COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

CSEE4840 Final Project: Tetris Game

Team Members: Zihao Wang | Shengyue Guo | Shibo Sheng

Instructor: Prof. Stephen A. Edwards

Presented On: May 12th, 2022

Table of Content

1. Introduction

1.1 Project Overview

The team has designed and implemented a classic Tetris Game which was a puzzle video game created by Soviet software engineer Alexey Pajitnov in 1984.

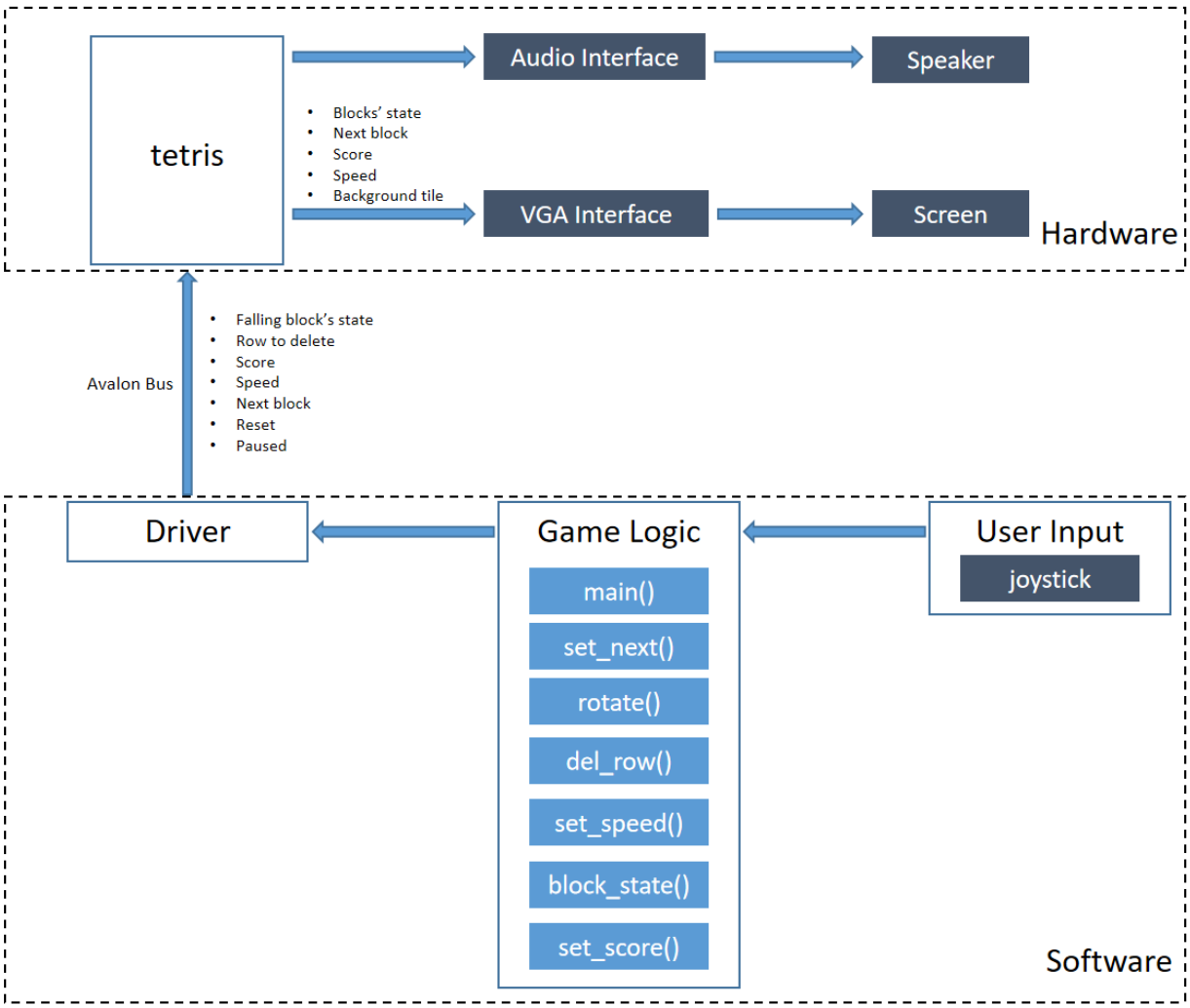
"In Tetris, players complete lines by moving differently shaped pieces (tetrominoes), which descend onto the playing field. The completed lines disappear and grant the player points, and the player can proceed to fill the vacated spaces. The game ends when the playing field is filled."

To fulfill the enjoyment of the classic Tetris Game, the team rebuilt the game that has 3 levels of difficulties with each level increasing the speed of the falling block along with quicker audio output. The team developed the game on a DE1-SoC FPGA board and players control the game via a PS2 controller for the purpose of more accurate and faster control. [3]

1.2 Game Information

Once the players pressed the "START" button on the PS2 controller, the Tetris Game would start with the basic difficulty level along with the normal speed bgm music. Once the players feel confident and want to challenge themselves, they can press the "SELECT" button on the PS2 controller to increase the difficulties by fastening the speed of the falling blocks. Once the difficulty level is increased, the bgm music speed would increase as well indicating the more challenging game with higher adrenaline levels. The game consists of 3 difficulty levels with the 3rd one being the fastest. Players have to make very quick and accurate decisions on this level.

2. System Architecture



This is a very high level architecture diagram. Detailed diagrams will be shown in the following sections.

In the software part, the team used a structure that mainly consists of a forever loop with timeout to model frames. In each frame, the program will first read from the joystick input buffer and handle the keys accordingly. The game status is stored as several parameters and a 20×10 matrix. The team used a block object to store the currently falling object, and most of the button presses and corresponding handling methods are implemented as functions. After the buttons are correctly handled, information will be sent to the hardware module according to our protocols. Automatic falling down is implemented by counting frames.

Hardware can be divided into two parts: VGA and audio. In the VGA part, the hardware will read images stored in BRAM and display them according to the data stored in registers. For the audio, the team loaded a music that is around 11-second long and played it in a loop as the background music. Besides, the team changed the speed of it according to the difficulty level of the game. The higher the level is, the faster the music is.

3. Hardware Design

3.1 Graphic Design

The team made very careful design on the image files since the space of BRAM is very limited. The team used a tile-and-sprite method to do our graphic display. The system has 3 layers in total: background image, blocks and numbers. Background image is still while blocks and numbers will change according to the data stored in registers.

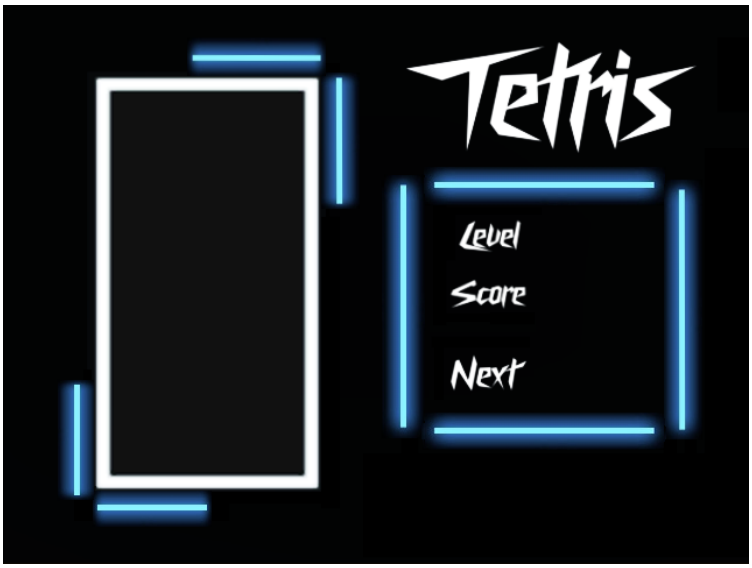
The sizes of each image is shown below:

Image	Shape	# of images	Size (bits)
Background	640×480	5	5×128×512×8 + 64×24 = 2622976
Block	16×16	1	16×16×2 = 512
Numbers	16×16	10	10×16×16×3 = 7680
Total			2631168

As block and number images are of size 16×16, the team can easily locate their position with the first 6 bits of *hcount* and *vcount*, while using the remaining 4 bits as the address to read data from corresponding memories. That is to say, the team divided the screen into 40×30 grids, and then the team can use coordinates to put images into correct positions.

