2048 Project

Jingtian Lin(jl6589) Kanghui Lin(kl3521) Yunhao Xing(yx2812)

May 2024

1 Introduction

The 2048 game is a classical merging tile puzzle game that aims to combine tiles with the same numbers to create a tile with the number 2048. This project aims to implement the 2048 game using a hardware-software approach on an Altera DE1-SoC development board with a PlayStation 5 (PS5) controller for user input and including audio feedback during the gameplay. The hardware part of the project involves creating a VGA display interface to render the game board. The hardware code is responsible for generating VGA signals and managing the display of the game tiles, scores, and necessary graphical elements. The core of the hardware design is implemented in SystemVerilog, which includes modules for sprite rendering and VGA signal generation. The software part of the project is designed to interact with the hardware by sending tile information and color data. The software is coded for managing the game logic, reading user input from the PS5 controller, and updating the hardware with the current state of the game board. The software communicates with the hardware through memory-mapped I/O, sending the necessary data to render the game on the VGA display.

2 Objective and current progress

The overall goal of the project is to create a fully functional 2048 game that can be controlled using a PS5 controller and includes audio feedback. However, the current progress of the project has several pending issues and incomplete features. Hardware-software connection issue The connection between the hardware and software components prevents the game from being demonstrated successfully. In addition, although the audio file has been created, it is not integrated into the software to provide audio feedback during gameplay. Finally, the implementation of the PS5 controller as the input device is not yet finished, and additional work is required to complete this feature.

3 System overview

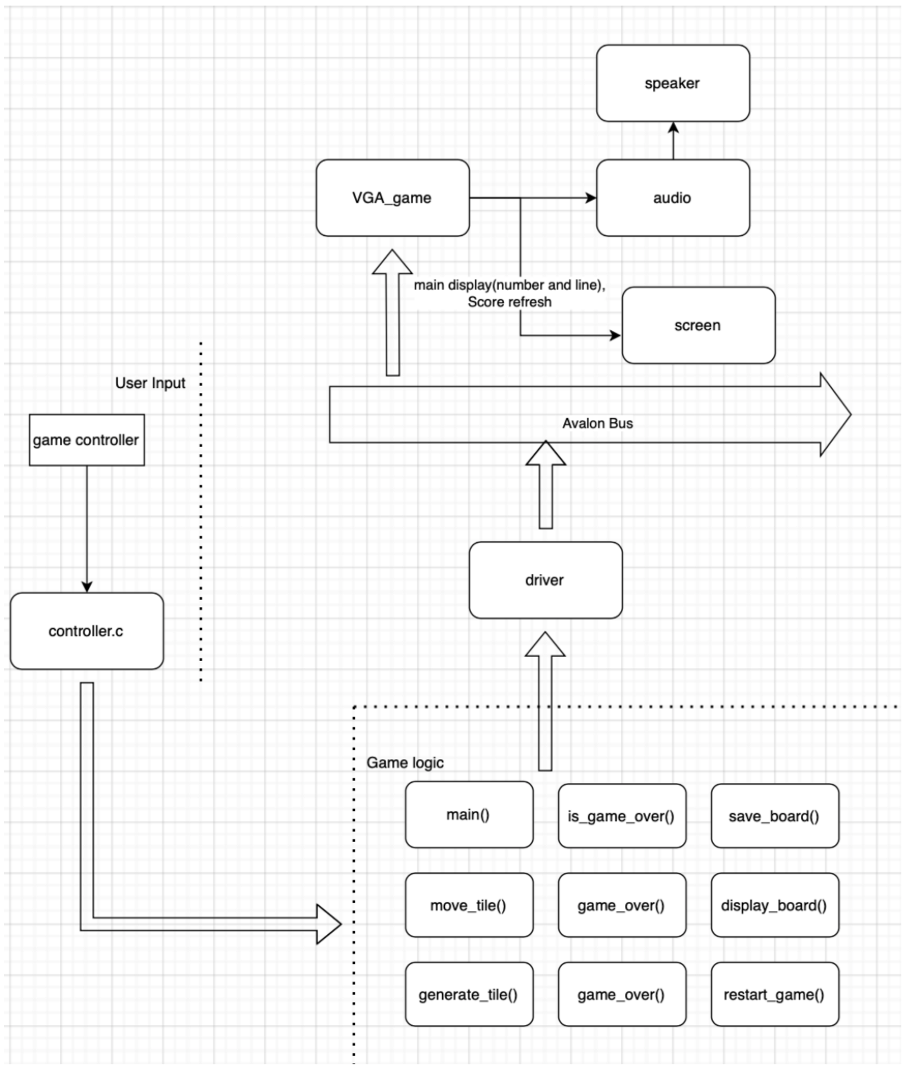


Figure 1: Overall design

4 Software

The software part consists of two main functionality: to create the animation and handle the game logic

The software part has the following functions:

- void init(): the function initialize the board and reset the value to 0

- void restart(): the function
- int game_over(): the function iterate over the board and judge whether there is still valid move that can be applied. if not return 1 to signalize game over else return 0 that game continues;
- void generate_tile(): the function iterate over the board and check for all empty spaces left; randomly select a empty space based on the random number generated, then randomly choose a 2 or a 4 to insert into the empty space;
- void shift_left(): the function perform a left shift on the board that moves all the tiles to the leftmost unoccupied space and merge the rightmost identical tiles in each row and add the score according to the tiles merged. Note each tile can only merge once during a move (eg. a row with 0-2-2-4 become 4-4-0-0 instead of 8-0-0-0 after a left shift)
- void shift_right(): similar to the logic of left shift. perform a right shift that moves all the tiles to the rightmost unoccupied space and merge the identical tiles in each row;
- void shift_up(): similar to previous shifts;
- void shift_down(): similar to previous shifts;
- void render_frames(): sent the current score, best score, and animation generated by player move in sequence

4.1 player moves

At each stage there are 5 possible move that player can make: move top, move down, move left, move right, and restart.

the move in direction logic function as follows:

- the function first initialize 6 frame arrays, frame1, frame2,..,frame6 each with size of 16*4. The frame array is used to hold the display information about the 16 grids during the player move.
- the function traverse the board matrix which holds the information on the board, and search from the opposite direction of player move to find the first zero in the row/column depending on the shift direction. then shift all the non-zero block to the closest zero position in the row/column depending on the shift direction. Eg. a row with 0-2-0-2 become 2-2-0-0 after a left shift
- then traverse the row/column to perform a merge if adjacent blocks have same value.

- during the shift and merge, if non of the tiles can be moved such that the board stays the same after the move, the move is deemed invalid. This is achieved using a variable valid_move which initialized as 0 signalizing no valid change has been performed yet and set to 1 whenever a shift or merge happened on the board. If non of the tile changes position or merges after the above logic, the move will be rolled back to its original state and hint the player that the move is invalid. Eg. a board (2-0-0-0; 0-0-0-0; 0-0-0-0; 0-0-0-0) stays the same after a left shift then left shift in this case is invalid and will be rolled back.
- if the move is valid, then it is garanteed that there is at least a empty space on the board for new tile to generate. The new tile will be generated by calling the generate_tile function. The function simply traverse through the current board and finds all empty space; randomly select a space and randomly pick 2 or 4 to insert into the space.
- after the tile has been generated, the function game_over() is called to check whether there is still empty space or if a move can be applied. If not the game is over and the score will be updated accordingly.

The code for move logic is as follows:

in the above code, the first for loop shift all tiles in a row to the left, and in the second shift perform a merge on adjacent tiles.

4.2 animation and display

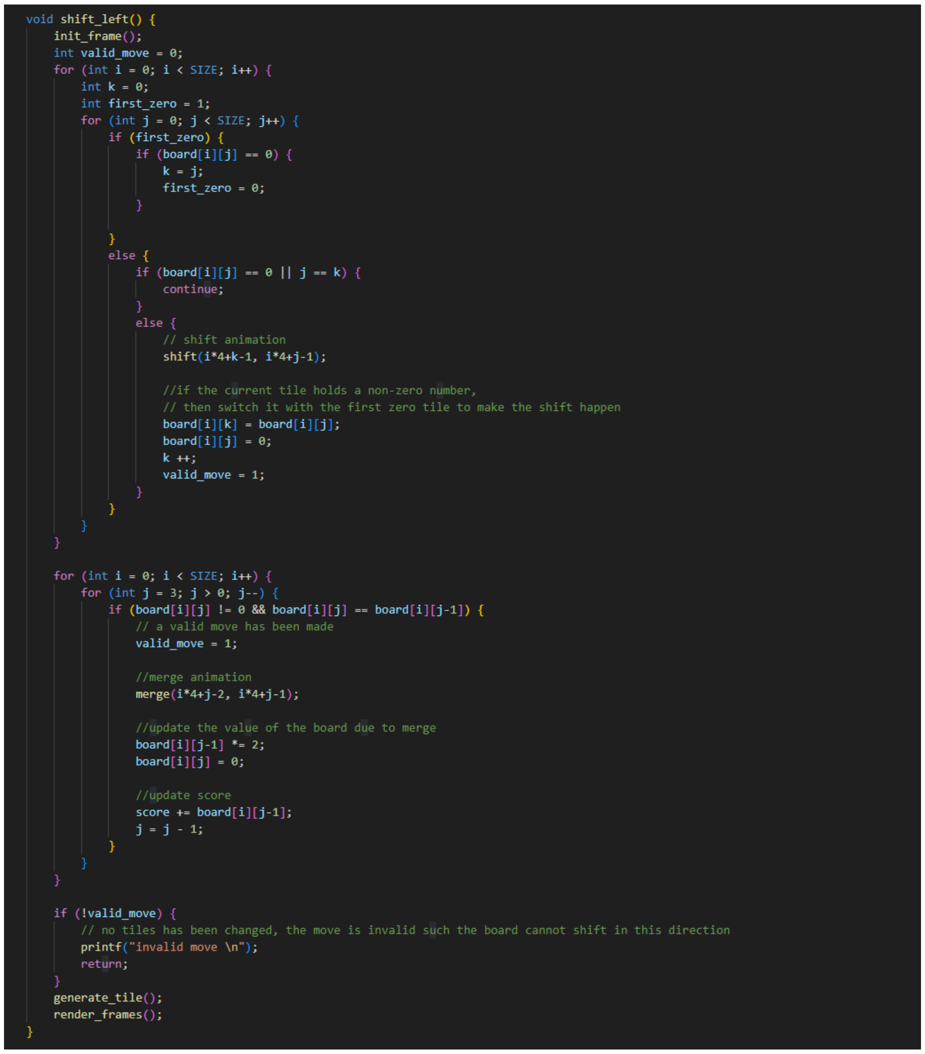
The software is also in charged of generating the animation for the player move and communicate with the hardware to render the display.

