Final Lab Report

Rachel Abraham (rachel20)
Albert Jiang (hongyi9)
TA: Henry Gillespie

Introduction

The goal of our final project is to design and implement a tile-matching game on the FPGA. The basic gameplay is to switch two adjacent tiles. Tiles will disappear if more than two of the same color are in the line, and tiles will fall down with new tiles refreshing at the top. Additional features include special items, score displays, and animation. The game will be handling USB keyboard input using MicroBlaze as in lab 6. MicroBlaze will also check for valid moves and update the tile position in the VRAM. The color video display will be first generated as VGA signal and then converted to HDMI signal as in lab 7. In the end, we successfully implemented the game with most of the desired features.

Design

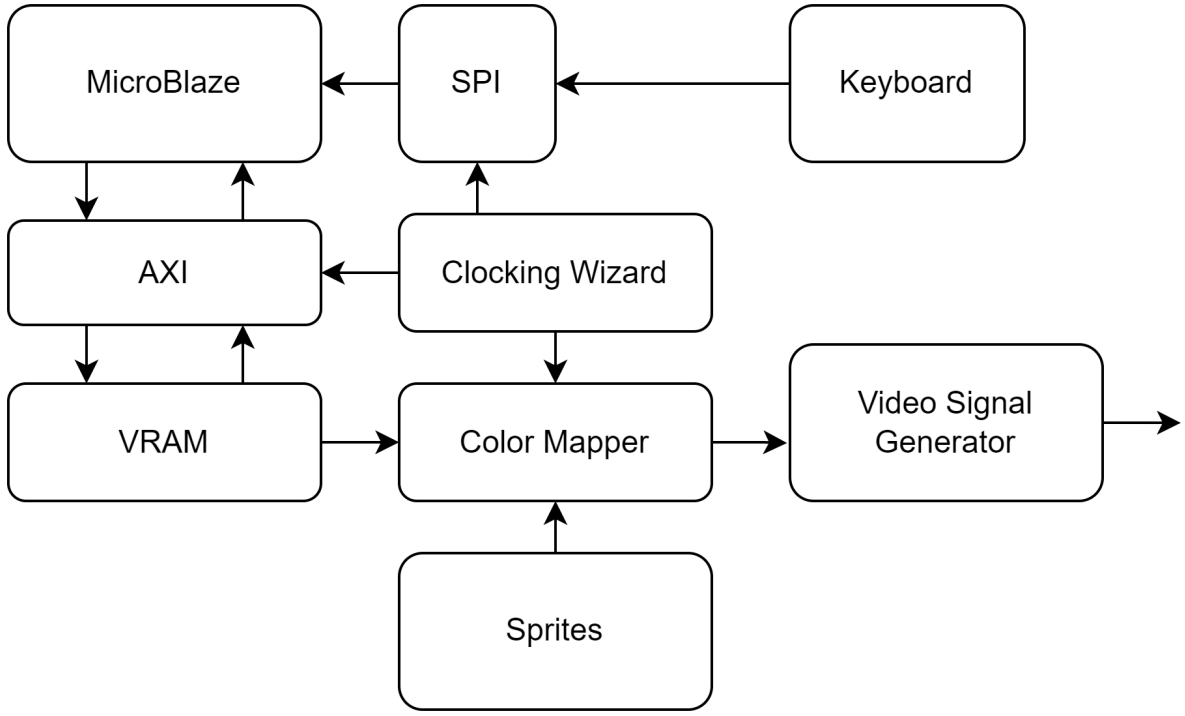


Figure X. Simplified Block diagram of the Game System

The general system in our final project is the combination of Lab 6 and 7. We used the same approach of creating our own IP as lab 7 and the same setup for receiving USB keyboard input as lab 6. Unlike lab 6, which directly sends the keycode to the hardware, we take in the keycode and evaluate it within the MicroBlaze. We used a similar memory-mapped I/O as lab 7 to modify VRAM using AXI protocol. The definition of memory map and bit encoding is listed below.

Word Address	Byte Address	Description
0x00 - 0x11	0x00 - 0x47	Tiles, 1 byte per tile 4 tile per word. 8 x 8 tile playfield and an invisible first row.

