MAXIMO, Calvin James T. May 21, 2023

2021-12444 CS 21

Machine Problem 1 Documentation

Tetrisito in MIPS

I. .data segment

```
.data
        .eqv    gridOne, $s0
        .eqv    gridTwo, $s1

        .eqv    grid, $s0
        .eqv    gridCopy, $s1
        .eqv    piece, $s2

        .eqv    currGrid, $s0
        .eqv    pieces, $s1
        .eqv    i, $s2

        .eqv    nextGrid, $s0

        .eqv    hashtag, 0x23
        .eqv    X, 0x58
        .eqv    dot, 0x2e

        yes:    .asciiz "YES"
        no:     .asciiz "NO"
```

In the .data segment of my source code, I renamed some of the s registers for easier tracing while writing the code.

For the is_equal_grids function, $s0 was renamed to gridOne and $s1 was renamed to gridTwo.

For the deepcopy function, $s0 was renamed to grid, and $s1 was renamed to gridCopy. The register grid ($s0) was also used for the freeze_blocks function. In addition to grid and gridCopy, the register $s2 was renamed to piece and used in the drop_piece_in_grid function.

For the backtrack function, $s0 was renamed to currGrid, $s1 was renamed to pieces, and $s2 was renamed to i.

For the line_clearing function, $s0 was renamed to nextGrid.

Also, hashtag is equivalent to 0x23 (ASCII value of # in hexadecimal), X is equivalent to 0x58 (ASCII value of X in hexadecimal), and dot is equivalent to 0x2e (ASCII value of . in hexadecimal).

Also contained in the .data segment is the string “YES” which is stored at the address 0x10010000, and the string “NO” which is stored at the address 0x10010004.

II. Macros

```
.macro allocate_heap(%num_of_bytes)
        li      $a0, %num_of_bytes      # Allocate # of bytes
        do_syscall(9)           # Allocate heap memory
.end_macro

.macro do_syscall(%callnumber)
        li      $v0, %callnumber
        syscall
.end_macro

.macro string_input(%num_of_bytes)
        allocate_heap(%num_of_bytes)    # Allocate %num_of_bytes bytes to heap memory
        move    $a0, $v0                # Move address of allocated heap memory to $a0
        addi    $a1, $0, %num_of_bytes  # Allows the user to input #-1 of characters
        do_syscall(8)                   # Syscall 8 for string input
.end_macro

.macro exit_program
        do_syscall(10)
.end_macro

.macro initialize_empty_rows
        allocate_heap(32)
        li      $t0, 0x2e2e2e2e # Equal to ....
        sw      $t0, 0($v0)
        sw      $t0, 8($v0)
        sw      $t0, 16($v0)
        sw      $t0, 24($v0)

        li      $t0, 0x00002e2e # Equal to \0\0..
        sw      $t0, 4($v0)
        sw      $t0, 12($v0)
        sw      $t0, 20($v0)
        sw      $t0, 28($v0)

        li      $t0, 0  # Clear temporary register
.end_macro
```

Macros were also used to make the process of writing the code easier, and also make it easier to read.

The first macro is allocate_heap, which takes in a parameter %num_of_bytes, or as it name implies, the number of bytes that will be allocated to the heap. After doing a syscall with call number 9, the address that points to the start of the allocated bytes is stored in $v0.

The second macro is do_syscall, which takes in a parameter %call_number. It loads it to $v0, then executes a syscall which depends on %call_number. It's basically just a shortcut instead of having to list li $v0, %call_number and syscall in the code.

The third macro is string_input, which takes in a parameter %num_of_bytes, which denotes the length of the string that will be input by the user. This macro is mainly used in the get_input_pieces function, which will be discussed later in this documentation. The first line of this macro is the macro allocate_heap, which takes in %num_of_bytes as its parameter, or %num_of_bytes bytes will be allocated in the heap. The address that points to the start of the allocated bytes is stored in $v0. After that, the value of $v0 is also stored in $a0. The value of %num_of_bytes is stored in $a1. Then, a syscall of call number 8 will be executed (for string input). The input string will then be stored in the address at $v0. $v0 will also be stored in

$gp (global pointer). By offsetting, we can store both the addresses of the start and final grids in $gp.

The fourth macro is exit_program, which just executes a syscall of call number 10 for terminating the program.

The fifth macro is initialize_empty_rows, which is used for initializing the first 4 rows of the start and final grids. Only dots are stored in the first 4 rows of these grids. First, this macro allocates 32 bytes to the heap, where each row of the grid uses up 8 bytes (the last two bytes will be empty or one byte has \n). The macro then stores "...." and "\0\0.." in each of the 8 bytes in the heap, effectively representing a row. Note that "\0" is stored in the last two bytes, since allocating 8 bytes for each row of the grid makes it easier to work around in the code. The address that points to the start of the grid is stored in $v0.