In the software perspective, after each player move six 16*4 matrix are generated each representing a frame.

The first two array stores the animation of the shift. The logic is as follows: The function first calculate the x and y coordinate of the destination tile, and the origin tile. According to the board layout shown in the design doc, the top left tile starts on (50,50), and each block is 100 by 100.

In the first frame, the center point of the origin block moves to 1/3 point between origin and destination, and the size of the block shrink down to 50% of its original size (2 in this case). The destination tile remains in its place and shrink to 50%. The array frame1 will be filled accordingly. The first column of origin/destination tile holds its x position in this frame, the second column holds its y position in this frame, the third column holds its size in this frame, and the last column holds its value in this frame. Suppose the origin tile starts at (150, 150) and is moving to (350, 150), then frame1[origin][0] = (350-150)/3 = 66, frame1[origin][1] = (350-150)/3 = 66, frame1[2] = 2, frame1[2] = its value, and the destination tile in frame1 will have its original disposition but a size 2.

In the second frame, the center point of the origin block moves to 1/2 point between origin and destination, and the size of the block goes to 75% of its



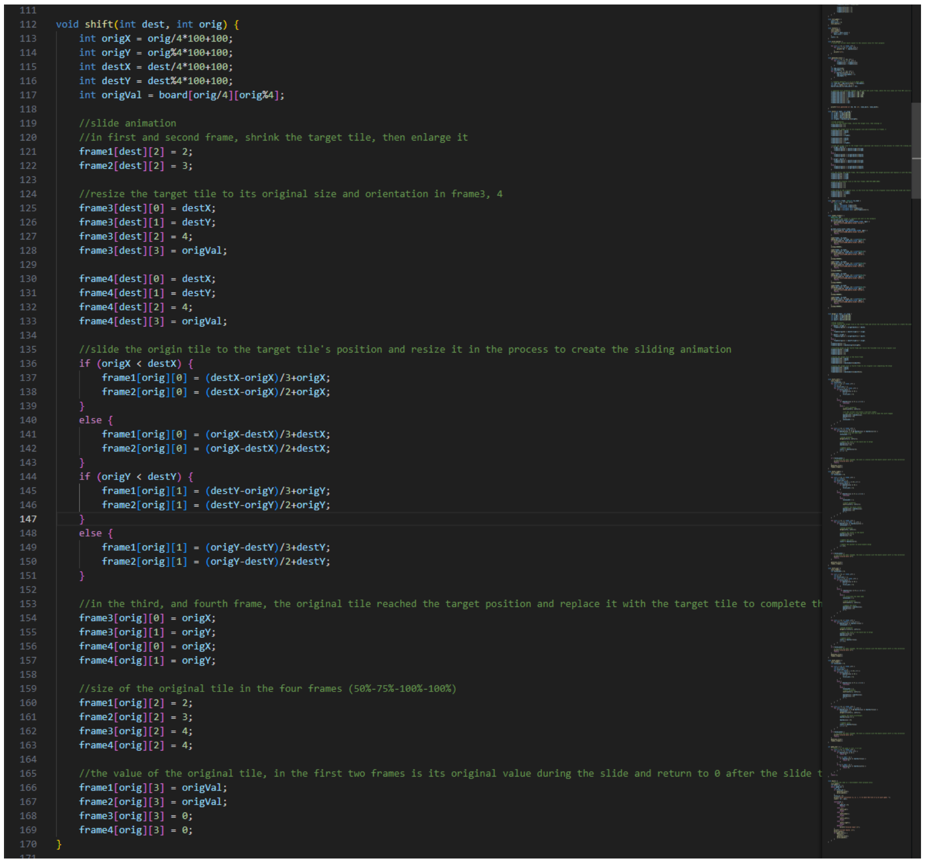
```c
void shift_left() {
    init_frame();
    int valid_move = 0;
    for (int i = 0; i < SIZE; i++) {
        int k = 0;
        int first_zero = 1;
        for (int j = 0; j < SIZE; j++) {
            if (first_zero) {
                if (board[i][j] == 0) {
                    k = j;
                    first_zero = 0;
                }

            }
            else {
                if (board[i][j] == 0 || j == k) {
                    continue;
                }
                else {
                    // shift animation
                    shift(i*4+k-1, i*4+j-1);

                    //if the current tile holds a non-zero number,
                    // then switch it with the first zero tile to make the shift happen
                    board[i][k] = board[i][j];
                    board[i][j] = 0;
                    k ++;
                    valid_move = 1;
                }
            }
        }
    }

    for (int i = 0; i < SIZE; i++) {
        for (int j = 3; j > 0; j--) {
            if (board[i][j] != 0 && board[i][j] == board[i][j-1]) {
                // a valid move has been made
                valid_move = 1;

                //merge animation
                merge(i*4+j-2, i*4+j-1);

                //update the value of the board due to merge
                board[i][j-1] *= 2;
                board[i][j] = 0;

                //update score
                score += board[i][j-1];
                j = j - 1;
            }
        }
    }

    if (!valid_move) {
        // no tiles has been changed, the move is invalid such the board cannot shift in this direction
        printf("invalid move \n");
        return;
    }
    generate_tile();
    render_frames();
}
```

