Tetris Game Based on De1-Soc

CSEEW4840 Embedded Systems

Author: Xinzi Yu(xy2590) , Chuyi Jiang(cj2792) **Department:** Electrical Engineering, Columbia Engineering **Instructor:** Prof.Stephen A. Edwards

Date: May 4, 2024

Contents

1 Introduction

1.1 Project Overview

We developed a version of classic tetris game, in which players strategically position varying shapes of blocks that drop into the game area. As these lines are completed, they vanish, awarding points to the player, who then uses the newly freed space to maneuver additional pieces. The game concludes once no further space is available for new pieces to fit. In this project, we have introduced three difficulty settings, enhancing the challenge by accelerating the descent of the tetrominoes and the tempo of the background music with each level. This version of Tetris was programmed on a DE1-SoC FPGA board, and it employs a PS2 joystick, enabling more precise and swift game-play management.

1.2 User Guide

Here's how to get started and play the game:

1. **Getting Started**

Power Up: Turn on your gaming console connected to the DE1-SoC FPGA board.

Launch Game: On terminal, move the game folder and enter './game'.

2. **Game Controls** The Fig.1 shows the details of the controller as an instrument. **Joystick:** Use the joystick on the PS2 controller to move the tetrominoes left, right, down and rotation.
START Button: Press this button to begin the game at the basic difficulty level with normal speed background music (BGM).
SELECT Button: Press this button to cycle through the difficulty levels. Each press increases the speed of the falling blocks and the tempo of the BGM, enhancing the challenge.

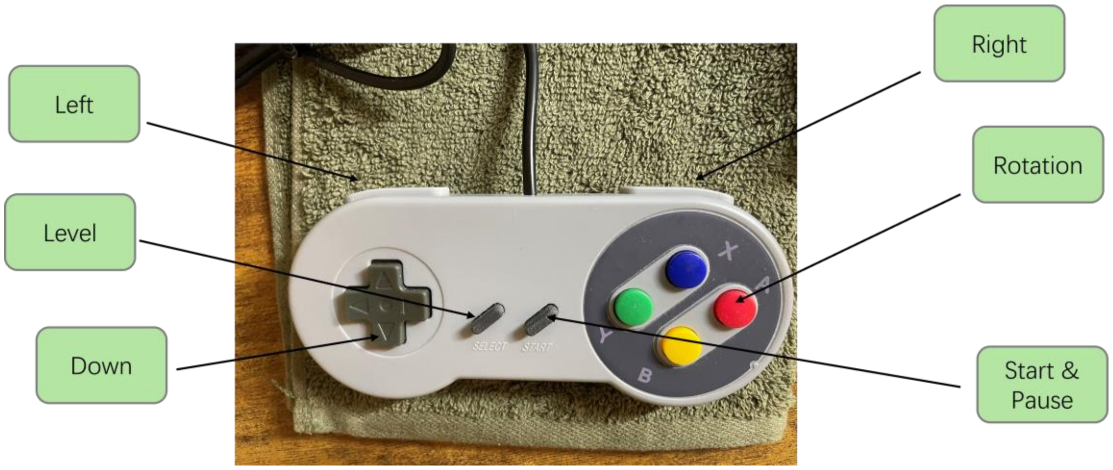


Figure 1: joysticks controller

3. **Gameplay**
Objective: Position the differently shaped blocks, or tetrominoes, that descend onto the playing field to complete lines. Every line you complete will disappear, granting you points and clearing space for more blocks.
Block Shapes: There are 7 unique block shapes to manipulate and fit into the playing field.
Difficulty Levels: The game features three levels of difficulty:
Level 1: Basic level with normal speed falling blocks and BGM. Level 2:
Intermediate difficulty with faster blocks and quicker BGM.
Level 3: Advanced difficulty where blocks drop very quickly, requiring fast and accurate decisions. The BGM at this level is the fastest, indicating a high-adrenaline gameplay environment.

4. **Tips for Success**
Anticipate: Plan where to place blocks based on the shapes and upcoming spaces.

React: As the game speeds up, quick and precise movements are crucial.
Strategize: Try to clear multiple lines at once to score more points and maintain manageable game- play speed.

5. Ending the Game
The game ends when you can no longer fit any new blocks into the playing field. Aim to reach the highest score before reaching this point!

2 System Block Diagram

The whole system, as shown in Figure.1, are composed of Input, Software, Hardware and Output sections.