3.1.1 Background Image

For the background image, the team used 64 24-bit colors. The shape of the background image is 640×480. In order to save space, the team used the palette method. Besides, to avoid too much overhead, the team divided the background image vertically into 5 parts, so the shape of each part is 128×480. However, due to the design of FPGA on-board memory, each image actually has the shape of 128×512.



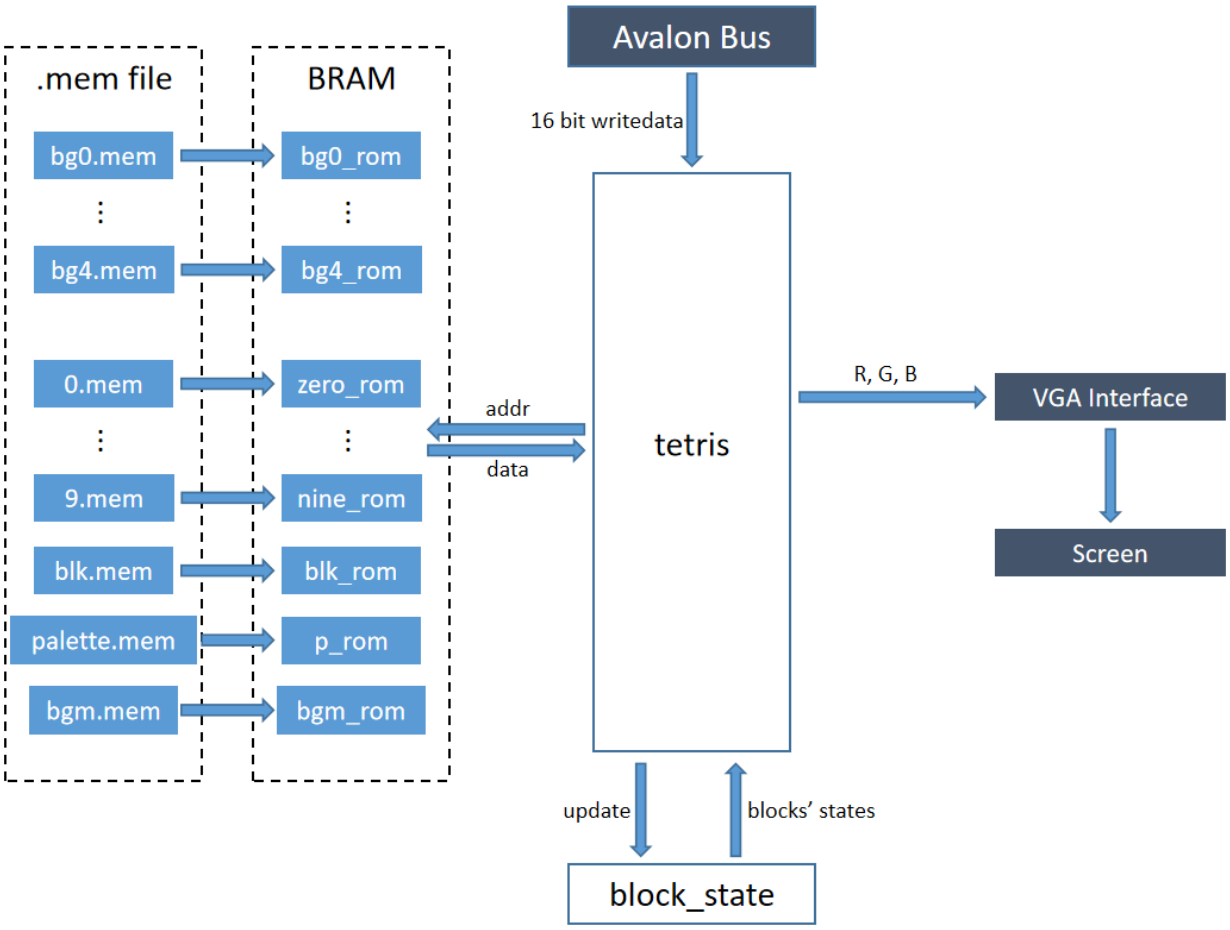
3.1.2 Block Image

The shape of each block is 16×16. To make the blocks look prettier without using too much space, the team stored 2 bits for each pixel as grayscale. Meanwhile, the system has 7 24-bit colors stored in registers. For each block, the team generated color by doing bit operation ({3{rom_data[1], {7{rom_data[0]}}}} & block_color). The generated block looks like this:

3.1.2 Number Image

Like blocks, the shape of each number is also 16×16. However, the team stored 3 bits for each pixel and used these 3 bits together with 5 bits of 0 as color for R, G and B channels. This method saves space but introduces slight color distortion. However, the influence is not that obvious.

3.2 Hardware Block Diagram



Above is the diagram of our hardware. The team implemented asynchronous ROMs to read the .mem files and store them. Because of different image sizes, the team used addresses of different lengths. Data lines have different lengths as well.

The team had a set of registers called *block_state* to store the state of each block. Its shape is 20×10×3. That is to say, the system has 20×10 registers and each has a length of 3-bit (the system has 7 types of building blocks). If the value in a register is not zero, then it will display a block at the corresponding position. The value of the register determines the block's color. Besides, in order to display the next building block, the team used another 3-bit register called *value_next*.

The system had a set of registers called *score* to store the player's score. Its shape is 4×4. The module will display the corresponding numbers at correct positions. It's the same for the speed number.

The above two kinds of images are both in the shape of 16×16, so the address is 8-bit long. For blocks, the output line is 2-bit long while 3-bit long for numbers.

For background images, the team used the palette method. The team had 5.mem files that store the indexes of colors. The team used the index to look for color in palette.mem file and then display it. As the shape of each background image is 128×512, the address line connected to bg_rom is 16-bit long. The output of bg_rom is 8-bit long. Because the system only has 64 colors in the palette, the team took the 6 least significant bits from the output and connected it to p_rom. The output of p_rom is 24-bit long.

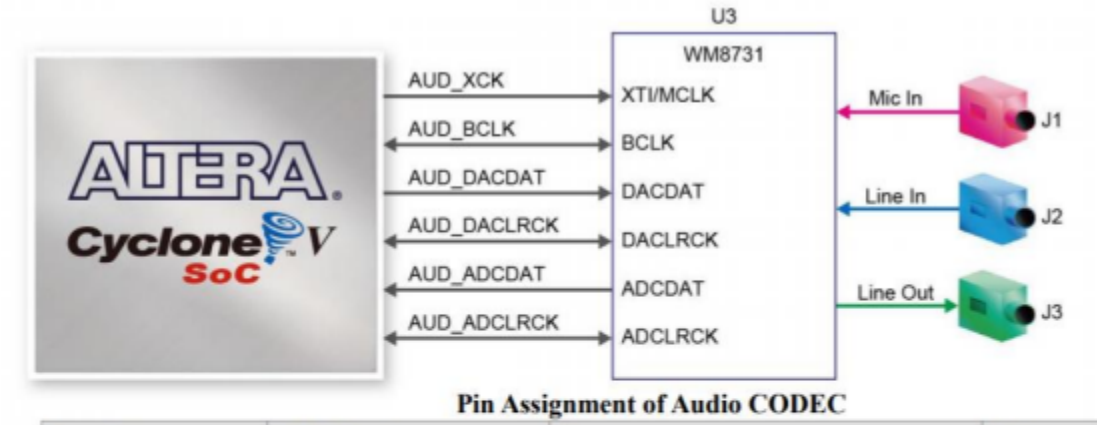
The size of our bgm file is 92897×16 bits. Therefore, the address is 17-bit long and the output is 16-bit long.

To reduce the number of communication between software and hardware, game logic of delet_row is implemented in the hardware. Once the hardware receives which row to delete, it will first empty this row. Then it will move rows above one grid down. With this function, the system only needs 1 communication instead of 20.