Figure 2: code for left shift logic

original size. The destination tile remains in its place and have a size of 75%. The frame2 matrix will be updated similar to the logic described above.

In the third frame, the value of the starting block will be set to 0, and the starting block will be rendered back to its original orientation and size. The destination block, however, will now have the value of the starting block in its disposition. This frame finishes the shift that the starting block now become 0 and the destination block now holds the value of the starting block. The frame3 matrix is updated accordingly as the logic explained above.

The third and fourth frame matrix store the animation of merging blocks.



```
void shift(int dest, int orig) {
    int origX = orig/4*100+100;
    int origY = orig%4*100+100;
    int destX = dest/4*100+100;
    int destY = dest%4*100+100;
    int origVal = board[orig/4][orig%4];

    //slide animation
    //in first and second frame, shrink the target tile, then enlarge it
    frame1[dest][2] = 2;
    frame2[dest][2] = 3;

    //resize the target tile to its original size and orientation in frame3, 4
    frame3[dest][0] = destX;
    frame3[dest][1] = destY;
    frame3[dest][2] = 4;
    frame3[dest][3] = origVal;

    frame4[dest][0] = destX;
    frame4[dest][1] = destY;
    frame4[dest][2] = 4;
    frame4[dest][3] = origVal;

    //slide the origin tile to the target tile's position and resize it in the process to create the sliding animation
    if (origX < destX) {
        frame1[orig][0] = (destX-origX)/3+origX;
        frame2[orig][0] = (destX-origX)/2+origX;
    }
    else {
        frame1[orig][0] = (origX-destX)/3+destX;
        frame2[orig][0] = (origX-destX)/2+destX;
    }
    if (origY < destY) {
        frame1[orig][1] = (destY-origY)/3+origY;
        frame2[orig][1] = (destY-origY)/2+origY;
    }
    else {
        frame1[orig][1] = (origY-destY)/3+destY;
        frame2[orig][1] = (origY-destY)/2+destY;
    }

    //in the third, and fourth frame, the original tile reached the target position and replace it with the target tile to complete th
    frame3[orig][0] = origX;
    frame3[orig][1] = origY;
    frame4[orig][0] = origX;
    frame4[orig][1] = origY;

    //size of the original tile in the four frames (50%-75%-100%-100%)
    frame1[orig][2] = 2;
    frame2[orig][2] = 3;
    frame3[orig][2] = 4;
    frame4[orig][2] = 4;

    //the value of the original tile, in the first two frames is its original value during the slide and return to 0 after the slide t
    frame1[orig][3] = origVal;
    frame2[orig][3] = origVal;
    frame3[orig][3] = 0;
    frame4[orig][3] = 0;
}
```

