CS 21 Project: 2048 in MIPS

Jan Albert Quidet
Judelle Clareese Gaza

Project Specifications

The task is to create a 2048 implementation written in assembly language and simulated using the MARS software. The game will be implemented using a 3 by 3 square grid and the player wins if one of the grids gets a tile numbered 512. A player loses if the 3 by 3 grid gets filled up with number tiles and is out of moves to combine similar tiles.

Our MIPS implementation will have the following features:

- Choose from a New Game that generates an initial state, or Starts from a Player's State
- Input Checking, our program can check for valid or invalid inputs
  - That is the game should not accept inputs not from valid choices:
    - W/A/S/D/X/3/4
    - W: Move Up, A: Move Left, S: Move Down, D: Move Right
    - X: Terminate Program
    - 3: Disable RNG
    - 4: Enable RNG
  - Our game also does not generate a 2 if the move by the Player does not change the board state (i.e spamming/swiping left where the tiles are at the edges, a 2 should not generate in the board since it is an invalid input)
- The game ends when the Win/Lose Condition is satisfied. That is:
  - Win State: Achieve **512!** Yay!
  - Lose State: No more valid moves (No more same adjacent tiles)/Grid is Full **:<**

MIPS Implementation

Important Data used in the Program:

```
.data
WELCOME:        .asciiz "Main Menu:\n[1] New Game\n[2] Start from a State\n[X] Quit\n"
BOARD_CONFIG:   .asciiz "Enter a board configuration (Invalid integer = back to Main menu):\n"
LINE:           .asciiz "+---+---+---+\n"
VERTICAL:       .asciiz "|"
NUMBER_0:       .asciiz "   "
NUMBER_2:       .asciiz " 2 "
NUMBER_4:       .asciiz " 4 "
NUMBER_8:       .asciiz " 8 "
NUMBER_16:      .asciiz " 16"
NUMBER_32:      .asciiz " 32"
NUMBER_64:      .asciiz " 64"
NUMBER_128:     .asciiz "128"
NUMBER_256:     .asciiz "256"
NUMBER_512:     .asciiz "512"
NEWLINE:        .asciiz "\n"
WIN:            .asciiz "Congratulations! You have reached the 512 tile!"
LOSE:           .asciiz "Game Over."
QUIT:           .asciiz "Program Terminated."
ENTER_MOVE:     .asciiz "Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG):\n"
INVALID_MOVE:   .asciiz "Invalid move.\n"
INVALID_INT:    .asciiz "Invalid integer.\n"
RNG_DISABLED:   .asciiz "RNG Disabled.\n"
RNG_ENABLED:    .asciiz "RNG Enabled.\n"
MOVE:           .word
ORIGIN_GRID:    .word   0:9             # Array to store 9 original values
```

We have the following data stored for the strings used in the game:

- **WELCOME**: The main menu text displayed at the start.
- **BOARD_CONFIG**: Message prompting the user to enter a board configuration.
- **LINE**: Horizontal line for formatting the game grid.
- **VERTICAL**: Vertical separator for grid columns.
- **NUMBER_0** to **NUMBER_512**: Strings for formatting grid numbers (e.g., 2, 4, 8, etc.) into a readable grid.
- **NEWLINE**: A newline character used to format the output.
- **WIN**: Message displayed when the player wins by reaching a tile of 512.
- **LOSE**: Message displayed when the player loses (no moves left).
- **QUIT**: Message displayed when the program terminates.
- **ENTER_MOVE**: Prompt for the user to enter a move.
- **INVALID_MOVE**: Error message for an invalid move.
- **INVALID_INT**: Error message for an invalid integer input.
- **RNG_DISABLED**: Message displayed when RNG is disabled.
- **RNG_ENABLED**: Message displayed when RNG is enabled.
- **MOVE**: Memory space for storing user input for moves.
- **ORIGIN_GRID**: Array for checking for grid changes

Overview of Key Sections

Welcome and Game Mode Selection

```
# ---------------------- MAIN PROGRAM ----------------------
.text
# First Load Initialize the Board
main:
        print_str WELCOME               # Print MAIN MENU PROMPT
        get_input MOVE                  # Read User Input STORE IN MOVE

        lw      $t0, MOVE               # load MOVE to t0
        beq     $t0, 0xa31, input_new   # If NEW GAME
        beq     $t0, 0xa32, input_state # If Start from a STATE
        beq     $t0, 0xa58, terminate   # X = Quit

        print_str INVALID_MOVE          # otherwise, Print INVALID MOVE
        j       main                    # Loop back to main (main menu)
```

The game prompts the user to choose between starting a new game with random tiles or entering a custom board configuration.

