CS 21 Project: 2048 in MIPS (Part II)

Jan Albert Quidet
Judelle Clareese Gaza

Project Specifications

Part II of the project involves extending our initial 2024 MIPS game by extending the game board and adding some features to the game.

Our current implementation has the following features:

- Start a New Game
- Start from a State
- Choose Board size N
- Moves Tracking
- Score Tracking
- Single-move Undo
- A Free Feature Cheat Code

Data and Macros

To make things easier to understand, first let us discuss the macros and the data used in our program.

```
.data
WELCOME:        .asciiz "Main Menu:\n[1] New Game\n[2] Start from a State\n[X] Quit\n"
GRID_SIZE:      .asciiz "Input the size of the board: \n"
BOARD_CONFIG:   .asciiz "Enter a board configuration (Invalid input = back to Main menu):\n"
INVALID_INPUT:  .asciiz "Invalid input.\n"
QUIT:           .asciiz "Program Terminated."
MOVE_NUMBER:    .asciiz "Moves:"
SCORE:          .asciiz " Score:"
NEWLINE:        .asciiz "\n"
BORDER:         .asciiz "+----"
CORNER:         .asciiz "+\n"
HEDGE:          .asciiz "|"
EDGE:           .asciiz "|\n"
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"
NUMBER_1024:    .asciiz "1024"
NUMBER_2048:    .asciiz "2048"
WIN:            .asciiz "Congratulations! You have reached the 2048 tile!"
LOSE:           .asciiz "Game Over."
ENTER_MOVE:     .asciiz "Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG, z=Undo):\n"
RNG_DISABLED:   .asciiz "RNG Disabled.\n"
RNG_ENABLED:    .asciiz "RNG Enabled.\n"
                .align 2
MOVE:           .space  4               # We store user input here
N:              .space  4               # We store the indicated number N for the NxN grid
NxN:            .space  4               # value of NxN
GRID_BASE:      .space  4               # grid base address
PREV_GRID:      .space  4               # prev grid base address
TEMP_GRID:      .space  4               # temporary grid for rotate and reverse functions
RNG_FLAG:       .word   4               # 3 = disabled, 4 = enabled
CURRENT_MOVES:  .word   0
CURRENT_SCORE:  .word   0
```

Above is the data segment of our code, which are the strings to be used for prompting and printing, as well as the variable names we will be using to reference in our code for easier understanding. You can refer to the comments for the variables used in our program.

```
# ---------------------- MACROS TO REDUCE REDUNDANCY ----------------------
# Print String
    .macro print_str %str
        li      $v0, 4
        la      $a0, %str
        syscall
    .end_macro
# Read String
    .macro read_str %address
        li      $v0, 8                          # Syscall for reading string
        la      $a0, %address                   # Address of buffer to store input
        li      $a1, 100                        # Maximum length of input
        syscall                                 # Read input from user
    .end_macro
# Print Integer
    .macro print_int %int
        li      $v0, 1
        lw      $a0, %int
        syscall
    .end_macro
# Read Integer
    .macro read_int %address
        li      $v0, 5                          # Syscall for reading string
        syscall                                 # Read input from user
        sw      $v0, %address
    .end_macro
# Randomizer
    .macro randomize %upperbound_reg
        move    $a1, %upperbound_reg            # Upperbound when generating a num
        li      $v0, 42                         # Randomizer syscall
        syscall
    .end_macro
# Incrementer
    .macro increment %reg
        addi    %reg, %reg, 1
    .end_macro
# Input the state of a tile
    .macro set_tile %reg
        li      $v0, 5
        syscall
        move    $a0, $v0
        jal     check_integer
        sw      $a0, (%reg)
    .end_macro
```

Above are our macros, `print_str`, `read_str`, `print_int`, `read_int` are self-explanatory macros, these are the macros responsible for printing strings, reading string inputs, printing integers, and reading integer inputs.

We also have the following macros that will help reduce the redundancy of our program: `randomize`, `increment`, `set_tile`.

`randomize` is a macro also used in the PART I of our project. It utilizes syscall 42, and takes in an upper bound value taken from the register.

`increment` is a macro that is also self explanatory, it increments the value of a data in register by 1.

`set_tile` is a helper macro used for setting the state of a tile, which is a helpful macro when creating a game from a state.

Part II changes
Starting a Game

```
# ---------------------- MAIN PROGRAM ----------------------
.text
# First Load Initialize the Board
main:
        print_str WELCOME               # Print MAIN MENU PROMPT
        read_str MOVE                   # Read User Input STORE IN MOVE
        print_str GRID_SIZE             # Ask for the grid size N
        read_int N                      # Read User Input STORE IN N

        lw      $a0, N
        jal     grid_malloc             # memory allocation function
        sw      $v0, GRID_BASE
        sw      $v1, NxN

        jal     make_history_grid
        jal     make_temp_grid
main_move:
        lw      $a0, MOVE               # load MOVE to a0
        beq     $a0, 0xa31, input_new   # If NEW GAME
        beq     $a0, 0xa32, input_state # If Start from a STATE
        beq     $a0, 0xa58, terminate   # X = Quit

        print_str INVALID_INPUT         # otherwise, Print INVALID INPUT
        j       main_move               # Loop back to main (main menu)
```

Similar to the previous project, the code above is responsible for initializing the board state whether we start from a New Game by generating 2 random tiles or by starting from a state.
Quick line-by-line walkthrough:

- First we print the **WELCOME** string:
  `"Main Menu:\n[1] New Game\n[2] Start from a State\n[X] Quit\n"`
- Then read and save the input to **MOVE**
- We then ask the user for the grid size by prompting with string:
  `"Input the size of the board: \n"`
- Then read and save the board size N input to **N**
- We then pass **N** as a function to `grid_malloc` by loading it to register $a0
- We call the `grid_malloc` function

grid_malloc:

```
grid_malloc:
        move    $s0, $a0                # Move input to s0
        mul     $s1, $s0, $s0           # S1 = S0 x S0 (NxN)
        mul     $s2, $s1, 4             # S2 = S1 x 4 -> This calculates the total memory required for the grid, assuming each element is 4 bytes

        li      $v0, 9
        move    $a0, $s2                # Memory allocation for the NxN grid in heap
        syscall
        move    $v1, $s1                # $v1 = NxN

        jr      $ra
```

Above is the `grid_malloc` function where we create the heap memory allocation for our grid. First we

take the input that is N (stored in $a0) multiply it by itself to for each "cell" in the grid, resulting to NxN, then we take this result (NxN) and multiply it by 4 (NxNx4), in which the result is the total memory required for the grid, since each cell in the grid (which are actually just integers) will have 4 bytes.
We then call **sbrk** to initialize the heap, by using the syscall 9 and allocate the necessary memory.

sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory

- Then we store the outputs of the function to **GRID_BASE** and **NxN** where
  - **GRID_BASE** is the base address of the allocated grid memory
  - And **NxN** is the the size of the grid
- We then call the functions: `make_history_grid` and `make_temp_grid`

```
make_history_grid:
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        lw      $a0, N
        jal     grid_malloc             # memory allocation function
        sw      $v0, PREV_GRID
        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr      $ra
make_temp_grid:
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        lw      $a0, N
        jal     grid_malloc             # memory allocation function
        sw      $v0, TEMP_GRID
        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr      $ra
```

Each function allocates and creates two grids: **PREV_GRID** and **TEMP_GRID**, wherein **PREV_GRID** will store the previous grid for the single undo function and where **TEMP_GRID** will be a temporary grid used for the rotate and reverse functions for movement.

- We then branch accordingly depending on the value of the player **MOVE,** wherein we create a **New Game** or **Start from a State** or **terminate the program.**
- If the player inputs are none of the following we print the input to be invalid and keep looping back to the `main_move` to ask for user input.

Initializing from New Game

```
# ======= NEW GAME ==========
input_new:                              # NEW GAME 1
        li      $t0, 0                  # counter so that there will be only two 2-tiles in the grid
        lw      $s0, NxN                # value of NxN
        lw      $s1, GRID_BASE          # grid base address
grid_position_reset:
        li      $t1, 0                  # grid position counter
zero_or_two_loop:
        sll     $t2, $t1, 2             # Multiply index by 4 for word offset
        add     $t2, $t2, $s1           # Get address of current element
        jal     zero_or_two             # Call zero_or_two function
        sw      $v0, ($t2)              # address will now have value of 0 or 2

        increment $t1
        beq     $t0, 2, main_loop               # If $t0 = 2, tile is ready to be printed
        beq     $t1, $s0, grid_position_reset   # If reached (N, N) in the grid, restart back to (0, 0)
        j       zero_or_two_loop                # loop back if conditions are not met

# --------------------- FUNCTIONS ---------------------
zero_or_two:
        li      $t3, 2
        randomize $t3                   # 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
        increment $t0                   # increment $t0 >> there are now $t0 tiles in the board
        jr      $ra                     # return 2
# --------------------- END ---------------------
```

Above is where we branch to if we wish to initialize from a New Game, similar to our previous implementation. The `input_new` function initializes the game grid for a new game, ensuring that exactly two tiles contain the value 2, while the rest are 0. It iterates through the grid positions, calculates the memory address of each cell, and uses `zero_or_two` to determine whether the cell will hold a 0 or a 2. Randomization is applied to decide the cell's value, but the function enforces a rule that no more than two 2-tiles are placed by maintaining a counter. If two 2-tiles are successfully placed, the function exits; otherwise, it resets and continues until the condition is met. This setup ensures that every new game begins with a randomized grid containing two initial tiles, replicating the starting state of the 2048 game.

Initializing from a State

```
# ======= NEW GAME FROM STATE ==========
input_state:                            # NEW GAME 2
        print_str BOARD_CONFIG
        li      $t0, 0                  # grid position counter
        lw      $s0, NxN                # value of NxN
        lw      $s1, GRID_BASE          # grid base address
board_config_loop:
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t1, $t1, $s1           # Get address of current position
        set_tile $t1

        increment $t0
        blt     $t0, $s0, board_config_loop

        j       main_loop
```

The `input_state` function initializes a new game from a predefined board configuration provided by the user. It starts by displaying a prompt (`BOARD_CONFIG`) and iterates through each grid position using a counter. For each position, it calculates the memory address within the grid using offsets and calls the `set_tile` function to set the value of the tile based on user input. The process continues until all positions in the grid are configured. Once completed, the function transitions to the main game loop (`main_loop`), allowing the game to proceed with the customized board setup. This feature allows users to start the game with specific tile values instead of a default grid.