In the software part, the structure revolves around an infinite loop with timeouts to simulate game frames. Each frame begins by processing joystick inputs, with game dynamics represented through various parameters. The currently active Tetris block is managed using a block object, with most game interactions handled through function-based methods. Once inputs are processed, the relevant data is communicated to the hardware using predefined protocols. Blocks descend automatically by counting the elapsed frames.

The hardware is segmented into VGA and audio components. The VGA module retrieves and displays images, based on the data held in specific registers. For audio, an approximately 6-second track is looped as background music, with playback speed varying according to the game's difficulty—faster music for higher levels.

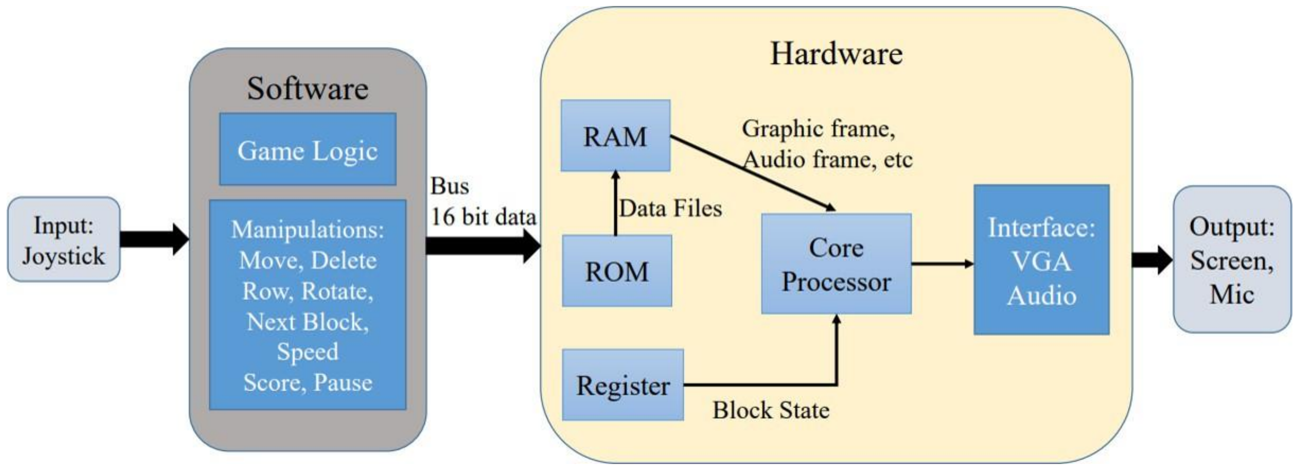


Figure 2: System Block Diagram

3 Software Design

In this section, we delve deep into the intricate design elements of the Tetris game software, encom- passing the core game logic and the interface with input devices and hardware.

3.1 Game Logic

As the flowchart in the following figure shows, the Tetris game begins with initialization, followed by the random generation of a block shape. The generated block automatically descends, but can be manipulated via joystick control to rotate and translate. Whether the block descends automatically or is manually controlled, the block's state is updated. After each state update, the game checks if the block has reached the bottom. If it hasn't, the block continues to descend automatically. If it has reached the bottom, the game evaluates if it is over. If the game is over, it ends and the score is updated. If not, the top row is cleared, and both the score and block state are updated.