0x12	0x48 - 0x51	Current selected tile. 0-5 bits represent the selected tile (0-71)
0x13	0x52 - 0x55	Score. Each byte represents a 1-digit decimal number
0x14	0x56 - 0x59	Frame clock. The least significant bit is the vertical sync provided by VGA controller
0x15	0x60 - 0x63	Laser. Parameters for laser animation.
0x16	0x64 - 0x67	Starting Screen, The least significant bit represents on/off of the starting screen.

Table x, Memory Map definition for

Bit	7 - 3	2 - 0
Function	Y coordinate pixel offset (0-31)	Tile type, 0-4 are ordinary color, 6 is special tile, 7 is empty

Table x, Bit Encoding for Tiles

Bit	31 - 24	10 - 8	4	2 - 0
Function	Covered pixel from the left	Color offset for rainbow in laser	on/off for laser	Selected row for laser animation

Table x, Bit Encoding for Laser

Software Design

The software program in MicroBlaze takes care of all the game logic and video animation. The memory mapping is defined as a Struct in the C program as below.

```
81 struct HDMI_TEXT_STRUCT {
82     uint8_t  VRAM [72];
83     uint32_t CTRL;
84     uint32_t SCORE;
85     uint32_t FRAME;
86     uint32_t LASER;
87     uint32_t START;
88 };
```

Figure X, Struct for Memory Map

And all the functions we defined are listed below

```
void hdmiClr();
void iniTile();
void getTiles(uint8_t * tiles);
void pushTiles(uint8_t * tiles);
void setTile(uint8_t tile, uint8_t offset, uint8_t color);
void setLaser(uint8_t on, uint8_t row, uint32_t offset, uint8_t spin);
void switchTile(uint8_t a, uint8_t b, uint8_t* arr);
void hdmiTestWeek1(uint8_t key);
uint8_t checkMatches(uint8_t* arr);
uint8_t checkMatchesSimple(uint8_t* arr);
void fillTiles(uint8_t* arr);
void printTiles(uint8_t* arr);
void addScore(uint8_t score);
void aniFill(uint8_t* arr);
void refresh(uint8_t* arr);
uint8_t empty(uint8_t* arr);
```

void hdmiClr();
This function initialize everything in the struct to be 0.

void iniTile();
This function keeps filling the playfield(tiles) and remove all the match of more than three until there is no match in the playfield. This is done using checkMatchesSimple(uint8_t* arr) and fillTiles(uint8_t* arr). So no special tiles or score is added.

void getTiles(uint8_t * tiles);
This functions takes in a uint8_t array with 64 tiles. It updates the array to have the same values as last 64 tiles stored in the hardware. This function provide a way to make sure the array in the software is the same as the hardware.

void pushTiles(uint8_t * tiles);
This functions takes in a uint8_t array with 64 tiles. It updates the last 64 tiles to have the same values as the array. This function doesn't update offset and only changes the color of the tile.

void setTile(uint8_t tile, uint8_t offset, uint8_t color);
This function takes in values for tile index, tile offset, and tile color. It updates the tiles information in the hardware based on the input values.

void setLaser(uint8_t on, uint8_t row, uint32_t offset, uint8_t spin);
This function takes in uint8_t values for on/off, index of row, covered pixels from the left, and color offset. If on value is 1, it sets the laser animation to that specific location with color offset. If on value is 0, it turns the laser animation off.

void switchTile(uint8_t a, uint8_t b, uint8_t* arr);

This function takes in two indexes of tiles and an array of 64 tiles. If at least one of the tiles is special tiles(number 6), it turned on the laser animation. The laser animation is done by calling setLaser per frame to update the laser location. The function detects the frame clock by reading value from FRAME in the struct. After the laser animation, the corresponding row is removed. If none of the tiles is special, the functions check whether this switch creates a match of at least three by calling checkMatches (uint8_t* arr). If there is a match, the function add the number of matches to the score and call aniFill(uint8_t* arr) to have the falling animation to fill the removed tiles. The function keeps calling checkMatches (uint8_t* arr) and aniFill(uint8_t* arr) until there is no match in the playfield. If there is no match, nothing will be changed.

void hdmiTestWeek1(uint8_t key);
This functions takes in the keycode. This is called in the main function after USB is initialized, so the main function keeps calling it when new keycode is sent. Within hdmiTestWeek1, it stores the index of the current selected tile and update the select tiles based on the input. When it is called for the first time, it will set the START as 1 so the hardware will stop displaying the starting screen. If the keycode input are “wasd”, it will move the select tile to that direction by one tile. The selection will loop back from the other side if it goes out of bound. If the keycode is “↑ ↓ → ← ”, it will call switchTile(uint8_t a, uint8_t b, uint8_t* arr) for the selected tile and the nearby tile in the provided direction. If the selection is out of bound for switch, nothing will be changed. If the keycode is “r”, refresh(uint8_t* arr) will be called to refresh the playfield. Nothing will be called for any other keycode. The SELECT is updated based on the selected tile.