- If the user chooses to create a New Game: Program branches to **input_new**
    - Two tiles in the grid will be randomly selected to initialize two “2” valued tiles.
    - After the initial board has been printed, the user may proceed to play the game using the inputs
- If the user chooses to Start from a State: Program branches to **input_state**
    - The user is prompted to provide the value of each tile (9 tiles in total) which will be saved in registers $t1 to $t9
- If the user chooses to input “X”, then the program branches to terminate, which terminates the game.

If the program doesn’t branch in any of the possible branches then the input must be valid, we loop back to main.

Game Loop

The game continuously prints the grid, checks for win/lose conditions, and processes player input. The loop ensures that the game state is updated after each move.

```
# ===== Main Loop - print grid - check win state - get input =====
main_loop:      # Print the Grid
        jal     print_grid      # Print the Grid
        jal     is_win          # Check if win
        jal     is_lose         # Check if lose
input_move:     # Get User Input
        print_str ENTER_MOVE    # Prompt Move
        get_input MOVE          # Get Input

        lw      $t0, MOVE
        beq     $t0, 0xa58, terminate   # X = Quit

        jal     store_origin
        beq     $t0, 0xa57, move_up     # W = move up
        beq     $t0, 0xa41, move_left   # A = move left
        beq     $t0, 0xa53, move_down   # S = move down
        beq     $t0, 0xa44, move_right  # D = move right

        beq     $t0, 0xa33, rng_disable # 3 = disable RNG
        beq     $t0, 0xa34, rng_enable  # 4 = enable RNG
# Skipped if move is valid -- else we jump here if board not changed
invalid_move:
        print_str INVALID_MOVE          # Print INVALID MOVE
        j       input_move              # Loop back to input_move
```

Above is the main loop of our program. We print the grid -> check the win state (if either is satisfied the program terminates) -> we get the input.

Note! Before getting the input we store the current board state (values of registers $t1 - $t9) in the origin grid :>

Getting the input:
First we prompt the user for the input by printing the ENTER_MOVE string.
Then we read the input using the macro get_input and store the input in MOVE.
If MOVE == X we quit.
If MOVE is either of the the following: W/A/S/D, we branch accordingly to:

- move_up
- move_left
- move_down
- move_right

If MOVE is 3 or 4, we branch to:

```
# ===== RNG =======
rng_disable:
        li      $s0, 3
        print_str RNG_DISABLED
        j       input_move
rng_enable:
        li      $s0, 4
        print_str RNG_ENABLED
        j       input_move
```

- rng_disable: store 3 to $s0, print prompt, then we loop back to taking input
- rng_enable: store 4 to $s0, print prompt, then we loop back to taking input

Grid and Tile Handling

If the player chooses to Start from a State, they are prompted to give the 9 states of the 9 tiles. To do that we used a macro **set_tile:**

```
# Input the state of a tile
        .macro set_tile %reg
                li      $v0, 5
                syscall
                move    %reg, $v0
                move    $a0, %reg
                jal     check_integer
        .end_macro
```

The **print_grid** function displays the grid, iterating through each tile and printing its value.

```
# ============== PRINT THE GRID BOARD =================
print_grid:                                 # GRID PRINTING
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        print_row $t1 $t2 $t3   # row 1
        print_row $t4 $t5 $t6   # row 2
        print_row $t7 $t8 $t9   # row 3
        print_str LINE          # bottom of the board
        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr      $ra
```

We first save the return address, then proceed to print the rows using our defined macro **print_row.**

print_row macro:

```
# Print Rows of board
    .macro print_row %reg1 %reg2 %reg3
        print_str LINE
        print_str VERTICAL
        move    $a0, %reg1
        jal     print_integer
        print_str VERTICAL
        move    $a0, %reg2
        jal     print_integer
        print_str VERTICAL
        move    $a0, %reg3
        jal     print_integer
        print_str VERTICAL
        print_str NEWLINE
    .end_macro
```

The macro prints the values in the row along with the vertical lines and top lines of the board

