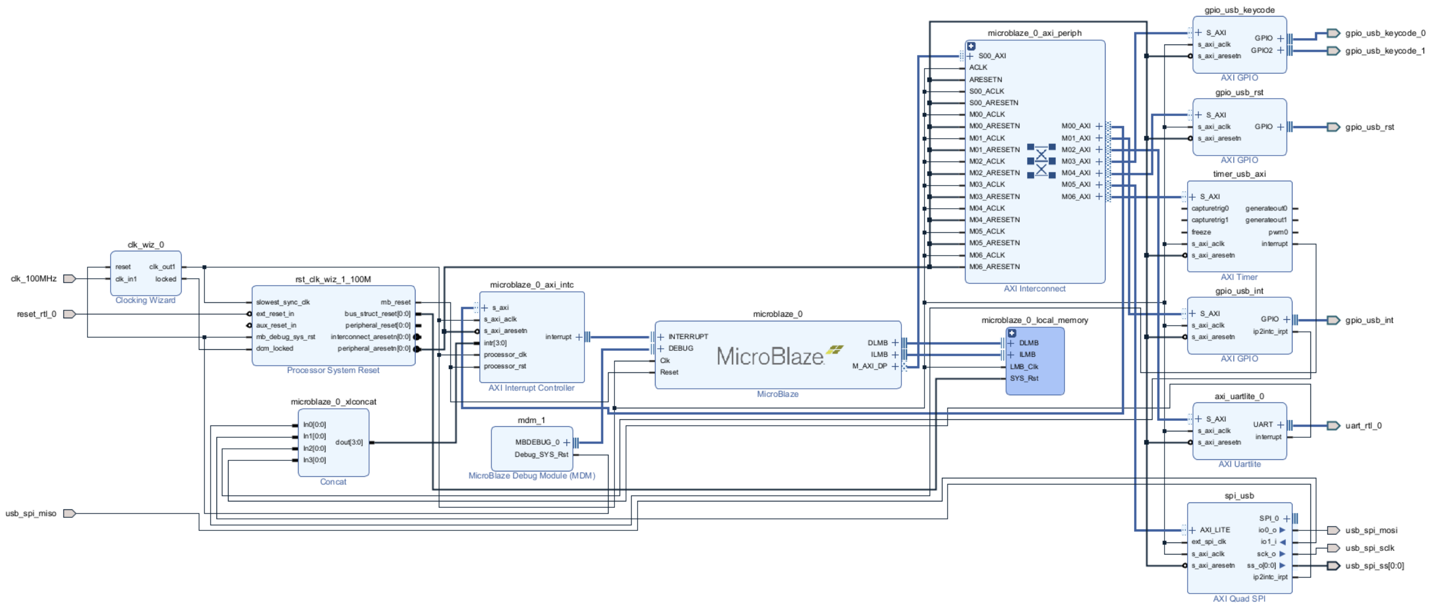


Figure: MicroBlaze USB setup from lab 6

The MicroBlaze and MAX3421E communicate with an SPI protocol, which sets up a master – slave relationship between the Microprocessor and the peripheral. In this case, the MicroBlaze processor is the master device, and the MAX3421E is its slave device. The MAX3421E is used as a USB host peripheral, when a key is pressed, the MAX3421E reads this as a keycode. The keycode depends on which key has been pressed. This keycode is sent from the MAX3421E to the MicroBlaze processor, which is then read by our combinational logic. The MAX3421E and SPI protocol of this project are software based and are programmed onto the MicroBlaze processor, allowing our input to be fully handled by our SoC processor.

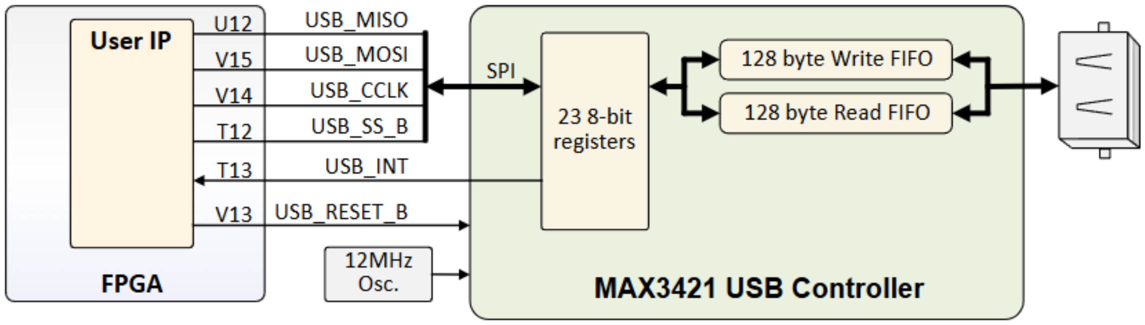


Figure 21. Urbana board USB host

HDMI output

The process of generating the designs HDMI signals starts at the provided VGA controller module. The VGA controller module generates VGA signals necessary for display. Such as hsync, vsync, and activenblank. Hsync or “Horizontal sync pulse” is a timing signal used to indicate the start of a new horizontal line, since VGA is drawn row by row. Vsync or “vertical sync pulse” is a timing signal used to indicate the start of a new frame. activenblank indicates when the pixels should be displayed. The VGA controller has two other outputs, DrawX and DrawY, which are 10-bit outputs used to map the current pixel being drawn. DrawX is a 10-bit output used to indicate horizontal position of the current pixel being drawn. Similarly, DrawY is a 10-bit output used to indicate the vertical position of the current pixel being drawn.

The Vsync signal is used by the block module as its frame clock, so the logic of the game can be synced with the framerate of the VGA signal. The DrawX and DrawY signal are used by the color mapper module. We use these signals to determine what is the current pixel that we are currently drawing, and based off that, what RGB values should be assigned to the current pixel. Static elements, such as on-screen text, or the play area, will cause the RGB values to change if they are within a certain range. For example, the tetris play area will be drawn if drawX is between 240 and 400, and when DrawY is between 80 and 400. As mentioned earlier, the RGB values depend on the current value in that grid, since we need to account for 6 different types of blocks, a placed block, and an empty grid. We do this by checking the value of grid[(DrawX-240)>>4][((DrawY-80)>>4)+2], and assign the colors accordingly. These RGB values are sent as outputs from the color mapper module and are sent as inputs into the VGA to HDMI converter IP. The VGA-to-HDMI converter IP is used to generate HDMI compatible signals, which can be used to display through an HDMI cable. This IP takes the 4-bit RGB values, vsync and hsync, in order to generate four HDMI compatible signals. A simplified block diagram of the VGA to HDMI conversion process is shown below.