3.3 Audio

The DE1-SoC FPGA board has three audio jacks: a line out, a line in, and a microphone jack. The jacks interface with a Wolfson WM8731 audio CODEC chip. This chip has ADCs and DACs connected to the analog jacks and connected to the FPGA via a digital interface. The Audio Core is a circuit which simplifies communication between the Nios II processor and the Audio CODEC responsible for converting sound into digital values and vice versa. It is

essentially 2 pairs of FIFOs each 128 entries deep with each pair responsible for sending/receiving samples to/from the LINEOUT/MIC. They are pairs because there are two different channels, left and right, allowing for stereo sound. The connection of the circuit is shown as follows. [4][5]

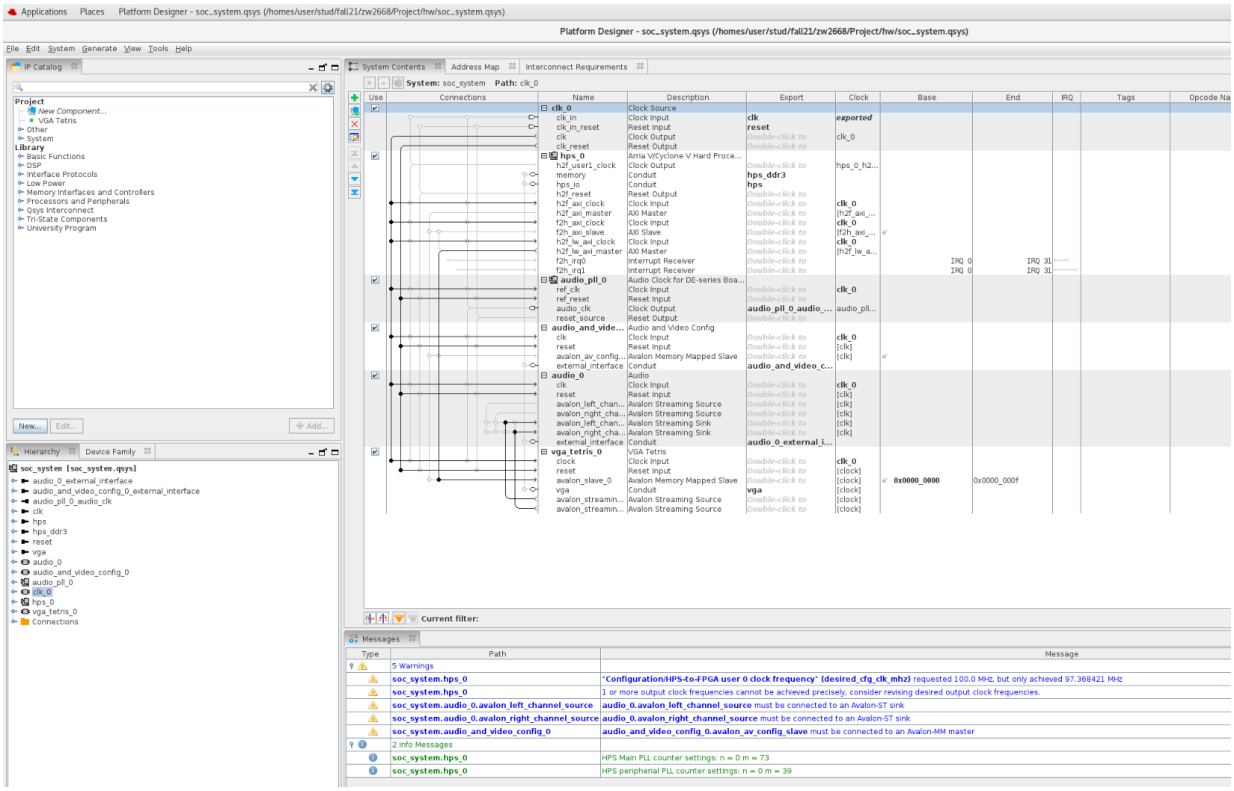


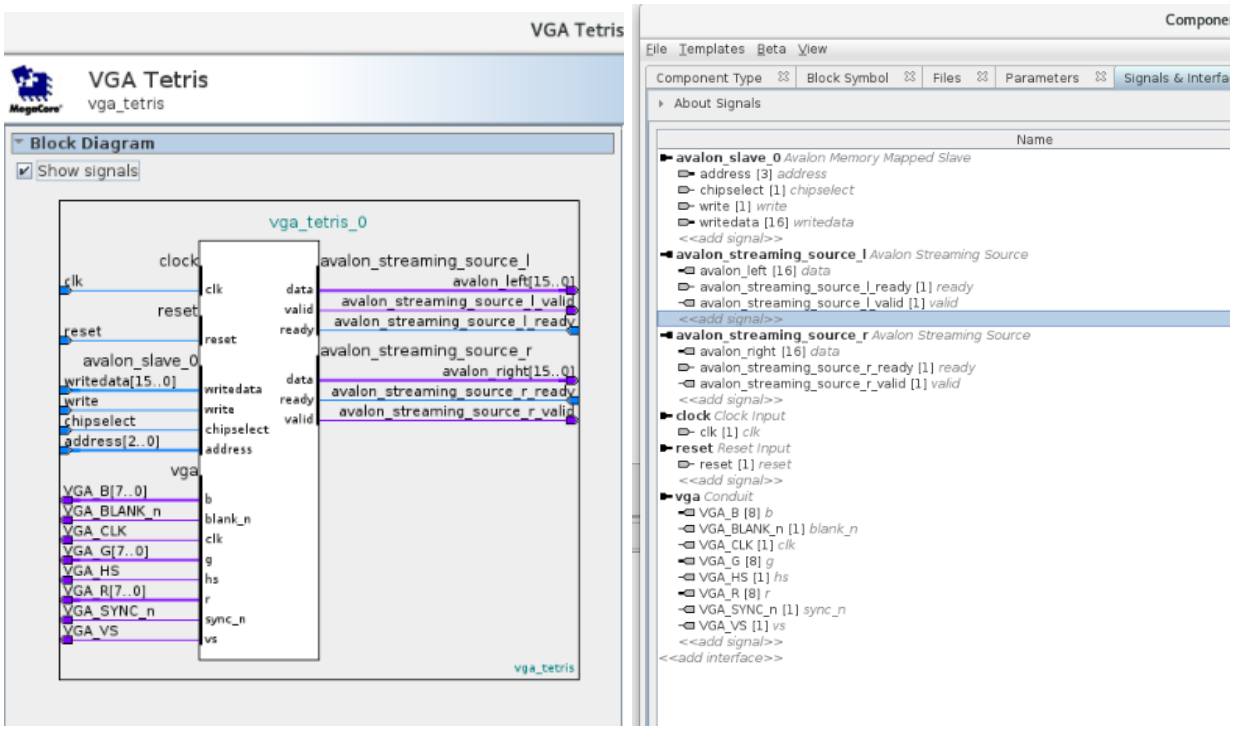
Pin Assignment of Audio CODEC

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
AUD_ADCLRCK	PIN_K8	Audio CODEC ADC LR Clock	3.3V
AUD_ADCDAT	PIN_K7	Audio CODEC ADC Data	3.3V
AUD_DACLRCK	PIN_H8	Audio CODEC DAC LR Clock	3.3V
AUD_DACDAT	PIN_J7	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_G7	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_H7	Audio CODEC Bit-stream Clock	3.3V
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data	3.3V

The team followed the report instructions in previous years' projects as well as the instruction written by Cambridge University which is partially incomplete. The team gathered all the incomplete instructions and piece them back together and finally figured out how to properly implement the audio output function on the DE1-SoC FPGA board. The team was the very first team to figure out the audio function implementation as the team was frequently asked and helped many other teams to finish their audio function parts so that they could finish their projects on time.

The team first read through the instructions by Cambridge University and partially set up the Qsys connection. Later on, the team realized that the instruction was incomplete. As a result, the team searched through the previous year's projects and realized that in order to combine the existing Qsys connection with the newly added Audio parts, the team needed to add several streaming interfaces and redo some of the connections. The final Qsys is shown below:





Final Qsys Configuration

After the team has successfully set up the Qsys connection, the team downloaded an 8-bit bgm music from the internet and converted it to 16 bits mono PCM.wav. The team then used Audacity to convert the music into an 8kHz one-channel wav file. The team also wrote a python script to convert the .wav file into .mem file, filter out the useless text in the .mem file and shorten the audio to eliminate the noise at the end of the audio file. Finally, the team wrote the audio output code in the main hardware code file. The team has to manually divide the frequency by 6250 in order to get the proper audio output (The clock's frequency is 50MHz and our music's sample frequency is 8kHZ). Later on, since the team decided to corporate the difficulty level increasing mechanism with the speed of the bgm music, the team modified the code so that once the difficulty level increases, the frequency divider decreases, thus increasing the speed of the bgm music.

4. Software Design

4.1 User Input

Users have control on 6 keys. The left and right shoulder keys are used for controlling the left and right movements of the currently falling block. Each press will cause the block to move by one unit. The down key on the direction pad is used for speeding up the falling down process, and users can hold the key while the block continues to fall. The "A" key will be used to rotate the block by 90 degrees clockwise. The team also used the start and select keys to control the main game, where start is used for pause and restart the game, and select is used for adjusting the speed level of the main game.