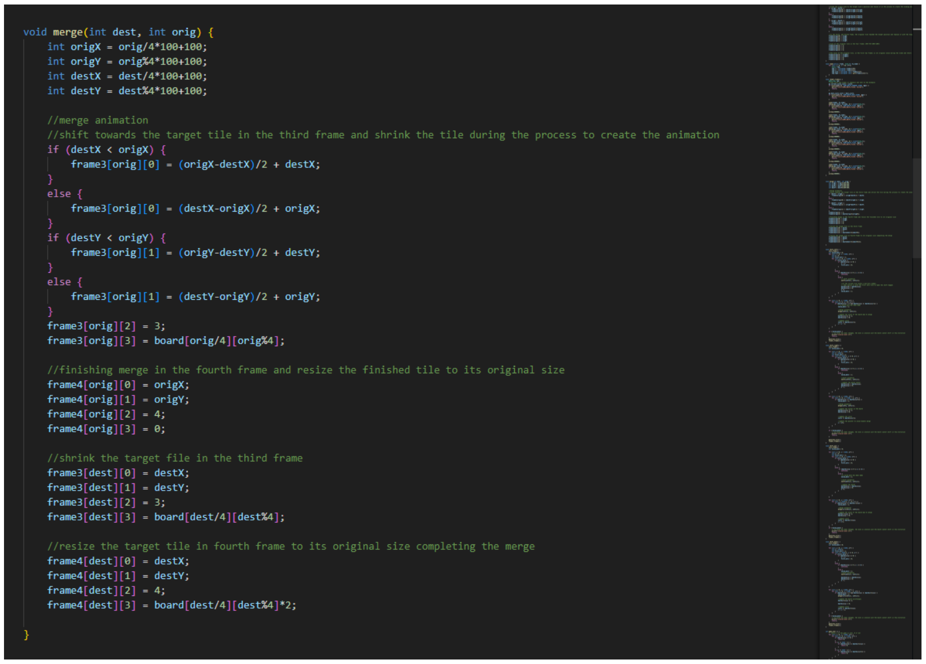
Figure 3: code for shift logic

The logic is as follows:

In the third frame, the starting block moves to middle point between starting block and destination block. The size of starting and destination block become 50% of its original size. The frame3 matrix is updated accordingly.

Note the shift animation finishes at frame3. The merge animation overrides the last frame of shift animation and makes the block remain 50% of its original size if the block first shift then merge. If the block only shifts then frame3 and frame4 will simply be the shifted block staying at its position. eg. suppose a row with value 0-2-2-0, a left shift first transform the row to 2-2-0-0 then merges and become 4-0-0-0. In this case the 3rd frame of shift animation will be override by the start of the merge animation. However, in a row with 0-2-4-0 become 2-4-0-0 after left shift which in this case merge wouldn't happen, and the third and fourth frame will simply render 2-4-0-0 with no animation.

In the fourth frame, the starting block adds its value to the destination block and goes back to its original disposition to finish the merge.



```c
void merge(int dest, int orig) {
    int origX = orig/4*100+100;
    int origY = orig%4*100+100;
    int destX = dest/4*100+100;
    int destY = dest%4*100+100;

    //merge animation
    //shift towards the target tile in the third frame and shrink the tile during the process to create the animation
    if (destX < origX) {
        frame3[orig][0] = (origX-destX)/2 + destX;
    }
    else {
        frame3[orig][0] = (destX-origX)/2 + origX;
    }
    if (destY < origY) {
        frame3[orig][1] = (origY-destY)/2 + destY;
    }
    else {
        frame3[orig][1] = (destY-origY)/2 + origY;
    }
    frame3[orig][2] = 3;
    frame3[orig][3] = board[orig/4][orig%4];

    //finishing merge in the fourth frame and resize the finished tile to its original size
    frame4[orig][0] = origX;
    frame4[orig][1] = origY;
    frame4[orig][2] = 4;
    frame4[orig][3] = 0;

    //shrink the target file in the third frame
    frame3[dest][0] = destX;
    frame3[dest][1] = destY;
    frame3[dest][2] = 3;
    frame3[dest][3] = board[dest/4][dest%4];

    //resize the target tile in fourth frame to its original size completing the merge
    frame4[dest][0] = destX;
    frame4[dest][1] = destY;
    frame4[dest][2] = 4;
    frame4[dest][3] = board[dest/4][dest%4]*2;

}
```

