c

ECE 385

Fall 2023

Final Project

Implementation of tetris using Microblaze in Systemverilog

Cher Rui Tan, Adithya Balaji
Section: Friday 11.30am
TA: Gene Lee

Introduction

In lab 6, we set up a microblaze and added a USB host controller, USB keyboard (MAX3241E) and VGA to the HDMI module to be able to output to a screen. This final project builds on that basis to build the game of tetris on Vivado using SystemVerilog.

We use a Microblaze device with HDMI peripherals setup as in Lab 6, and implement the game of tetris with basic functionality of moving blocks, bounds checking and collision detection, as well as additional features of rotation, rotation collision checking, as well as start/end blue screen of death and score counter based on number of rows cleared.

The user will be able to enter the game by pressing the spacebar, then subsequently play using the A and D keys for left and right movement as well as the Q and E keys for counterclockwise and clockwise rotation respectively.

Written description and diagrams of Microblaze System

*****Note that the descriptions are exactly the same as in lab 6, but with additional modules required for the game logic of tetris.

Written description of all SV module

Module: mb_usb_hdmi_top.sv:

Inputs & Outputs:

```
input logic Clk,
input logic reset_rtl_0,

//USB signals
input logic [0:0] gpio_usb_int_tri_i,
output logic gpio_usb_rst_tri_o,
input logic usb_spi_miso,
output logic usb_spi_mosi,
output logic usb_spi_sclk,
output logic usb_spi_ss,

//UART
input logic uart_rtl_0_rxd,
output logic uart_rtl_0_txd,

//HDMI
output logic hdmi_tmds_clk_n,
output logic hdmi_tmds_clk_p,
output logic [2:0]hdmi_tmds_data_n,
output logic [2:0]hdmi_tmds_data_p,
```

```
//HEX displays
output logic [7:0] hex_segA,
output logic [3:0] hex_gridA,
output logic [7:0] hex_segB,
output logic [3:0] hex_gridB
```

Description: Top level module which instantiates USB signals, HDMI, UART and HEX displays. Does this by instantiating a clocking wizard, VGA sync signal generator, VGA to HDMI converter, as well as the ball module, gridmapper, color mapper, randnumgen, rotate, gamefsm, updategridarray and movementchecker. Furthermore, it contains a mux which decides input between the random gen and the rotation function.

Purpose: Used as a top level to instantiate and link all the relevant modules to one another

Module: hex.sv:

```
Inputs:     input  logic       clk,
            input  logic       reset,
            input  logic [3:0] in[4],

Output:     output logic [7:0] hex_seg,
            output logic [3:0] hex_grid
```

Description: Implements a 4-digit hexadecimal display driver by using a counter to cycle through 4 digits, and selecting the appropriate pattern based on the input nibbles. Also contains a module within for nibble to hex.

Purpose: To display hexadecimal numbers on the 4-digit 7-segment display

Module: nibble_to_hex (.sv)

```
Inputs:     input  logic [3:0] nibble,
Outputs:    output logic [7:0] hex
```

Description: Takes asynchronous signals and holds them in a flip flop allowing for them to be implemented on a clock signal. This results in the signals being converted from asynchronous to synchronous signals.

Purpose: To eliminate asynchronous components such that the entire circuit remains synchronous to reduce the chance of static hazards arising from timing issues and ensure proper signal handling in FPGA designs.

Module: vga_controller (.sv)

Inputs: pixel_clk,
reset

Outputs: hs, vs, active_nblank, sync, [9:0]drawX, [9:0]drawY

Description: Generates VGA signals to drive display. It manages horizontal and vertical synchronization pulses, provides pixel coordinates for rendering graphics, and controls pixel display. The module uses a 50 MHz clock and includes 10-bit counters to track pixel positions. It also adjusts the timing to match the 25 MHz pixel clock and ensures that only valid pixels are displayed.

Purpose: To output the VGA signal to be used in the display

Module: ball (.sv)

Inputs: input logic Reset, frame_clk,
input logic [7:0] keycode,

Outputs: output logic [9:0] BallX, BallY, BallS

Description: Controls the motion of the ball on VGA display. Handles user input for moving the ball up, down, left, or right, by checking which key (W,A,S,D) is pressed and its hex character. Ball's position is updated based on its current coordinates and boundary limits. The conditions include state values from the FSM as well as movement checking to ensure that the Ball only moves to valid positions
Purpose: Control movement of ball on screen.