Game Loop

```
# ===== Main Loop - print grid - check win state - get input =====
main_loop:
        jal     calculate_score
        jal     print_grid              # Print the Grid
        jal     is_win                  # Check if win
        jal     is_lose                 # Check if lose
input_move:     # Get User Input
        sw      $zero, MOVE             # Reset MOVE value to 0

        print_str ENTER_MOVE            # Prompt Move
        read_str MOVE                   # Get Input

        lw      $t0, MOVE
        beq     $t0, 0xa58, terminate   # X = Quit
        beq     $t0, 0xa33, rng_disable # 3 = RNG disable
        beq     $t0, 0xa34, rng_enable  # 4 = RNG enable
        beq     $t0, 0xa7a, undo        # z = undo
        beq     $t0, 0x31325343, cheat  # CS21 cheat code

        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

# Skipped if move is valid -- else we jump here if board changed
invalid_move:
        print_str INVALID_INPUT         # Print INVALID INPUT
        j       input_move              # Loop back to input_move
```

Above is the main game loop of our 2048 game. This is after the creation of the game state (whether we start from a new game or if we initialize from a game state). The main game loop will do the following:

- Call the function `calculate_score` to calculate the current score in the **n x n** board
- Print the grid
- Check if the current state of the game is a win state
- Check if the current state of the game is a lose state
- Take the input of the user:
    - First we print the prompt `"Enter a move (W=Up, A=Left, S=Down, D=Right, X=Quit, 3=Disable RNG, 4=Enable RNG, z=Undo):\n"`
    - Then we read the prompt and store in MOVE
- After the taking the input of the user, the program branches accordingly:
    - X to terminate
    - 3 to RNG Disable

- 4 to RNG Enable
- z to for Single-move Undo
- And our **SECRET CHEAT CODE!!! <3**
    - Our cheat code corresponds to the string "**CS21**"
    - This is our bonus feature for task 5 :>>
    - Inputting this secret cheat code will double the values of all tiles in the grid.
- W to move up
- A to move left
- S to move down
- D to move right

- If the user input is a WASD or the cheat code move, we first store the current grid by calling the `store_current` function

We will discuss the printing of the grid, the movements, and the score tracking, single-move undo, the secret code in their respective sections. I will first discuss the win conditions, and the RNG enabling in the sections below.

Checking win conditions

The `is_win` function checks if any tile in the grid has the value 2048, and if found, it prints "WIN" and exits the game. The `is_lose` function checks if the player has lost by ensuring that all tiles are filled and there are no adjacent **tiles that can combine. If no moves are possible, it prints "LOSE" and ends the game.**

is_win:

```
is_win:
        li      $t0, 0                  # grid position counter
        lw      $s0, NxN                # value of NxN
        lw      $s1, GRID_BASE          # grid base address
is_win_loop:
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t1, $t1, $s1           # Get address of current position
        lw      $t2, ($t1)              # Load current element
        beq     $t2, 2048, win

        increment $t0
        blt     $t0, $s0, is_win_loop

        jr      $ra
win:
        print_str WIN                   # End if WIN!
        j       exit
```

Initialization:

- A counter (`$t0`) is initialized to `0`, which will be used to iterate through all grid positions.
- The value of `NxN` (size of the grid) is loaded into `$s0`.
- The base address of the grid (`GRID_BASE`) is loaded into `$s1`.

Loop to Check Each Grid Position:

- The `is_win_loop` loop iterates over every grid position by incrementing `$t0`.
- The index `$t0` is multiplied by 4 (`sll $t1, $t0, 2`) to compute the memory offset for the grid position since each tile is a word (4 bytes).
- The address of the current grid position is computed and loaded into `$t1`, and the value at that position is loaded into `$t2`.
- If the value of the tile is `2048`, the game has been won, and the program jumps to the `win` label.
- If no `2048` tile is found after checking all positions, the function exits.

Win Condition:

- When a tile with value `2048` is found, the string "WIN" is printed, and the program jumps to the `exit` label, ending the game.

Exit:

- If no winning condition is met, control is returned to the caller (`jr $ra`).

is_lose:

```
is_lose:
        li      $t0, 0              # grid position counter
        lw      $s0, N
        lw      $s1, NxN            # value of NxN
        lw      $s2, GRID_BASE      # grid base address
is_lose_loop:
        sll     $t1, $t0, 2         # Multiply index by 4 for word offset
        add     $t1, $t1, $s2       # Get address of current position
        lw      $t2, ($t1)          # Load current element
        beqz    $t2, False          # Check if there's an empty tile

        # check right
        addi    $t3, $t0, 1         # check right position
        div     $t3, $s0
        mfhi    $t3
        blt     $t3, $t0, check_down  # Skip if out of bounds
        lw      $t4, 4($t1)         # Load right value
        beq     $t2, $t4, False
check_down:
        add     $t3, $t0, $s0       # get y coordinate
        bge     $t3, $s1, skip_down # Skip if out of bounds
        sll     $t3, $t3, 2         # N * 4
        add     $t3, $t3, $s2       # adds offset to original position's address
        lw      $t4, ($t3)          # Load down value
        beq     $t2, $t4, False
skip_down:
        increment $t0
        blt     $t0, $s1, is_lose_loop

        print_str LOSE              # otherwise, game over
        j exit
False:
        jr      $ra
```

The `is_lose` function checks if the player has lost by verifying two conditions:

- There are no empty spaces left on the grid (i.e., no tile with value `0`).
- No adjacent tiles (either horizontally or vertically) can combine.
- Initialization:
    - Similar to the `is_win` function, the counter `$t0` is initialized to `0`, and the values of `N` (grid size), `NxN` (total grid positions), and `GRID_BASE` (grid base address) are loaded into `$s0`, `$s1`, and `$s2`, respectively.
- Loop to Check Each Grid Position:
    - The `is_lose_loop` loop checks each tile on the grid.
    - If a tile is `0`, it means there is an empty space, so the function will jump to `False` and not check for the lose-condition.
- Horizontal and Vertical Check for Adjacent Tiles:
    - Right Check: The right-adjacent tile is checked by incrementing the index `$t0` and ensuring it is within bounds. If the value of the current tile (`$t2`) matches the value of the tile to the right (`$t4`), the game is not over.
    - Down Check: Similarly, the down-adjacent tile is checked by adding `N` to the current position (`$t3`). If the current tile (`$t2`) matches the value of the tile below (`$t4`), the game is not over.
- Lose Condition:
    - If all grid positions are filled and no adjacent tiles can combine (i.e., there are no possible moves), the string "LOSE" is printed, indicating that the game is over, and the program jumps to the `exit` label.
- Exit:
    - If neither of the conditions for a loss is met, the function returns control to the caller (`jr $ra`).