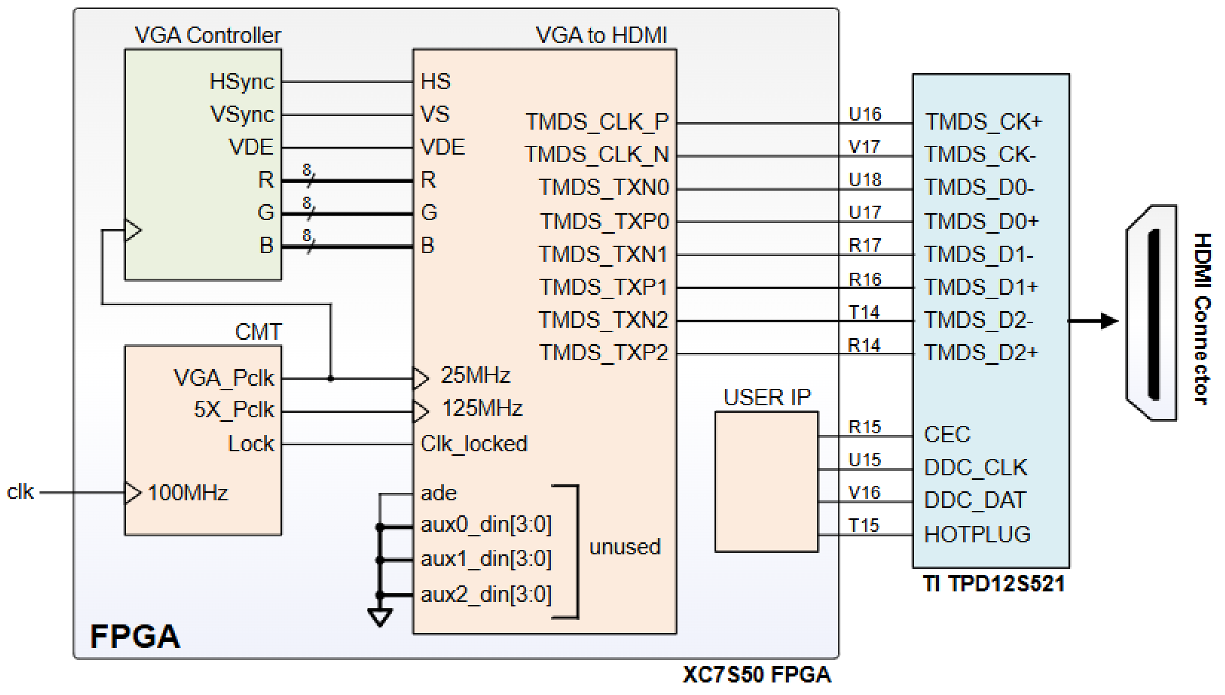


Figure 18. HDMI controller top-level block diagram

Audio output

The tetris theme module generates audio signals to play the tetris theme song on the Urbana boards audio output. We achieve this by creating a digital waveform using pulse width modulation, we cycle through a series of musical notes, stored in a ROM file. We have a 6-bit running counter "q", that runs with the input clock at about

100 MHz, we take the ones bit position q[1] to create a clock signal of 25MHz. This 25MHz clock signal is used to increment a 31-bit counter "tone", at every positive edge of that clock signal. We only use the upper bits of tone (tone[29:22]) to serve as our address pointer to the music ROM, because the lower bits of tone flip way too fast. We use these specific bits to control the timing of the note, tone[22] flips every 2^{22} counts, which in real time would be about 0.168 seconds per note, based on the 25MHz clock ($\frac{25MHz}{2^{22}} = 0.168s$). The upper 8 bits of tone are used to address the music ROM file. The contents of the music ROM file itself are from a publicly available project, that used this music ROM data on an Arduino device, outputted to a piezo buzzer. The music notes range in value from 0 to 39, with 0 being silent. Besides 0, the other notes range from 27 to 39. For the music ROM file, the ROM gives us a note number, and these numbers are mapped to musical notes, and gives us both the actual note, and the octave, which is used here to determine the pitch of the characters. To get the note and octave from the 6-bit value in the music ROM, we need to divide it by 12 in order to retrieve the octave. To get the note, we take the remainder of the value in music ROM and 12. For example, the first note of the music ROM is 34, which is 6'b100010. Dividing this by 12, we find the octave to be 2. Taking the reminder, we get a value of 10, which maps to the note 'G'. The music ROM encodes the information of the octave and note into a single value, this is done as encoded note = 12 * Octave + note.

In western music, an octave consists of 12 distinct notes. Each octave repeats this pattern, for example, Middle C and high C are one octave apart. When the information of the note is put into a single value such as in the music ROM, dividing by 12 tells us how many complete octaves the note has passed, while taking the remainder 12 tells us the specific note within the octave. Back to the example with 34, we found the note to be 12, which is equivalent to G, and the octave to be 2, which is represented as G2 in music. Next, we use the note value to select a note frequency. For G, the value received is 286. This value tells us how many clock cycles we should wait for before toggling the speaker. To implement the octave, we have an extra delay, that is calculated as 255 divided by 2 to the power of the octaves value. For an octave of two, the delay is around 64, So we actually toggle every 286 * 64 cycles, or 18304 cycles. Finally, the frequency f = $\frac{Module\ clock}{2 \cdot clock\ delay} = \frac{100MHz}{36.6KHz} = 2731Hz$. This frequency is used to generate the actual audio output for the FPGA.

A generated waveform for the audio is outputted from the module into the top-level module. This signal needs to be connected to the audio output of the FPGA. From the Urbana board reference manual, the diagram below shows that the audio out consists of two different ports on the Urbana board, B13 and B14, for left and right audio respectively. We needed to connect the waveform output from the music module into both the left and right audio outputs, so we set it equal to another variable in the top-level module and set the two audio outputs to pin B13 and B14 in the constraints file. The audio should simply work by connecting an audio device via aux cord to the audio out onto the FPGA. For my project demonstration, I connected the Audio out of the FPGA to a pair of headphones, but any audio device, such as a speaker should work, as long as the right cable and connection is used.

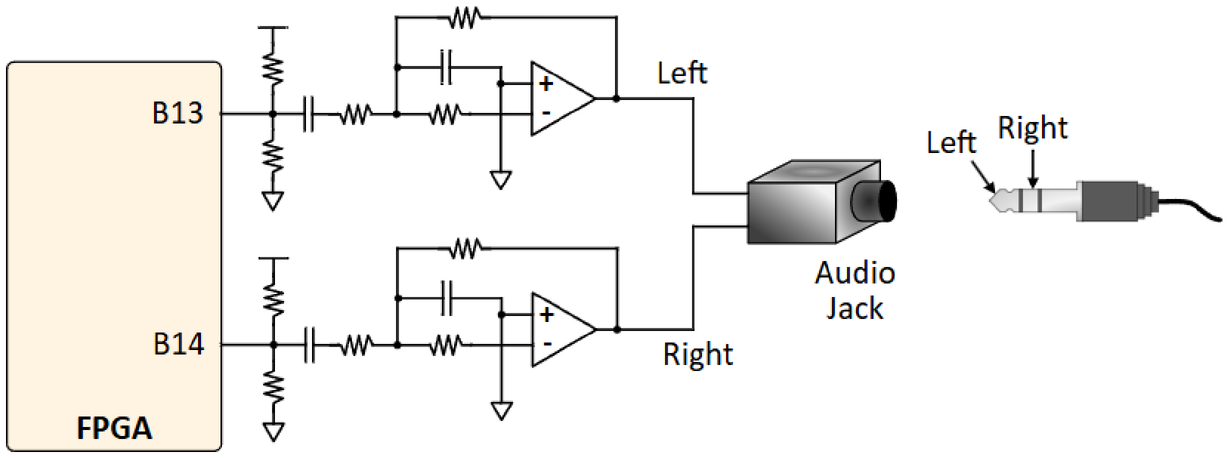


Figure 13. Urbana board audio output circuits

Module descriptions

clk_wiz_0 IP:

This clocking wizard module is used to generate clock signals for our design. For lab 6, the clocking wizard generates both a 25MHz and 125MHz clock signal.

clk_wiz_0 signals:

clk_out1: 25MHz clock signal output

clk_out2: 125MHz clock signal output

reset: Active high reset

clk_n1: input clock connected to primary clk, which is used as a reference frequency

hdmi_tx_0 IP:
This module is responsible for converting our FPGA VGA signal to HDMI signals, which can be used as output to our FPGA HDMI port.
hdmi_tx_0 signals:
pix_clk, pix_clkx5, pix_clk_locked, rst, red, green, blue, hsync, vsync, vde, TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N

Module: mb_usb_hdmi_top.sv
inputs: Clk, reset_rtl_0, gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
outputs: gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, speaker, speaker1
Description: This is the top-level module that instantiates every module used in this design. Connecting the modules including the block diagrams. The connected modules in this top-level module create four HDMI signals to be sent to the HDMI output, as well as two signals to be sent to the AUDIO out. The module also contains the addresses of the ASCII characters to be printed onto the display.
purpose: Instantiate game logic and connect multiple modules, IPs, and MicroBlaze USB block diagram.