Module: mb_intro_top (.sv)

Inputs: input logic clk,
input logic [3:0] btn,
input logic uart_txd,
input logic [15:0] sw);

Outputs: output logic [15:0] led,
output logic uart_rxd,

Description: Interfaces the processor with LEDs, buttons, UART, and switches. Connects signals between MicroBlaze and external peripherals

Purpose: Replacement block diagram wrapper file and top level for use with ECE 385

Module: color_mapper(.sv)

Inputs: input logic [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
input logic grid,
input logic [15:0] blockgrid,
input logic [1:0] arraygrid[25:0][11:0],
input logic blockexist,
input logic die,
input logic startGame,

Outputs: output logic [3:0] Red, Green, Blue);

Description: Contains a RGB color code for the blocks, grid, block on the grid and start and end screen. There is also decoder for blockgrid which uses the BallX and BallY center and offsets it to color in the specific tetris shape.

Purpose: To assign colors to the pixels being drawn based on the position of objects from other modules.

Module: randnumgen(.sv)

Inputs: input logic Clk,
input logic reset,

Outputs: output logic [15:0] blockgrid
);

Description: Contains a 16 bit shift register which XNORs the 13th bit and the 15th bit together every cycle. It also contains an initial seed which triggers on reset. It also contains a mux which selects blockgrid based on the last 3 bits of the LFSR.

Purpose: To acts as a linear feedback shift register, which creates a random number every cycle of the clock. This is then fed into the mux to determine the random block which is instantiated whenever the state is in Newblock.

Module: rotation(.sv)

Input logic: input logic Clk,
input logic Reset,
input logic [7:0] keycode,
input logic [15:0] blockgrid,
input logic canrotate,

Outputs: output logic [15:0] blockgridrotate

Description: Contains a mux for clockwise and anticlockwise rotation of shape.

Purpose: Takes in the input of the current block and outputs the encoding for the rotated block if the rotation was deemed possible by the movementchecker module.

Module: gamefsm

Inputs: input logic Clk,
input logic Reset,
input logic [7:0] keycode,
input logic canmovedown,

Outputs: output logic startGame,
output logic newblock,
output logic blockcheck,
output logic rowcheck,
output logic updategrid

Description: Contains a moore state machine, encoding the basic game functionality states, startGame, newblock, blockcheck and updategrid.

Purpose: To ensure that operations happen sequentially such that they do not interact with the same element at the same time. This is especially important for updategrid and rowcheck, as if they occur on the same clock cycle, there might be errors as a result.

Module: updatearraygrid

Inputs: input logic Clk,
input logic updategrid,
input logic rowcheck,
input logic startGame,
input logic [9:0] gridrow,
input logic [9:0] gridcol,
input logic [15:0] blockgrid,
input logic [9:0] DrawX, DrawY,

Outputs: output logic die,
output logic [1:0] arraygrid [25:0][11:0],
output logic blockexist,
output logic [15:0] score

Description: Contains the 26x12 array in row major order consisting of 2 bit registers.

This acts as a register memory space for the tracking of blocks in a game cycle.

Purpose: When the updategrid state is entered after the block has stopped. The center position of BallX and BallY as well as the blockgrid encoding is used to determine the position of the block and what squares it has filled. This is then updated as a 1 in the register array.

Module: movementchecker

Inputs: input Clk,
input logic [15:0] blockgrid,
input logic [9:0] BallX, BallY,
input logic [7:0] keycode,
input logic [1:0] arraygrid[25:0][11:0],

Outputs: output logic canmoveleft, canmoveright, canmovedown, canrotate,
output logic [9:0] gridrow, gridcol

Description: Creates a virtual block in a future movement position for down, left, right as well as rotate and compares it to the current register array to see if it overlaps with any filled blocks. Then returns if the movement is possible.

Purpose: To determine if a particular value of movement is possible.

Written description of modules created for block design

Microblaze

Creates a basic microblaze block diagram with its associated peripherals, with a microcontroller preset which is relevant for this low-level program.

AXI GPIO

Serves as a link between the AXI bus as well as input/output pins on the FPGA, used to read and write data from/to the FPGA. Allocate and maps memory in the Microblaze for different peripherals

AXI Uartlite

Universal asynchronous receiver transmitter, allows for asynchronous UART communication via the AXI bus, handling data transmission and reception. Used for communication with microcontrollers.

AXI Interconnect

Allows for connection and transmitting data between multiple AXI modules, by managing data traffic by handling routing and latency control. Serves to make the system more efficient and flexible.