RNG Enable\Disable

```
rng_disable:
        li      $s0, 3
        sw      $s0, RNG_FLAG
        print_str RNG_DISABLED
        j       input_move
rng_enable:
        li      $s0, 4
        sw      $s0, RNG_FLAG
        print_str RNG_ENABLED
        j       input_move
```

`rng_disable` sets the RNG flag to a value representing a disabled state (3), prints a message indicating the RNG is disabled, and then jumps to the `input_move` label.

`rng_enable` sets the RNG flag to a value representing an enabled state (4), prints a message indicating the RNG is enabled, and then jumps to the `input_move` label.

Task 1: Extending the Game Board

For this part of the project we chose to pursue the **n × n** grid board extension, wherein the user can specify the size of the board. Prompt the user for the number n, then make the board size **n × n.**

Below are grid boards with sizes **3, 4, 5, 6, 7, 8.**

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board:
3
Moves:0 Score:4
+----+----+----+
|  2 |  2 |    |
+----+----+----+
|    |    |    |
+----+----+----+
|    |    |    |
+----+----+----+
```

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board:
4
Moves:0 Score:4
+----+----+----+----+
|    |  2 |    |  2 |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
```

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board
5
Moves:0 Score:4
+----+----+----+----+----+
|    |  2 |  2 |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
+----+----+----+----+----+
|    |    |    |    |    |
+----+----+----+----+----+
```

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board:
6
Moves:0 Score:4
+----+----+----+----+----+----+
|  2 |  2 |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
```

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board:
7
Moves:0 Score:4
+----+----+----+----+----+----+----+
|    |    |    |  2 |    |  2 |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+
```

```
Main Menu:
[1] New Game
[2] Start from a State
[X] Quit
1
Input the size of the board:
10
Moves:0 Score:4
+----+----+----+----+----+----+----+----+----+----+
|    |  2 |  2 |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
```

print_grid:

```
print_grid:                         # GRID PRINTING
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        print_str MOVE_NUMBER     # Print string "Move:"
        print_int CURRENT_MOVES   # Print the counter
        print_str SCORE           # Print string "Score:"
        print_int CURRENT_SCORE   # Print the score
        print_str NEWLINE         # New line

        jal     print_border
        jal     print_rows

        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr      $ra
```

Above is the section of our MIPS code responsible for printing the grid. First we initialize the preamble, then print the necessary statistics. Print the string "Move:" along with the integer current_moves. We also print the string "Score:" along with the current score state of the board. After that we call the functions `print_border`, and `print_rows`.

print_border:

```
print_border:
        ##### preamble #####
        addi    $sp, $sp, -8
        sw      $s0, 0($sp)
        sw      $t0, 4($sp)
        ##### preamble #####
        li      $t0, 0                  # grid position counter
        lw      $s0, N                  # value of N
border_loop:
        print_str BORDER
        increment $t0
        blt     $t0, $s0, border_loop
        print_str CORNER

        ##### end #####
        lw      $s0, 0($sp)
        lw      $t0, 4($sp)
        addi    $sp, $sp, 8
        ##### end #####
        jr      $ra
```

The `print_border` function prints the top and bottom borders of the grid. This function does the following:

Preamble: It saves the state by storing `$s0` and `$t0` registers to the stack.

Loop for Printing Borders: border_loop is a loop used to print the horizontal border (`BORDER`) `N` times.

After the loop, a corner (`CORNER`) is printed at the end which is the "+" character.

End: The state is restored from the stack, and the function returns control to the caller.

print_rows:

```
print_rows:
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        ##### preamble #####
        li      $t0, 0                  # grid position counter
        lw      $s0, N                  # value of N
        lw      $s1, NxN                # value of NxN
        lw      $s2, GRID_BASE          # grid base address
row_loop:
        print_str HEDGE
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t1, $t1, $s2           # Get address of current position
        lw      $a0, ($t1)              # Load current element
        jal     print_integer

        addi    $t1, $t0, 1
        div     $t1, $s0
        mfhi    $t2
        bnez    $t2, is_not_edge
        print_str EDGE
        jal     print_border
```

The `print_rows` function prints the individual rows of the grid.

- Preamble: It saves the return address (`$ra`) to the stack.
- Loop for Printing Rows: It loops through all grid positions and prints the elements in each row.

- Each element is printed using `print_integer`, which calls `get_integer` to convert the tile value into a string representation (e.g., "0", "2", "4").
- If an element is at the edge of the grid (i.e., a corner or border row), the EDGE string is printed, followed by a border.

- End: The state is restored from the stack, and the function returns to the caller.