Module: Block.sv
inputs: Reset, frame_clk, [7:0] keycode
outputs: gameover_flag, [4:0] score, [2:0] grid[10][20]
Description: This module contains the logic for the tetris blocks, including their size, movement, and behavior. The game play area is expressed as a 10x20 grid, Which the value in the grid determines which type of block is present. This module determines most of the game's behavior, and the values of the blocks are sent as outputs to the color mapper. The module also takes keycodes from our USB input to determine what the tetris block should do next.
purpose: To instantiate game behavior, as well as the behavior and size of the blocks.

Module: Color_mapper.sv
inputs: [6:0] lines_text[13], [6:0] welcome_text[17], [6:0] gameover_text[8], [2:0] grid [10][20], [9:0] DrawX, [9:0] DrawY, [4:0] score, gameover_indicator
outputs: [3:0] Red, [3:0] Green, [3:0] Blue

Description: This module contains the logic to determine the colors of what is printed onto the screen. This module takes the values of text arrays that contain the hex values of ASCII characters to be printed onto the screen, the module contains internal logic to control where this text is printed onto the screen, and what color it should be, including the score. The module also takes the value of the grid, and prints certain colors depending on the value, with each Tetris block having a different value and color. The positions of objects are determined using DrawX and DrawY.
purpose: To determine the color of the current pixel being printed, as well as the position of text to be printed onto the screen.

Module: TetrisTheme.sv
inputs: clk, reset
outputs: speaker
Description: This module is used to generate the tetris theme, using a counter, it reads the music notes from a music ROM that's in the correct order for the tetris theme. We then decode the ROM output for the correct octave and note of the current note to be played, and we generate a frequency to be sent to to the audio out.
purpose: To create an audio output that sounds similar to the original Tetris theme, to be sent as an output to the AUDIO OUT of the Urbana board.

module: VGA_controller.sv
inputs: pixel_clk, reset
outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: The VGA module is a provided module. This module is used to generate VGA video signals, as well as track pixel and line positions. The module manages the drawing coordinates on the display, using DrawX and DrawY as outputs. Horizontal and vertical sync are used as display frequencies.
Purpose: This module created pixel coordinates and VGA signals for controlling the display.

Module: font_rom.sv
inputs: [10:0] addr
outputs: [7:0] data
Description: This module contains the ROM data for ASCII characters. Each character has a width of 8 pixels and a length of 16 pixels. The module takes the

address as an input, which selects a certain row of data in a character. the module then will output an 8-bit wide value of that current row.
Purpose: module is used to print text characters onto the screen in a simple way.

Components of Block Diagram:

mb_block module:

This module instantiates our MicroBlaze processor and contains all its peripherals for the microcontroller preset. This module also contains IP's we want to connect to the microblaze, like GPIO's.

mb_block signals:

clk_100MHz, gpio_usb_keycode_0, gpio_usb_keycode1, gpio_usb_int, gpio_usb_rst, reset_rt1_0, uart_rtl_0, usb_spi_mosi, usb_spi_sclk, usb_spi_ss

IP's within mb_block:

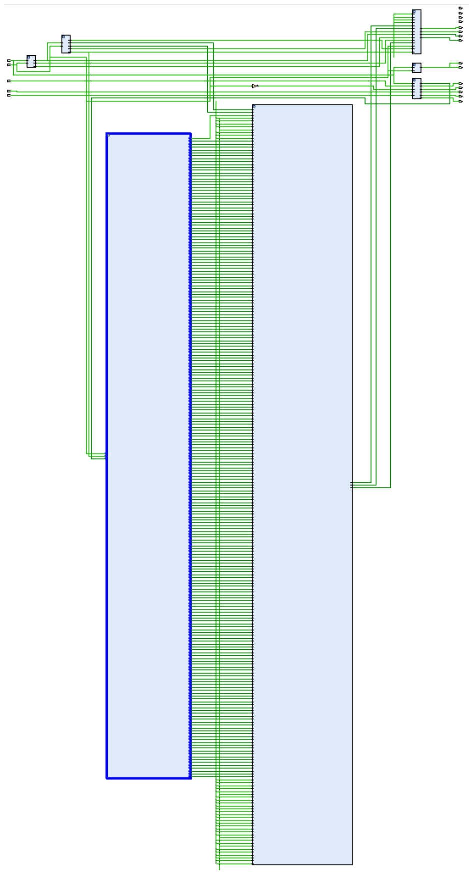
AXI Uartlite:
This IP is used to set up asynchronous serial communication between the MicroBlaze and our serial monitor in Vitis IDE.

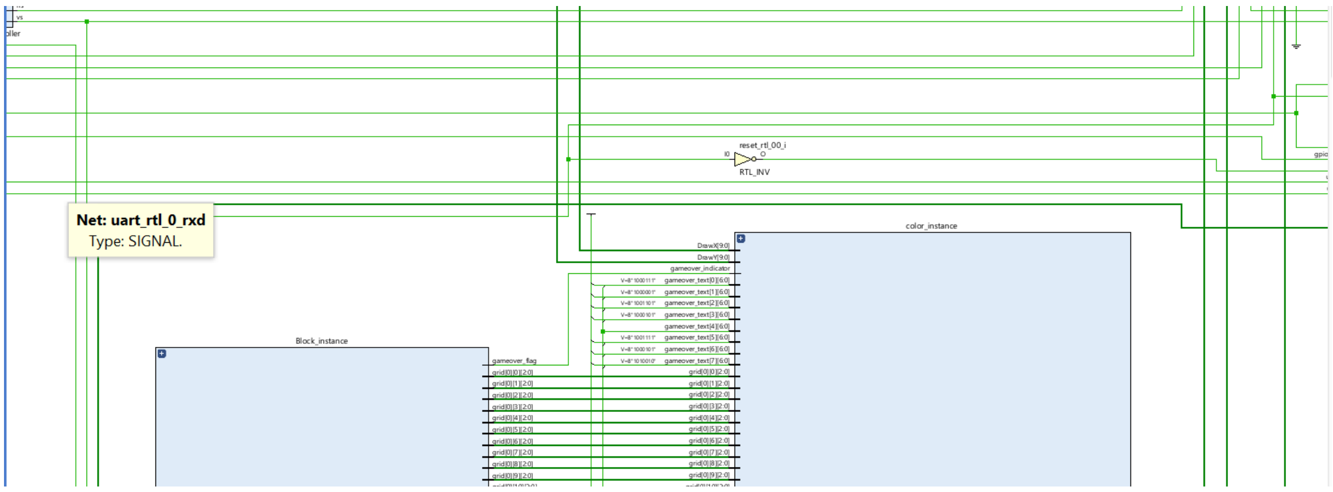
AXI GPIO:
GPIO stands for "General purpose input output". This IP allows our MicroBlaze processor to interact with peripherals, by either receiving inputs, or sending signals to outputs. For week 1, we used the GPIO IP to blink an LED, and for the microcontroller to receive data from our FPGA switches. For week 2, they are used for USB keyboard communication.

AXI TIMER:
This IP is used to set up asynchronous serial communication between the MicroBlaze and our serial monitor in Vitis IDE.

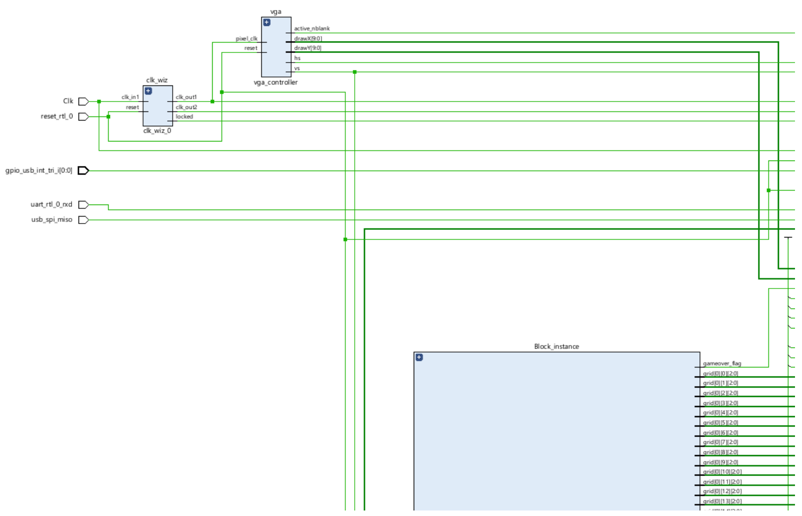
AXI Quad SPI:
This IP is used to set up a SPI interface between the MicroBlaze and slave devices. Its four signals, MOSI (Master out slave in), MISO (master in slave out), serial clock, and slave select, are used for communication between master and its slave devices. In this case the master device is the MicroBlaze processor.