AXI QUAD SPI

Handles high speed data transfer involving memory mapped access to external devices (in this case, the USB) but with 4 data lines instead of the usual 1. For week 2 we can see that the keycodes were obtained through the SPI then transmitted to the ball for movement decoding.

Clocking wizard
Takes a 100Mhz Clk input and outputs two clocks of 25Mhz and 125Mhz to support the TMDS portion of the VGA to HDMI converter IP.

AXI Interrupt Controller
Converts signals into a single signal to be sent to the processor

Processor Sytem Reset
Allows for enabling or disabling based on user, such as active high

Concat
Feeds concataneted interrupt signals into interrupt controller

Microblaze Debug module
Allows for JTAG based debugging

AXI Timer
Timer to the interface that provides reset and clk signals

Description of IO:

The GPIO acts as a bridge from the Microblaze to the FPGA logic. They are used as data transmitters that map to specific points in memory that can be accessed by software. They can be uni or bi-directional depending on how they are defined and have a specified data bit width but a total of 32 bits. The GPIOs like LED are an output which transmits the overflows bit and Switches GPIO takes in the value of the switches. There is also the Run accumulate GPIO which takes in the value of the push button and the Reset GPIO which takes in the value of the reset button.

Interaction of MicroBlaze with MAX3421 and ball components

The Microblaze accesses the AXI4 bus which is a memory mapped interface. This much like MEM2IO, intercepts data from specific locations and sends them to peripherals and receives data and puts them in specific memory locations. The Quad SPI takes the keycode values and passes them to the memory locations where the Microblaze is able to access and decodes them before inputting them to the ball motion sv modules, which result in a different type of motion depending on the keycodes.

Interaction of VGA controller with Ball and Color mapper

VGA controller is the main file for the drawing of pixels on the screen, it handles the total number of bits on the screen and the drawing process with DrawX and DrawY as outputs. These values are then passed to the colormapper.sv module. The ball.sv module purely defines how the ball moves and interacts with the rest of the screen, including edge detection and keycode decoding for extra movement with the keyboard. This information is then passed to colormapper.sv where the RGB values are determined based on if the pixel is within the ball or not within the ball.

Written Description of implementation of Tetris using SystemVerilog

Overall system description

Finite State Machine

We use a finite state machine to design and dictate the game logic and overall feature and design. We designed the state machine with extra states to handle more implemented logic, however the important states are as follows: Start, Newblock, Blockcheck, Rowcheck, Updategrid and die. In Start, the screen displays a complete blue screen and no other objects are shown. In Newblock, a random shape is generated from our fibonacci linear feedback shift register and instantiated on the board. Blockcheck acts as a holding state to constantly check if the block will be moving into a valid position by keycodes or by falling. If the block has another piece under it, it moves to the Updategrid state. The Updategrid state then updates the entire board array and draws the new piece on the board. This is followed by the Rowcheck state ,checking for filled rows and deleting them if needed, updating the scoring. Rowcheck is split into 4 states to allow for 4 cycles of checking rows individually since that is the maximum number of rows that can be completed at a time.

Block drawing and handling

We used a 16-element array *blockgrid[15:0]* with row-major order operation to handle a 4x4 grid which we used as a standard template to draw our shapes on. The center of this grid we leave as BallX and BallY from lab 6.2. This allows to index each of these squares to the coordinates by determining how far the grid's center is from the game boundaries and mapping each square to an x and y coordinate to be colored in.

We then had bit strings corresponding to the squares to be filled for each of the shapes
For example, for the bit string 16'b0010001000100010, squares 1, 5, 9, and 13 are filled which create a straight line, and so on for any shape we wanted to draw.

This blockgrid array was also used extensively in collision checking.

Array grid

The main data structure used to handle the Tetris game logic was arrays, specifically a 2 dimensional array *arraygrid* instantiated as 26 rows by 12 columns wide to handle a total of 312 'squares' on the game grid, which each square essentially corresponding to a 16 pixel x 16 pixel block on the screen. This included a ring a of 1s around the boundary of the array to represent the boundary of the board, making bounds checking more seamless.

For the mapping of the grid to the actual x and y coordinates on the screen, we used a simple math algorithm as follows:

gridrow = ((BallY - 32) / 16) - 1 ;
gridcol = ((BallX - 240) / 16) - 1 ;

Where the gridrow and gridcol index would be the top left of the squares. Note that 32 and 240 are the pixel starting coordinates of the game grid.