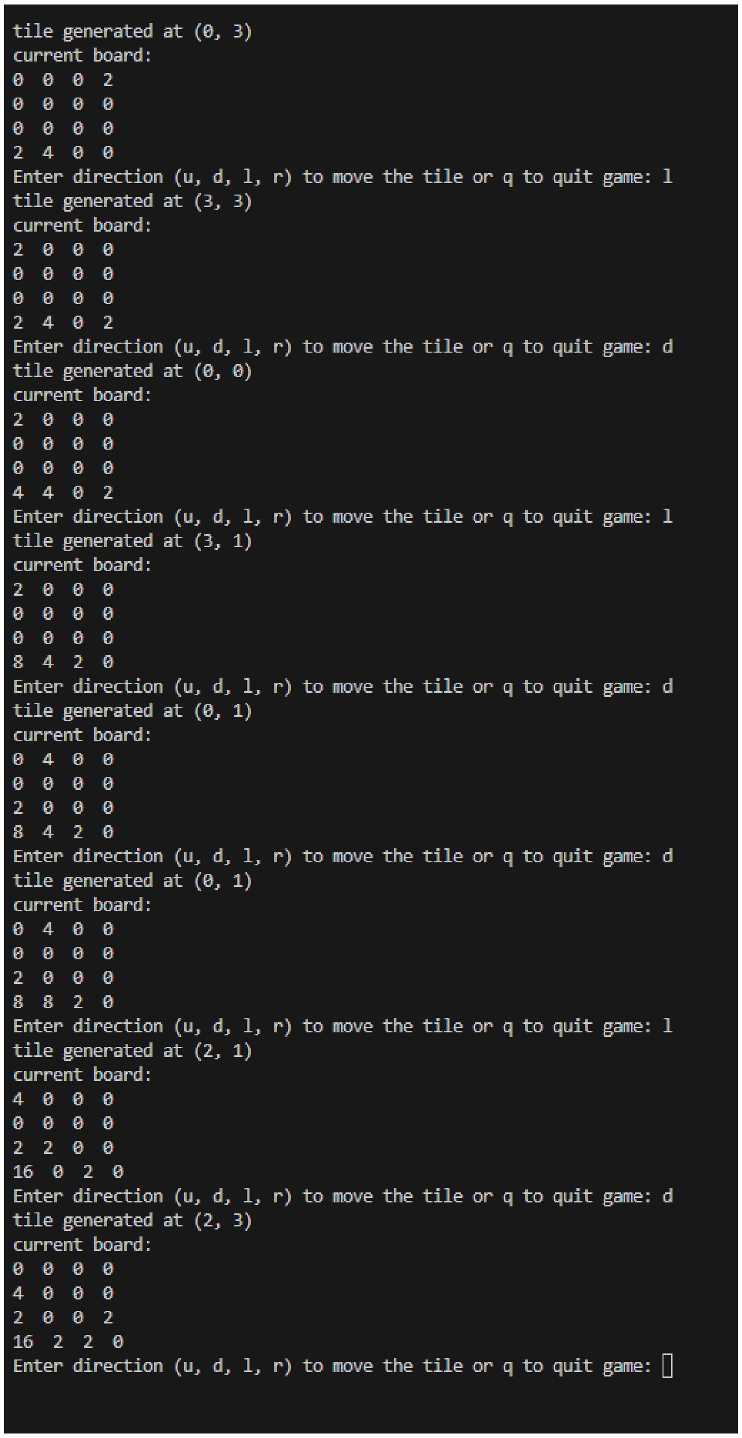
Figure 4: code for merge logic

The fifth and sixth frames store the animation of generating a new tile. The logic is simply merging a tile in the place where the tile is generated by gradually enlarging its size (50% in frame5, and 100% in frame 6).

After all frames are generated, the render_frame() function will then be called to sent the frame information to the hardware. The frames will be loaded into a predefined tile_t[16] array. The tile_t strut contains two unsigned variables x and y which correspond to the x and y position of the tile, and two unsigned char size and type which correspond to the size of the tile and the value of the tile (type = log_2(tile_value). The array is then sent in sequence to the board to render the display.

The future step would be to coordinate with the hardware part and make sure the render function performs smoothly. Since the move is split into three independent part, this approach is allows scalability if more frames are needed for smoother transition animation. If the animation appeared rigid, simply adding more transition frames into each stages with a smaller step size and smaller size change between each frame move.

4.3 test and validation of software



```
tile generated at (0, 3)
current board:
0  0  0  2
0  0  0  0
0  0  0  0
2  4  0  0
Enter direction (u, d, l, r) to move the tile or q to quit game: l
tile generated at (3, 3)
current board:
2  0  0  0
0  0  0  0
0  0  0  0
2  4  0  2
Enter direction (u, d, l, r) to move the tile or q to quit game: d
tile generated at (0, 0)
current board:
2  0  0  0
0  0  0  0
0  0  0  0
4  4  0  2
Enter direction (u, d, l, r) to move the tile or q to quit game: l
tile generated at (3, 1)
current board:
2  0  0  0
0  0  0  0
0  0  0  0
8  4  2  0
Enter direction (u, d, l, r) to move the tile or q to quit game: d
tile generated at (0, 1)
current board:
0  4  0  0
0  0  0  0
2  0  0  0
8  4  2  0
Enter direction (u, d, l, r) to move the tile or q to quit game: d
tile generated at (0, 1)
current board:
0  4  0  0
0  0  0  0
2  0  0  0
8  8  2  0
Enter direction (u, d, l, r) to move the tile or q to quit game: l
tile generated at (2, 1)
current board:
4  0  0  0
0  0  0  0
2  2  0  0
16  0  2  0
Enter direction (u, d, l, r) to move the tile or q to quit game: d
tile generated at (2, 3)
current board:
0  0  0  0
4  0  0  0
2  0  0  2
16  2  2  0
Enter direction (u, d, l, r) to move the tile or q to quit game:
```

Figure 5: running software in C environment

All the software function is tested and validated in the C environment. The main function is implemented for the test purpose. The game can be played by compiling executable of the C file. The game asks for 5 kinds of command in each round (u,d,l,r,q) corresponding to move up, down, left, right, quit in each round. All the frames generated have been tested to hold the expected value. The future step would be to connect with the board.