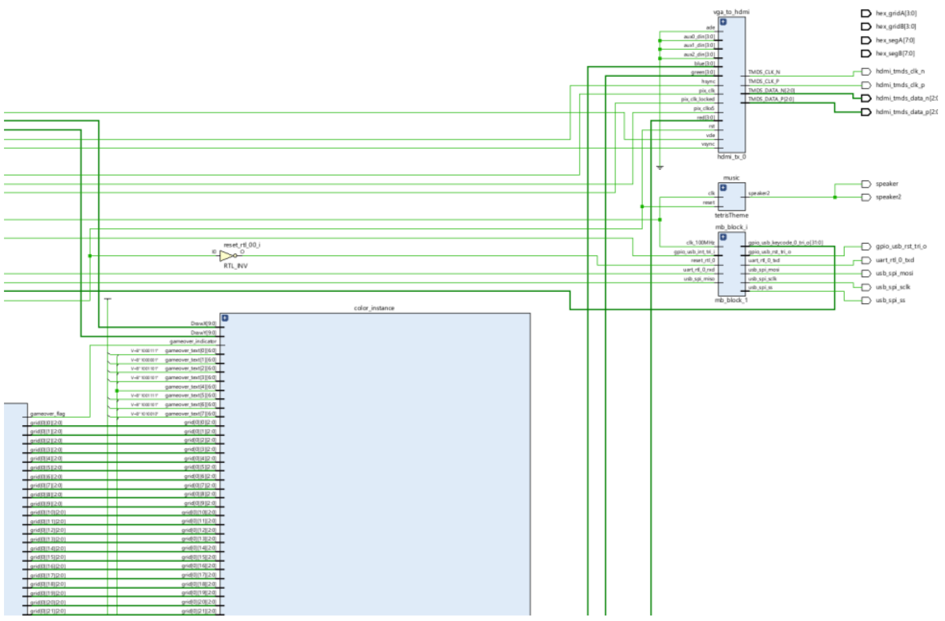
RTL Diagrams





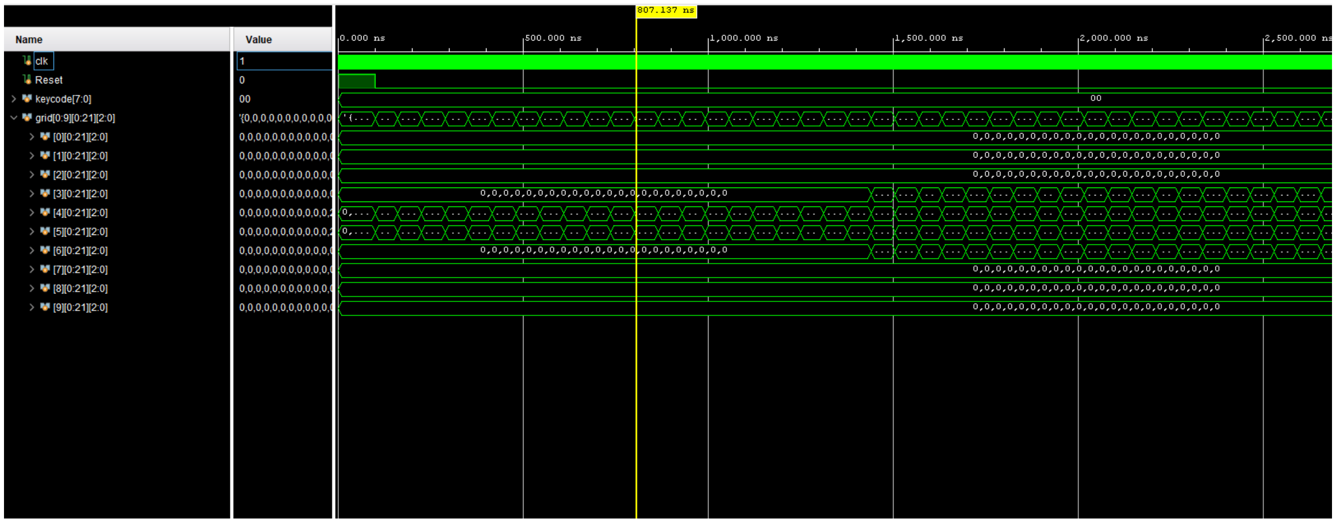
vs
oller
reset_rtl_00_i
I0
O
RTL_INV
Net: uart_rtl_0_rxd
Type: SIGNAL
color_instance
DrawX[9:0]
DrawY[9:0]
gameover_indicator
V=8'h1000111' gameover_text[0][6:0]
V=8'h1000001' gameover_text[1][6:0]
V=8'h1001101' gameover_text[2][6:0]
V=8'h1000101' gameover_text[3][6:0]
gameover_text[4][6:0]
V=8'h1001111' gameover_text[5][6:0]
V=8'h1000101' gameover_text[6][6:0]
V=8'h1010010' gameover_text[7][6:0]
Block_instance
gameover_flag
grid[0][0][2:0]
grid[0][1][2:0]
grid[0][2][2:0]
grid[0][3][2:0]
grid[0][4][2:0]
grid[0][5][2:0]
grid[0][6][2:0]
grid[0][7][2:0]
grid[0][8][2:0]
grid[0][9][2:0]
gpio



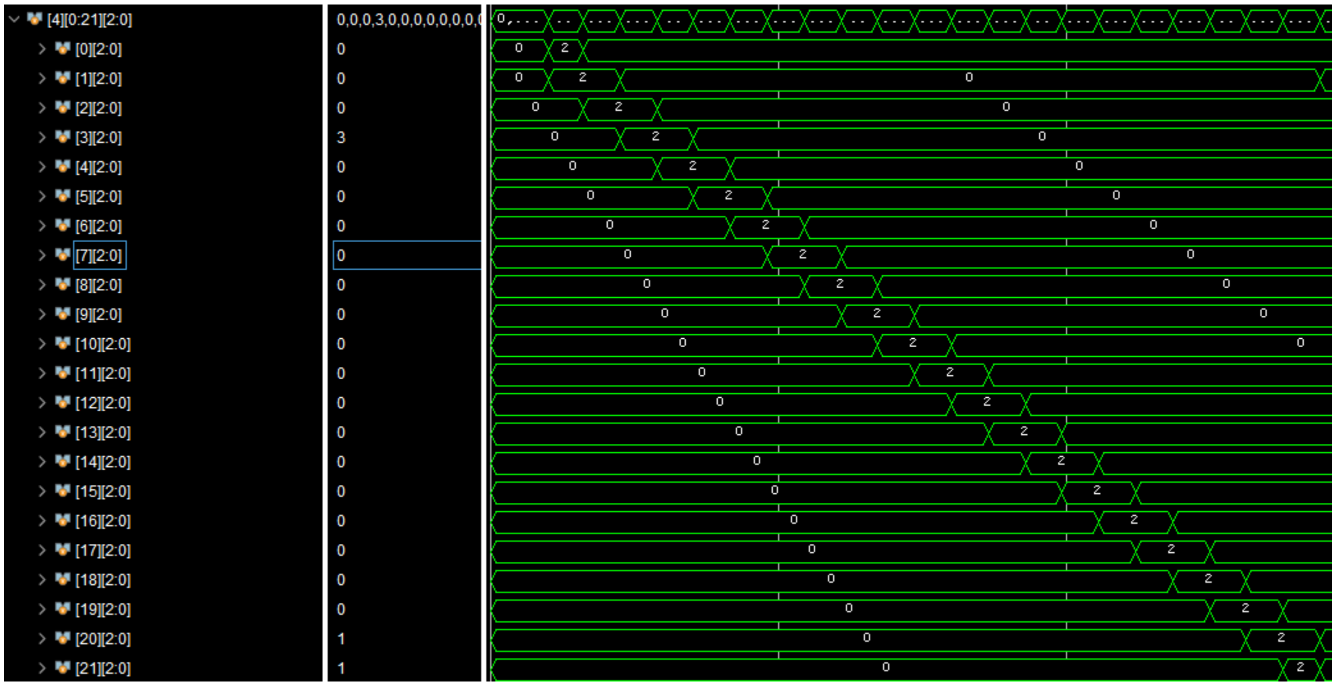


Simulated waveforms

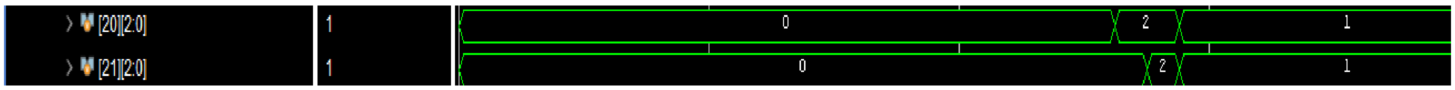
Simulated grid no keycode:



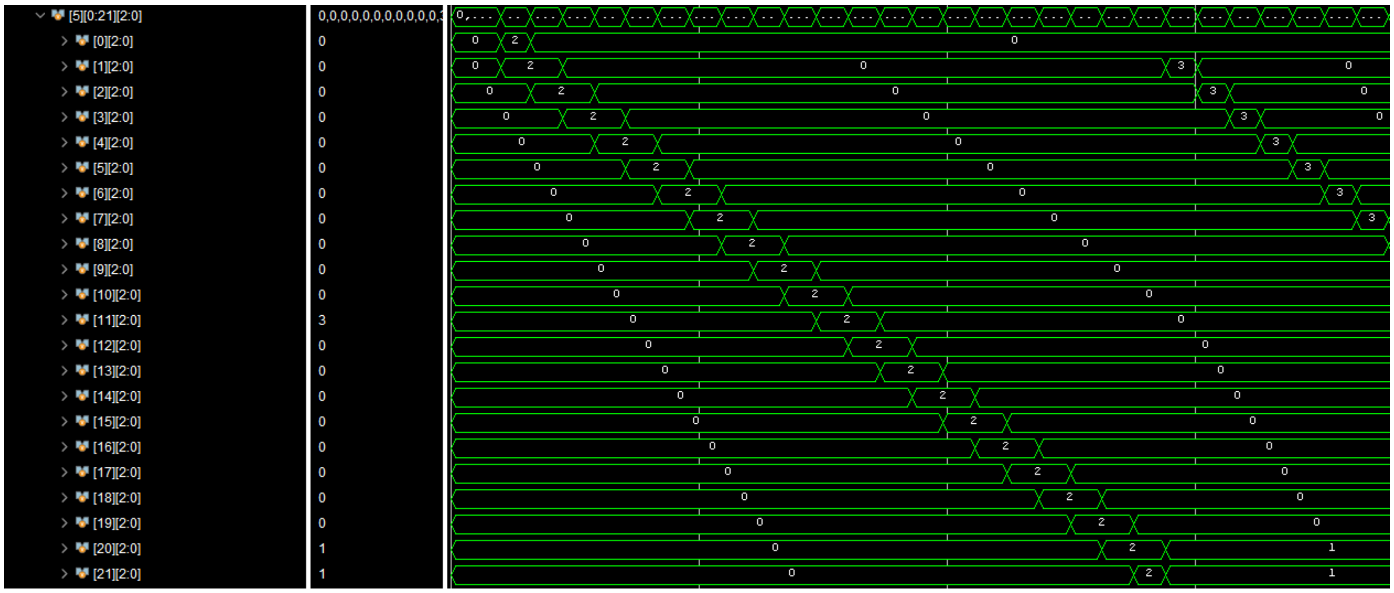
The full simulation of the game without input shows the grid falling in the y direction due to the game's gravity logic. With no input, the block does not change direction horizontally. The simulation can show each individual column, and their respective row values. Since we are starting with the square block, we know that columns 4 and 5 should be the only columns with changing values, until that block is placed.



The above simulation image shows column 4, and how it changes as the square block is falling. We can see that the block starts at the top two rows where it is spawned in, and progressively falls towards the bottom two rows, the block will occupy two rows at a time due to its height.

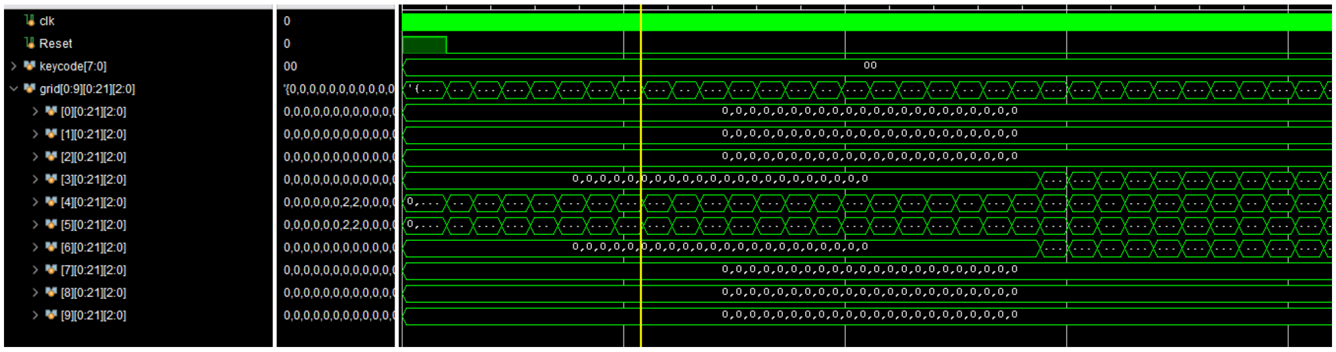


Once the block is settled at the bottom two rows, the block is placed and changed into a" settled" block which can no longer be moved, only cleared. We see this in the simulation, as column 4 rows 20 and 21 take on the value of 1 after the square block is placed.

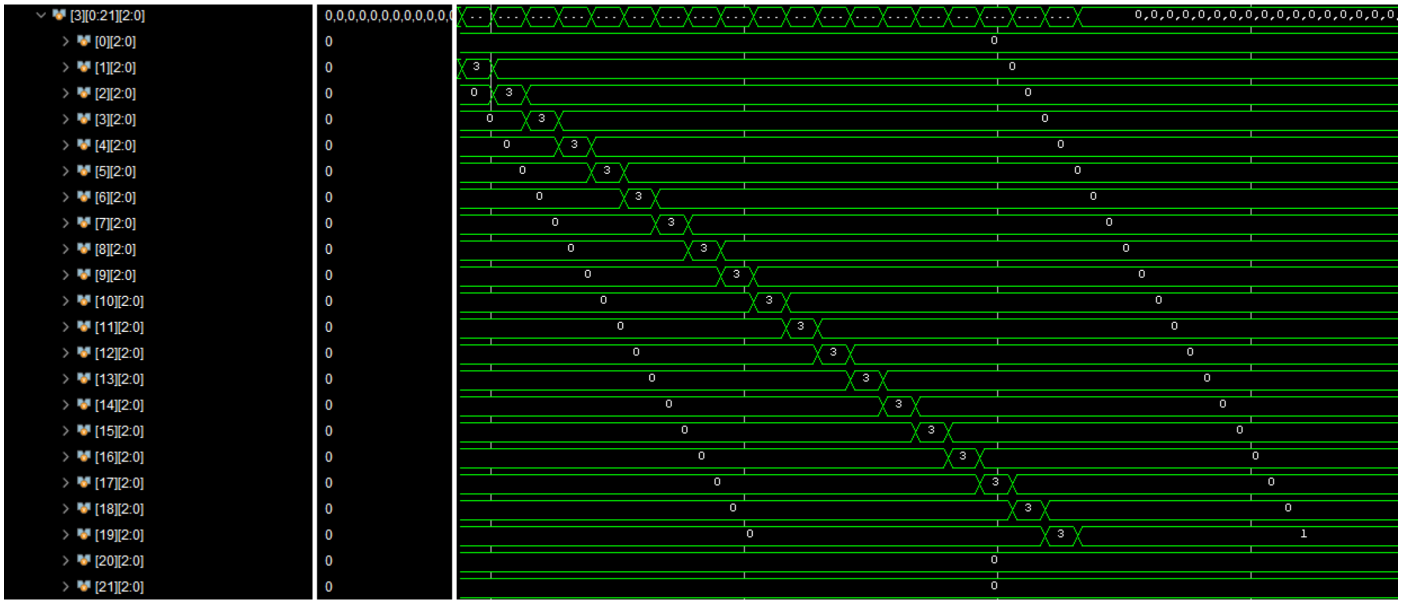


Column 5 shown above goes through the exact same process, because the square block is the same length in both columns. The block also gets placed in the same spot, shown by the 1s. The 3 in the simulation shows as the next block is generated.