Collision detection

We then used different variations of this to access the specific arraygrid index we want to check for movement left, right, and down. Essentially, we would check the future left, right or down movement by doing:

```
if(arraygrid[gridrownextleft + i][gridcolnextleft + j] == 1 && blockgrid[i*4+ j] == 1) begin
        canmoveleft = 1'b0;
   end
```

where if any of the 16 blockgrid squares happened to be filled with a 1 at the same spot as the arraygrid square, the signal for movement in that direction is set to 0 to be passed into the Ball.Sv module. The same is done for other direction movements.

Rotation

Similar to movement, we first checked if a rotation is possible by checking the future shape with the present arraygrid to ensure no overlaps of 1s (representing filled). If rotation was possible, we use a hardcoded logic for every possible shape rotating into its next iteration, which is passed back to the top level into the Draw Modules.

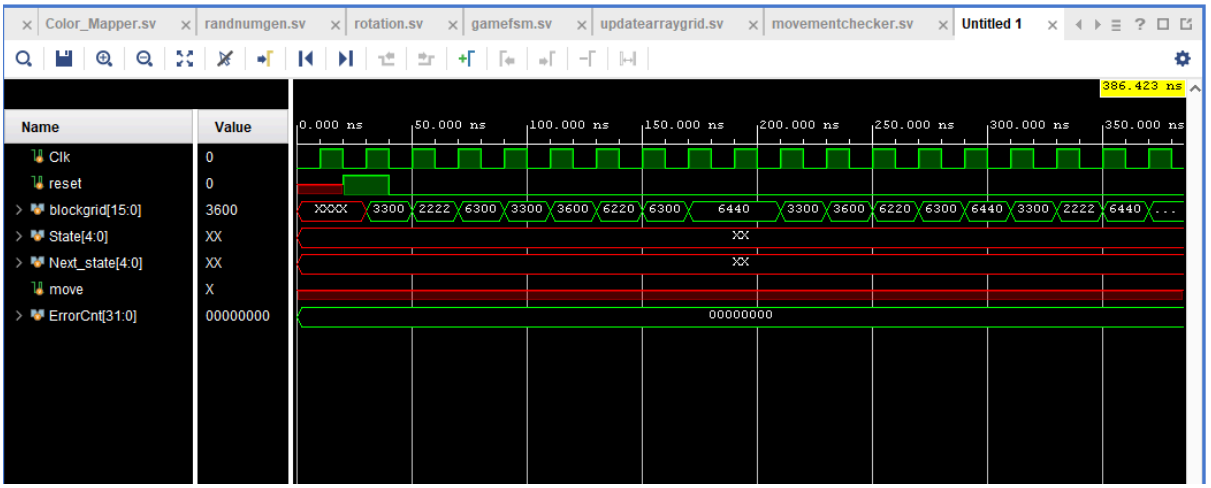
Row checking

Row checking was done using arraygrid, where if the entire row arraygrid[row][i] was filled with 1s, the row above is moved down using a simple backtracking algorithm, where arraygrid[row][col] is assigned arraygrid[row-1][col], which worked for every case since there is no case where any already updated row is being used in another operation.

Additionally, since this logic was in an always_ff block to accommodate for assigning arraygrid values, we had to include 4 wait states corresponding to 4 clock cycles to allow for the case of 4 rows being filled, which is the maximum number of rows that can all be filled at once using the straight line block.

Random Number Generator

We used a fibonacci linear feedback shift register (LFSR) to get encoding of random blocks every cycle. This was done through the use of a 16 bit shift register seeded on reset. Every clock cycle, the thirteenth and the fifteenth bit were XNOR-ed together and left-shifted into the register, allowing for a pseudorandom number to be generated every clock cycle, which was sampled in the new block state to generate a new block as seen below.