```
is_not_edge:
        increment $t0
        blt     $t0, $s1, row_loop

        ##### end #####
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        jr      $ra

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
        beq     $a0, 1024, print_1024
        beq     $a0, 2048, print_2048
```

```
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_1024:
        la      $v0, NUMBER_1024
        jr      $ra
print_2048:
        la      $v0, NUMBER_2048
        jr      $ra
```

The `get_integer` function maps specific tile values (like 0, 2, 4, 8, etc.) to predefined string labels (`NUMBER_0`, `NUMBER_2`, etc.) and loads the corresponding address into `$v0` for printing.

- The function checks the value of the input integer (`$a0`) and loads the corresponding string address into `$v0` using `la`.
- For example, if `$a0` is 2, it loads the address of `NUMBER_2` into `$v0`.

Task 2: Moves Tracking

Next we will discuss how movement is handled in our game. In this section the **TEMP_GRID** will be utilized to move the tiles. First it's important to note that each movement is called the `compress`, and `merge` functions, which by default compresses and merges towards the left of the grid. Which is why, when we call other directional movements like `move_up`, `move_down`, `move_right`, we have helper functions to rotate the grid accordingly to orient it properly when moving.

```
move_up:
        jal     store_current

        jal     rotate_right
        jal     rotate_right
        jal     rotate_right

        jal     compress
        jal     merge
        jal     compress

        jal     rotate_right
        j       add_random_tile         # Add a random tile
move_left:
        jal     store_current

        jal     compress
        jal     merge
        jal     compress

        j       add_random_tile         # Add a random tile
move_down:
        jal     store_current

        jal     rotate_right

        jal     compress
        jal     merge
        jal     compress

        jal     rotate_right
        jal     rotate_right
        jal     rotate_right
        j       add_random_tile         # Add a random tile
move_right:
        jal     store_current

        jal     reverse

        jal     compress
        jal     merge
        jal     compress

        jal     reverse
        j       add_random_tile         # Add a random tile
```

Each of the following branches above have the common instructions:

- `jal store_current`:
  - This call saves the current board state, which will be used for undo functionality and tracking purposes.
- `jal compress`:
  - This operation moves all tiles to one side of the board, removing gaps.
- `jal merge`:
  - This operation merges adjacent tiles with the same value.
- `jal add_random_tile`:
  - After processing, a random tile is added to an empty cell on the board.
  - This is also where the **CURRENT_MOVES** is updated which tracks and counts the player currently has.
- Rotation and Reversal:
  - The functions utilize rotations and reversals to adjust the board's orientation, allowing reuse of the `compress` and `merge` logic for all directions.

compress:

```
compress:
        li      $t0, 0                  # grid position counter
        lw      $s0, GRID_BASE          # grid base address
        lw      $s1, NxN                # value of NxN
        li      $s2, 0                  # nonzero position counter
        lw      $s3, N                  # value of N
compress_loop:
        addi    $t4, $t0, 1
        div     $t4, $s3
        mfhi    $t4                     # index mod N = row counter

        # Load current element
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t1, $t1, $s0           # Get address of current element
        lw      $t2, ($t1)              # Load current element

        # Check if current element is non-zero
        beqz    $t2, zero_tile          # If zero, skip to next element

        # If non-zero, place at leftmost available position
        sll     $t3, $s2, 2             # Get offset for non-zero position
        add     $t3, $t3, $s0           # Get address for placement
        sw      $t2, ($t3)              # Store non-zero element
        increment $s2                   # Increment non-zero position counter
zero_tile:
        increment $t0                   # Increment loop counter
        beqz    $t4, autofill_loop      # If reached end of row, fill rest with zeros
        j       compress_loop
autofill_loop:
        div     $s2, $s3
        mfhi    $t4                     # index mod N = row counter
        beqz    $t4, compress_next_row  # If reached end of row, go to next row

        sll     $t1, $s2, 2             # Get offset
        add     $t1, $t1, $s0           # Get address
        sw      $zero, ($t1)            # Store zero
        increment $s2                   # Increment counter

        j       autofill_loop
compress_next_row:
        move    $s2, $t0
        blt     $t0, $s1, compress_loop
        jr      $ra
```

The compress function organizes tiles in each row of a grid by shifting all non-zero elements to the left while maintaining their order. It also fills the remaining positions in the row with zeros which represents a white space.

`compress_loop`

The loop iterates over all grid positions and processes each element:

- Non-Zero Tile Handling:
  - If the current grid element ($t2) is non-zero:
    - Place it in the leftmost available position (tracked by $s2).
    - Increment $s2 to point to the next available slot for a non-zero tile.
- Skip Zero Tile:
  - If the current element is zero, it's ignored.

`autofill_loop`

At the end of each row:

- Calculate index mod N to determine the current row position.
- If there are remaining positions in the row, fill them with zeros.
- Increment $s2 until the end of the row is reached.

Row Transition

When one row is completed:

- Reset $s2 to the starting position of the next row (move $s2, $t0).
- Check if the loop should continue to process more rows or terminate.

`autofill_loop`

- If there are unfilled slots in the row (determined by mod N), write zeros in the remaining positions.
- Increment the $s2 counter until the row is complete.

End of Compression

When the grid position counter $t0 reaches $s1 (the total number of elements), the function completes and returns to the caller with jr $ra.

merge:

```
merge:
        li      $t0, 0                  # grid position counter
        lw      $s0, GRID_BASE          # grid base address
        lw      $s1, NxN                # value of NxN
        lw      $s2, N                  # value of N
merge_loop:
        div     $t0, $s2
        mfhi    $t7                     # index mod N = row counter

        addi    $t1, $t0, 1
        div     $t1, $s2
        mfhi    $t8                     # index mod N = next counter

        bgt     $t7, $t8, skip          # If current row index is end, next row

        # Load current and next elements
        sll     $t2, $t0, 2             # Current index * 4
        add     $t2, $t2, $s0           # Current address
        lw      $t3, ($t2)              # Current value

        sll     $t4, $t1, 2             # Next index * 4
        add     $t4, $t4, $s0           # Next address
        lw      $t5, ($t4)              # Next value