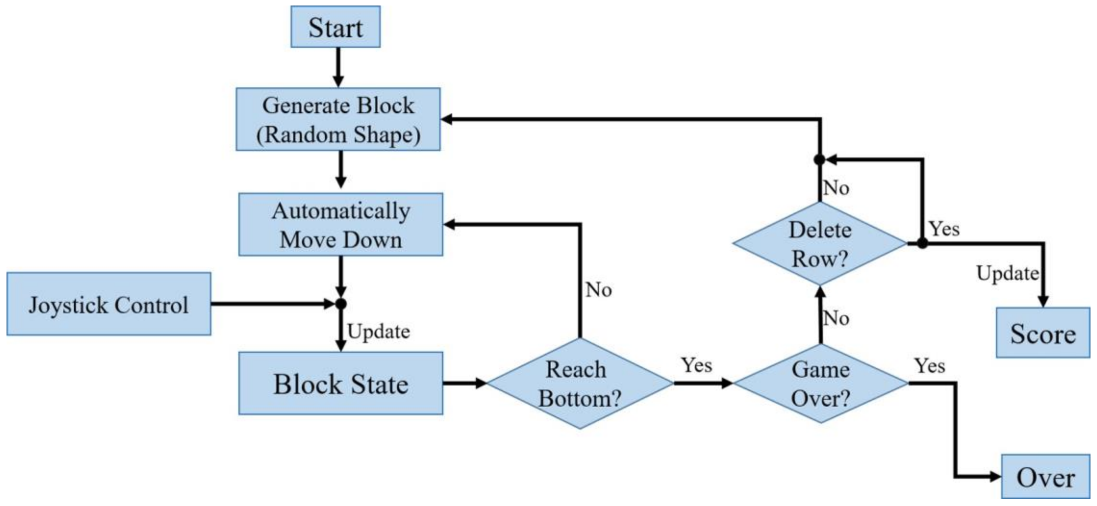


Figure 3: Flowchart of Game Logic

3.2 Implementation

1. Initialization.

The game board represented by the blocks array is initialized by the function emptyBlocks(), which iterates over each cell in the array and sets its value to zero, indicating an empty space on the game board.

2. Block Manipulation.

1).[Block State] We use function flip(int r, int c) to toggle the state of the block located at position (r, c) on the game board. If the block is currently filled, it becomes empty, and if it's empty, it becomes filled.
2).[Next Block] By using set next(int tetris fd, int current), we determines the type of the next Tetris block to be displayed on the game board. It randomly selects a block type different from the current one and transmits this information to the hardware device.
3).[Line Deleting] In function testLine(int r), it examines whether a specific row r on the game board is completely filled with blocks. If all cells in the row are occupied, it returns true; otherwise, it returns false.

4. Tetris Shape Generation.

1).[Block Shape] The function shapeAssign(int a, int b, int c, int d, int* shape) assigns a Tetris shape to the provided array shape. It sets the elements at indices a, b, c, and d to 1, representing filled blocks within the shape.
2).[Shape Placement] We write getShape(int type, int rotation) to generates the 4x4 matrix rep- resenting a Tetris shape based on its type and rotation. It computes the shape's configuration and returns it as a matrix for placement on the game board.
3).[Block Display]The function printBlocks() aids in debugging by printing the current state of the game board to the console. It displays each cell as either filled (represented by '*') or empty (represented by a space), providing a visual representation of the game board's layout

5. Game Parameter Control.

1).[Score] To update the game score displayed on the Tetris hardware device, we applied function set score(int tetris fd, const int score), which converts the score to a suitable format, such as a 4-digit number, and sends it to the device via the ioctl system call.

2).[Speed] In function set speed(int tetris fd, const int speed) , the speed of the game is adjusted, controlling the rate at which Tetris blocks descend on the game board. It communicates the new speed setting to the hardware device using the ioctl function.
3).[Pause] We employed the set pause function, which communicates directly with the hardware to toggle the game's state between paused and active. It sends a specific command through a system call, which instructs the hardware to alter its operational status based on our input. If this system call encounters any issues, it triggers an error message, clearly indicating a failure to execute the pause command. This mechanism ensures robust and responsive control over the game's pause functionality, facilitating a seamless interaction between the software and the hardware.

6. Joystick Control

In our Tetris game, joystick control enables real-time interaction. Device files for the joystick and game open in non-blocking mode to keep the game responsive. A loop with a poll system call monitors joystick inputs, categorizing them into button presses and axis movements. Button presses control block rotation, horizontal movement, game pausing, and speed adjustment. Axis movements quicken block descent.
Each joystick button is linked to a specific function, and the y-axis adjusts the descent speed, influencing gameplay pace based on player input. The game responds to these inputs by executing game functions, clearing lines, updating scores, or checking for game over conditions when the bottom is reached.

3.3 I/O Communication

We utilized the iowrite16 function for communication between software and hardware. The 'e' value in the communication protocol signifies whether a block should be erased or a new block should be created. By adopting this method, we can substantially reduce the number of required communications from eight to just two, thereby not only simplifying the update process but also significantly decreasing the computational load on the hardware. The table below outlines how various instructions and data are encoded within a 16-bit address space for different operations such as block state management, row deletion, score updates, block generation, game speed adjustment, system reset, and pause or game-over status.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Remark
0	row					column				type			rotation				Block Information
1												num for deleted row					Row to Delete
2	score 1				score 2				score 3				score 4				Scores
3														next type			Next Block Type
4															speed		Speed
5															reset		Reset
6															o	p	Pause or Over

Table 1: I/O Interface

4 Hardware Design

4.1 Hardware Block Diagram

The block diagram of hardware is shown in the following figure.