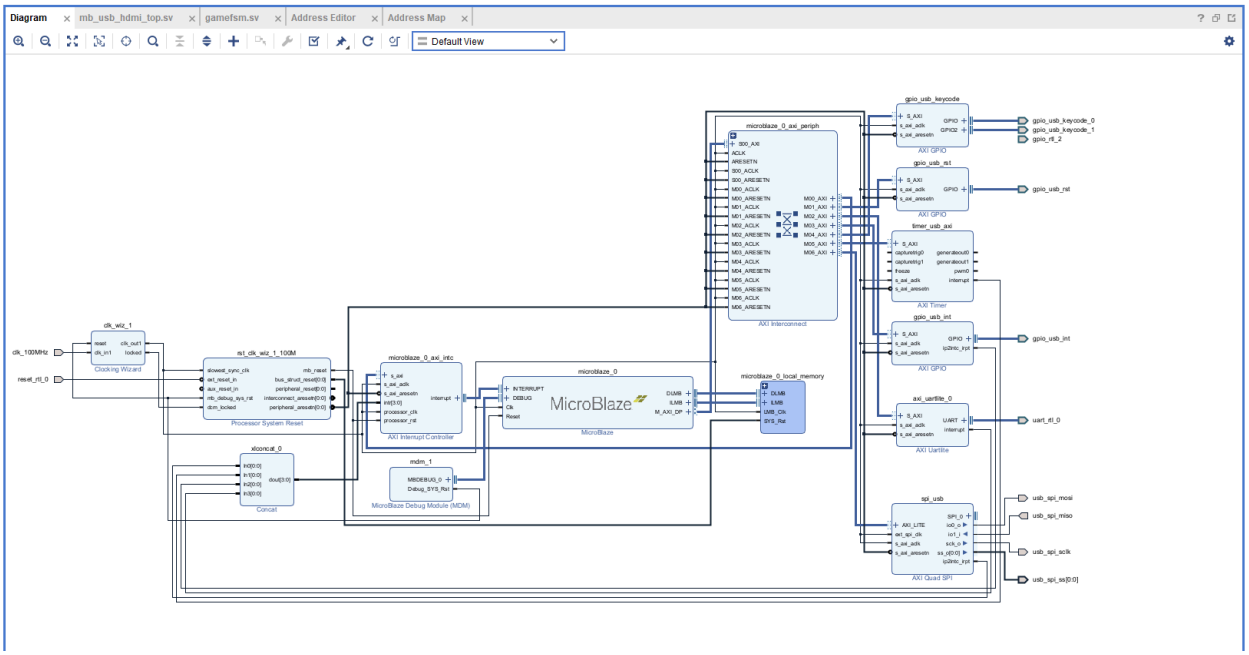
Scoreboard

The scoreboard was incremented with a counter every time a row was removed, this output was then sent to the HexA driver which would display the score on the Urbana board.