4.2 Game Logic

The software is designed to have only one thread, since this is enough for the game. The main structure is a forever loop with timeout. Each loop can be considered a frame, and in one single frame, the program will read from the joystick input buffer, handle the keys and send corresponding information to the hardware side. The length of a frame is kept as 5 ms so that the update rate is high enough to keep the minimum number of simultaneous presses in one frame. This is achieved with the timeout mechanism of the reading function from a buffer. Though in theory a faster press would speed up the frames, considering that normal presses from human hands are less slower than the above threshold, and we still need time to process keys, this implementation would not influence the normal game play. Of course a better implementation would be to keep a timer for the loop, but our implementation can also be considered a guidance for the users to take minimum possible moves.

The game status is stored as main parameters such as game speed, key status, game paused, score and several frame counts. The tetris grid is stored as a binary matrix of size 20×10, and the

falling block is stored as a structure that consists of the coordinates of the up-left corner, the rotation and the type. Notice that the falling block will not be updated to the grid matrix until the block completes a successful move.

There are methods to rotate and move the falling block. These methods will first try to do the corresponding operations, which is a function that flips four values in the grid. A valid flip operation should flip four bit of 0 to 1, which represent assigning a new block, or flip four bit of 1 to 0, which represent erasing an existing block. Thus, a correct operation should first erase the block of the old status in the grid if needed, and try to assign the block of the new status. If any flip is not valid, that flip will be done again to keep the status of the grid, otherwise, an operation is considered successfully handled, and the update of the falling block will be sent to the hardware side.

When the block tries to move down but returns invalid operation, the game will trigger a function that tries to discover if any line(s) can be erased or not. If successfully erased, the score will be added according to the formula: line_erased×(speed + line_erased). This encourages users to try higher speed and erase multiple lines at the same time. The line erased and the change of score will be updated to the hardware side. The game grid will be updated with memory copy in software.

4.3 Avalon Bus Interface

The team used *iowrite16* for software and hardware communication. Notice that the *e* value means whether we should erase the block or create a new block. In this way, instead of updating 8 slots, we just need two communications to complete the update, and also saves the computational time needed in hardware.

Addr	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Comment

0	row					col				value					e		Block's state
1												row_to_del						Row to delete
2	score[3]				score[2]				score[1]				score[0]					Player's score
3														value_next				Next block
4															speed			Game speed
5															reset			Reset from sw
6															o	p		Paused or not Over or not

5. Problem Encountered

Hardware:

The most challenging part is how to put large images into limited storage space. The team first tried to put the background image directly into the board, but the team found that overhead was huge (640×480 was extended to 1024×512). Then the team came up with the idea of splitting the image. The overhead problem is solved, but the size of the image is still too large. Finally, the team used the palette method. The size of the image decreased from 640×512×24 = 7864kb to 640×512×8 + 64×24 = 2623kb.

Another challenge is the latency. The team found that using case statements in comb can introduce very large latency. The latency caused the vga display to keep shaking, which made the screen look very weird. To deal with this problem, the team stored the data wanted in registers in advance and accessed these data with indexes. What's more, the team optimized the drawing-block algorithm by eliminating loops inside. With these improvements, the latency became so small that the screen could display everything normally.

Software:

One of the major issues the team encountered at the starting stage of the project was how to read from the joystick. Initially, the team tried to use library functions. This works well on workstations, but since the linux version on the DE1-SoC board does not have corresponding joystick drivers, the library functions cannot work. The barrier arises where the system cannot distinguish the joystick from other types of USB inputs, thus the library function which only takes joystick type input cannot run properly. The team tried to update the linux version or install the missing drivers, but all of them seem not to be working. One difference appears in the system is whether the device has the correct prefix. All the USB device can appear with an *event* prefix, and if correct drivers are installed, it should have a *js* prefix.

Finally, the team tried to directly pull input from the USB port. The team reads the raw input using the ***poll*** function from the library ***poll.h***. We firstly open the device port as a file in *NON-BLOCK* mode, and use the above function to try to fetch from the buffer. The raw information consists of several 16 bits information, each key action could correspond to two or three such information. We classify the bits to get necessary information, and the input packages will eventually be mapped to a key, value pair, where the first one represents the key type, and the second one represents the motion press or release. These two values will then be processed in a looped manner, as there can be multiple key presses within a single frame. Another potential way to solve this problem could be the device tree, but ***poll*** is good enough for our application.

Audio:

The team encountered various problems when implementing the audio function for the Tetris Game. The team first followed the instruction by Cambridge University to reconfigure the Qsys connection. After matching the instruction completely, the audio function was still not working.

After careful investigation on multiple projects involving audio functions from previous years, the team realized that the team needed to add a few interfaces under the main function part. Thus, the team added 2 Avalon streaming sources with a choice of "data" under the VGA_ball part and had them connected to the 2 Avalon streaming sinks in the audio_0 part. After properly connecting all the clocks, resets, and exports, the audio function can finally output some random noise.

After successfully configuring the Qsys connection, the team downloaded a 8-bit gaming bgm music from the internet. At first, the team had no idea how to play the bgm music as it was initially an mp3 file. The team later realized that in order to play the bgm music, the team needed to first convert the mp3 file to wav file and further convert it to mem file as well as filtering out the noise data. The team wrote a python script to do the work and successfully played it.

6. Reference

1. *"Computer Laboratory." Computer Laboratory – Course Pages 2016–17: ECAD and Architecture Practical Classes – Tutorial, https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/optional-tonegen.html.*
2. *"General Information." CSEE 4840 Embedded System Design, http://www.cs.columbia.edu/~sedwards/classes/2020/4840-spring/index.html.*
3. *"Tetris." Wikipedia, Wikimedia Foundation, 1 May 2022, https://en.wikipedia.org/wiki/Tetris.*
4. *De1-SOC Interfaces and Peripherals, https://class.ece.uw.edu/271/hauck2/de1/.*
5. *"Audio Core." Audio Core on DE1-SoC, http://www-ug.eecg.utoronto.ca/desl/nios_devices_SoC/dev_audio.html.*

7. Appendix

Hardware:

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 * Columbia University
 */

module vga_tetris(
    input logic clk,
    input logic reset,
    input logic [15:0] writedata,
    input logic write,
    input chipselect,
    input logic [2:0] address,

    output logic [7:0] VGA_R, VGA_G, VGA_B,
    output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
    output logic VGA_SYNC_n,

    // The music related IO.
    input avalon_streaming_source_l_ready, avalon_streaming_source_r_ready,
    output logic[15:0] avalon_left, avalon_right,
    output logic avalon_streaming_source_l_valid, avalon_streaming_source_r_valid
    );

    logic [10:0] hcount;
    logic [9:0] vcount;

    vga_counters counters(.clk50(clk), .*);

    // The pixels equal to 16 * 16 here.
    localparam SPR_PIXELS = 256;
    localparam COLR_BITS = 1;
    localparam SPR_ADDRW  = $clog2(SPR_PIXELS);
    logic [SPR_ADDRW - 1:0] rom_addr;
    logic [1:0] block_rom_data;
    logic [2:0] digit_rom_data[0:9];

    // The pixels equal to 128 * 512 here.
    localparam BG_PIXELS = 65536;
    localparam BG_ADDRW  = $clog2(BG_PIXELS);
    localparam BG_IDX_BITS = 8;
    logic [BG_ADDRW - 1:0] bg_addr;
    logic [BG_IDX_BITS - 1:0] bg_idx_rom_data[0:4];
    localparam BG_COLOR_BITS = 24;
    logic [6 - 1:0] bg_idx;
    logic [BG_COLOR_BITS - 1:0] bg_rom_data;

    // bgm
    logic [15:0] bgm_rom_data;
    logic [16:0] bgm_rom_addr;
```

```
genvar i;
generate
    // Background segements.
    for (i = 0; i < 5; i++) begin: generate_bg
        rom_async_hex #(
            .WIDTH(BG_IDX_BITS),
            .DEPTH(BG_PIXELS),
            .INIT_F({"./image/bg", "0" + i, ".mem"})
        ) bg0_rom (
            .addr(bg_addr),
            .data(bg_idx_rom_data[i]));
    end

    // The digits for scores
    for (i = 0; i < 10; i++) begin: generate_digits
        rom_async #(
            .WIDTH(3),
            .DEPTH(SPR_PIXELS),
            .INIT_F({"./image/", "0" + i, ".mem"})
        ) zero_rom (
            .addr(rom_addr),
            .data(digit_rom_data[i]));
    end
endgenerate