FUNCTIONS

- All functions have their necessary preambles and postambles (s registers and t registers, if needed, used in the function are saved in the stack then restored after the function ends).

III. input_6x6

```
101 input_6x6:      # Input last 6 rows of start and final grid
102         ####preamble####
103         addi    $sp, $sp, -32
104         sw      $s0, 28($sp)
105         sw      $s1, 24($sp)
106         sw      $s2, 20($sp)
107         sw      $ra, 16($sp)
108         ####preamble####
109         addi    $s0, $0, 0 # i = 0
110         input_6x6_loop:
111                 beq     $s0, 6, exit_input_6x6  # branch when i = 6
112                 allocate_heap(8)        # allocate 8 bytes
113                 addi    $s1, $v0, 0     # $s1 = address at $v0
114                 addi    $a0, $v0, 0     # $a0 = address at $v0
115                 li      $a1, 8          # Input 8-1 = 7 characters long
116                 li      $v0, 8          # Syscall 8 for string input
117                 syscall
118                 addi    $s2, $0, 0      # j = 0
119                 frozen:                 # mark frozen blocks as 'X'
120                         beq     $s2, 6, exit_frozen     # exit when j = 6
121                         li      $t0, hashtag            # Loads ASCII value of # into $t0
122                         lb      $t1, 0($s1)             # Get byte at addr[$s1]
123                         beq     $t0, $t1, append_x      # Branch if row[j] == '#'
124                         addi    $s1, $s1, 1             # increment address by 1
125                         addi    $s2, $s2, 1             # increment j
126                         j frozen
127                         append_x:
128                         li      $t0, X          # Loads ASCII value of X into $t0
129                         sb      $t0, 0($s1)     # Stores ASCII value of X in addr[$s1]
130                         addi    $s1, $s1, 1     # increment address by 1
131                         addi    $s2, $s2, 1     # increment j
132                         j       frozen
133                 exit_frozen:
134                         addi    $s0, $s0, 1     # increment i
135                         j       input_6x6_loop
136         exit_input_6x6:
137         ####end####
138         lw      $s0, 28($sp)
139         lw      $s1, 24($sp)
140         lw      $s2, 20($sp)
141         lw      $ra, 16($sp)
142         addi    $sp, $sp, 32
143         ####end####
144         jr      $ra
```

The input_6x6 function allows the user to input the last 6 rows of the start/final grids. In line 109, $s0 is initialized to 0, or i = 0. Then, a for loop is started from lines 110-135, which loops 6 times. Lines 112-117 are for allocating 8 bytes to the heap (represents a row). The address pointing to the start of these allocated bytes is stored in $v0. This is also stored in $s1 (to be used in the function) and $a0 (to be used for storing the user's input). After that, the user is asked to input the row.

After getting the user's input, $s2 is initialized to 0, or j = 0 in line 118. This will be used as the counter in freezing the blocks or converting the #s from the user's input to Xs. This is a for loop that is inside the for loop in getting the user's input. The implementation of the input_6x6 function is coded in a way that after every time the user inputs a row of the grid, the whole row will be frozen before proceeding to ask the user the next input/row. In lines 119-132, the ASCII value of # is stored in $t0. Then, the byte at the current value of $s1 will be loaded to $t1. If $t1 == '#', then it will be frozen or changed to an 'X', then proceeds to the other elements in the row. Otherwise, it proceeds to the other elements in the row (increment $s1) and keeps looping until $s2/j is equal to 6. Once $s2/j is equal to 6, it exits out of the frozen loop (lines 119-132) and proceeds to lines 133-135, which increments $s0/i and jumps back to line 110. The user is then asked to input again, and it repeats until the user has inputted all 6 rows.