uint8_t checkMatches(uint8_t* arr);
This function takes in an array of 64 tiles. It checks for horizontal and vertical matches of at least 3. For match of 4, it will replace one of the matching tiles by special tile. For match of 5, it will replace two of the matching tiles by special tiles. For the rest, It will replace those tiles with removed tile(number 7). And it call pushTiles(uint8_t* arr) to update the screen. The function returns the number of total matches. Match of 4 counts as 2 matches, and match of 5 counts as 3 matches.

uint8_t checkMatchesSimple(uint8_t* arr);
This function is essentially the same as checkMatches(uint8_t* arr) except that it doesn’t generate any special tile and just replace matching tiles with removed tiles.

void fillTiles(uint8_t* arr);
This function takes in an array of 64 tiles. It moves tiles downward to fill in all the removed tiles. Then it replaces all the removed with randomly generated normal tiles(number 0 to 4).

void aniFill(uint8_t* arr);
This function takes in an array of 64 tiles. It assigns randomly generated normal tiles to the invisible first row in the hardware. The function iterate through the tile array in the hardware. For any tile in the array, if there is a removed tile below it, it will add an offset value in the hardware by calling setTile(uint8_t tile, uint8_t offset, uint8_t color). The offset value starts with 1 and increased by 1 per frame until 31. When the offset value is 31, the tiles that have removed tiles

below it will be moved downward by one. This process repeats until there is no removed tiles in the array.

void printTiles(uint8_t* arr);
This functions takes in an array and print every value in it. This is for debugging purposes.

void addScore(uint8_t score);
This function takes in an uint8_t value. It break the number into four digits in decimal and add those value to the SCORE.

void refresh(uint8_t* arr);
This function takes in an array of 64 tiles. It clear the array and the tile array in the hardware by setting every tile to removed tile(number 7). Then it calls aniFill(uint8_t* arr) and checkMatchesSimple(uint8_t* arr) until there is no matches or removed tiles in the array. This doesn't add score.

uint8_t empty(uint8_t* arr);
This function takes in an array of 64 tiles and check if there is any removed tile in the array.

The state machine is shown in the figure below

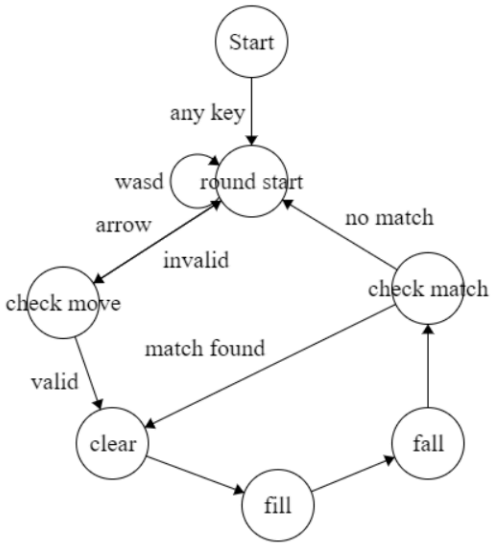


Figure X, FSM for the Game

At the starting state, the program will wait for the USB driver to be initialized. So before receiving any keycode input, the Microblaze will keep the START value as 0, which means displaying the starting screen. At this state, the program will call hdmiClr() to initialize all value in the struct to be 0. And then it will call iniTile() to fill the playfield with no matching tiles. After USB port is initialized, hdmiTestWeek1(uint8_t key) is called with keycode input from the USB keycode.

After receiving any keycode input, START is set to 1. And within the function hdmiTestWeek1, the selected tile is updated based on the keycode and switchTile(uint8_t a, uint8_t b, uint8_t*

arr) is called to check for moves. If there is any matches, the tiles are removed and filled until there is no match. Then it returns and wait for more keycode input.