It calls the print_integer function to know which one to print

print_integer function:

```
print_integer:
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        jal     get_integer     # Get integer to Print
        move    $a0, $v0
        li      $v0, 4
        syscall
        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr $ra
```

```
get_integer:
        beq     $a0, 0, print_0
        beq     $a0, 2, print_2
        beq     $a0, 4, print_4
        beq     $a0, 8, print_8
        beq     $a0, 16, print_16
        beq     $a0, 32, print_32
        beq     $a0, 64, print_64
        beq     $a0, 128, print_128
        beq     $a0, 256, print_256
        beq     $a0, 512, print_512
```

```
print_0:
        la      $v0, NUMBER_0
        jr      $ra
print_2:
        la      $v0, NUMBER_2
        jr      $ra
print_4:
        la      $v0, NUMBER_4
        jr      $ra
print_8:
        la      $v0, NUMBER_8
        jr      $ra
print_16:
        la      $v0, NUMBER_16
        jr      $ra
print_32:
        la      $v0, NUMBER_32
        jr      $ra
print_64:
        la      $v0, NUMBER_64
        jr      $ra
print_128:
        la      $v0, NUMBER_128
        jr      $ra
print_256:
        la      $v0, NUMBER_256
        jr      $ra
print_512:
        la      $v0, NUMBER_512
        jr      $ra
```

print_integer function calls the get_integer function. **print_integer** will do the syscall to output the terminal while the **get_integer** returns the mapping of integers to its string equivalent

Random Tile Generation

The **zero_or_two** function generates either a 0 or 2 for the tile placement. This is called when the user wants to create a NEW GAME (that is, we generate two random tiles to have the initial state of the board).

```
# ======= NEW GAME ==========
input_new:                          # NEW GAME 1
        li      $t0, 0
input_random_loop:
        call_zero_or_two $t1        # Calls the zero_or_two function (generates random 0 or 2) then store to t1
        call_zero_or_two $t2        # Store to t2
        call_zero_or_two $t3        # Store to t3
        call_zero_or_two $t4        # Store to t4
        call_zero_or_two $t5        # Store to t5
        call_zero_or_two $t6        # Store to t6
        call_zero_or_two $t7        # Store to t7
        call_zero_or_two $t8        # Store to t8
        call_zero_or_two $t9        # Store to t9
        blt     $t0, 2, input_random_loop # If $t0 < 2 then input random loop again, the board has to have ATLEAST 2 "2" initialized tiles
        j       main_loop           # proceeds to game loop

# Used Functions for randomized zero or two #
```

When creating a new game, we use the macro **call_zero_or_two** which is a macro that calls the function **zero_or_two** which in turn uses the macro **randomize**.

```
# Generate zero or two
    .macro call_zero_or_two %reg
        jal zero_or_two             # Call zero_or_two function
        move %reg, $v0              # returns a zero or two
    .end_macro
```

Call_zero_or_two calls **zero_or_two**
And saves the output to the designated register

```
# Used Functions for randomized zero or two #
zero_or_two:
        randomize 2             # randomize a number in [0,1]
        beq     $a0, 0, zero    # if num == 0 return zero
        beq     $a0, 1, two     # if num == 1 return two
zero:
        li      $v0, 0
        jr      $ra             # return 0
two:
        bge     $t0, 2, zero    #
        li      $v0, 2
        addi    $t0, $t0, 1     # increment $t0 >> there are now $t0 tiles in the board
        jr      $ra
# End #

# Randomizer
    .macro randomize %upperbound
        li      $a1, %upperbound        # Upperbound when generating a num
        li      $v0, 42                 # Randomizer syscall
        syscall
    .end_macro
```

Zero_or_two utilized the macro **randomize** to generate a number between 0 or 1.
If 0, we return 0.
If 1, we check if t0 >= 2, (t0 checks the number of "2s" in the board) if not then we increment t0 and return 2.

randomize macro takes in an integer as the upper bound. It uses syscall 42 for the randomizer, and returns the number.