IV. get_input_pieces

```
146 get_input_pieces: # Gets the input pieces from the player
147         ####preamble####
148         addi    $sp, $sp, -32
149         sw      $ra, 28($sp)
150         sw      $s0, 24($sp)
151         sw      $s1, 20($sp)
152         sw      $s2, 16($sp)
153         sw      $s3, 12($sp)
154         ####preamble####
155         addi    $s0, $0, 0      # counter of for _ in range(numPieces)
156         lw      $s1, 8($gp)     # $s1 = numPieces
157         get_input_pieces_outer:
158                 beq     $s0, $s1, exit_get_input_pieces # if $s0 == numPieces, exit the loop
159                 addi    $s2, $0, 0      # counter for for _ in range(4)
160                 get_input_pieces_inner:
161                         beq     $s2, 4, exit_get_input_pieces_inner     # if $s2 == 4, exit inner loop
162                         string_input(8) # Input for each row of the piece
163                         addi    $s2, $s2, 1     # Increment $s2/inner counter
164                         j       get_input_pieces_inner
165                 exit_get_input_pieces_inner:
166                 addi    $s0, $s0, 1     # Increment $t0/outer counter
167                 j       get_input_pieces_outer
168         exit_get_input_pieces:
169         move    $t0, $a0        # Get address of last row of last input piece
170         addi    $t1, $0, 32     # Load 32 to $t0 for multiplying to numPieces
171         mult    $s1, $t1        # numPieces * 32
172         mflo    $t1             # Computes the address of first row of first input piece in the heap
173         sub     $t0, $t0, $t1   #
174         addi    $t0, $t0, 8     # Address of first row of first input piece in the heap
175         move    $v0, $t0        # Store address in $v0
176         ####end####
177         lw      $ra, 28($sp)
178         lw      $s0, 24($sp)
179         lw      $s1, 20($sp)
180         lw      $s2, 16($sp)
181         lw      $s3, 12($sp)
182         addi    $sp, $sp, 32
183         ####end####
184         jr      $ra
```

The get_input_pieces_function, when called, gets the input pieces from the user. Before this function is called in main, the user is asked to input the number of pieces , then the value will be stored in $gp. In line 155, the counter used for the loop is initialiazed to 0 and stored in $s0. In line 156, numPieces was loaded from $gp and stored to $s1. It proceeds to a for loop which exits once $s0 = $s1, or when the user has inputted all the pieces. Inside this for loop is another for loop that is used for inputting each piece. The counter for this loop is initialized to 0 in line 159 and is stored in $s2. This inner for loop then calls the macro string_input, with 8 as its parameter (allocate 8 bytes in heap for each row of the piece). Once the user has input a row of the piece, it repeats the inner loop and will keep on asking input from the user until the user inputs the last row of the piece. After that, it exits the inner for loop then asks the user to input another piece if $s0 is not yet equal to $s1. Once the user has finished inputting all the pieces, it exits out of the outer loop.

Note that the macro string_input is called in the inner loop (input each row of piece). Before it exits out of that loop, $v0 contains the address that points to the start of the last row of the input piece. This is then stored in $a0 in the macro string_input. Once the user has finished inputting all pieces, we can take advantage of this in computing for the address that points to the start of the first input piece. This is because in heap, all of the pieces are next to each other. Now, we store the value of $a0 to $t0 (line 169). We temporarily load 32 to the register $t1 in line 170 (32 because there are a total of 32 bytes allocated for each piece). We then multiply it to $s1/numPieces and load the product from mflo to $t1. We then subtract $t1 from $t0. $t0 is not yet the address that points to the first row of the first piece, so 8 is added to $t0. Then, $t0 will be copied to $v0. The value of $v0 then is the address that points to the start of the 'array' of the input pieces. This will then be stored to $gp in the main function after calling get_input_pieces.

V. convert_piece_to_pairs

```
convert_piece_to_pairs: # Converts each # of piece to a tuple of coordinates (row, col)
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      $s0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        sw      $s4, 8($sp)
        ####preamble####
        addi    $t0, $0, 8      # 8 bytes per set of coordinates of a piece
        lw      $t1, 8($gp)     # $t1 = numPieces
        mult    $t1, $t0        # Gets the total number of bytes to allocate in heap memory for storing the c
        mflo    $a0             # Store the result in $a0
        do_syscall(9)           # Allocate bytes in heap memory

        move    $s0, $v0        # Store the address of converted_pieces
        move    $t0, $s0        # Store a temporary copy of addr[converted_pieces] in $t0 to be used in this
        lw      $t1, 12($gp)    # Store a temporary copy of addr[pieceAscii] in $t4 to be used in this functi

        addi    $s1, $0, 0      # counter for outer loop
        lw      $s2, 8($gp)     # $s2 = numPieces
        convert_piece_to_pairs_outer:
                beq     $s1, $s2, exit_convert_piece_to_pairs   # If $s1 = numPieces, exit
                addi    $s3, $0, 0              # row = 0
                convert_piece_to_pairs_row:
                        beq     $s3, 4, exit_convert_piece_to_pairs_row
                        addi    $s4, $0, 0              # col = 0
                        convert_piece_to_pairs_col:
                                beq     $s4, 4, exit_convert_piece_to_pairs_col
                                lb      $t2, 0($t1)     # Get byte at address at $t1
                                beq     $t2, hashtag, store_pair # Store (row,col) if $t5 == '#'
                                addi    $t1, $t1, 1     # Increment addr[pieceAscii] by 1
                                addi    $s4, $s4, 1     # Increment col
                                j       convert_piece_to_pairs_col
                        store_pair:
                                sb      $s3, 0($t0)     # Store row coordinate in converted_pieces
                                addi    $t0, $t0, 1     # Increment addr[converted_pieces] by 1
                                sb      $s4, 0($t0)     # Store col coordinate in converted_pieces
                                addi    $t0, $t0, 1     # Increment addr[converted_pieces] by 1
                                addi    $t1, $t1, 1     # Increment addr[pieceAscii] by 1
                                addi    $s4, $s4, 1     # Increment col
                                j       convert_piece_to_pairs_col
                        exit_convert_piece_to_pairs_col:
                                addi    $t1, $t1, 4     # Increment addr[pieceAscii] by 4 (move to next row o
                                addi    $s3, $s3, 1     # Increment row
                                j       convert_piece_to_pairs_row
                exit_convert_piece_to_pairs_row:
                        addi    $s1, $s1, 1     # Increment outer loop counter
                        j       convert_piece_to_pairs_outer
        exit_convert_piece_to_pairs:
        move    $v0, $s0        # $v0 = addr[converted_pieces]
        ####end####
        lw      $ra, 28($sp)
        lw      $s0, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        lw      $s4, 8($sp)
        addi    $sp, $sp, 32
        ####end####
        jr      $ra
```