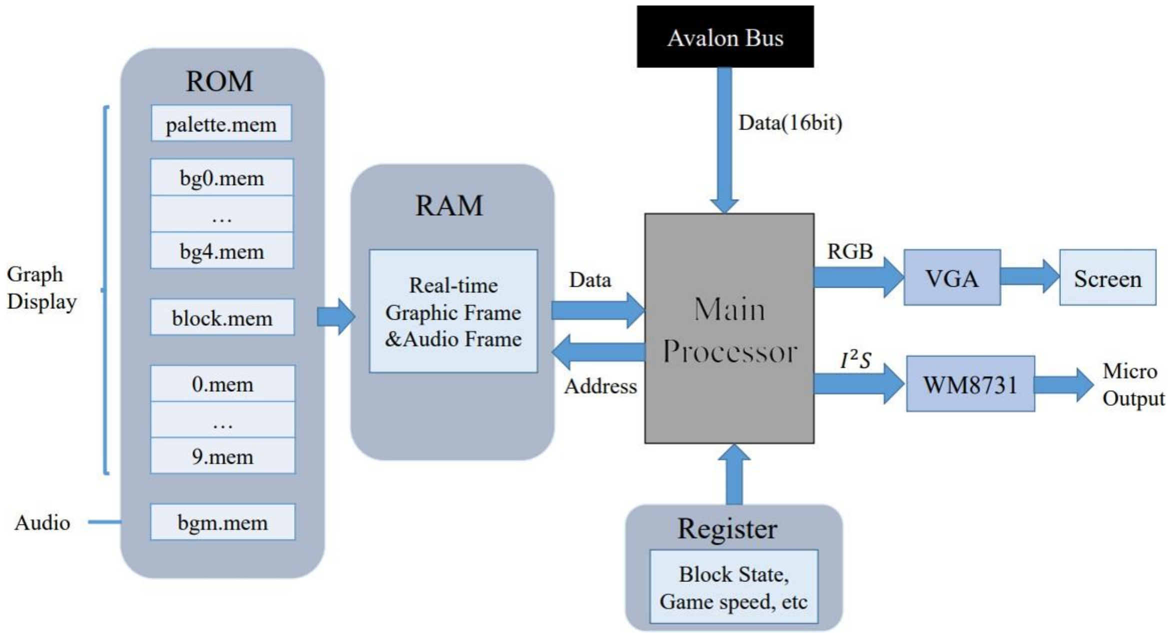


Figure 4: Hardware Block Diagram

4.2 Graphic Design

This section will discuss the display of the Background Image, Block Image, and Number Image, as well as the calculation of storage space for the Graphic Design component. In this part, we utilize .mem files to store image display information. The entire page size is set at 640×512 pixels. It is subdivided into a grid of 40×32, with each cell measuring 16×16 pixels. This size is consistent across each grid comprising the blocks in the game and the size used to display each number. As we implemented ram to store the information, the address is used by hcount and vcount when we need to get relative data form RAM. As hcount is 10:1 and vcount is 9:0,it is sufficient to cover all address we need. In each clock, the code will scan all data in hcount and vcount. That means we could use the value as the address.

4.2.1 Background Image

1. Color Palette.
To conserve storage space, we applied a K-Means clustering algorithm in Python to process the color scheme of the background image, reducing the number of colors used to 64. The analysis of the algorithm is as follows.
We employ the RGB color model to generate colors, with each channel occupying 8 bits, and each color totaling 24 bits as illustrated in the diagram below.

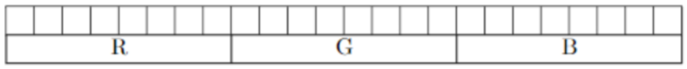


Figure 5: Memory Allocation in RGB Color Model

2. Display
We divided a 640×512 background into 5 equal sections, each measuring 128×480(as shown in the figure below). However, due to the design of the FPGA on-board memory, each image actually has the dimensions of 128×512. In displaying the images, to save space, we use 6-bit storage for the color index of each pixel, which saves a significant amount of space compared to storing 24 bits for each pixel's color directly.