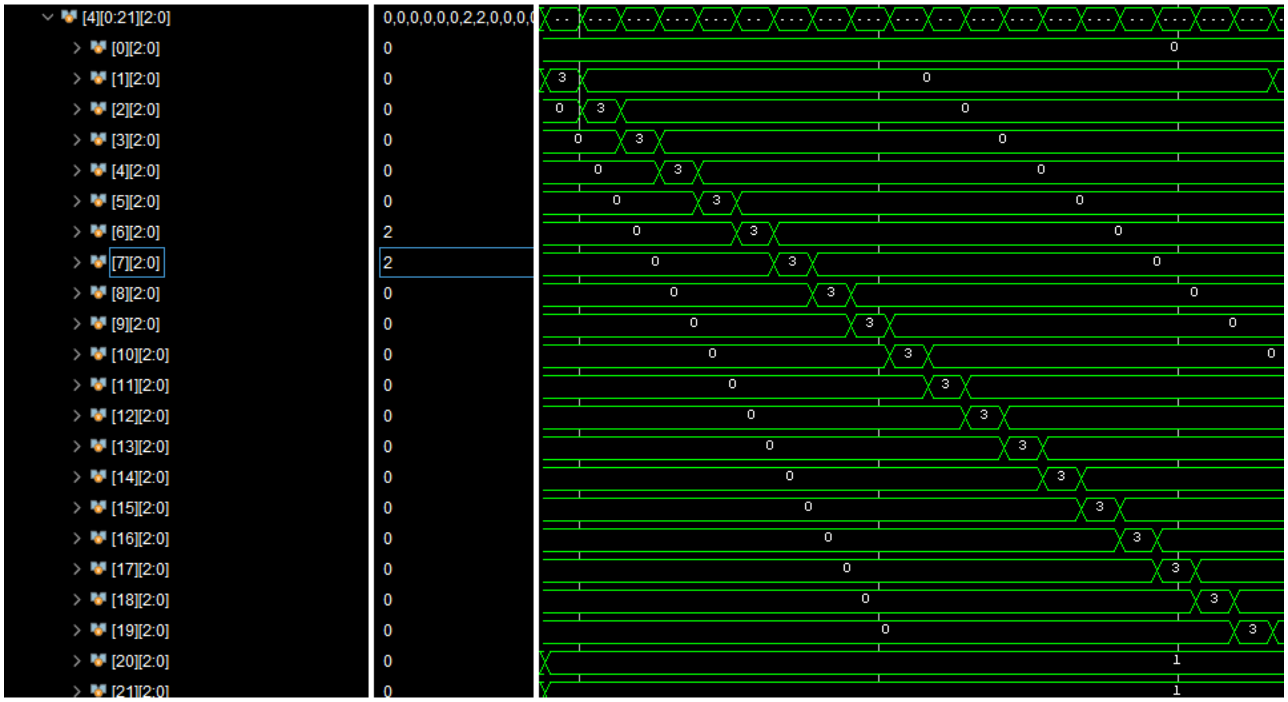
the grid value for the next block is 3, which we know to be the line block. The line block spawns in lying flat, so it should occupy 4 columns and 1 row each. This looks correct based off the simulation.



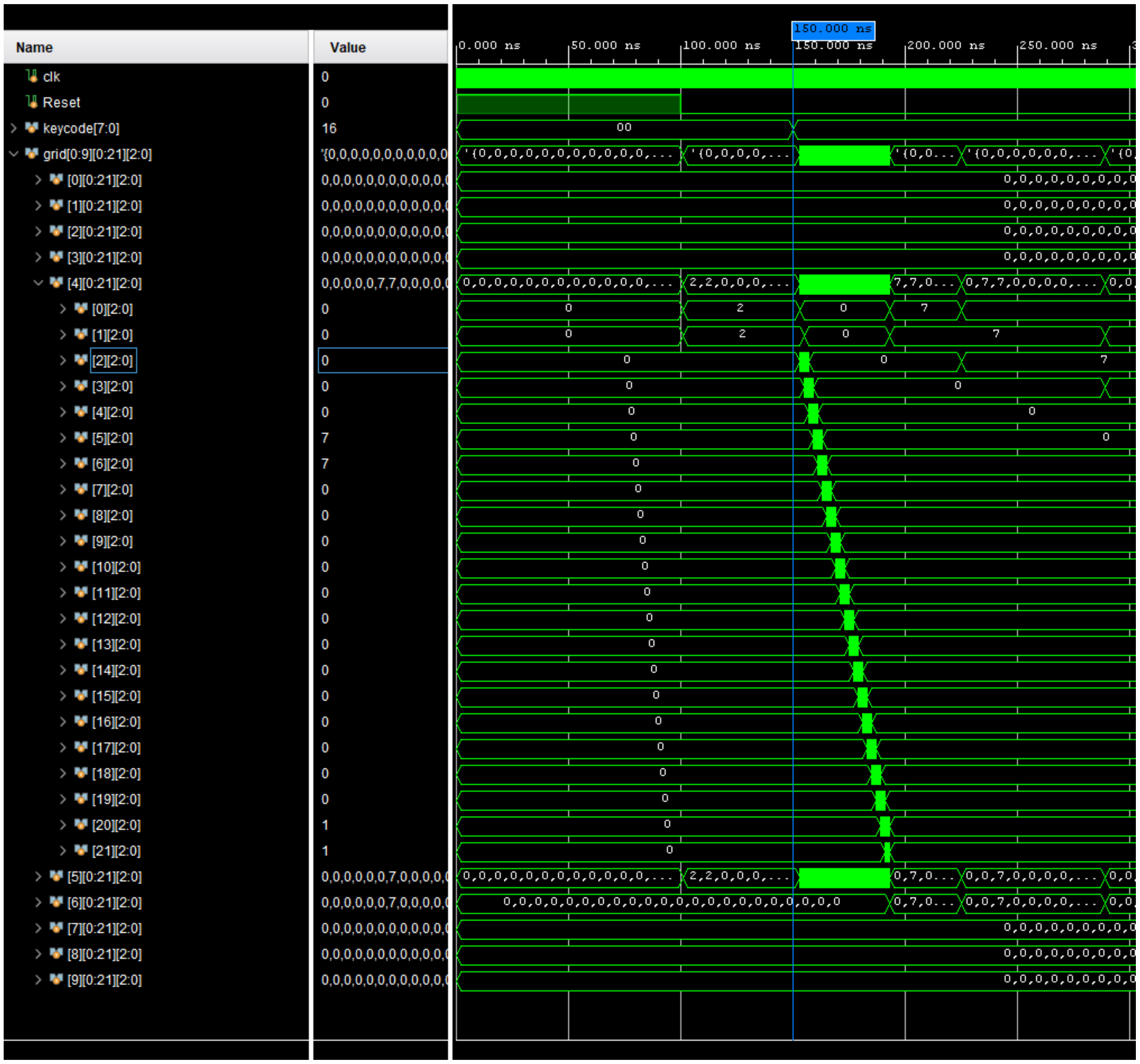
From the simulation above, we can see that the four columns the line block occupies when in its default position have values other than zero, meaning one of their rows is occupied by a block.