This function is called after the code finishes getting the input pieces from the user (get_input_pieces function). This code stores the row and column coordinates of each block of each piece. The row coordinate is equivalent to the y value, while the column coordinate is equivalent to the x value. The first thing this function does is temporarily assigning 8 to register $t0, then getting the number of pieces from $gp and loads it to register $t1. Note that 8 bytes will be allocated for each piece's coordinates. We multiply $t0 to $t1 then get the result from mflo and store it to $a0, then the macro allocate_heap is called. The address that points to the start of the allocated bytes for the coordinates is stored in $v0. We copy this value to $s0, and to $t0 as well to be used in the function (lines 202 and 203). We also load the address of the input pieces from $gp and store it to $t1 (line 204). After that, we initialize a counter value to 0, stored in $s1, for the outermost loop which denotes which piece we're calculating the coordinates of. We also load again the number of pieces from $gp to $s2. After the outermost loop, another counter is initialized to 0 and assigned to $s3. This is a loop that is used to iterate through each row of the piece, which exits out of the loop when $s3 is 4. After this loop, there is another loop that is used to iterate through each column of the piece. A counter is initialized to 0 and assigned to $s4, which exits out of the loop when $s4 is 4. Inside this innermost for loop is where each of the bytes of the piece are checked whether they are a "#" or ".". If it is a "#", then it stores its row and column coordinates in the heap. The row coordinate is stored in the first byte, then $t0 is incremented by 1, then the column coordinate is stored in the second byte. Once all columns have been checked for #s, $t1 is incremented by 4 to move to the next row of the piece. This repeats until the program has iterated through the 4x4 grid of the piece. Once done, it exits out of the inner loops and loops back to the outer loop to proceed to calculate the coordinates of the next piece. The process repeats until all the pieces have been iterated through. The address that points to the start of the allocated bytes for the coordinates is stored to $v0, which will then be stored to $gp in the main function after calling convert_piece_to_pairs.

VI. is_equal_grids

```
is_equal_grids: # Compare if two grids (gridOne and gridTwo) are equal, returns result
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      gridOne, 24($sp)
        sw      gridTwo, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        sw      $s4, 8($sp)
        ####preamble####
        move    gridOne, $a0    # Move address of $a0 (1st input grid) to gridOne
        move    gridTwo, $a1    # Move address of $a1 (2nd input grid) to gridTwo
        move    $t0, gridOne    # Move gridOne to $t0 for manipulating in function
        move    $t1, gridTwo    # Move gridTwo to $t1 for manipulating in function

        addi    $s2, $0, 1      # result = True
        addi    $s3, $0, 0      # i = 0

        is_equal_grids_i:
                beq     $s3, 10, exit_is_equal_grids    # Exit when i = 10
                addi    $s4, $0, 0      # j = 0
                is_equal_grids_j:
                        beq     $s4, 6, exit_is_equal_grids_j   # Exit when j = 6
                        lb      $t2, 0($t0)     # Load character at addr[gridOne]
                        addi    $t0, $t0, 1     # Increment addr[gridOne] by 1
                        lb      $t3, 0($t1)     # Load character at addr[gridTwo]
                        addi    $t1, $t1, 1     # Increment addr[gridTwo] by 1
                        bne     $t2, $t3, is_equal_grids_false  # return result = False
                        addi    $s2, $0, 1      # result = result AND (gridOne[i][j] == gridTwo[i][j])
                        addi    $s4, $s4, 1     # Increment j
                        j       is_equal_grids_j
                exit_is_equal_grids_j:
                addi    $t0, $t0, 2     # Increment addr[gridOne] by 2 to move to next row of gridOne
                addi    $t1, $t1, 2     # Increment addr[gridOne] by 2 to move to next row of gridOne
                addi    $s3, $s3, 1     # Increment i
                j       is_equal_grids_i
        is_equal_grids_false:
                addi    $s2, $0, 0      # result = False
        exit_is_equal_grids:
        move    $v0, $s2        # Store result in $v0
        ####end####
        lw      $ra, 28($sp)
        lw      gridOne, 24($sp)
        lw      gridTwo, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        lw      $s4, 8($sp)
        addi    $sp, $sp, 32
        ####end###
        jr      $ra
```