// Background plaette
rom_async_hex #(
    .WIDTH(BG_COLOR_BITS),
    .DEPTH(64),
    .INIT_F("./image/bg_palette.mem"))
bg_palette_rom (
    .addr(bg_idx),
    .data(bg_rom_data));

// Block mask
rom_async #(
    .WIDTH(2),
    .DEPTH(SPR_PIXELS),
    .INIT_F("./image/blk.mem")
) block_rom (
    .addr(rom_addr),
    .data(block_rom_data));

// BGM
rom_async_hex #(
    .WIDTH(16),
    .DEPTH(92897),
    .INIT_F("music/bgm.mem")
) bgm_rom (
    .addr(bgm_rom_addr),
    .data(bgm_rom_data));

logic [3:0] score[0:3];
logic [1:0] speed;
logic [2:0] block_state[0:19][0:9];
logic [2:0] value, value_next;
```

```
logic [15:0] offset, offset_next;
logic [4:0] row;
logic [3:0] col;
logic [4:0] row_to_del;
logic del_row, write_block, reset_sw, paused, over;
logic [12:0] divider = 0;
logic [12:0] interval[0:2];
logic[23:0] block_color[0:7];

initial begin
    // Initialize block colors.
    block_color[0] = {8'h00, 8'h00, 8'h00};
    block_color[1] = {8'hd5, 8'h00, 8'h00};
    block_color[2] = {8'h4c, 8'haf, 8'h50};
    block_color[3] = {8'h9e, 8'h9e, 8'h9e};
    block_color[4] = {8'hff, 8'heb, 8'h3b};
    block_color[5] = {8'h03, 8'ha8, 8'hf4};
    block_color[6] = {8'hd5, 8'h00, 8'hf9};
    block_color[7] = {8'hff, 8'h98, 8'h00};

    // Initialize music intervals.
    interval[0] = 6250;
    interval[1] = 5000;
    interval[2] = 4000;

    // Initialize parameters
    speed = 1;
    for (int i = 0; i < 4; i++) score[i] = 4'd0;
    del_row = 0;
    write_block = 0;
    reset_sw = 0;
    paused = 0;

    for (int i = 0; i < 20; i++)
        for (int j = 0; j < 10; j++)
            block_state[i][j] = 3'd0;
    // Initialize painting blue.
    block_state[5][1] = 5;
    block_state[5][8] = 5;
    block_state[6][1] = 5;
    block_state[6][2] = 5;
    block_state[6][7] = 5;
    block_state[6][8] = 5;
    block_state[7][1] = 5;
    block_state[7][2] = 5;
    block_state[7][3] = 5;
    block_state[7][4] = 5;
    block_state[7][5] = 5;
    block_state[7][6] = 5;
    block_state[7][7] = 5;
    block_state[7][8] = 5;
    block_state[8][1] = 5;
    block_state[8][3] = 5;
    block_state[8][4] = 5;
    block_state[8][5] = 5;
    block_state[8][6] = 5;
```

```
    block_state[8][8] = 5;
    block_state[9][1] = 5;
    block_state[9][4] = 5;
    block_state[9][5] = 5;
    block_state[9][8] = 5;
    block_state[10][1] = 5;
    block_state[11][1] = 5;
    block_state[12][1] = 5;
    block_state[10][8] = 5;
    block_state[11][8] = 5;
    block_state[12][8] = 5;
    block_state[13][2] = 5;
    block_state[13][7] = 5;
    block_state[14][3] = 5;
    block_state[14][4] = 5;
    block_state[14][5] = 5;
    block_state[14][6] = 5;

    // Initialize painting grey.
    block_state[10][2] = 3;
    block_state[10][3] = 3;
    block_state[10][6] = 3;
    block_state[10][7] = 3;
    block_state[12][4] = 3;
    block_state[12][5] = 3;

    // Initialize painting yellow.
    block_state[12][2] = 4;
    block_state[12][3] = 4;
    block_state[12][6] = 4;
    block_state[12][7] = 4;
    block_state[13][3] = 4;
    block_state[13][4] = 4;
    block_state[13][5] = 4;
    block_state[13][6] = 4;
end

always_ff @(posedge clk) begin
    if (reset_sw) begin
        speed <= 1;
        for (int i = 0; i < 4; i++) score[i] <= 4'd0;
        for (int i = 0; i < 20; i++)
            for (int j = 0; j < 10; j++)
                block_state[i][j] <= 3'd0;
        reset_sw <= 0;
    end
    else if (chipselect && write) begin
        case (address)
        3'h0: begin
            row <= writedata[15:11];
            col <= writedata[10:7];

            case (writedata[6:2])
            // I
            5'b00100, 5'b00110: offset <= 16'b0000000100100011;
            5'b00101, 5'b00111: offset <= 16'b0000010010001100;
```

```
    // O
    5'b01000, 5'b01001, 5'b01010, 5'b01011: offset <= 16'b0000010000010101;
    // T
    5'b01100: offset <= 16'b0000000100100101;
    5'b01101: offset <= 16'b0100000101011001;
    5'b01110: offset <= 16'b0001010001010110;
    5'b01111: offset <= 16'b0000010010000101;
    // L
    5'b10000: offset <= 16'b0000000100100100;
    5'b10001: offset <= 16'b0000000101011001;
    5'b10010: offset <= 16'b0010010001010110;
    5'b10011: offset <= 16'b0000010010001001;
    // J
    5'b10100: offset <= 16'b0000000100100110;
    5'b10101: offset <= 16'b1000000101011001;
    5'b10110: offset <= 16'b0000010001010110;
    5'b10111: offset <= 16'b0000010010000001;
    // Z
    5'b11000, 5'b11010: offset <= 16'b0000000101010110;
    5'b11001, 5'b11011: offset <= 16'b0001010001011000;
    //S
    5'b11100, 5'b11110: offset <= 16'b0001001001000101;
    5'b11101, 5'b11111: offset <= 16'b0000010001011001;
    endcase

    value <= {3{writedata[1]}} & writedata[6:4];
    write_block <= 1;
end
3'h1: begin
    row_to_del <= writedata[4:0];
    del_row <= 1;
end
3'h2: begin
    score[0] <= writedata[3:0];
    score[1] <= writedata[7:4];
    score[2] <= writedata[11:8];
    score[3] <= writedata[15:12];
end
3'h3: begin
    value_next <= writedata[2:0];

    case (writedata[2:0])
    3'd1: offset_next <= 16'b0000000100100011;
    3'd2: offset_next <= 16'b0000010000010101;
    3'd3: offset_next <= 16'b0000000100100101;
    3'd4: offset_next <= 16'b0000000100100100;
    3'd5: offset_next <= 16'b0000000100100110;
    3'd6: offset_next <= 16'b0000000101010110;
    3'd7: offset_next <= 16'b0001001001000101;
    endcase
end
3'h4:
    speed <= writedata[1:0];
3'h5:
    reset_sw <= 1;
3'h6: begin
```

```
        paused <= writedata[0] + 1;
        over <= writedata[1];
    end
    endcase
end

if (write_block) begin
    // Update block state
    block_state[row + offset[15:14]][col + offset[13:12]] <= value;
    block_state[row + offset[11:10]][col + offset[9:8]] <= value;
    block_state[row + offset[7:6]][col + offset[5:4]] <= value;
    block_state[row + offset[3:2]][col + offset[1:0]] <= value;
    write_block <= 0;
end
if (del_row) begin
    // Delete row
    for (int i = 19; i >= 1; i--)
        if (i <= row_to_del) block_state[i] <= block_state[i - 1];
    for (int j = 0; j < 10; j++) block_state[0][j] <= 3'd0;
    del_row <= 0;
end

if (over) begin
    for (int i = 0; i < 20; i++)
            for (int j = 0; j < 10; j++)
                block_state[i][j] = 3'd0;
    // Initialize painting blue.
    block_state[5][1] = 5;
    block_state[5][8] = 5;
    block_state[6][1] = 5;
    block_state[6][2] = 5;
    block_state[6][7] = 5;
    block_state[6][8] = 5;
    block_state[7][1] = 5;
    block_state[7][2] = 5;
    block_state[7][3] = 5;
    block_state[7][4] = 5;
    block_state[7][5] = 5;
    block_state[7][6] = 5;
    block_state[7][7] = 5;
    block_state[7][8] = 5;
    block_state[8][1] = 5;
    block_state[8][3] = 5;
    block_state[8][4] = 5;
    block_state[8][5] = 5;
    block_state[8][6] = 5;
    block_state[8][8] = 5;
    block_state[9][1] = 5;
    block_state[9][4] = 5;
    block_state[9][5] = 5;
    block_state[9][8] = 5;
    block_state[10][1] = 5;
    block_state[11][1] = 5;
    block_state[12][1] = 5;
    block_state[10][8] = 5;
    block_state[11][8] = 5;
```