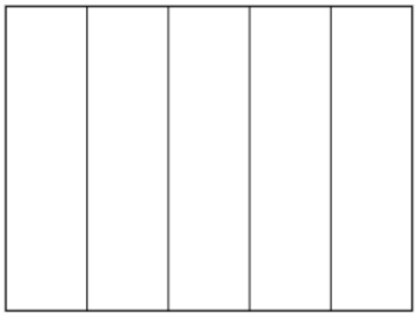


Figure 6: Background Section Division The

interface display is as shown in the following figure.

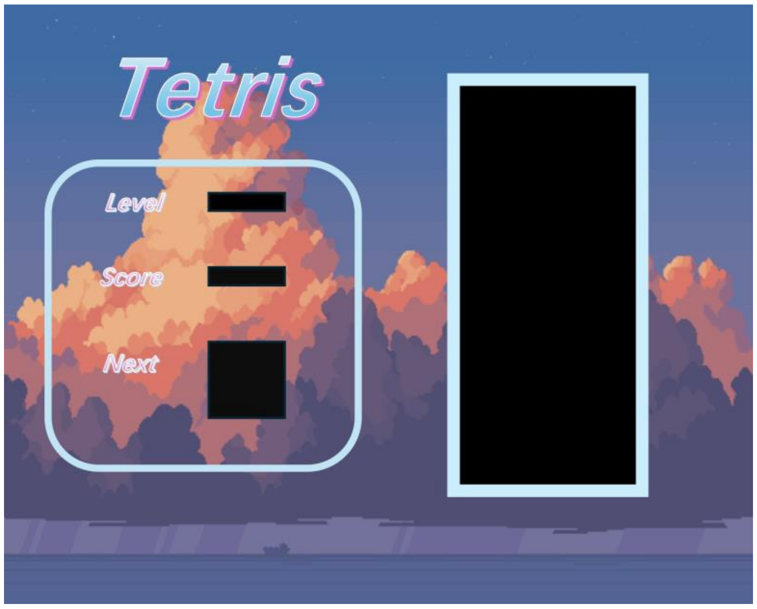


Figure 7: Background Display

4.2.2 Block Image

For the display of blocks, each block is sized at 16 × 16 pixels. We store the colors for seven different Tetris block shapes in seven registers, each color encoded in 24-bit RGB format. Additionally, for aesthetic purposes, we utilize grayscale extraction to determine the color index of the block, represented by 2 bits. This is achieved through the algorithm: [3*{*block rom data[1], *{*7*{*block rom data[0]*}}* & block color[block state[vcount[9:4] - 5][hcount[8:5] - 6]]]. The color index is then matched to the corresponding color in the palette, allowing for different colors within each block and its borders, as illustrated in the figure below.

Figure 8: Block Display

4.2.3 Number Image

For the display of numbers, the approach is similar to that used for block displays, still utilizing a 16×16 grid. The difference lies in the use of 3 bits to represent the color of each pixel. Regarding the style of the numbers, we have adopted the same format used by previous students, as illustrated in the figure below.

Figure 9: Block Display

4.2.4 Storage Allocation

Based on the analysis above, it is straightforward to calculate the storage space occupied by each of the three parts of the graph display, as well as the total storage space used, as shown in the following Table.2.

Table 2: Storage Space - Graphic Design

Component	Dimensions	Calculation
Background	640×512	5×128×512×8+64×24 = 2622976
Block	16×16	16×16×2 = 512
Numbers	16×16	10×16×16×3 = 7680
Total		**2631168**

4.3 Audio Design

4.3.1 Audio Processing and Playback

1. **Audio Processing.** The audio configuration is set to a 16-bit resolution, 8 kHz sample rate, and monaural output. This setup provides a balance between audio quality and system resource utilization, ideal for DE1-SoC. The process begins with the conversion of audio files from the standard .wav format to a .mem format. This transformation is crucial for compatibility with the FPGA's memory handling capabilities, which facilitates direct audio data manipulation and playback.