Hardware Design

The Microblaze will write to the memory stored within our customized IP HDMI_text_controller. The memory is memory-mapped to Microblaze so the code may access it through pointers. Each 2 byte corresponds to one tile of 16x8 pixel on the 480x640 screen. Each tile stores the character code and the color. The RGB information of color is stored in the corresponding palette. The color mapper module will read from memory and provide the corresponding RGB information to the VGA to HDMI converter to output the HDMI signal.

For HDMI Text Mode controller IP, it will take input from the Microblaze through the AXI interface. For a read operation, Microblaze will put the read address onto the bus and put up the valid signal for the address and the ready signal for receiving data. The IP will respond with the ready signal for the receiving address, and then it will put the requested data onto the bus and put up the valid signal for data. For a write operation, Microblaze will put the writing address and data onto the bus and put up valid signals for them. The IP will respond with ready signals indicating it received the data. After the writing process is finished, the IP will return valid signal and responses.

In our hardware, we will generate the frame based on the memory set by the software control. The specific coloring of images is stored as Sprites in BRAM. And the hardware will choose the Sprites based on the memory

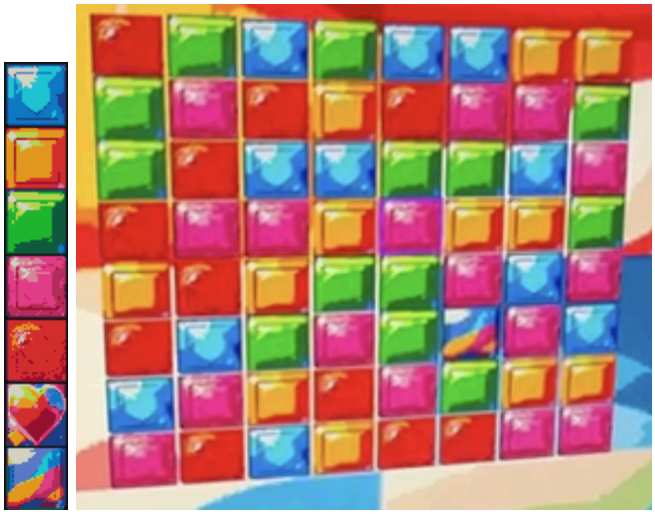
All sprites are modified AI-generated images. Then we convert each image file into COE files and palette files. Each color pixel in the image is simplified to an index number in the palette that contains the closest color to the pixel. These index numbers are stored in row-major order in the COE file. So after initializing BRAM modules with the COE file, we could recreate the image display signal by inputting the pixel coordinates to the BRAM and reading the corresponding index in the palette.

There are several layers of selection for displaying the corresponding Sprite:

1. Starting screen. Before initialization of USB port. This is displayed till a key is pressed on the keyboard. It fills the whole screen so the memory look up is directly correlated to the position on screen.

2. Background. The background is displayed in every area that the tiles or score are not displayed. It also fills the whole screen so the memory look up is directly correlated to the position on screen.

3. Tiles. The tiles are arranged as 6 different types of tiles. The tiles are arranged in an 8 by 8 pattern. Each tile has a different color which is from 0 to 6, 7 reserved for removed tile that directly display the background. The VRAM has which tile is in which location. Based on this, the corresponding tile is drawn in the correct spot to create the grid. The coordinate within the tile is calculated by taking the last 5 bit of the difference between DrawX and DrawY. The display logic also considers the Y coordinate offset of each tile. So the logic will compare the current coordinate within the tile and the offset of the tile above it to determine which tile it's drawing. The BRAM for the tiles is all together so that tile 0 is the first 32 rows and so forth. The address within the BRAM can be calculated based on this arrangement. And color black(RGB = 0, 0, 0) is set as transparent so those pixels will display the background color.

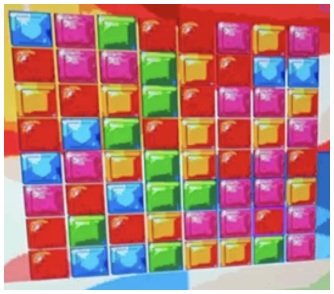
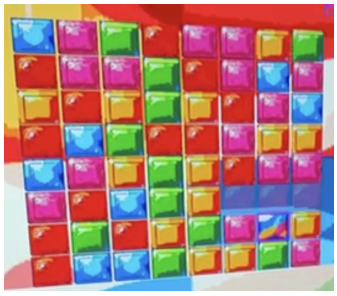


4. Selection. The outer boarder of the tile will be colored purple if the index of that tile equals to the value of selection register. This selection is on and off based on a counter that counts for 16 frame cycle. So the selection is blinking.