```
            block_state[12][8] = 5;
            block_state[13][2] = 5;
            block_state[13][7] = 5;
            block_state[14][3] = 5;
            block_state[14][4] = 5;
            block_state[14][5] = 5;
            block_state[14][6] = 5;

            // Initialize painting grey.
            block_state[10][2] = 3;
            block_state[10][3] = 3;
            block_state[10][6] = 3;
            block_state[10][7] = 3;
            block_state[12][4] = 3;
            block_state[12][5] = 3;

            // Initialize painting yellow.
            block_state[12][2] = 4;
            block_state[12][3] = 4;
            block_state[12][6] = 4;
            block_state[12][7] = 4;
            block_state[13][3] = 4;
            block_state[13][4] = 4;
            block_state[13][5] = 4;
            block_state[13][6] = 4;

            divider <= 0;
            bgm_rom_addr <= 0;
            over <= 0;
        end

        // music
        if (divider < interval[speed - 1]) begin
            divider <= divider + paused;
            avalon_streaming_source_l_valid <= 0;
            avalon_streaming_source_r_valid <= 0;
        end
        else begin
            divider <= 0;
            bgm_rom_addr <= bgm_rom_addr + paused;
            if (bgm_rom_addr >= 92897)
                bgm_rom_addr <= 0;
                avalon_streaming_source_l_valid <= paused;
                avalon_streaming_source_r_valid <= paused;
                avalon_left <= bgm_rom_data;
                avalon_right <= bgm_rom_data;
            end
        end

    always_comb begin
        rom_addr = {vcount[3:0], hcount[4:1]};
        bg_addr = {vcount[8:0], hcount[7:1]};
        bg_idx = bg_idx_rom_data[hcount[10:8]][5:0];
        {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff};
        if (VGA_BLANK_n) begin
            // bg
```

```
        {VGA_R, VGA_G, VGA_B} = bg_rom_data;
        // draw sprites
        //blocks
        if (hcount[10:9] == 0 && hcount[8:5] >= 6 && vcount[9:4] >= 5 && vcount[9:4] <= 24)
            {VGA_R, VGA_G, VGA_B} = { 3{block_rom_data[1], {7{block_rom_data[0]}}} } &
block_color[block_state[vcount[9:4] - 5][hcount[8:5] - 6]];
        else if (hcount[10:5] >= 30 && hcount[10:5] <= 33)
            // score number
            if (vcount[9:4] == 15)
                {VGA_R, VGA_G, VGA_B} = {3{digit_rom_data[score[33 - hcount[10:5]]], 5'd0}};
            // speed number
            else if(hcount[10:5] == 30 && vcount[9:4] == 12)
                {VGA_R, VGA_G, VGA_B} = {3{digit_rom_data[speed], 5'd0}};
            // next blocks
            else if ((hcount[10:5] == 30 + offset_next[13:12] && vcount[9:4] == 19 + offset_next[15:14])
                || (hcount[10:5] == 30 + offset_next[9:8] && vcount[9:4] == 19 + offset_next[11:10])
                || (hcount[10:5] == 30 + offset_next[5:4] && vcount[9:4] == 19 + offset_next[7:6])
                || (hcount[10:5] == 30 + offset_next[1:0] && vcount[9:4] == 19 + offset_next[3:2]))
                 // Set the next value block's color
                 {VGA_R, VGA_G, VGA_B} = {3{block_rom_data[1], {7{block_rom_data[0]}}}} &
block_color[value_next];
      end
   end
endmodule

module vga_counters(
 input logic                                                                                                clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic                                        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0            1279       1599 0
 *             _______________           ________
 * ____________|    Video      |_____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _____________________     _____________
 * |____|       VGA_HS          |____|
 */
  // Parameters for hcount
  parameter HACTIVE      = 11'd 1280,
            HFRONT_PORCH = 11'd 32,
            HSYNC        = 11'd 192,
            HBACK_PORCH  = 11'd 96,
            HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                           HBACK_PORCH; // 1600

  // Parameters for vcount
  parameter VACTIVE      = 10'd 480,
            VFRONT_PORCH = 10'd 10,
            VSYNC        = 10'd 2,
            VBACK_PORCH  = 10'd 33,
```

```
        VTOTAL      = VACTIVE + VFRONT_PORCH + VSYNC +
                      VBACK_PORCH; // 525

  logic endOfLine;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)          hcount <= 0;
    else if (endOfLine) hcount <= 0;
    else                                                                    hcount <= hcount + 11'd 1;

  assign endOfLine = hcount == HTOTAL - 1;

  logic endOfField;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)          vcount <= 0;
    else if (endOfLine)
      if (endOfField)   vcount <= 0;
      else              vcount <= vcount + 10'd 1;

  assign endOfField = vcount == VTOTAL - 1;

  // Horizontal sync: from 0x520 to 0x5DF (0x57F)
  // 101 0010 0000 to 101 1101 1111
  assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                                                                          !(hcount[7:5] == 3'b111));
  assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

  assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

  // Horizontal active: 0 to 1279     Vertical active: 0 to 479
  // 101 0000 0000  1280                                                  01 1110 0000  480
  // 110 0011 1111  1599                                                  10 0000 1100  524
  assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &


                                                                  !( vcount[9] | (vcount[8:5] == 4'b1111) );

  /* VGA_CLK is 25 MHz
   *
   * clk50    __|  |__|  |__|
   *
   *
   *           _____      __
   * hcount[0]__|     |_____|
   */
  assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule

module rom_async #(
  parameter WIDTH=8,
  parameter DEPTH=256,
  parameter INIT_F="")
  (
    input wire logic [ADDRW-1:0] addr,
```

```
   output     logic [WIDTH-1:0] data
  );

  localparam ADDRW=$clog2(DEPTH);
  logic [WIDTH-1:0] memory [DEPTH];

  initial begin
    if (INIT_F != 0) begin
      $display("Creating rom_async from init file '%s'.", INIT_F);
      $readmemb(INIT_F, memory);
    end
  end

  always_comb data = memory[addr];
endmodule

module rom_async_hex #(
  parameter WIDTH=8,
  parameter DEPTH=256,
  parameter INIT_F="")
  (
    input wire logic [ADDRW-1:0] addr,
    output     logic [WIDTH-1:0] data
  );

  localparam ADDRW=$clog2(DEPTH);
  logic [WIDTH-1:0] memory [DEPTH];

  initial begin
    if (INIT_F != 0) begin
      $display("Creating rom_async from init file '%s'.", INIT_F);
      $readmemh(INIT_F, memory);
    end
  end

  always_comb data = memory[addr];
endmodule
```

Software

Tetris.h:

```
#ifndef _TETRIS_H
#define _TETRIS_H

#include <linux/ioctl.h>

typedef struct
{
                                                                    unsigned short p;
} tetris_arg_t;

typedef struct
{
                                                                        int x;
                                                                        int y;
```

```
    int type;
    int rotation;
} tetris_block;

#define TETRIS_MAGIC 'q'