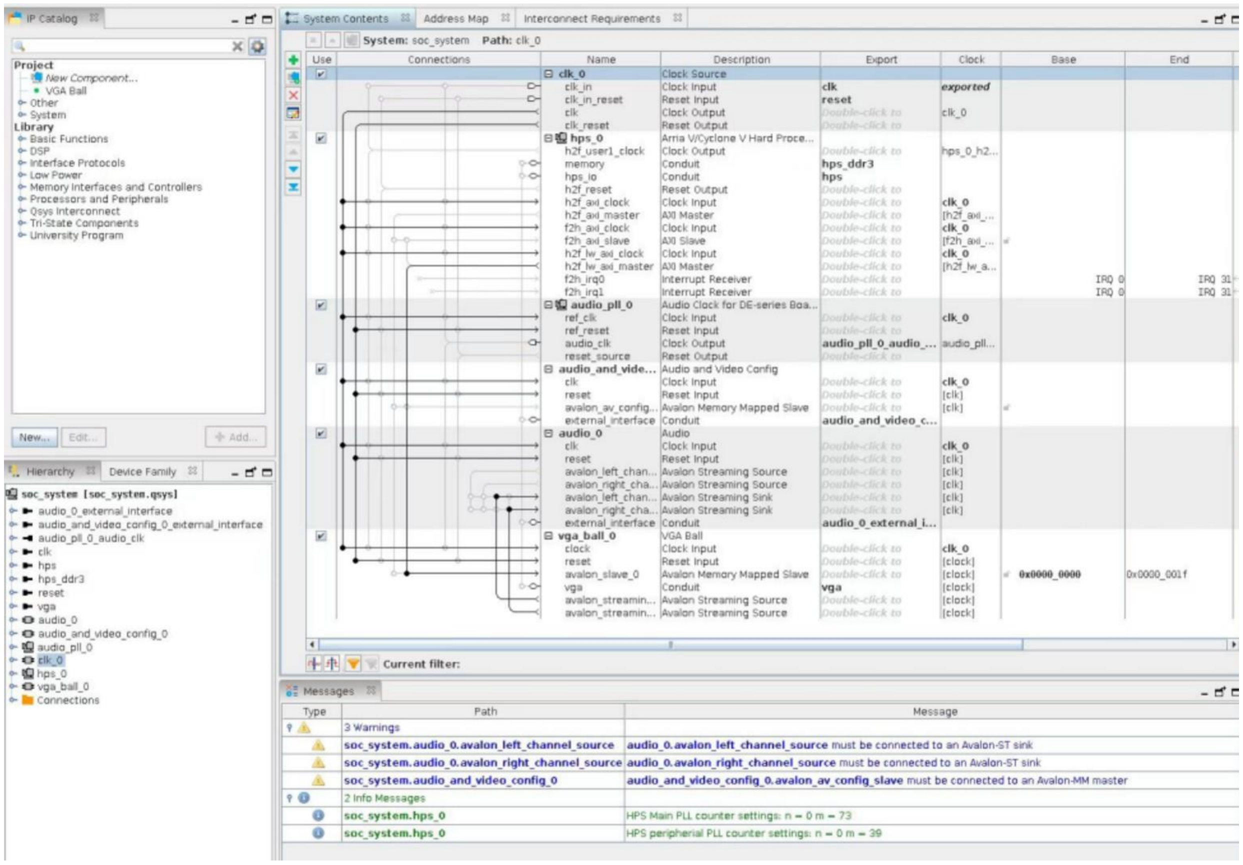
2. **Playback.** Given that the DE1-SoC operates with a 50 MHz clock frequency, and our audio sample frequency is set at 8 kHz, it is necessary to synchronize the audio playback rate with the system clock. To achieve this, we employ a constant divider method. By dividing the main clock frequency by a calculated constant, we adjust the clock to match the audio sample frequency. This ensures that each audio sample is played at the correct rate of 8 kHz, maintaining the integrity and tempo of the sound throughout the game.Furthermore, to accommodate variations in audio playback speed, different constants can be applied to adjust the clock divider. This flexibility allows the audio playback speed to be dynamically altered in response to game events, adding an immersive element to the Tetris gameplay.

4.3.2 Storage Allocation

Given that the total number of frames for the stereo audio is 47,000, with a sampling rate of 8kHz, the total duration of the audio is approximately 6 seconds. It is looped during the game play. Each frame is allocated 16 bits of storage space, thus the total space required is: 47,000 ×16×2 = 1, 504, 000.

4.4 Qsys Configuration

By referring to the DE1-Soc manual and the experiences of former students, we completed the config- uration, as shown in the following Figure.