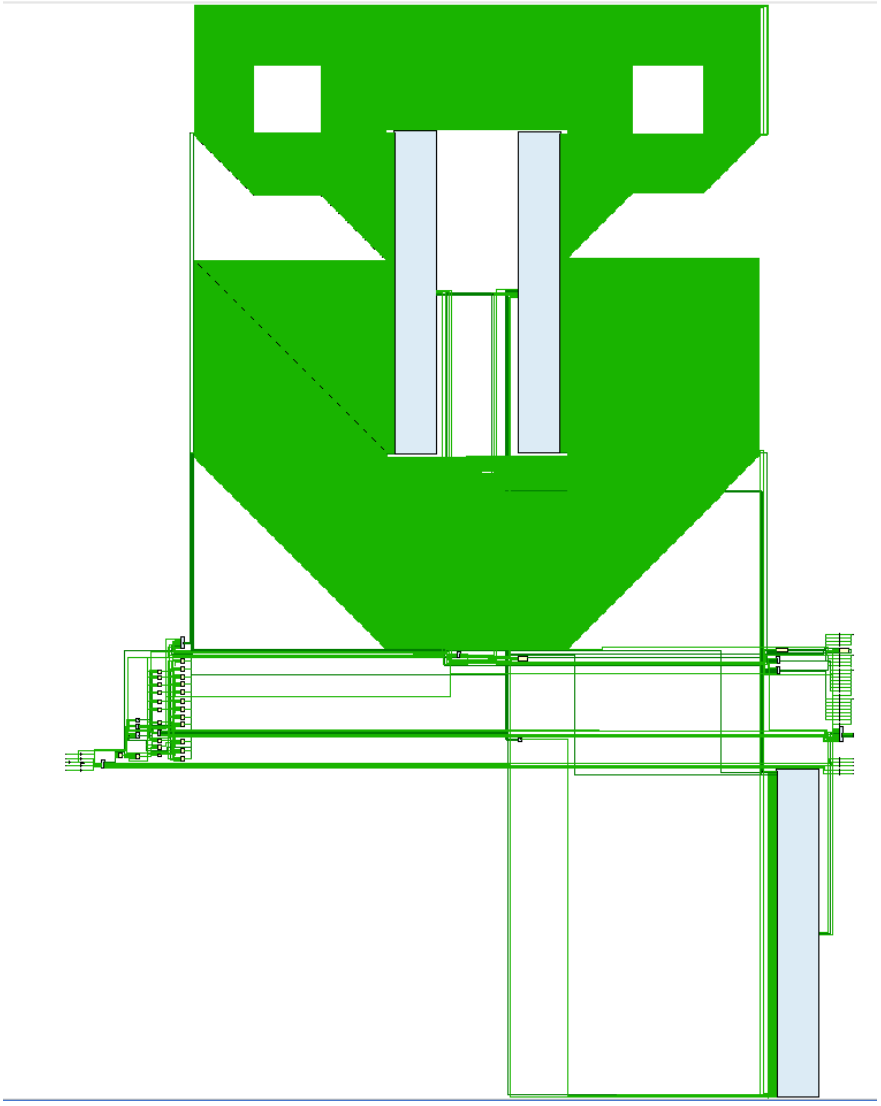
Start and end screen

The start and end screen were implemented as part of our FSM, where the topmost row is checked for any filled squares, which means the game has ended, before a signal is passed to break away to the blue screen of death.

Block Design



Top Level Block Diagram



Software component

As in lab 6.2, we had low-level C code to implement the SPI protocol. At a broad level, this involves a synchronous serial communication protocol between the master device (in this case, the CPU) and the slave device (MAX3421E). There are 4 key signals that this protocol works on, which are the SClk, MOSI, MISO as well as the CS (Chip select). Data is transmitted one bit at a time over multiple clock cycles.

For both reading and writing, the MAX3421E waits for the CS signal to be lowered for a command to be passed to the slave. Based on the !R/W signal in the 8 bit string, if reading the master sends a command with the register being read from, and the data is returned from the slave (in which case the MISO is high), and if writing the master sends a command to write followed by as many 8-bit long data strings as it wants to write until the CS goes back to high.

The functions we modified and their purposes are as follows:

void MAXreg_wr(BYTE reg, BYTE val):
Used for single host register writing. Writes the value of the register being written two which is offset by 2 to set the write bit to 1 subsequently. The value is written via the SPI and the return code is read to check if successful, before deselecting the MAX3421E

BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
Used for multiple byte writing to different registers. The same algorithm as single host writing, but the main difference is having an array to store data of size (nbytes+1) to write. Additionally, returns a pointer to the memory location after the last written data.

BYTE MAXreg_rd(BYTE reg)
Used for single host register reading. Create a BYTE array to store the register that it wants to read from and send this information through the XSpi_transfer function, and the data being read is also assigned into another BYTE array, with an error message printed if unsuccessful, before deselecting the MAX3241E

BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
Used for multiple byte reading. Similar algorithm as single host reading, but uses a BYTE array of size (nbytes + 1) to send in the register values being read from as well as an output BYTE array of size (nbytes + 1) to support the multiple data bytes being read, which are subsequently reassigned back to the BYTE* data, skipping the first bit of the output BYTE array which represents the register data.

Design Resources and Statistics

The frequency is calculated by doing 1/(10ns-wns);

LUT	8029
DSP	3
Memory (BRAM)	9
Flip-Flop	5671
Latches*	0
Frequency	0.112Mhz
Static Power	0.076W
Dynamic Power	0.486W
Total Power	0.561W

Conclusion

We managed to build a completely bug-free version of a tetris game entirely written on Systemverilog apart from the interaction of the MAX3241E with the Microblaze, which we believe is a success. The tetris variant we had seemed very smooth and not glitchy, perhaps reiterating the focus of this class which is using SystemVerilog to effectively program FPGA systems.

There are many extensions to this project that we considered, such as using a frame buffer implementation for added graphics, as well as implementing tetris with many more features such as combination row clearing, a preview of the next shape and being able to hold the current shape to come back to.

There were several bugs we encountered. Firstly, there were signals we did not include in the top level as logic wires, which caused ports to be unmapped or mapped wrongly, leading to entire modules disappearing. Additionally, we had to modify our fsm several times to account for the different stages in a tetris game, as sometimes it would move on to the new block state before it had even cleared all the rows, or it would prematurely move to the update grid state before the block had even hit something below it. This has given us a greater appreciation as to how code written into hardware has to be as precise as possible, and the need to juggle all the different elements of logic instead of simply writing out game logic in software using C or C++.

In conclusion, this project allowed us to gain new insights into the workings of Vivado and particularly improved our debugging skills since the code itself was logical and intuitive, but difficult to link all the signals without experiencing random bugs and errors.