5 Hardware

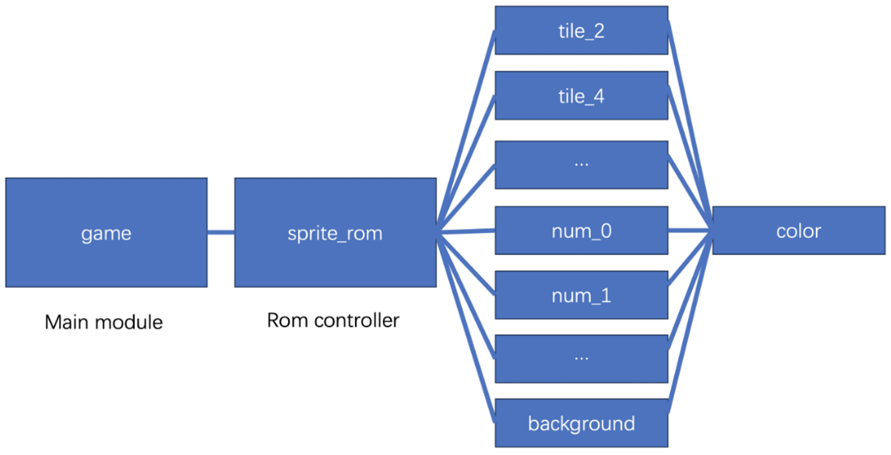


Figure 6: hardware diagram

5.1 VGA display

The VGA display progress for the 2048 game on the Altera DE1-SoC board is shown as follows, with the core functionality now rendering the game board and scores. The 'sprite_rom' and game modules written in SystemVerilog are responsible for generating VGA signals and rendering tiles based on their positions and values. It shows the game grid, current score, best score, and a prompt to start a new game. However, there are issues with the correct rendering of some tiles, resulting in black blocks as shown in the provided image. These issues indicate potential problems in the 'DRAW-TILE' state or address calculations within the hardware code. Efforts are ongoing to debug and resolve these rendering issues to achieve a fully functional and visually accurate game display.

To be more detailed, 'game' is the main hardware module which takes data from the driver and use the rom controller 'sprite_rom' to generate correct RGB value for each pixel. The 'sprite_rom' contains a state machine to draw each pixel and tell the main module whether it finishes drawing so that the main module could continue processing next pixel. The roms are verilog files generated by Quartus using .mif files which representing each 'image'. Except for 'color' rom, we use limited bits to represent the color type of each pixel in the image and then connect the output of the other roms with the input of the 'color' rom to find the exact color of that type in the specific image. In this way we can reduce the memory budget about storing images. After sufficient tests, this part works fine.

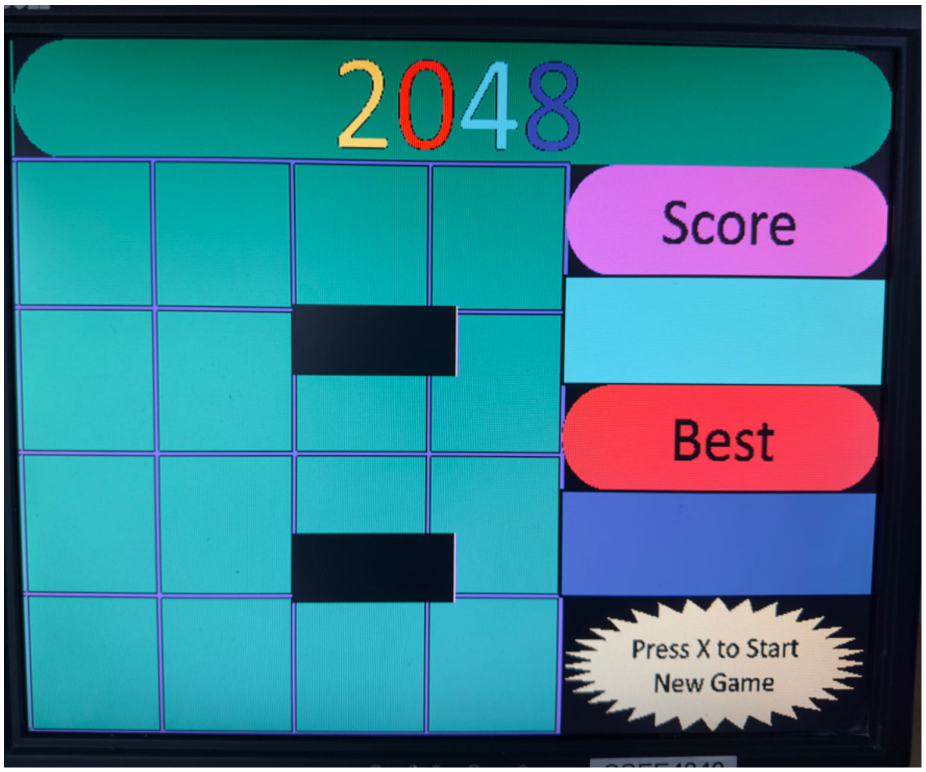
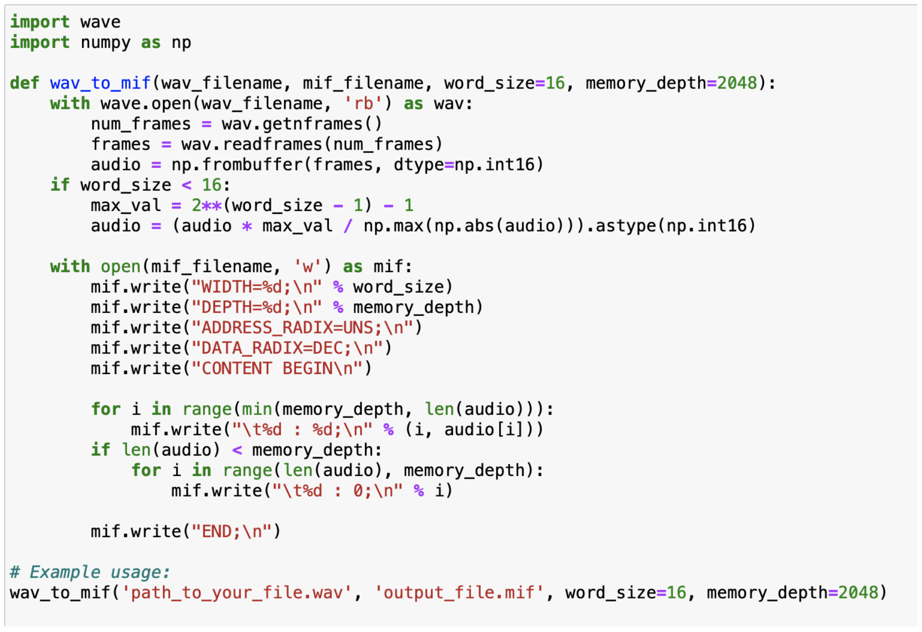