The **add_random_tile** function randomly places a tile (2) on the grid after each move if it is enabled ($s0 = 4). The position is selected randomly, and the tile value is set accordingly.

```
# ====== adding a random tile =======
add_random_tile:
        jal     check_origin        # Checks if grid changed
        beq     $s0, 3, main_loop   # If $s0 is 3 - RNG disabled no need to add_random_tile
        randomize 9                 # Choose a tile! from 1 to 9 >> (note zero indexed so choose a number in [0,8]
        beq     $a0, 0, tile_t1
        beq     $a0, 1, tile_t2
        beq     $a0, 2, tile_t3
        beq     $a0, 3, tile_t4
        beq     $a0, 4, tile_t5
        beq     $a0, 5, tile_t6
        beq     $a0, 6, tile_t7
        beq     $a0, 7, tile_t8
        beq     $a0, 8, tile_t9

tile_t1: generate_tile $t1
tile_t2: generate_tile $t2
tile_t3: generate_tile $t3
tile_t4: generate_tile $t4
tile_t5: generate_tile $t5
tile_t6: generate_tile $t6
tile_t7: generate_tile $t7
tile_t8: generate_tile $t8
tile_t9: generate_tile $t9
```

```
.macro generate_tile %reg
        bnez    %reg, add_random_tile
        li      %reg, 2
        j       main_loop
.end_macro
```

add_random_tile calls **check_origin** if the grid has changed. Then if $s0 == 3 (RNG disabled) we jump to the main loop - no need to randomize. However, if $s0 is not 3, then we call **randomize 9** macro where we generate a number from 0 to 8 (this represents the tiles). We then use the macro **generate_tile** to set the random tile to 2 if it is unoccupied, else if occupied we get a new random number, then we jump back into the main loop.

Win/Loss Condition

The game checks if the player has reached the 512 tiles (win condition) or NO empty cells & NO equal adjacent cells (lose condition). The checks use the **is_win** and **is_lose** functions, which look for the 512 tile or for any possible moves or combinations left on the grid.

```
# =========== CHECK WIN/LOSE CONDITION =============
is_win:
        beq     $t1, 512, win   # Checks if 512
        beq     $t2, 512, win
        beq     $t3, 512, win
        beq     $t4, 512, win
        beq     $t5, 512, win
        beq     $t6, 512, win
        beq     $t7, 512, win
        beq     $t8, 512, win
        beq     $t9, 512, win
        jr      $ra
win:
        print_str WIN           # End if WIN!
        j       exit
```

```
# ---- terminate -----
terminate:
        print_str QUIT
exit:
        li      $v0, 10
        syscall
```

```
is_lose:
        beqz    $t1, False              # Check if there's an empty tile
        beqz    $t2, False
        beqz    $t3, False
        beqz    $t4, False
        beqz    $t5, False
        beqz    $t6, False
        beqz    $t7, False
        beqz    $t8, False
        beqz    $t9, False

        beq     $t1, $t2, False         # Check if adjacent tiles can combine
        beq     $t1, $t4, False
        beq     $t2, $t3, False
        beq     $t2, $t5, False
        beq     $t3, $t6, False
        beq     $t4, $t5, False
        beq     $t4, $t7, False
        beq     $t5, $t6, False
        beq     $t5, $t8, False
        beq     $t6, $t9, False
        beq     $t7, $t8, False
        beq     $t8, $t9, False
game_over:
        print_str LOSE
        j exit
False:
```

Is_win exhaustively checks each tile if it contains 512. If it does then we print the WIN string, and exit the program.
Is_lose will check if there are no more empty tiles and that there are no more adjacent strings that can be merged. If this is satisfied then we print the LOSE string and exit the program.

Movement and Input Handling

```
# ========= MOVEMENT ==========
move_up:
        move_tiles $t1 $t4 $t7  # Merge and Compress Column 1 upward
        move_tiles $t2 $t5 $t8  # Merge and Compress Column 2 upward
        move_tiles $t3 $t6 $t9  # Merge and Compress Column 3 upward
        j       add_random_tile # Add a random tile
move_left:
        move_tiles $t1 $t2 $t3  # Merge and Compress Row 1 to the left
        move_tiles $t4 $t5 $t6  # Merge and Compress Row 2 to the left
        move_tiles $t7 $t8 $t9  # Merge and Compress Row 3 to the left
        j       add_random_tile # Add a random tile
move_down:
        move_tiles $t7 $t4 $t1  # Merge and Compress Column 1 downward
        move_tiles $t8 $t5 $t2  # Merge and Compress Column 2 downward
        move_tiles $t9 $t6 $t3  # Merge and Compress Column 3 downward
        j       add_random_tile # Add random tile
move_right:
        move_tiles $t3 $t2 $t1  # Merge and Compress Row 1 to the right
        move_tiles $t6 $t5 $t4  # Merge and Compress Row 2 to the right
        move_tiles $t9 $t8 $t7  # Merge and Compress Row 3 to the right
        j       add_random_tile # Add random tile
```