/* ioctls and their arguments */
#define TETRIS_WRITE_POSITION _IOW(TETRIS_MAGIC, 1, tetris_arg_t *)
#define TETRIS_DEL_ROW  _IOW(TETRIS_MAGIC, 2, tetris_arg_t *)
#define TETRIS_WRITE_SCORE  _IOW(TETRIS_MAGIC, 3, tetris_arg_t *)
#define TETRIS_WRITE_NEXT  _IOW(TETRIS_MAGIC, 4, tetris_arg_t *)
#define TETRIS_WRITE_SPEED  _IOW(TETRIS_MAGIC, 5, tetris_arg_t *)
#define TETRIS_RESET  _IOW(TETRIS_MAGIC, 6, tetris_arg_t *)
#define TETRIS_PAUSE  _IOW(TETRIS_MAGIC, 7, tetris_arg_t *)
#endif
```

Tetris.c:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#include "Tetris.h"
#include "Tool.h"

#define DRIVER_NAME "Tetris"

struct tetris_dev
{
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   tetris_arg_t t_arg;
} dev;

// Write the args
static void write(tetris_arg_t *t_arg, int flag)
{
   iowrite16(t_arg->p, dev.virtbase + 2 * flag);
   dev.t_arg = *t_arg;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
```

```
static long tetris_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
   tetris_arg_t vla;

   switch (cmd)
   {
  case TETRIS_WRITE_POSITION:
   if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
  write(&vla, 0);
  break;
  case TETRIS_DEL_ROW:
  if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
  write(&vla, 1);
   break;
  case TETRIS_WRITE_SCORE:
   if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
   write(&vla, 2);
   break;
  case TETRIS_WRITE_NEXT:
   if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
   write(&vla, 3);
   break;
  case TETRIS_WRITE_SPEED:
  if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
  write(&vla, 4);
   break;
   case TETRIS_RESET:
  if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
  write(&vla, 5);
   break;
   case TETRIS_PAUSE:
  if (copy_from_user(&vla, (tetris_arg_t*) arg, sizeof(tetris_arg_t))) return -EACCES;
  write(&vla, 6);
   break;
  default: return -EINVAL;
   }

   return 0;
}

/* The operations our device knows how to do */
static const struct file_operations tetris_fops =
{
   .owner = THIS_MODULE,
   .unlocked_ioctl = tetris_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice tetris_misc_device =
{
   .minor                                                                  = MISC_DYNAMIC_MINOR,
   .name                                                                        = DRIVER_NAME,
   .fops                                                                          = &tetris_fops,
};

/*
```

```
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init tetris_probe(struct platform_device *pdev)
{
   int ret;
   ret = misc_register(&tetris_misc_device);

   /* Get the address of our registers from the device tree */
   ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
   if (ret)
   {
                                                                                    ret = -ENOENT;
                                                                                goto out_deregister;
   }

   /* Make sure we can use these registers */
   if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME) == NULL)
   {
                                                                                    ret = -EBUSY;
                                                                                goto out_deregister;
   }

   /* Arrange access to our registers */
   dev.virtbase = of_iomap(pdev->dev.of_node, 0);
   if (dev.virtbase == NULL)
   {
                                                                                    ret = -ENOMEM;
                                                                            goto out_release_mem_region;
   }

   return 0;

   out_release_mem_region:
                                                        release_mem_region(dev.res.start, resource_size(&dev.res));
   out_deregister:
                                                                        misc_deregister(&tetris_misc_device);
   return ret;
}

/* Clean-up code: release resources */
static int tetris_remove(struct platform_device *pdev)
{
   iounmap(dev.virtbase);
   release_mem_region(dev.res.start, resource_size(&dev.res));
   misc_deregister(&tetris_misc_device);
   return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id tetris_of_match[] =
{
   { .compatible = "csee4840,vga_tetris-1.0" },
   {},
};
```

```
MODULE_DEVICE_TABLE(of, tetris_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver tetris_driver =
{
   .driver    = {
                                                                    .name    = DRIVER_NAME,
                                                                    .owner    = THIS_MODULE,
                                                          .of_match_table = of_match_ptr(tetris_of_match),
   },
   .remove    = __exit_p(tetris_remove),
};

/* Called when the module is loaded: set things up */
static int __init tetris_init(void)
{
   pr_info(DRIVER_NAME ": init\n");
   return platform_driver_probe(&tetris_driver, tetris_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit tetris_exit(void)
{
   platform_driver_unregister(&tetris_driver);
   pr_info(DRIVER_NAME ": exit\n");
}

module_init(tetris_init);
module_exit(tetris_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CSEE4840, Columbia University");
MODULE_DESCRIPTION("Tetris driver");
```

Tool.h:

```
#ifndef _TOOL_H
#define _TOOL_H

extern void emptyBlocks(void);
extern void printBlocks(void);
extern int flip(int, int);
extern int testLine(int);
extern int* getShape(int, int);
extern void set_score(int, const int);
extern void set_speed(int, const int);
extern int set_next(int, int);
extern void set_del_row(int, const int);
extern void set_block(int, const int, const int, const int, const int, const int);
extern void set_pause(int, const int);

#endif
```

Tool.c:

```
#include <fcntl.h>
#include <unistd.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "Tetris.h"

int blocks[20][10];

void emptyBlocks() { memset(blocks, '\0', 240 * sizeof(int)); }

// For debug only.
void printBlocks()
{
   printf("----------\n");
   int i = 0;
   for (i = 0; i < 200; i++)
   {
      int px = i % 10;
      int py = i / 10;
      if (px == 0) printf("|");
      if (blocks[py][px] == 1) printf("*");
      else if (blocks[py][px] == 0) printf(" ");
      else printf("%d", blocks[py][px]);
      if (px == 9) printf("|\n");
   }
   printf("----------\n");
}


// Flip the target block and return the flipped value.
int flip(int r, int c)
{
   blocks[r][c] = 1 - blocks[r][c];
   return blocks[r][c];
}

// Test a line and return if the line is connected or not.
int testLine(int r)
{
   int c = 0;
   for (c = 0; c < 10; c++)
   {
      if (blocks[r][c] == 0) return 0;
   }
   for (c = r; c > 0; c--) memcpy(blocks[c], blocks[c - 1], sizeof(int) * 10);
   memset(blocks, '\0', sizeof(int) * 10);
   return 1;
}
```

```
// Assign the 4 bits to 1 in the shape.
void shapeAssign(int a, int b, int c, int d, int* shape)
{
   shape[a] = 1;
   shape[b] = 1;
   shape[c] = 1;
   shape[d] = 1;
}

// Return the 4*4 matrix of the teris shape.
int* getShape(int type, int rotation)
{
   static int r[16];
   memset(r, '\0', 16 * sizeof(int));

   int merged_type;
   if (type == 0 || type >= 5) merged_type = 10 * type + rotation % 2;
   else if (type == 1) merged_type = 10;
   else merged_type = 10 * type + rotation % 4;

   switch (merged_type)
   {
      // I for type 0
      case 0:
         shapeAssign(0, 1, 2, 3, r);
         break;
      case 1:
         shapeAssign(0, 4, 8, 12, r);
         break;
      // O for type 1
      case 10:
         shapeAssign(0, 1, 4, 5, r);
         break;
      // T for type 2
      case 20:
         shapeAssign(0, 1, 2, 5, r);
         break;
      case 21:
         shapeAssign(1, 4, 5, 9, r);
         break;
      case 22:
         shapeAssign(1, 4, 5, 6, r);
         break;
      case 23:
         shapeAssign(0, 4, 5, 8, r);
         break;
      // L for type 3
      case 30:
         shapeAssign(0, 1, 2, 4, r);
         break;
      case 31:
         shapeAssign(0, 1, 5, 9, r);
         break;
      case 32:
         shapeAssign(2, 4, 5, 6, r);
```

```
            break;
        case 33:
            shapeAssign(0, 4, 8, 9, r);
            break;
        // J for type 4
        case 40:
            shapeAssign(0, 1, 2, 6, r);
            break;
        case 41:
            shapeAssign(1, 5, 8, 9, r);
            break;
        case 42:
            shapeAssign(0, 4, 5, 6, r);
            break;
        case 43:
            shapeAssign(0, 1, 4, 8, r);
            break;
        // Z for type 5
        case 50:
            shapeAssign(0, 1, 5, 6, r);
            break;
        case 51:
            shapeAssign(1, 4, 5, 8, r);
            break;
        // N for type 6
        case 60:
            shapeAssign(1, 2, 4, 5, r);
            break;
        case 61:
            shapeAssign(0, 4, 5, 9, r);
            break;
    }
    return r;
}

// Set the score value.
void set_score(int tetris_fd, const int score)
{
    int temp = score > 9999? 9999 : score;
    tetris_arg_t vla;
    vla.p = 0;

    for(int i = 0; i < 4; i++)
    {
        vla.p += ((temp % 10) << (4 * i));
        temp /= 10;
    }
    if (ioctl(tetris_fd, TETRIS_WRITE_SCORE, &vla))
    {
        perror("ioctl(TETRIS_WRITE_SCORE) failed");
        return;
    }
}

// Set the speed of the game.
void set_speed(int tetris_fd, const int speed)
```

```
{
    tetris_arg_t vla;
    vla.p = speed;
    if (ioctl(tetris_fd, TETRIS_WRITE_SPEED, &vla))
    {
        perror("ioctl(TETRIS_WRITE_SPEED) failed");
        return;
    }
}

// Set the next block.
int set_next(int tetris_fd, int current)
{
    int type = rand() % 7;
    if (type == current) type = (type + 1 + rand() % 6) % 7;
    tetris_arg_t vla;
    vla.p = type + 1;
    if (ioctl(tetris_fd, TETRIS_WRITE_NEXT, &vla))
    {
        perror("ioctl(TETRIS_WRITE_NEXT) failed");
        return -1;
    }
    return type;
}

// Set the lines to erase.
void set_del_row(int tetris_fd, const int count)
{
    tetris_arg_t vla;
    vla.p = count;
    if (ioctl(tetris_fd, TETRIS_DEL_ROW, &vla))
    {
        perror("ioctl(TETRIS_DEL_ROW) failed");
        return;
    }
}

// Send the information about the new block position.
void set_block(int tetris_fd, const int x, const int y, const int type, const int rotation, const int value)
{
    tetris_arg_t vla;
    vla.p = (x << 11) + (y << 7) + ((type % 7 + 1) << 4) + ((rotation % 4) << 2) + (value << 1);
    if (ioctl(tetris_fd, TETRIS_WRITE_POSITION, &vla))
    {
        perror("ioctl(TETRIS_WRITE_POSITION) failed");
        return;
    }
}

// Pause or end game, 1 for pause, 11 (3) for end + pause.
void set_pause(int tetris_fd, const int paused)
{
    tetris_arg_t vla;
    vla.p = paused;
    if (ioctl(tetris_fd, TETRIS_PAUSE, &vla))
    {
```

```
        perror("ioctl(TETRIS_PAUSE) failed");
        return;
    }
}
```