Name	Description	Export	Clock	Base	End
clk_0	Clock Source				
clk_in	Clock Input	**clk**	*exported*		
clk_in_reset	Reset Input	**reset**			
clk	Clock Output	*Double-click to*	clk_0		
clk_reset	Reset Output	*Double-click to*			
hps_0	Arria V/Cyclone V Hard Proce...				
h2f_user1_clock	Clock Output	*Double-click to*	hps_0_h2...		
memory	Conduit	**hps_ddr3**			
hps_io	Conduit	**hps**			
h2f_reset	Reset Output	*Double-click to*			
h2f_axi_clock	Clock Input	*Double-click to*	**clk_0**		
h2f_axi_master	AXI Master	*Double-click to*	[h2f_axi_...		
f2h_axi_clock	Clock Input	*Double-click to*	**clk_0**		
f2h_axi_slave	AXI Slave	*Double-click to*	[f2h_axi_...		
h2f_lw_axi_clock	Clock Input	*Double-click to*	**clk_0**		
h2f_lw_axi_master	AXI Master	*Double-click to*	[h2f_lw_a...		
f2h_irq0	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31
f2h_irq1	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31
audio_pll_0	Audio Clock for DE-series Boa...				
ref_clk	Clock Input	*Double-click to*	**clk_0**		
ref_reset	Reset Input	*Double-click to*			
audio_clk	Clock Output	**audio_pll_0_audio_...**	audio_pll...		
reset_source	Reset Output	*Double-click to*			
audio_and_vide...	Audio and Video Config				
clk	Clock Input	*Double-click to*	**clk_0**		
reset	Reset Input	*Double-click to*	[clk]		
avalon_av_config...	Avalon Memory Mapped Slave	*Double-click to*	[clk]		
external_interface	Conduit	**audio_and_video_c...**			
audio_0	Audio				
clk	Clock Input	*Double-click to*	**clk_0**		
reset	Reset Input	*Double-click to*	[clk]		
avalon_left_chan...	Avalon Streaming Source	*Double-click to*	[clk]		
avalon_right_cha...	Avalon Streaming Source	*Double-click to*	[clk]		
avalon_left_chan...	Avalon Streaming Sink	*Double-click to*	[clk]		
avalon_right_cha...	Avalon Streaming Sink	*Double-click to*	[clk]		
external_interface	Conduit	**audio_0_external_i...**			
vga_ball_0	VGA Ball				
clock	Clock Input	*Double-click to*	**clk_0**		
reset	Reset Input	*Double-click to*	[clock]		
avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to*	[clock]	**0x0000_0000**	0x0000_001f
vga	Conduit	**vga**	[clock]		
avalon_streamin...	Avalon Streaming Source	*Double-click to*	[clock]		
avalon_streamin...	Avalon Streaming Source	*Double-click to*	[clock]		

Type	Path	Message
⚠	3 Warnings	
⚠	**soc_system.audio_0.avalon_left_channel_source**	**audio_0.avalon_left_channel_source** must be connected to an Avalon-ST sink
⚠	**soc_system.audio_0.avalon_right_channel_source**	**audio_0.avalon_right_channel_source** must be connected to an Avalon-ST sink
⚠	**soc_system.audio_and_video_config_0**	**audio_and_video_config_0.avalon_av_config_slave** must be connected to an Avalon-MM master
●	2 Info Messages	
●	**soc_system.hps_0**	HPS Main PLL counter settings: n = 0 m = 73
●	**soc_system.hps_0**	HPS peripherial PLL counter settings: n = 0 m = 39

Figure 10: Qsys Configuration

References

[1] De1-soc user manual. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836, 2015. Accessed: 2024-05-14.

[2] EDWARDS, S., AND OTHER STUDENTS. Analysis of tetris game development on fpga. https://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/reports/Tetris-report.pdf, 2022. Accessed: 2024-05-14.

[3] GAJSKI, D. D., AND RAMACHANDRAN, L. Silicon compilation. *Kluwer Academic Publishers* (1994).

[4] HAUCK, S., AND DEHON, A. Reconfigurable computing: The theory and practice of fpga-based computation. *Systems on Silicon* (2008).

[5] MAXFIELD, C. M. *FPGAs: Instant Access*. Elsevier, 2004.

[6] SMITH, J., AND DOE, J. Synthesis of vhdl code for fpga-based video games. In *Proceedings of the International Conference on Computer Design* (2006), pp. 45–50.