Input is handled by checking the ASCII value of the user's keystrokes. The **move_up**, **move_left**, **move_down**, and **move_right** functions handle the logic for each of these moves, adjusting the grid accordingly. Each of the move branches call the **move_tiles** macro, that passes the 3 tiles in a row or column onto the function variables $a0, $a1, $a2. The macro also calls the functions: **compress** and **merge**.

```
    .end_macro
# Move tiles - Move the row/column -> compress -> merge adjacent -> compress
    .macro move_tiles %r1 %r2 %r3
        move    $a0, %r1
        move    $a1, %r2
        move    $a2, %r3
        jal     compress
        jal     merge
        jal     compress
        move    %r1, $a0
        move    %r2, $a2
        move    %r3, $a2
    .end_macro
```

```
+---+---+---+
| $t1 | $t2 | $ t3 |
+---+---+---+
| $t4 | $t5 | $t6 |
+---+---+---+
| $t7 | $t8 | $t9 |
+---+---+---+
```

Each one of the move labels does the same thing depending on the orientation of the direction. For instance: when we move up and down, we modify by columns. When we move left and right, we modify by rows. Meaning we always only modify three registers per row.

The **compress** and **merge** functions manipulate the grid after a move, which are called in the move branches:

```
compress:
        bnez    $a1, compress_1 # If tile not empy
        move    $a1, $a2        # If tile empty move
        li      $a2, 0
compress_1:
        bnez    $a0, compress_2 # If tile not empty
        move    $a0, $a1        # If tile empty move
        li      $a1, 0
compress_2:
        bnez    $a1, compress_end # If tile not empty
        move    $a1, $a2          # If tile empty move
        li      $a2, 0
compress_end:
        jr      $ra               # return
```

```
merge:
        bne     $a0, $a1, merge_1 # If adjacent tiles not equal
        add     $a0, $a0, $a1     # Merge the first two tiles since equal >> add
        li      $a1, 0
merge_1:
        bne     $a1, $a2, merge_end # If adjacent last two tiles not equal
        add     $a1, $a1, $a2       # Merge the last two tiles since equal >> add
        li      $a2, 0
merge_end:
        jr      $ra
```

Compress: Moves tiles towards one side (up, down, left, or right)

Merge: Combines adjacent tiles of the same value, doubling them.

State Saving and Comparison

```
        syscall
# store the state of the grid!
store_origin:
        sw      $t1, ORIGIN_GRID
        sw      $t2, ORIGIN_GRID+4
        sw      $t3, ORIGIN_GRID+8
        sw      $t4, ORIGIN_GRID+12
        sw      $t5, ORIGIN_GRID+16
        sw      $t6, ORIGIN_GRID+20
        sw      $t7, ORIGIN_GRID+24
        sw      $t8, ORIGIN_GRID+28
        sw      $t9, ORIGIN_GRID+32
        jr      $ra
```