5. Score. The score is drawn using the font rom. The numbers 0 to 9 are in the font rom so that it can be addressed for each number that makes up the score. If the font rom has 0 in the bit the background is drawn instead so it looks transparent.

6. Laser(rainbow). The laser register is read from the AXI. If it is on, the row specified in the register gets animated with the rainbow color. The rainbow is drawn based on the offset and pixel given in the register so it moves from left to right on the screen. The color is read from the rainbow palette.

Here is an example game move.

1.  tile(6, 6) is select, and we press ↑ to switch with the tile above it
2.  a match of four is found. So it is removed and a special tile is generated, and tiles above them are falling off.
3.    More matches are found after new tiles are generated, so more tiles are generated and removed.
4.  It ends at a stable state that no more match exist.

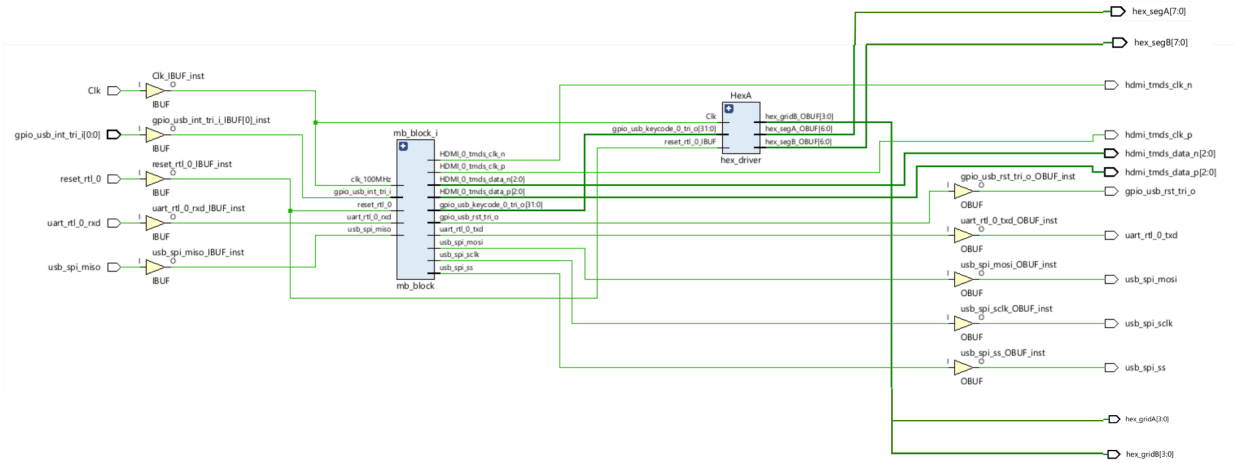


Figure 11: Top level diagram

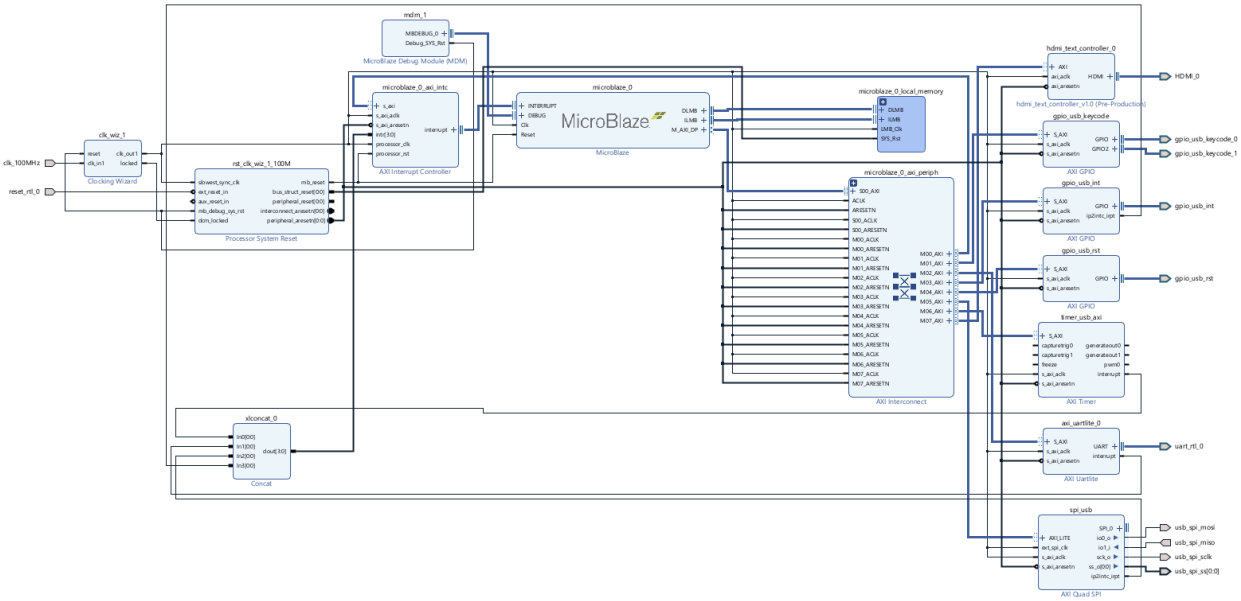


Figure X Block Design Diagram

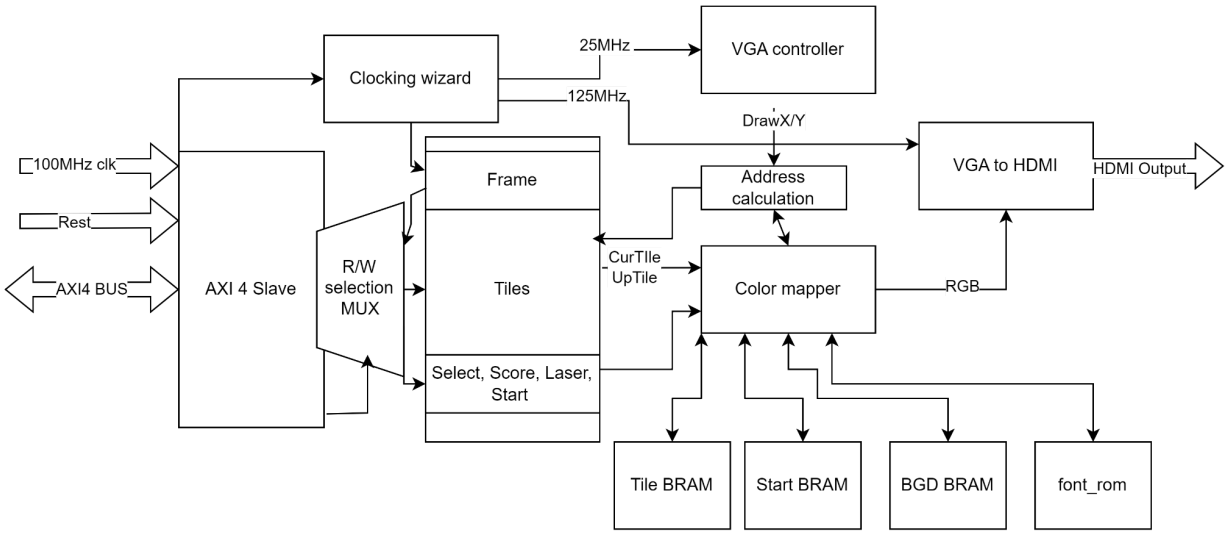


Figure X Block Diagram of IP

Modules:
Name: mb_usb_hdmi_top
Input: Clk, reset_rtl_0, uart_rtl_0_rxd
Output: uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0], hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB

Description: This is the top level that initializes the microblaze block with active low reset. Instantiate Hex Driver that prints the keycode pressed.

Purpose: Based on the uart input, the hdmi output is used to display things on screen.

Name: axi_uartlite_0 (AXI Uartlite:2.0)
Description: This is a serial communication protocol with interrupt mechanisms. It has serial console access with transfer and receive protocols.
Purpose: Allows for logging the keystrokes in the program.

Name: clk_wiz_1 (Clocking Wizard:6.0)
Description: It takes an input clock and manipulates the frequency, phase, or duty cycle.
Purpose: Creates a 25MHz and 125MHz clock from 100MHz for the VGA to HDMI signal.

Name: mdm_1 (Microblaze Debug Modle (MDM):3.2)
Description: This is the debugger for the system that can handle multiple processors in one design to debug.
Purpose: Make sure it is a valid design that can be put onto the fpga.