Figure 7: vga display

5.2 Audio

For the Audio part, the hardware implementation is not finished yet. However, the original audio file is converted into a monotype with a 44100Mhz WAV type file with Audacity. Then the following Python script is used to convert the WAV file into .mif file so that it can be embedded in the ROM of Quartus II:



```python
import wave
import numpy as np

def wav_to_mif(wav_filename, mif_filename, word_size=16, memory_depth=2048):
    with wave.open(wav_filename, 'rb') as wav:
        num_frames = wav.getnframes()
        frames = wav.readframes(num_frames)
        audio = np.frombuffer(frames, dtype=np.int16)
    if word_size < 16:
        max_val = 2**(word_size - 1) - 1
        audio = (audio * max_val / np.max(np.abs(audio))).astype(np.int16)

    with open(mif_filename, 'w') as mif:
        mif.write("WIDTH=%d;\n" % word_size)
        mif.write("DEPTH=%d;\n" % memory_depth)
        mif.write("ADDRESS_RADIX=UNS;\n")
        mif.write("DATA_RADIX=DEC;\n")
        mif.write("CONTENT BEGIN\n")

        for i in range(min(memory_depth, len(audio))):
            mif.write("\t%d : %d;\n" % (i, audio[i]))
        if len(audio) < memory_depth:
            for i in range(len(audio), memory_depth):
                mif.write("\t%d : 0;\n" % i)

        mif.write("END;\n")

# Example usage:
wav_to_mif('path_to_your_file.wav', 'output_file.mif', word_size=16, memory_depth=2048)
```

Figure 8: code to provide .hex audio file

Then the future step is to implement a I2C interface to send a command to the WM7831 codec on board to set the volume and other necessary parameters for audio output.

6 Hardware-Software interface

In the driver, we defined a argument that is used to transfer information between hardware and software. The argument contains the information about our tiles, game status and scores. From software, we update our game after user does some input and then set the argument correctly and use ioctl() to pass the argument to our driver. To hardware, the driver will convert the argument to specific register format and use Avalon bus to tell the hardware about what to do. The communication between hardware and software is one-way. The hardware does not need to tell the software anything.

Tile registers: 10 bits for x position, 9 bits for y position, 3 bits for tile size, 4 bits for tile type, 26/32 bits in use.

Current score register: 4 bits for each digit, 5 digits in total, 2 bits for game status, 2 bits for audio selection, 24/32 bits in use. Best score register is same as current score register but no game status and audio part, 20/32 bits in use.

7 Team contribution

Jingtian Lin: Focus on software and game logic implementation. Wrote all the software parts, explicitly tested all the software logic and implemented the animation generation logic. Also coordinate with the team in discussing and coming up with the framework.

Kanghui Lin: Focus on the audio and controller implementation. And test the connection between the hardware part and the software part as well as assist in debugging in VGA-display.

Yunhao Xing: Focus in the video implementation and hardware-software interface.