store_origin saves the initial configuration of the grid before a move so that it can later be compared for changes (useful for random tile addition). This checks if the move is valid by checking if the grid has changed by the inputs W/A/S/D, if the move is not valid, we tell the user that the move is “Invalid” and prompt the user to enter a move again.

```
# -------- CHECK BOARD STATE IF MOVE INPUT IS VALID ------------
check_origin:
        ##### preamble #####
        addi    $sp, $sp, -16   # Load the origin board
        sw      $s0, 0($sp)
        sw      $s1, 4($sp)
        sw      $s2, 8($sp)
        sw      $ra, 12($sp)
        ##### preamble #####
        li      $s0, 0                  # Change flag (0 = unchanged, 1 = changed)
        li      $s1, 0                  # Loop counter
check_origin_loop:
        # Calculate offset
        mul     $s2, $s1, 4             # Multiply counter by 4 (word size)
        # Load original and current values
        lw      $s2, ORIGIN_GRID($s2)   # Original value
        # Load current register value based on counter
        beq     $s1, 0, check_t1
        beq     $s1, 1, check_t2
        beq     $s1, 2, check_t3
        beq     $s1, 3, check_t4
        beq     $s1, 4, check_t5
        beq     $s1, 5, check_t6
        beq     $s1, 6, check_t7
        beq     $s1, 7, check_t8
        beq     $s1, 8, check_t9
```

```
check_t1: check_tile_change $s2, $t1
check_t2: check_tile_change $s2, $t2
check_t3: check_tile_change $s2, $t3
check_t4: check_tile_change $s2, $t4
check_t5: check_tile_change $s2, $t5
check_t6: check_tile_change $s2, $t6
check_t7: check_tile_change $s2, $t7
check_t8: check_tile_change $s2, $t8
check_t9: check_tile_change $s2, $t9
# Check if changed
        .macro check_tile_change %reg1 %reg2
                bne %reg1, %reg2, grid_changed
        j       continue_loop
        .end_macro
```

```
continue_loop:
        # Increment counter
        addi    $s1, $s1, 1
        blt     $s1, 9, check_origin_loop

        # If we get here, no changes detected
        j       check_result
grid_changed:
        # Set change flag
        li      $s0, 1
check_result:
        # Result based on change flag
        beq     $s0, 1, return_nothing
        # Else
        j       return_invalid
return_nothing:
        ##### end #####
        lw      $s0, 0($sp)
        lw      $s1, 4($sp)
        lw      $s2, 8($sp)
        lw      $ra, 12($sp)
        addi    $sp, $sp, 16
        ##### end #####

        jr      $ra
return_invalid:
        lw      $s0, 0($sp)
        lw      $s1, 4($sp)
        lw      $s2, 8($sp)
        lw      $ra, 12($sp)
        addi    $sp, $sp, 16
        ##### end #####
        j       invalid_move
```

The **check_origin** function checks if the current board state has changed compared to the saved state. It uses the store_origin function to compare the current grid state with the saved state. If the grid has changed (i.e., a move was made), it updates the s0 register to reflect this. **check_origin_loop** is where the comparison between the saved (origin) grid and the current board grid happens. mul $s2, $s1, 4 calculates the offset for the current tile in the grid. Each tile is 4 bytes, so the loop counter ($s1) is multiplied by 4 to get the correct byte offset for the tile being checked. Each tile is checked by checking its index position in the board. [0-8]. It branches to check_tile: which calls the macro **check_tile_change**, which just checks if there is a change in the tile. If no change was detected ($t0 == $t1), the loop counter ($s1) is incremented by 1 (addi $s1, $s1, 1), and the loop continues.

If the loop finishes and no changes are detected (i.e., all tiles match), the function jumps to the return address (jr $ra), signaling no change. If a change was detected (i.e., the tiles were different at any position), the origin_changed label is triggered:
li $s0, 1: This sets the flag in $s0 to 1, indicating that the board has changed.
jr $ra: The function then returns, completing the check.
When the board does not change, the move is invalid. Prompting the user to put a valid input.

Limitations and Challenges
Our MIPS implementation utilizes brute force and exhaustion, because it is only a 3x3 grid with 9 tiles. Since we assigned one register per one tile, we exhaustively compared each tile when checking for the win and lose states, as well as when we update its values. This can be quite tedious and time-consuming to implement seeing that our program has about 500 lines of code. This also means that our implementation is not *scalable*. For an NxN grid, it would be harder to compare n tiles to each possible win/lose state. For a specific limitation in our implementation, when starting a game from a state we cannot “X” \quit in the middle of setting the states of the tiles.

The most difficult part in implementing the 2048 game was figuring out how to do the movement algorithm. It took some time to fully understand and grasp it, but it was resolved by doing 3 simple steps: compress (pushing the numbers to the edge) -> merge (merge/add the adjacent values) and -> compress again.

Example Gameplay

Below are example gameplays from our game.

New Game

Below is the Main Menu for starting a new game, we input ‘1’

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
+---+---+---+
| 2 | 2 |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG):
A
```

Above is the generated initial state for the board.
Below is the game play-through in which we terminate after 4 rounds.

```
+---+---+---+
| 2 | 2 |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
Enter a move (
A
```

Initial Board State

```
Enter a move
A
+---+---+---+
| 4 |   | 2 |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
```

```
Enter a move
S
+---+---+---+
|   |   |   |
+---+---+---+
|   | 2 |   |
+---+---+---+
| 4 |   | 2 |
+---+---+---+
```

```
Enter a move
S
+---+---+---+
|   |   |   |
+---+---+---+
|   | 2 |   |
+---+---+---+
| 4 | 2 | 2 |
+---+---+---+
```

```
Enter a move
A
+---+---+---+
|   |   |   |
+---+---+---+
| 2 |   | 2 |
+---+---+---+
| 4 | 4 |   |
+---+---+---+
```

We have the Inputs: “A” -> “S” -> “S” -> “A”

That is we move **LEFT -> DOWN -> DOWN -> LEFT.**
From the images above, you can see that the game plays as intended.