This function checks whether two grids, gridOne and gridTwo, are equal. The address of the first grid is in $a0 then it is stored to gridOne, and the address of the second grid is in $a1 then is stored to gridTwo. gridOne is copied to $t0, and gridTwo is copied to $t1. Two for loops are then used for checking if they are equal, the first loop iterates through each row, and the second inner loop iterates through each column. A boolean variable called result is initialized to TRUE/1 and stored to $s2. The two loops keep running as long as $s2 remains TRUE. Once the code has detected that gridOne[i][j] and gridTwo[i][j] are not equal, it exits out of the loops and sets $s2 to FALSE/0. Otherwise, it keeps on looping until it has checked every element of the grid. The result/value of $s2 is then stored to $v0.

VII. get_max_x_of_piece

```
get_max_x_of_piece:     # Get x value of rightmost block of piece
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      $s0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        ####preamble####
        move    $s0, $a0                # $s0 = addr[piece]
        move    $t0, $s0                # $t0 = $s0 for use in function
        addi    $s1, $0, -1             # max_x = -1
        addi    $s2, $0, 0              # block = 0
        get_max_x_of_piece_loop:
                beq     $s2, 4, end_get_max_x_of_piece  # Exit once finished iterating through all blocks of
                lb      $t1, 1($t0)             # Load to $t3 the x-coordinate of the block
                bgt     $s1, $t1, get_max_x_continue    # Compare max_x to $t3, branch when max_x is greater
                move    $s1, $t1                # max_x = block[1]
                get_max_x_continue:
                addi    $t0, $t0, 2     # Increment addr[block] by 2 to move to next coordinate
                addi    $s2, $s2, 1     # Increment counter
                j       get_max_x_of_piece_loop
        end_get_max_x_of_piece:
        move    $v0, $s1        # Return max_x
        ####end####
        lw      $ra, 28($sp)
        lw      $s0, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        addi    $sp, $sp, 32
        ####end####
        jr      $ra
```

This function takes in the address of the coordinates of a piece in $a0, then finds the maximum x coordinate among its blocks. A variable named max_x is initialized to -1 and is stored in $s1. A counter variable is then initialized to 0 and is stored in $s2. The for loop is used to iterate through each set of coordinates, and compares its column/x coordinate to max_x. If max_x is less than the column/x coordinate of the block, then the new value of max_x is the column/x coordinate. Otherwise, it remains the same and continues iterating through the rest of the coordinates until it finishes comparing their column/x coordinates. The loop exits once $s2 is equal to 4 (since there are 4 sets of coordinates for each piece). The value of max_x /$s1 is then stored to $v0. This function is also called inside the backtrack function, which will be expounded on later.

VIII. deepcopy

```
deepcopy:
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      grid, 24($sp)
        sw      gridCopy, 20($sp)
        sw      $s2, 16($sp)
        ####preamble####
        move    grid, $a0       # $s0 = addr[grid]
        move    $t0, grid       # $t0 = $s0 for manipulating in function
        addi    $a0, $0, 80     # Allocate 80 bytes for copying grid
        do_syscall(9)           # Syscall 9 for allocating bytes to heap

        # $v0 contains the address pointing to the start of gridCopy
        move    gridCopy, $v0   # $s1 = addr[gridCopy]
        move    $t1, gridCopy   # $t1 = $s1 for manipulating in function
        addi    $s2, $0, 0      # i = 0
        deepcopy_loop:
                beq     $s2, 20, exit_deepcopy  # Exit when i = 20
                lw      $t2, 0($t0)             # Get word from original grid and store in $t3
                sw      $t2, 0($t1)             # Store the word in gridCopy
                addi    $t0, $t0, 4     # Increment addr[grid] by 4 for word alignment
                addi    $t1, $t1, 4     # Increment addr[gridCopy] by 4 for word alignment
                addi    $s2, $s2, 1     # Increment loop counter by 1
                j       deepcopy_loop
        exit_deepcopy:
        move    $v0, gridCopy           # $v0 = address pointing to the start of gridCopy
        ####end####
        lw      $ra, 28($sp)
        lw      grid, 24($sp)
        lw      gridCopy, 20($sp)
        lw      $s2, 16($sp)
        addi    $sp, $sp, 32
        ####end####
        jr      $ra
```