game.c:

```
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "Tetris.h"
#include "Tool.h"

// The block of the game.
int score = 0;
int downCount = 0;
int speed = 0;
int tetris_fd;
int pauseFlag = 1;
int ignoreNext = 0;
int down_key = 0;
int gameEnd = 1;

// A correct assign should return either 0 or 4.
int assign(int x, int y, int type, int rotation)
{
    int* shape = getShape(type, rotation);
    int sum = 0, i;
    for (i = 0; i < 16; i++)
    {
        // Flip the items in the shape.
        if (shape[i] == 1)
        {
            int dx = x + i % 4;
            int dy = y + i / 4;
            if (dx < 10 && dx >= 0 && dy < 20 && dy >= 0) sum += flip(dy, dx);
        }
    }
    return sum;
}

// Rotate the current shape by 90 degree clockwise.
void rotate(tetris_block *input)
{
    assign(input->x, input->y, input->type, input->rotation);
    if (assign(input->x, input->y, input->type, input->rotation + 1) != 4)
    {
```

```
        assign(input->x, input->y, input->type, input->rotation);
        assign(input->x, input->y, input->type, input->rotation + 1);
        return;
    }

    // Successfully rotate.
    set_block(tetris_fd, input->y, input->x, input->type, input->rotation, 0);
    set_block(tetris_fd, input->y, input->x, input->type, input->rotation + 1, 1);
    input->rotation = (input->rotation + 1) % 4;
}

// Return -1 if unable to create.
int testAndCreate(int type, tetris_block *input)
{
    int line_num = 0;

    // Test for any existing lines to erase.
    int count = 0;
    for (count = 0; count < 20; count++)
    {
        if (testLine(count) == 1)
        {
            if (ignoreNext == 0)
            {
                usleep(100000);
                ignoreNext = 1;
            }
            line_num += 1;
            set_del_row(tetris_fd, count);
        }
    }

    // Create a new shape.
    input->y = 0;
    input->rotation = 0;
    input->type = type;
    input->x = type == 0? 3 : 4;
    if (assign(input->x, 0, type, 0) != 4)
    {
        assign(input->x, 0, type, 0);
        return -1;
    }
    else set_block(tetris_fd, 0, input->x, type, 0, 1);

    // Set the score.
    score += line_num * (speed + line_num);
    set_score(tetris_fd, score);

    return line_num;
}

// Reset the parameters.
int reset(tetris_block *input)
{
    srand(time(NULL));
    emptyBlocks();
```

```
    tetris_arg_t vla;
    vla.p = 0;
    if (ioctl(tetris_fd, TETRIS_RESET, &vla))
    {
        perror("ioctl(TETRIS_RESET) failed");
        return -1;
    }

    downCount = 0;
    speed = 0;
    score = 0;
    down_key = 0;
    pauseFlag = 0;
    gameEnd = 0;
    testAndCreate(rand() % 7, input);

    set_pause(tetris_fd, 0);
    return set_next(tetris_fd, input->type);
}

// xy = 0 for x, xy = 1 for y.
int move(int d, int xy, tetris_block *input)
{
    int new_x = input->x + d * (1 - xy);
    int new_y = input->y + d * xy;
    if (assign(input->x, input->y, input->type, input->rotation) != 0)
    {
        assign(input->x, input->y, input->type, input->rotation);
        return -1;
    }
    else if (assign(new_x, new_y, input->type, input->rotation) != 4)
    {
        assign(input->x, input->y, input->type, input->rotation);
        assign(new_x, new_y, input->type, input->rotation);
        return 1;
    }

    set_block(tetris_fd, input->y, input->x, input->type, input->rotation, 0);
    set_block(tetris_fd, new_y, new_x, input->type, input->rotation, 1);
    input->x = new_x;
    input->y = new_y;

    return 0;
}

// Joystick control.
int main(int argc, char *argv[])
{
    char input_dev[] = "/dev/input/event0";
    static const char filename[] = "/dev/Tetris";
    const int input_size = 512;
    unsigned char input_data[input_size];
    struct pollfd fds[1];

    // Open the device.
```

```
    if ((fds[0].fd = open(input_dev, O_RDONLY | O_NONBLOCK)) == -1 || (tetris_fd = open(filename, O_RDWR))
== -1)
    {
        fprintf(stderr, "Check the joystick or the module\n");
        return -1;
    }

    int i = 0;
    int key_type = 0, key = 0, value = 0;
    tetris_block block;
    int new_type = 0;

    fds[0].events = POLLIN;
    int downThreshold[3] = {120, 80, 40};

    while(1)
    {
        int ret = poll(fds, 1, 5);

        /* The key values
        * A: 33
        * B: 34
        * X: 32
        * Y: 35
        * L: 36
        * R: 37
        * SELECT: 40
        * START: 41
        * y-axis: 1
        * x-axis: 0
        */
        if (ignoreNext == 1) ignoreNext = 0;
        else if (ret > 0)
        {
            key = -1;
            value = -1;
            memset(input_data, 0, input_size);
                                                    ssize_t read_res = read(fds[0].fd, input_data, input_size);

                                                                for(i = 0; i < read_res / 16 - 1; i++)
                                                                                {

                key_type = input_data[i * 16 + 8];
                if(key_type == 1)
                {
                    // button
                    key = input_data[i * 16 + 10];
                    value = input_data[i * 16 + 12];
                }
                else if(key_type == 3)
                {
                    //axis
                    key = input_data[i * 16 + 10];
                    value = input_data[i * 16 + 12];
                    switch(value){
                        case 0:
                            value = -1;
```

```
                break;
            case 255:
                value = 1;
                break;
            default:
                value = 0;
                break;
        }
    }


                                                                    // Received a new button press.
    if (value == 1)
    {
        // A, rotate.
        if (key == 33 && pauseFlag == 0) rotate(&block);
        // L & R, move.
        else if (key == 36 && pauseFlag == 0) move(-1, 0, &block);
        else if (key == 37 && pauseFlag == 0) move(1, 0, &block);
        // Start, pause or reset the game status.
        else if (key == 41)
        {
            if (gameEnd == 1) new_type = reset(&block);
            else
            {
                pauseFlag = 1 - pauseFlag;
                set_pause(tetris_fd, pauseFlag);


                                                                                    }
        }
        // Down, speed up to the bottom.
        else if (key == 1 && pauseFlag == 0) down_key = 1;
        // Select, change speed.
        else if (key == 40 && pauseFlag == 0)
        {
            speed = (speed + 1) % 3;
            set_speed(tetris_fd, speed + 1);
        }
    }
    else if (key == 1) down_key = 0;
                                                                                    }
}
else if (pauseFlag == 0 && (downCount++ >= downThreshold[speed] || (down_key == 1 && downCount++
>= 10)))
{
    downCount = 0;
    if (move(1, 1, &block) == 1)
    {
        if (testAndCreate(new_type, &block) < 0)
        {
            printf("Game Over!\n");
            gameEnd = 1;

                                                                usleep(1000000);
```

```
                                                                    set_pause(tetris_fd, 3);
            }
            else
            {
               new_type = set_next(tetris_fd, new_type);
               down_key = 0;
            }
         }
      }
   }

   close(fds[0].fd);
   return 0;
}
```