        # Compare values
        bne     $t3, $t5, skip          # If not equal, skip

        # Merge equal values
        add     $t6, $t3, $t5           # Add values
        sw      $t6, ($t2)              # Store sum in first position
        sw      $zero, ($t4)            # Store zero in second position
skip:
        increment $t0
        blt     $t0, $s1, merge_loop
        jr      $ra
```

The merge function in this MIPS code merges adjacent equal values in each row of the grid. This function will Identify pairs of adjacent tiles in each row that have the same value. Merge these tiles by adding their values together. Replace the second tile of the pair with zero (to indicate it's now empty).

A quick rundown of the function would be the following:

`merge`:

- The grid position counter ($t0) starts at the first tile.
- GRID_BASE holds the starting address of the grid, NxN is the total number of tiles, and N is the row length.

Row Boundary Check `(merge_loop and skip)`:

- computes the current tile's row index ($t7) and the next tile's row index ($t8) using the modulus operation (div and mfhi).
- If the two tiles are in different rows, it skips the merge process for this tile (label skip).

Tile Comparison `(merge_loop and skip)`:

- If the two tiles are in the same row, their values are loaded into $t3 (current tile) and $t5 (next tile).
- If the values are different, no merging happens, and the function proceeds to the next tile (skip).

Merging `(merge_loop and skip)`:

- If the tiles have the same value:
  - Their sum is stored in the current tile's position ($t6 is written to the current tile address).
  - A zero is written to the next tile's position to "clear" it.

Incrementing Position `(merge_loop)`:

- The grid position counter ($t0) is incremented, and the process repeats until all tiles have been processed.

`merge_loop Exit`:

- Once all positions have been examined ($t0 reaches NxN), the function returns to the caller (jr $ra).

rotate_right:

```
rotate_right:
        lw      $s0, GRID_BASE          # Grid base address
        lw      $s1, TEMP_GRID          # Temp array base address
        lw      $s2, N                  # N (size)

        li      $t0, 0                  # row counter
transfer_row_loop:
        li      $t1, 0                  # column counter
transfer_col_loop:
        # Calculate source index: row * N + col
        mul     $t2, $t0, $s2
        add     $t2, $t2, $t1
        sll     $t2, $t2, 2             # multiply by 4 for word offset
        add     $t2, $t2, $s0
        lw      $t3, ($t2)              # load value

        # Calculate destination index: col * N + (N-1-row)
        mul     $t4, $t1, $s2
        sub     $t5, $s2, 1
        sub     $t5, $t5, $t0
        add     $t4, $t4, $t5
        sll     $t4, $t4, 2
        add     $t4, $t4, $s1
        sw      $t3, ($t4)              # store in temp array

        increment $t1                   # increment col
        blt     $t1, $s2, transfer_col_loop

        increment $t0                   # increment row
        blt     $t0, $s2, transfer_row_loop
# Copy back to original array
        mul     $t7, $s2, $s2           # total elements
        li      $t0, 0                  # counter
copy_loop:
        sll     $t1, $t0, 2
        add     $t2, $s1, $t1           # temp address
        add     $t3, $s0, $t1           # grid address
        lw      $t4, ($t2)              # load from temp
        sw      $t4, ($t3)              # store in grid
        increment $t0
        blt     $t0, $t7, copy_loop

        jr      $ra
```

The `rotate_right` function performs a 90-degree clockwise rotation of the grid. It uses a temporary grid (`TEMP_GRID`) to hold the rotated values during the process and then copies the result back to the original grid (`GRID_BASE`).

A quick rundown of the function would be the following:

Initialize Variables:

- $s0: Base address of the original grid (`GRID_BASE`).
- $s1: Base address of the temporary grid (`TEMP_GRID`).
- $s2: The size of the grid (N), representing the number of rows or columns.

Transfer Elements to the Temporary Grid (`transfer_row_loop and transfer_col_loop`):

- Iterates through the rows ($t0) and columns ($t1) of the original grid.
- For each element, calculates:
  - **Source index**: row * N + col, the current position in the grid.
  - **Destination index**: col * N + (N - 1 - row), the rotated position in the temporary grid.
- Transfers the value from the source index to the destination index in `TEMP_GRID`.

Copy Back to the Original Grid (`copy_loop`):

- Iterates over all elements in the temporary grid and copies them back to the original grid.

reverse:

```
reverse:
        lw      $s0, GRID_BASE          # Grid base address
        lw      $s1, TEMP_GRID          # Temp array base address
        lw      $s2, N                  # N (size)
        li      $t0, 0                  # row counter
rev_row:
        li      $t1, 0                  # left column
        sub     $t2, $s2, 1             # right column
rev_col:
        bge     $t1, $t2, rev_next_row  # Check if left >= right

        # Calculate left index
        mul     $t3, $t0, $s2
        add     $t3, $t3, $t1
        sll     $t3, $t3, 2
        add     $t3, $t3, $s0

        # Calculate right index
        mul     $t4, $t0, $s2
        add     $t4, $t4, $t2
        sll     $t4, $t4, 2
        add     $t4, $t4, $s0