This function is called to store a copy of an input grid, stored in $a0, in memory. We copy the address of the input grid to $t0. The function then allocates 80 bytes in the heap to store the copy of the input grid. After that, $v0 contains the address pointing to the start of the allocated bytes. We then move it to $t1, which now contains the address of gridCopy. A loop then runs 20 times, which copies each 'word'/row of the input grid to gridCopy. Both $t0 and $t1 are incremented by 4 every iteration of the loop for word alignment. The loop runs 20 times since each row of the grid uses up 2 words in memory, and there are 10 rows in a grid, which means it uses up 20 words. Once it has finished copying the input grid to gridCopy, the address of gridCopy is stored in $v0.

IX. freeze_blocks

```
freeze_blocks:
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      grid, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        ####preamble####
        move    grid, $a0               # grid = $a0
        move    $t0, grid               # $t0 = addr[grid]
        addi    $s1, $0, 0              # i = 0
        freeze_blocks_outer:
                beq     $s1, 10, exit_freeze_blocks     # Exit when i = 10
                addi    $s2, $0, 0                      # j = 0
                freeze_blocks_inner:
                        beq     $s2, 6, exit_freeze_blocks_inner
                        sll     $t1, $s1, 3     # 8i
                        add     $t1, $t1, $s2   # 8i + j
                        add     $t0, $t0, $t1   # Get address of grid[i][j]
                        lb      $t2, 0($t0)     # $t2 = grid[i][j]
                        seq     $t2, $t2, hashtag       # grid[i][j] == '#'?
                        beq     $t2, 0, fb_not_hash     # Continue looping if grid[i][j] != '#'
                        addi    $t2, $0, X      # $t2 = 'X'
                        sb      $t2, 0($t0)     # grid[i][j] = 'X'
                        fb_not_hash:
                        move    $t0, grid       # Restore addr[grid] to $t0
                        addi    $s2, $s2, 1     # Increment j
                        j       freeze_blocks_inner
                exit_freeze_blocks_inner:
                addi    $s1, $s1, 1     # Increment i
                j       freeze_blocks_outer
        exit_freeze_blocks:
        move    $v0, grid               # $v0 = addr[grid]
        ####end####
        lw      $ra, 28($sp)
        lw      grid, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        addi    $sp, $sp, 32
        ####end####
        jr      $ra
```

This function takes an input grid ($a0) and freezes all blocks in the grid, meaning it converts all #s in the grid to Xs. $a0 is stored in the register grid, then the register grid is stored in $t0. A counter is initialized to 0 and stored in $s1. It is the counter used for the outer loop, or for iterating through each row of the grid. It exits when $s1 is 10. A counter is initialized to 0 and stored in $s2. It is the counter used for the inner loop, or for iterating through each column of the row. Inside the inner loop is where it checks whether the column is a #. If it is a #, then it changes it to an X and proceeds to the next columns. Otherwise, it proceeds to the next columns and finishes when it has finished iterating through each column. It then exits out of the inner loop and moves on to the next row. The process repeats until it has converted all the #s to Xs. The address of grid is then copied to $v0, and the function returns $v0.