Name: microblaze_0 (MicroBlaze:11.0)
Description: This is a 32 or 64 bit processor with Reduced Instruction Set Computing that uses register memory and bus interfaces to do computations with peripherals.
Purpose: This is the fundamental unit for running the video display and getting the text display

Name: microblaze_0_axi_intc (AXI Interrupt Controller:4.1)
Description: This manages multiple interrupt signals with the capability of prioritization. This helps manage and control the processor tasks.
Purpose: Used for the UART that requires interrupt when sending text to the console.

Name: rst_clk_wiz_1_100M (Processor System Reset:5.0)
Description: This allows for a full reset of the microblaze processor with a universal reset signal. This keeps the order of reset correct if this is important in a design.
Purpose: Makes a full reset of the processor to start over the program.

Name: background_palette, newStartScreen_palette.sv, rainbow_palette.sv, tiles_424_palette.sv
Input: [3:0] index
Output: [3:0] red, [3:0] green, [3:0] blue

Description: Defines 16 different rgb colors. Outputs the rgb colors based on which palette color is picked.

Purpose: The palette reduces the size of memory for the background image so that a single index is stored rather than the whole color for every pixel.

Name: Color_Mapper
Input: [9:0] DrawX, [9:0] DrawY, frame_clk, clk_100MHz, clk_25MHz, Reset, [31:0] up_reg, [31:0] cur_reg, [31:0] score, [31:0] select, [31:0] laser, [31:0] start
Output: [6:0] upTile, [6:0] curTile, [3:0] Red, [3:0] Green, [3:0] Blue

Description: score_on is set based on an offset of where we want the score. Field on is set based on the size of the 8 by 8 tiles grid. The bgd_addr is the address in the bram to look at for the background color based on DrawX and DrawY. upTile is the current tile being drawn out of 64 which includes the invisible tiles. curTile is the current tile without the invisible tiles.Num_addr is the number needed to be displayed for the score. The rainbow is based on the laser. THe laser[2:0] is the row getting lasered, laser[4] is whether the laser should be on or off, laser[10:8] is the color offset to create the swirl, and laser[31:24] is the pixel. Updown is 0 for back, 1 for up , and 2 for cur. Based on tile_y and up_r, updown is set. If updown is 1 then the SP_addr_read is set. The font_reom takes in the num_addr and outputs the bits for the current number for the score. The blk_mem_gen_0 is for the tiles and takes in the SP_addr_read and outputs the current color. The background blk_mem_gen_1 does the same. The startscreen blk_mem_gen_2 also does the same. The rainbow_palette is used for the laser and has rainbow colors. The bgd_palette has the background colors. The newStartScreen_palette has the start_screen palette. The select_blink always_ff makes the currently selected tile blink purple. If start is 1 then the start screen is displayed. Otherwise the background or tiles are drawn based on the field. If the laser is on for a row then the rainbow colors are displayed. The score is displayed at the top right in the score_on location.

Purpose: The color_mapper outputs the color to be drawn per pixel based on the current score, and the tiles that should be displayed. There is also a start screen that is displayed at the beginning. The background is displayed in all the places where other things are not displayed.

Name: font_rom
Input: [10:0] addr
Output: [7:0] data

Description: The pixel description for every character in the font is written out with 8 bit rows and 16 columns. The addr input corresponds to a line of the memory and the data of that line is output.

Purpose: The font rom describes how each number for the score looks so that it can be drawn on screen based on the character selection.

Name: hdmi_text_controller_v1_0_AXI
Input: [15:0] regAddr, [3:0] fgdIndex, [3:0] bkgIndex, S_AXI_ACLK, S_AXI_ARESETN, [15:0] S_AXI_AWADDR, [2:0] S_AXI_AWPROT, S_AXI_AWVALID, [31:0] S_AXI_WDATA, [3:0] S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, [15:0] S_AXI_ADDR, [2:0] S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY, frame_clk, [4:0] curTile, [4:0] upTile
Output: [31:0] currReg, [15:0] fgd, [15:0] bkg, S_AXI_AWREADY, S_AXI_WREADY, [1:0] S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, [31:0] S_AXI_RDATA, [1:0] S_AXI_RRESP, S_AXI_RVALID, [31:0] upReg, [31:0] currReg, [31:0] conReg, [31:0] scoreReg, [31:0] laserReg, [31:0] startReg

Description: Initializes the BRAM that outputs the currReg at Index of regAddr. It writes to register memory based on S_AXI_WDATA. If reset is low axi_awready is deasserted. Otherwise, if the slave is ready to accept a write address and there is a valid write address and write data, axi_awready is high. The axi_awaddr is latched when both awvalid and wvalid are high. Wready is asserted when awvalid and wvalid are high. It is deasserted otherwise. Every slv_reg is looped through in 7.1 and the corresponding data is written to it. If reset, 0 is written to every register. The control register is outputted in 7.1. In 7.2, addrb is used for the palette registers. Fgd and bkg color is equal to the palette at the foreground and background index. Based on the axi_awaddr, laser, select, score, or start will be written to. The corresponding Register is equal to that value. The same is true for read.