Column 3 for example, shows the portion of the line block in column 3 progressively falls, and finally settles in row 19. The line block settles in row 19 rather than lower because rows 20 and 21 are already occupied by the square block that settled earlier. We see that the line block does not occupy more than one row in any column, because the line block is lying flat.

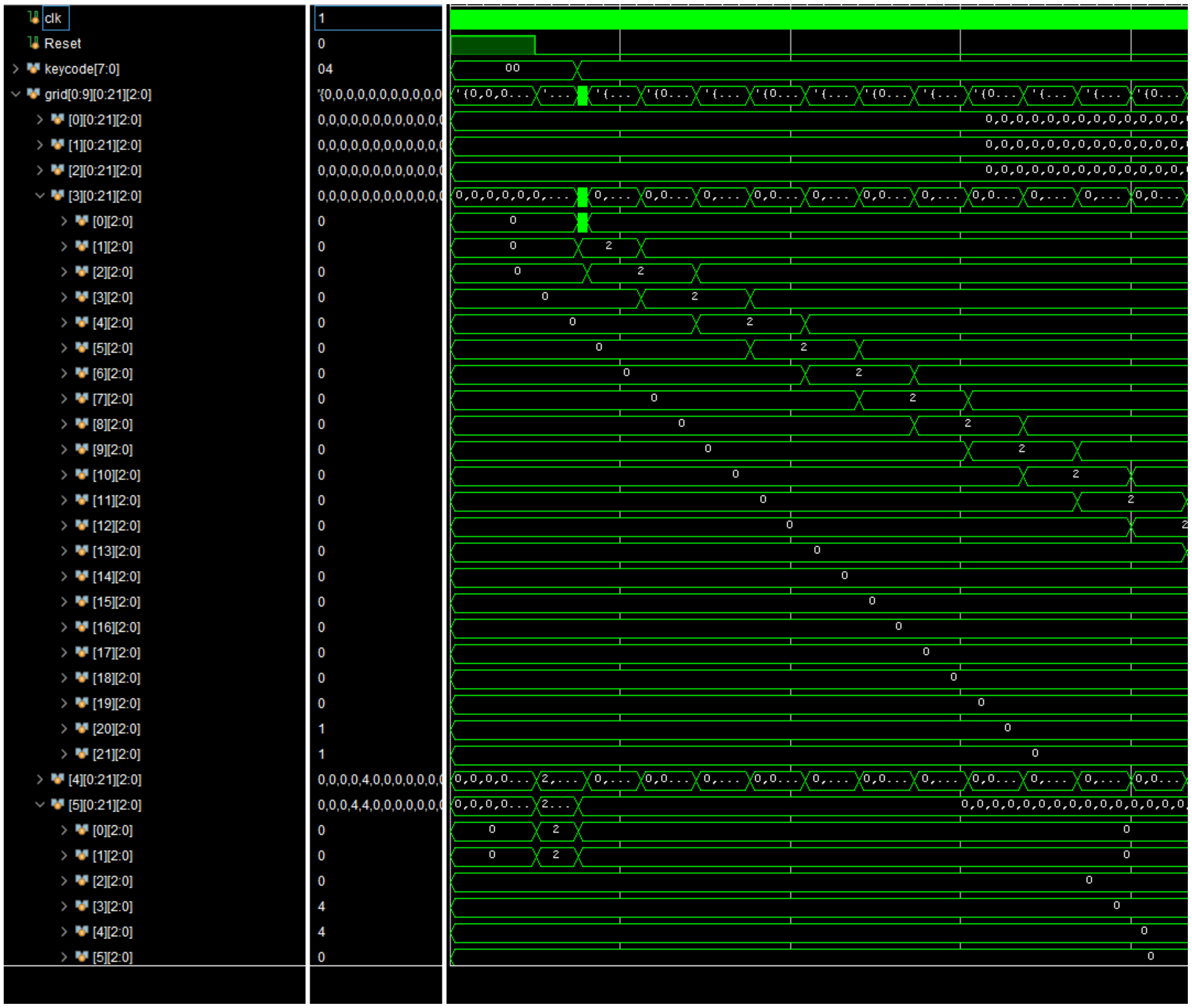


Column 4 appears to be identical to column 3, as well as the other two columns occupied by the line block. This just simulates the block falling, but we can also simulate the USB inputs using the keycodes.



Name	Value
clk	0
Reset	0
keycode[7:0]	16
grid[0:9][0:21][2:0]	'{0,0,0,0,0,0,0,0,0,0,0
[0][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[1][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[2][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[3][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[4][0:21][2:0]	0,0,0,0,0,7,7,0,0,0,0,0
[0][2:0]	0
[1][2:0]	0
[2][2:0]	0
[3][2:0]	0
[4][2:0]	0
[5][2:0]	7
[6][2:0]	7
[7][2:0]	0
[8][2:0]	0
[9][2:0]	0
[10][2:0]	0
[11][2:0]	0
[12][2:0]	0
[13][2:0]	0
[14][2:0]	0
[15][2:0]	0
[16][2:0]	0
[17][2:0]	0
[18][2:0]	0
[19][2:0]	0
[20][2:0]	1
[21][2:0]	1
[5][0:21][2:0]	0,0,0,0,0,0,7,0,0,0,0,0
[6][0:21][2:0]	0,0,0,0,0,0,7,0,0,0,0,0
[7][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[8][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[9][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0

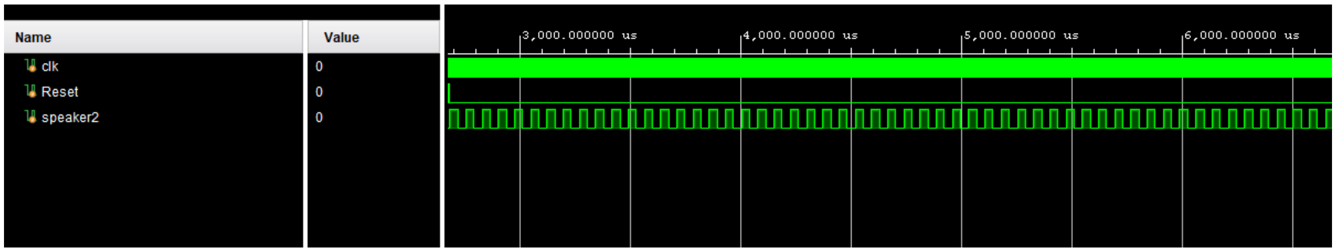
This is a simulated drop of the square block. We can see around the blue marker, at 150ns, we input the value of 16 into the keycode, which corresponds to the ‘s’ key. This key is used to drop the block quickly, as it goes through all the rows as quickly as possible. We can see around the same time at the same marker, the square block travels through the rows very quickly, much faster than if we had not pressed any key.



clk	1
Reset	0
keycode[7:0]	04
grid[0:9][0:21][2:0]	'{0,0,0,0,0,0,0,0,0,0,0
[0][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[1][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[2][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[3][0:21][2:0]	0,0,0,0,0,0,0,0,0,0,0,0
[0][2:0]	0
[1][2:0]	0
[2][2:0]	0
[3][2:0]	0
[4][2:0]	0
[5][2:0]	0
[6][2:0]	0
[7][2:0]	0
[8][2:0]	0
[9][2:0]	0
[10][2:0]	0
[11][2:0]	0
[12][2:0]	0
[13][2:0]	0
[14][2:0]	0
[15][2:0]	0
[16][2:0]	0
[17][2:0]	0
[18][2:0]	0
[19][2:0]	0
[20][2:0]	1
[21][2:0]	1
[4][0:21][2:0]	0,0,0,0,4,0,0,0,0,0,0,0
[5][0:21][2:0]	0,0,0,4,4,0,0,0,0,0,0,0
[0][2:0]	0
[1][2:0]	0
[2][2:0]	0
[3][2:0]	4
[4][2:0]	4
[5][2:0]	0