X. drop_piece_in_grid

```
drop_piece_in_grid:
        ####preamble####
        addi    $sp, $sp, -44
        sw      $ra, 40($sp)
        sw      grid, 36($sp)
        sw      gridCopy, 32($sp)
        sw      piece, 28($sp)
        sw      $s3, 24($sp)
        sw      $s4, 20($sp)
        sw      $s5, 16($sp)
        sw      $s6, 12($sp)
        ####preamble####
        move    grid, $a0               # grid = $a0
        jal     deepcopy                # gridCopy = deepcopy(grid)
        move    gridCopy, $v0           # store address from $v0 to gridCopy
        move    $t0, gridCopy           # $t0 holds address of gridCopy for manipulating
        move    piece, $a1              # piece = $a1
        move    $t1, piece              # $t1 holds address of piece for manipulating
        move    $s3, $a2                # $s3 holds xOffset
        addi    $s4, $0, 0              # block = 0
        drop_piece_in_grid_block_loop:
                beq     $s4, 4, exit_block_loop # Exit when block = 4
                lb      $t2, 0($t1)             # $t2 = block[0]
                sll     $t2, $t2, 3             # $t2 = 8 * block[0]
                add     $t0, $t0, $t2           # $t0 = gridCopy[block[0]]
                lb      $t2, 1($t1)             # $t2 = block[1]
                add     $t2, $t2, $s3           # $t2 = block[1] + xOffset
                add     $t0, $t0, $t2           # $t0 = gridCopy[block[0]][block[1] + xOffset]
                addi    $t2, $0, hashtag        # $t2 = ASCII value of '#'
                sb      $t2, 0($t0)             # gridCopy[block[0]][block[1] + xOffset] = '#'
                addi    $t1, $t1, 2             # Move to next set of coordinates
                addi    $s4, $s4, 1             # Increment block by 1
                move    $t0, gridCopy           # $t1 = Restore value of addr[gridCopy]
                j drop_piece_in_grid_block_loop # Loop back
        exit_block_loop:
        move    $t0, gridCopy           # $t1 = Restore value of addr[gridCopy]
        addi    $s4, $0, 1              # canStillGoDown = True
        addi    $s5, $0, 0              # i = 0
        drop_piece_for_loop_while:
                beq     $s5, 10, exit_drop_piece_for_loop_while         # Exit when i = 10
                addi    $s6, $0, 0                                      # j = 0
                drop_piece_for_loop_while_inner:
                        beq     $s6, 6, exit_for_loop_inner             # Exit when j = 6
                        sll     $t2, $s5, 3     # 8i
                        add     $t2, $t2, $s6   # 8i + j
                        add     $t0, $t0, $t2   # gridCopy[i][j]
                        lb      $t2, 0($t0)     # $t2 <- gridCopy[i][j]
                        seq     $t2, $t2, hashtag       # Check if gridCopy[i][j] == '#
                        addi    $t3, $s5, 1     # $t3 = i + 1
                        seq     $t3, $t3, 10    # Check if i + 1 == 10
                        lb      $t4, 8($t0)     # $t4 <- gridCopy[i+1][j]
                        seq     $t4, $t4, X     # Check if gridCopy[i + 1][j] == 'X'
                        or      $t3, $t3, $t4   # i + 1 == 10 or gridCopy[i + 1][j] == 'X'
                        and     $t2, $t2, $t3   # gridCopy[i][j] == '#' and (i + 1 == 10 or gridCopy[i + 1][j
                        beq     $t2, 0, continue_inner  # If false, continue for loop; otherwise, canStillGoD
                        addi    $s4, $0, 0              # canStillGoDownFalse = False
                        #j      exit_drop_piece_for_loop_while
                        continue_inner:
                        move    $t0, gridCopy   # Restore gridCopy to $t0
                        addi    $s6, $s6, 1     # Increment j
                        j       drop_piece_for_loop_while_inner # Loop back
                exit_for_loop_inner:
                        addi    $s5, $s5, 1     # Increment i
                        j       drop_piece_for_loop_while
        exit_drop_piece_for_loop_while:
```

```
	exit_drop_piece_for_loop_while:
	move	$t0, gridCopy	# Restore gridCopy to $t0
	beq	$s4, $0, break_out_of_while	# if canStillGoDown == False, break
	addi	$s5, $0, 8			# i = 8
	if_canStillGoDown_loop: # for i in range(8, -1, -1)
		beq	$s5, -1, exit_ifCanStillGoDown	# Exit when i = -1
		addi	$s6, $0, 0		# j = 0
		if_canStillGoDown_loop_inner:	# for j in range(6)
			beq	$s6, 6, exit_ifCanStillGoDown_inner	# Exit when j = 6
			sll	$t2, $s5, 3	# 8i
			add	$t2, $t2, $s6	# 8i + j
			add	$t0, $t0, $t2	# address of gridCopy[i][j]
			lb	$t2, 0($t0)	# $t2 <- gridCopy[i][j]
			seq	$t2, $t2, hashtag	# if gridCopy[i][j] == '#'
			beq	$t2, $0, continue_cSGD_inner # branch if gridCopy[i][j] != '#'
			addi	$t2, $0, hashtag	# $t2 = '#'
			sb	$t2, 8($t0)	# gridCopy[i+1][j] = '#'
			addi	$t2, $0, dot	# $t2 = '.'
			sb	$t2, 0($t0)	# gridCopy[i][j] = '.'
			continue_cSGD_inner:
			move	$t0, gridCopy	# restore addr[gridCopy] to $t0
			addi	$s6, $s6, 1	# Increment j
			j	if_canStillGoDown_loop_inner
		exit_ifCanStillGoDown_inner:
		addi	$s5, $s5, -1		# Decrement i by 1
		j	if_canStillGoDown_loop
	exit_ifCanStillGoDown:
	j	exit_block_loop			# Keep looping while True
	break_out_of_while:
	move	$t0, gridCopy	# Restore gridCopy to $t0
	addi	$s4, $0, 100		# maxY = 100
	addi	$s5, $0, 0		# i = 0
	maxY_loop:
		beq	$s5, 10, exit_maxY_loop		# Exit when i = 10
		addi	$s6, $0, 0			# j = 0
		maxY_loop_inner:
			beq	$s6, 6, exit_maxY_loop_inner	# Exit when j = 6
			sll	$t2, $s5, 3	# 8i
			add	$t2, $t2, $s6	# 8i + j
			add	$t0, $t0, $t2	# Get address of gridCopy[i][j]
			lb	$t2, 0($t0)	# $t2 <- gridCopy[i][j]
			seq	$t2, $t2, hashtag	# gridCopy[i][j] == '#'
			beq	$t2, 0, continue_maxY_loop_inner
			blt	$s4, $s5, continue_maxY_loop_inner	# if maxY > i, maxY = i
			move	$s4, $s5	# maxY = i
			continue_maxY_loop_inner:
			move	$t0, gridCopy	# Restore addr[gridCopy] to $t0
			addi	$s6, $s6, 1	# Increment j
			j	maxY_loop_inner
		exit_maxY_loop_inner:
		addi	$s5, $s5, 1		# Increment i
		j	maxY_loop
	exit_maxY_loop: # $t2 = maxY
	bgt	$s4, 3, return_freeze_blocks	# if maxY <= 3, return grid, False
	move	$v0, grid			# $v0 = addr[grid]
	addi	$v1, $0, 0			# $v1 = False
	j	end_drop_piece_in_grid

	return_freeze_blocks:		# if maxY > 3, return freeze_blocks(gridCopy), true
	move	$a0, gridCopy		# $a0 = gridCopy
	jal	freeze_blocks		# $v0 = addr[freeze_blocks(gridCopy)]
	addi	$v1, $0, 1			# $v1 = True
	end_drop_piece_in_grid:
	####end####
	lw	$ra, 40($sp)
	lw	grid, 36($sp)
	lw	gridCopy, 32($sp)
	lw	piece, 28($sp)
	lw	$s3, 24($sp)
	lw	$s4, 20($sp)
	lw	$s5, 16($sp)
	lw	$s6, 12($sp)
	addi	$sp, $sp, 44
	####end####
	jr $ra
```