[1] [2] [3] [4] [5] [6]

Appendix

Hardware: vga_tetris.sv

```
/*
 * 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 #(
      .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 #(
    .WIDTH(BG_COLOR_BITS),
    .DEPTH(64),
    .INIT_F("./image/palette.mem")
) bg_palette_rom (
    .addr(bg_idx),
    .data(bg_rom_data));

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

// BGM
rom_async_hex #(
    .WIDTH(16),
    .DEPTH(47000),
    .INIT_F("./image/bgm_mem.hex")
) 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[0:21][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];

/*logic[23:0] rand_block_color[0:7];
int rand_val;*/

initial begin
        int rand_val = 1;
    // 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};

        // rand_block_color = block_color;
    // 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 < 22; i++)

        for (int j = 0; j < 10; j++)

            block_state[i][j] = 3'd0;

// write a smile face

     block_state[7][3] = 4;

     block_state[7][4] = 4;

     block_state[7][5] = 4;

     block_state[7][6] = 4;

     block_state[8][2] = 4;

     block_state[8][7] = 4;

     block_state[9][1] = 4;

     block_state[9][8] = 4;

     block_state[10][1] = 4;

     block_state[10][8] = 4;

     block_state[11][1] = 4;

     block_state[11][8] = 4;

     block_state[12][1] = 4;

     block_state[12][8] = 4;

     block_state[13][1] = 4;

     block_state[13][8] = 4;

     block_state[14][2] = 4;

     block_state[14][7] = 4;

     block_state[15][3] = 4;
```

```
        block_state[15][4] = 4;

        block_state[15][5] = 4;

        block_state[15][6] = 4;


        // eyes

        block_state[9][3] = 3;

        block_state[9][6] = 3;


        // mouth

        block_state[12][3] = 1;

        block_state[12][6] = 1;

        block_state[13][5] = 1;

        block_state[13][4] = 1;




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 < 22; 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];


            /*for(int ii = 1; ii < 8; ii ++)

                     rand_block_color[ii] = block_color[(ii + rand_val)%7 +1];

            rand_val ++;*/


   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 = 21; 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 < 22; i++)
      for (int j = 0; j < 10; j++)
        block_state[i][j] = 3'd0;
  // show over
     // o
     block_state[2][5] = 4;
     block_state[3][4] = 4;
     block_state[3][6] = 4;
     block_state[4][6] = 4;
     block_state[4][4] = 4;
     block_state[5][5] = 4;


     // v
     block_state[7][4] = 4;
     block_state[7][6] = 4;
```

```
        block_state[8][4] = 4;

        block_state[8][6] = 4;

        block_state[9][5] = 4;


        // e

        block_state[11][5] = 4;

        block_state[12][4] = 4;

        block_state[12][6] = 4;

        block_state[13][4] = 4;

        block_state[13][5] = 4;

        block_state[14][4] = 4;

        block_state[15][5] = 4;


        // r

        block_state[17][5] = 4;

        block_state[17][6] = 4;

        block_state[18][4] = 4;

        block_state[19][4] = 4;

        block_state[20][4] = 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 >= 47000)
                    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[10:5] >= 24 && hcount[10:5] <=33 && vcount[9:4] >= 4 && vcount[9:4] <= 25)
            {VGA_R, VGA_G, VGA_B} = { 3{block_rom_data[1], {7{block_rom_data[0]}}} } &
block_color[block_state[vcount[9:4] - 4][hcount[8:5] - 24]];
         else if (hcount[10:5] >= 11 && hcount[10:5] <= 14)
            // score number
            if (vcount[9:4] == 14)
               {VGA_R, VGA_G, VGA_B} = {3{digit_rom_data[score[33 - hcount[10:5]]], 5'd0}};
            // speed number
            else if(hcount[10:5] == 14 && vcount[9:4] == 10)
               {VGA_R, VGA_G, VGA_B} = {3{digit_rom_data[speed], 5'd0}};
            // next blocks
            else if ((hcount[10:5] == 11 + offset_next[13:12] && vcount[9:4] == 19 + offset_next[15:14])
                || (hcount[10:5] == 11 + offset_next[9:8] && vcount[9:4] == 19 + offset_next[11:10])
                || (hcount[10:5] == 11 + offset_next[5:4] && vcount[9:4] == 19 + offset_next[7:6])
                || (hcount[10:5] == 11 + 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: 1) 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>
```

```c
#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");
```

2) 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[22][10];

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

// 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 row)
{
    int col = 0;
    for (col = 0; col < 10; col++)
    {
        if (blocks[row][col] == 0) return 0;
    }
    for (col = row; col > 0; col--) memcpy(blocks[col], blocks[col - 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;
}

// Send the value to vla as a signal.
// 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;
    }
}
```

3) 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 < 22 && 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 < 22; 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;
}
```