Purpose: Write to registers what is stored in memory to read and write data from the AXI so that what is drawn on screen matches with updates in code.

Name: hdmi_text_controller_v1_0
Input: axi_aclk, axi_aresetn, [15:0] axi_awaddr, [2:0] axi_awprot, axi_awvalid, [31:0] axi_wdata, [3:0] axi_wstrb, axi_wvalid, [15:0] axi_araddr, [2:0] axi_arpot, axi_arvalid, axi_rready

Output: hdmi_clk_n, hdmi_clk_p, [2:0] hdmi_tx_n, hdmi_tx_p, axi_awready, axi_wready, [1:0] axi_bresp, axi_bvalid, axi_bready, axi_arready, [15:0] axi_rdata, [1:0] axi_rresp, axi_rvalid

Description: Initializes the hdmi_text_AXI which outputs the foreground and background color based on input indexes. Instantiates the clk wizard, vga controller, vga to hdmi, and color mapper modules to output the correct hdmi signal based on the axi inputs.

Purpose: Outputs the hdmi signal based on axi inputs.

Name: VGA_controller
Input: pixel_clk, reset
Output: hs, vs, active_nblank, sync, [3:0] drawX, [3:0] drawY

Description: If reset is pressed, vs and hs reset to 0. Otherwise if hc is at the end of width of the screen, hc resets to 0 and If vc is at the end of length of the screen, vc resets to 0. If hc is not at the end of the row, hc increments. drawX is hc and drawY is vc. A horizontal sync pulse is 96 pixels to ensure a clean output waveform. The vertical sync pule is 800 pixels for a clean output waveform. Only the pixels between 640 x 480 are displayed.

Purpose: Output the draw signal based on VGA signal (top to bottom not interlaced).

Name: HexDriver
Input: clk, reset, [3:0] in[4]
Output: [7:0] hex_seg, [3:0] hex_grid

Description: defines the bits that need to be high for each hex digit to be displayed correctly on the Hex Display. The driver takes 16 bits input and divides them into four 4 bit value which will be displayed as hex number.

Purpose: Displays the keycode of the key being pressed.

Name: nibble_to_hex
Input: [3:0] nibble
Output: [7:0] hex

Description: Based on the 4 nibble bits (which is 1 hex), it sets hex to the 8 bit signal corresponding to 0 to F.

Purpose: The hex output lights up different LEDs that create the shape of 0 to F in hex to be able to visualize the input data.

Design Resource and Statistics

	Multiplier
LUT	3825
WNS	-0.032
DSP	0
Memory (BRAM)	36
Flip-Flop	3430
Latch	0
Frequency (MHz)	99.68102073
Static Power (W)	0.077
Dynamic Power (W)	0.404
Total Power (W)	0.481

Table x, Design Statistics

We have a violation of the timing since it has a slightly negative WNS. This doesn't affect the performance because the pixel clock is 25MHz and clock for reading from BRAM is 100MHz. So the delay on selecting different reads from memory doesn't affect performance. Also, the frame updates is about 25Hz, the delay on the registers would hardly affect the actual video display. Since the game stays in steady state for many clock cycles this does not affect game play and is not visible.

Conclusion:

The game works as expected and accurately represents a tile-matching game. In the future, we would like to expand this to have more power-ups for different types of matches such as 5 in a row. Right now the board can be changed to any square size because of how the code is written but it could be interesting to change the board to a nonsquare shape. A leaderboard or different levels could be added as well. Finding an artist to draw custom sprites could improve the aesthetic quality of the background and tiles. Doing this project allowed us to fully understand AXI read and write. This project is a good start for making any game for the fpga since we have the connections set up properly to modify the VRAM for any game logic in the Microblaze. The python script to change a PNG to a .COE and palette was especially useful for this final project and was a great help for creating nice looking sprites and backgrounds.