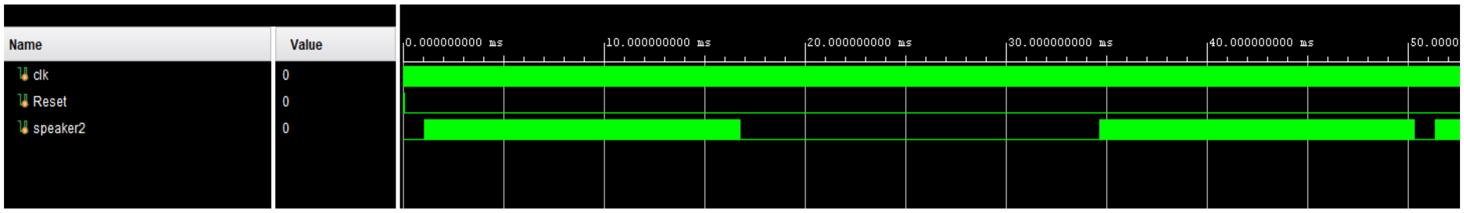
In this waveform, we simulated the press of the 'a' key, which causes the block to shift to the left. We can see from this simulation, that column 3, which previously did not hold any value from the square block, has the leftmost part of the square block shifted into its top 2 rows. Also seen from the simulation, is that the rows from column 5 now contain zero values, as we have shifted the block left, so the right part of the square block is now in column 4 instead of column 5.

Audio simulation



Name	Value
clk	0
Reset	0
speaker2	0

The audio algorithm consists of square waves as well as pauses between subsequent notes, these can be shown when zooming out on the waveform. The pauses are due to portions in our music ROM file in between notes that have zero values.



Design Analysis

LUT	**29978**
DSP	**3**
Memory (BRAM)	**32**
Flip-Flop	**3485**
Latches	**0**
Frequency	**119.7MHz**
Static power	**0.075W**
Dynamic power	**0.661W**
Total power	**0.736W**

Conclusion

Functionality of Design

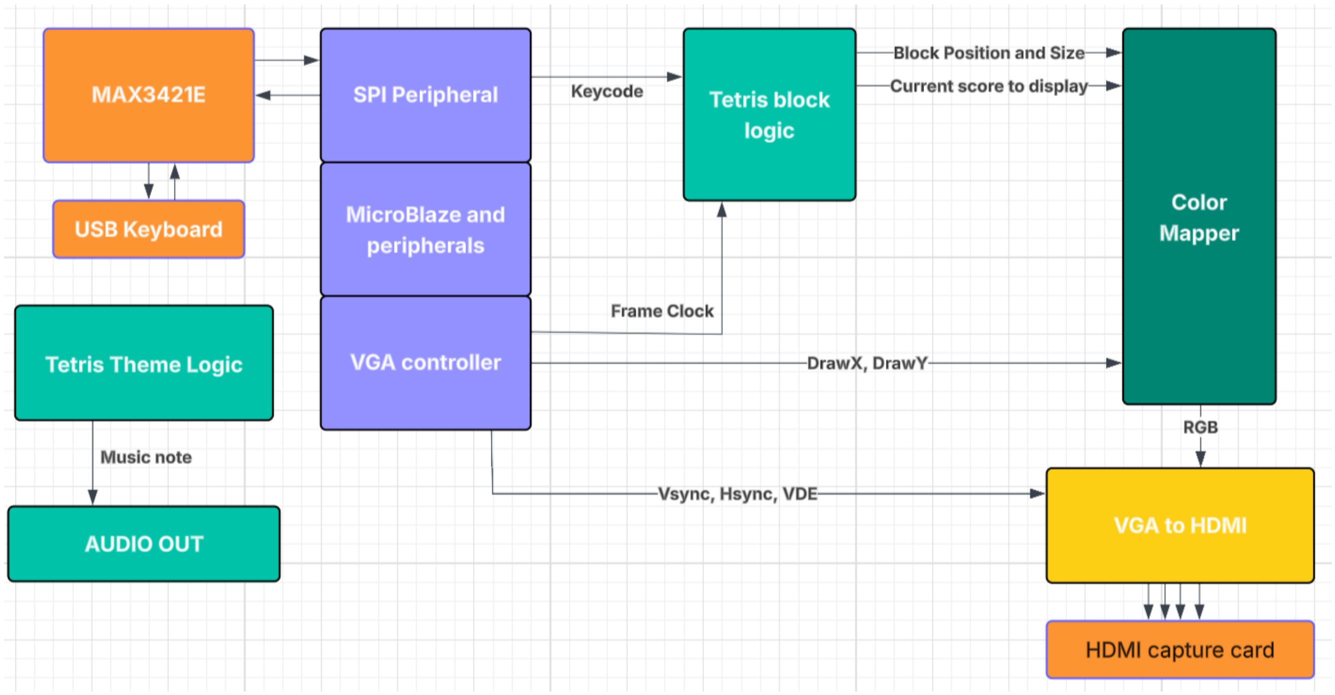
Implementation of the main functionalities of this game had been a success, as well as incorporating audio and on-screen score tracking as additional features. Although, we did run into a few issues in the game implementation. For one, this tetris game does not feature all 7 tetris blocks, as we are missing the T block. The reason for this is with how I was designing the grids, the size of our FPGA was the limiting factor. Our original design had the 20x22 grid array be 3 bits wide, so we can have a value between 0 and 7. This was all taken up, with 0 being used to express an empty grid, 1 being a placed block, and 2 through 7 being used to express different tetris blocks, we did not have room to add the additional T block. To add it, we would need to extend the 20x22 grid array to be 4 bits wide. This caused the implementation to fail, as now we had run into a size issue. The design quickly exceeded the number of LUT's that were available on the chip, as now we needed at least 40,000 LUT's for this design. To fix this problem, we would need to rethink the entire design of the play area. One possible solution is to not be expressed a place or “settled” block as a different grid value. In our current design, when a tetris block is placed, the new grid value changes from the block value to a value of 1, greying out the block as its now placed. Instead of doing this, we could forget about greying out the block, and just have it placed while keeping its original block value, we would just need to rework the logic in order to ensure that these blocks function the same as the placed blocks. This frees up 1 extra bit in the 10x22 grid array, allowing us to assign this to a T block instead using it for placed blocks. Another design issue is random but infrequent glitches of a generated block, where it will randomly have a piece missing from the grid, or a piece of the grid in a random spot. I found the problem to be pressing the w key during block generation, as the w key is used to rotate the block. I reproduced it by spamming the w key after the previous block is placed, however it still does not occur frequently, and I had not noticed it occurring otherwise.

For potential changes in the future, I would like to make the game more visually appealing. The game appears rather dull, since we only used the color mapper and combinational logic to print colors onto the screen with the VGA controller, and the text looking plain since it only came from the font ROM. In the future, I want to use a frame buffer to store sprites, as this will allow me to display more complex and visually appearing images and text.

Summary

Our implementation of Tetris on the FPGA was well designed and functioned properly for the most part, as well as having additional features such as music audio and on-screen score tracking. The game features USB input through the MicroBlaze, that interacts with a USB keyboard through the MAX3421E SPI protocol. The game features a display through the HDMI output, which is generated with combinational logic and the VGA controller, that sends color data to the VGA to HDMI converter IP to generate HDMI signals. We also create music audio which sounds very similar to the original Tetris theme, and we do this by using combinational logic to decode a music ROM to create signals sent to the audio out. I enjoyed this final project, I was able to apply all the topics I had learned throughout the semester, as well as new concepts such as creating audio using combinational logic. I hope to take advantage of what I learned during this course to work on

Design Block Diagram


MAX3421E
SPI Peripheral
Keycode
Tetris block logic
Block Position and Size
Current score to display
Color Mapper
USB Keyboard
MicroBlaze and peripherals
Frame Clock
Tetris Theme Logic
VGA controller
DrawX, DrawY
RGB
Music note
Vsync, Hsync, VDE
VGA to HDMI
AUDIO OUT
HDMI capture card