```
+---+---+---+
|   |   | 2 |
+---+---+---+
| 4 |   |   |
+---+---+---+
| 8 |   |   |
+---+---+---+
Enter a move (W=Up,
X
Program Terminated.
```

Input: “X” terminates the program

An input “X” will terminate the program.

Starting from a State

Below is the Main Menu for starting from a state, we input ‘2’’

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
2
Enter a board configuration (Invalid integer = back to Main menu):
128
64
64
32
256
128
0
0
0
```

From the inputs above we have the following initial Board:

```
+---+---+---+
|128| 64| 64|
+---+---+---+
| 32|256|128|
+---+---+---+
|   |   |   |
+---+---+---+
Enter a move
D
```

Initial Board State

```
Enter a move (
D
+---+---+---+
|   |128|128|
+---+---+---+
| 32|256|128|
+---+---+---+
|   | 2 |   |
+---+---+---+
```

Input “D”, move RIGHT (Merges 64 & 64)

```
Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG):
3
RNG Disabled.
```

Input “3”, disables RNG (Disable spawning of a 2)

```
Enter a move (
A
+---+---+---+
|256|   |   |
+---+---+---+
| 32|256|128|
+---+---+---+
| 2 |   |   |
+---+---+---+
```

Input “A”, move LEFT

Notice that no tile has spawned a “2”, and the existing tiles move to the Left.

```
Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG):
4
RNG Enabled.
```

Input “4”, enables RNG (Enable spawning of a 2)

```
Enter a move (
D
+---+---+---+
|   | 2 |256|
+---+---+---+
| 32|256|128|
+---+---+---+
|   |   | 2 |
+---+---+---+
```

```
Enter a move (
A
+---+---+---+
| 2 |256|   |
+---+---+---+
| 32|256|128|
+---+---+---+
| 2 |   | 2 |
+---+---+---+
```

```
Enter a move (W=Up, A=Left, S=Down, D=Right, X=Qu
S
+---+---+---+
| 2 | 2 |   |
+---+---+---+
| 32|   |128|
+---+---+---+
| 2 |512| 2 |
+---+---+---+
Congratulations! You have reached the 512 tile!
-- program is finished running --
```

Inputs “D” -> “A” -> “S” moves the tiles RIGHT -> LEFT -> DOWN triggering win state

Notice now that with RNG enabled, “2” tiles are spawning. After the last input (D) a 512 tile has appeared, triggering the win state and terminating the program.

Losing State

```
Enter a board configuration (Invalid integer = back to Main menu):
8
16
32
0
64
0
0
128
0
+---+---+---+
| 8 | 16| 32|
+---+---+---+
|   | 64|   |
+---+---+---+
|   |128|   |
+---+---+---+
```

From the inputs above we have the following initial board state:

```
+---+---+---+
| 8 | 16| 32|
+---+---+---+
|   | 64|   |
+---+---+---+
|   |128|   |
+---+---+---+
```

Initial Board State

```
Enter a move
A
+---+---+---+
| 8 | 16| 32|
+---+---+---+
| 64|   |   |
+---+---+---+
|128|   | 2 |
+---+---+---+
```

```
Enter a move
S
+---+---+---+
| 8 |   | 2 |
+---+---+---+
| 64|   | 32|
+---+---+---+
|128| 16| 2 |
+---+---+---+
```

```
Enter a move
D
+---+---+---+
|   | 8 | 2 |
+---+---+---+
| 2 | 64| 32|
+---+---+---+
|128| 16| 2 |
+---+---+---+
```

```
Enter a move
W
+---+---+---+
| 2 | 8 | 2 |
+---+---+---+
|128| 64| 32|
+---+---+---+
| 2 | 16| 2 |
+---+---+---+
Game Over.
```

We have the Inputs: “A” -> “S” -> “D” -> “W”

That is we move **LEFT -> DOWN -> RIGHT -> UP.**
In the last frame, we see that the Lose State appears since there are no more possible moves. The grid is full and there are no more valid moves since there are no adjacent tiles with the same numbers.

Thank you for reading our documentation!