        # Swap values
        lw      $t5, ($t3)              # left value
        lw      $t6, ($t4)              # right value
        sw      $t6, ($t3)              # store right in left
        sw      $t5, ($t4)              # store left in right

        addi    $t1, $t1, 1             # increment left
        sub     $t2, $t2, 1             # decrement right
        j       rev_col
rev_next_row:
        addi    $t0, $t0, 1             # next row
        blt     $t0, $s2, rev_row

        jr      $ra
```

The `reverse` function flips the elements of each row in the grid horizontally. For example, the first element of a row becomes the last, the second becomes the second-to-last, and so on.

Setup**:**
- $s0: Base address of the original grid (GRID_BASE).
- $s1: Base address of a temporary grid (TEMP_GRID).
- $s2: The size of the grid (N), representing the number of rows and columns.
- $t0: Row counter, initialized to 0.

Iterate Through Rows `(rev_row)`**:**
- The outer loop processes each row of the grid by iterating the row counter $t0.

Reverse Each Row `(rev_col)`**:**
- Initialize t1 (left pointer) to 0: Points to the leftmost column.
- Initialize t2 (right pointer) to N-1: Points to the rightmost column.
- Swap elements pointed to by t1 and t2 until t1 >= t2:
    - Calculate the **memory addresses** of the elements using:
        - row * N + col for the column index.
        - Word offset (*4 due to 32-bit values).
    - **Swap the values** by loading them into temporary registers (t5 and t6) and storing them in their swapped positions.
- Increment t1 (move inward from the left) and decrement t2 (move inward from the right).

Move to the Next Row `(rev_next_row)`**:**
- After completing the reversal of a row, increment $t0 to move to the next row.
- Continue the process until all rows are processed.

Return:
- Once all rows are reversed, return to the calling function with jr $ra.

After doing the appropriate movements depending on the user input, we then call the `add_random_tile` function to generate a 2-tile in the grid.. This function is also responsible for updating the number of moves the player has done by calling the function `add_random_tile`.

Add_random_tile:

The `add_random_tile` function is responsible for adding a new tile to the game grid after each valid move, provided that the grid is not full or a RNG flag is not disabled.

```
add_random_tile:
        jal     check_previous          # Checks if grid changed
        jal     increment_move_tally

        lw      $s0, RNG_FLAG
        beq     $s0, 3, main_loop       # If $s0 is 3 - RNG disabled no need to add_random_tile
randomize_N:
        li      $t0, 0                  # grid position counter
        lw      $s0, GRID_BASE          # grid base address
        lw      $s1, NxN                # size
        randomize $s1
random_tile_loop:
        bne     $a0, $t0, next_index

        sll     $t1, $t0, 2             # Current index * 4
        add     $t1, $t1, $s0           # Current address
        lw      $t2, ($t1)              # Current value
        bnez    $t2, randomize_N

        li      $t3, 2
        sw      $t3, ($t1)
        j       main_loop
next_index:
        increment $t0
        j       random_tile_loop
# ---------------------- FUNCTIONS ----------------------
increment_move_tally:
        lw      $s0, CURRENT_MOVES
        increment $s0
        sw      $s0, CURRENT_MOVES
        jr      $ra
# ---------------------- END ----------------------
```

Check if Grid Changed (`check_previous`):
- The function first calls check_previous, which verifies whether the grid state has changed after the last move. If no changes occurred, adding a tile may not be necessary.

Update Move Count (`increment_move_tally`):
- The increment_move_tally function is called to increase the count of valid moves.

Check RNG Flag:
- $s0 is loaded with RNG_FLAG to determine whether random tile addition is disabled:
  - If $s0 == 3, no tile is added, and the function jumps to main_loop.

Randomize Tile Placement:
- Randomize an Index **`(randomize $s1)`**:
  - A random index ($a0) within the grid's range (0 to NxN-1) is generated.
- Iterate Through Grid `(random_tile_loop)`:
  - The loop checks grid positions starting from the beginning ($t0).
  - If the current position ($t0) matches the random index ($a0), proceed to check if it's empty:
    - Load the grid value at the position.
    - If non-zero (bnez $t2), the space is occupied, and a new random index is generated (randomize_N).
  - If the position is empty (t2 == 0), place a tile with value 2 (li $t3, 2 and sw $t3).

Return to Main Loop:
- Once a tile is successfully added, the function jumps to main_loop.

`increment_move_tally`

- Increments the global counter CURRENT_MOVES to track the number of moves made in the game:
  - Load the current move count.
  - Increment it.
  - Store it back.

Randomization

- The `randomize` is the macro that generates a random number to determine the index of the next tile placement.

Task 3: Score Tracking

```
# ---------------------- FUNCTIONS ----------------------
calculate_score:
        li      $t0, 0                  # grid position counter
        lw      $s0, NxN                # value of NxN
        lw      $s1, GRID_BASE          # grid base address
        li      $s2, 0                  # score total
calculate_loop:
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t2, $t1, $s1           # Get address of current position
        lw      $t3, ($t2)              # Load current element
        add     $s2, $s2, $t3

        increment $t0
        blt     $t0, $s0, calculate_loop

        sw      $s2, CURRENT_SCORE
        jr      $ra
```

The `calculate_score` function computes the total score of the grid by summing all non-zero tile values.

Below is a quick run through of the function:

Initialize Variables:
- $t0: Counter for iterating through grid positions (initialized to 0).
- $s0: Holds the total number of grid tiles (NxN).
- $s1: Base address of the grid (GRID_BASE).
- $s2: Accumulator for the total score (initialized to 0).

Loop Through Grid:
- Calculate Address:
  - For each grid position, calculate the memory address of the current tile using: `Address = GRID_BASE + (Position * 4)`
  - The shift left logical (sll) operation is used to multiply the index by 4, as each tile is a word (4 bytes).
- Load Tile Value:
  - Load the current tile value into $t3 using the calculated address.
- Add to Score:
  - Accumulate the tile's value into $s2.

Increment Counter:
- Increment $t0 to process the next grid position.
- Continue looping until $t0 < NxN (process all tiles).

Store the Score:
- Once all tiles are processed, the total score in $s2 is stored in the memory location for `CURRENT_SCORE`.

Return::
- The function ends with jr $ra, returning control to the calling function.

Task 4: Single-move Undo

```
undo:
        jal     check_previous

        lw      $s0, CURRENT_MOVES
        beqz    $s0, invalid_move
        addi    $s0, $s0, -1
        sw      $s0, CURRENT_MOVES

        li      $t0, 0                  # grid position counter
        lw      $s0, NxN                # value of NxN
        lw      $s1, PREV_GRID          # previous grid base address
        lw      $s2, GRID_BASE          # current grid base address
undo_loop:
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t2, $t1, $s1           # Get address of current position
        lw      $t3, ($t2)              # Load current element
        add     $t4, $t1, $s2
        sw      $t3, ($t4)

        increment $t0
        blt     $t0, $s0, undo_loop

        j       main_loop
```

The `undo` function restores the grid to its previous state, allowing the player to revert their last move. This is achieved by copying the data from a stored "previous grid" `(PREV_GRID)` back into the current grid `(GRID_BASE)`.

First we check if the `undo` move is valid.

- $s0 holds the count of moves (`CURRENT_MOVES`).
- If $s0 is zero, it means no moves have been made, so the undo action is invalid, and control is directed to the invalid_move label.

Decrement Move Counter:

- If a move exists (`$s0` is not zero), the function decrements the `CURRENT_MOVES` by 1.
- This is done by subtracting 1 from `$s0` and storing the updated value back in the `CURRENT_MOVES` memory location.

Initialize Variables for `Undo`:

- The loop begins by initializing a counter `$t0` to 0, which will iterate over all grid positions.
- `$s0` is loaded with the grid size (NxN), which tells us how many positions we need to loop over.
- `$s1` holds the base address of the previous grid (`PREV_GRID`), which contains the state to restore.
- `$s2` holds the base address of the current grid (`GRID_BASE`), where the previous state will be copied back.

Restore Grid State:

- The main undo loop (`undo_loop`) starts by calculating the address of the current position in both grids (`PREV_GRID` and `GRID_BASE`).
  For the previous grid, the address is calculated using:
  ```
  add $t2, $t1, $s1   # Address in PREV_GRID
  ```
  For the current grid, the address is calculated using:
  ```
  add $t4, $t1, $s2   # Address in GRID_BASE
  ```
- The value from `PREV_GRID` is loaded into register `$t3`, and then it is stored back into the corresponding position in `GRID_BASE`.

Iterate Over All Grid Positions:

- The loop counter `$t0` is incremented at the end of each iteration, and the loop continues as long as `$t0` is less than `NxN` (the total number of grid positions).

Return to Main Loop:

- Once all positions have been restored, the function jumps to the `main_loop` label to continue the main game logic

Task 5: Free Feature - x2 multiplier cheat code

We implemented a bonus feature: a cheat code. Similar to old arcade games, there are always tricks and hacks to make the game more enjoyable to play. We decided to create a secret cheat code that when the player inputs the string "**CS21**" , every piece on the board will be multiplied by 2. This is so that it's easier to achieve 2048 in the game. Note that as long as the first 4 characters in the user input is **CS21** our code will know that the user is prompting the secret cheat code, for example the following string inputs are valid:

- "**CS21 is the best subject ever"**
- **"CS21 please let us pass"**
- **"CS21 CS21 CS21"**
- **"CS21 meowowowowowo"**

cheat:

```
cheat:
        ##### preamble #####
        addi    $sp, $sp, -4
        sw      $t0, 0($sp)
        ##### preamble #####
        jal     store_current

        li      $t0, 0                  # grid position counter
        lw      $s0, NxN                # value of NxN
        lw      $s1, GRID_BASE          # current grid base address
cheat_loop:
        sll     $t1, $t0, 2             # Multiply index by 4 for word offset
        add     $t2, $t1, $s1           # Get address of current position
        lw      $t3, ($t2)              # Load current element
        sll     $t3, $t3, 1
        sw      $t3, ($t2)

        increment $t0
        blt     $t0, $s0, cheat_loop
        ##### end #####
        lw      $t0, 0($sp)
        addi    $sp, $sp, 4
        ##### end #####
        j       main_loop
```

In summary, the cheat function doubles the values in the grid, saves the state before modifying the grid, and ensures that the function can safely restore the state of registers afterward.

The undo does the following:

First we have to save the state using preamble:

- Initial Preamble: The function starts by saving the current state of the $t0 register onto the stack. This ensures that the value of $t0 is preserved and can be restored later.
- The function then calls `store_current`, which stores the current grid state for potential future undo operations.

Doubling Grid Values:

- Grid Iteration: The function sets up a loop to iterate over each position in the grid. The counter ($t0) is initialized, and the grid size (NxN) is loaded into $s0.
- The loop uses this counter to calculate the address of each grid element and doubles its value. The doubling is done by performing a left shift on the current value of each element.
- After processing each grid element, the counter is incremented, and the loop continues until all grid positions are processed.

Restoring State and Returning:

- Once the loop is completed, the function restores the original value of $t0 from the stack to maintain the calling function's state.
- The function then jumps back to the `main_loop`, which is where the main game logic continues, effectively completing the "cheat" operation.

Thank you for reading our documentation!