This function, as its name implies, drops the input piece in the grid. Before that, the function deepcopy is called first, which makes a copy of the input grid that is in $a0. After calling deepcopy, the address of the copy is stored in the register gridCopy. It is also copied to $t0, which is used to manipulate the gridCopy in the function. The drop_piece_in_grid function takes in the address of a piece in $a1, stores it in the regisrter piece, and then stores it also in $t1 to be used in the function. Lastly, the last input parameter function is $a2, which houses the xOffset value. It is stored in $s3 to be used in the function. The first for loop in the function basically places the input piece in the top 4 rows of gridCopy. After that, we restore the value of gridCopy to $t0, which will be used for the next loop.

The next loop then is a while loop that first checks if a piece can be dropped, then it drops the piece one row at a time in gridCopy. A boolean variable named canStillGoDown is initially set to TRUE/1 and stored in $s4. For checking if a piece can still go down, if gridCopy[i][j] is a #, and if the loop is at the 10^{th} row (i + 1 == 10) or if the block below gridCopy[i][j], which is gridCopy[i+1][j], is an X, then the piece cannot go down anymore. It sets the value of canStillGoDown/$s4 to FALSE/0. It exits out of this while loop. Otherwise, if the piece can still go down, then the piece moves down 1 row, and the while loop repeats until the piece can no longer go down.

The next loop after the while loop checks gridCopy for the minimum y value that contains a #. An integer variable maxY is initialized to 100 and stored in $s4. It is compared to the y coordinate of gridCopy[i][j], or we get the minimum between maxY and i. If maxY is less than i/y coordinate, then the new value of maxY is i. Otherwise, retain the same value and keep comparing until it has finished iterating through the whole grid. After checking the whole grid, we check if maxY is less than or equal to 3. If it is less than or equal to 3, then the piece dropped above the 6x6 grid, or it protrudes from the top of it. This means we return the address of the original grid, stored in the register grid, and move it to $v0, and we set the other return value $v1 to FALSE/0. Otherwise, if maxY is greater than 3, then it means the piece can be dropped, so we first move the address of gridCopy to $a0, then call the freeze_blocks function. Recall that this function converts all #s in the grid to Xs, effectively freezing them in place. Once done, freeze_ blocks function returns the frozen gridCopy in $v0, which is also the return $v0 of the drop_piece_in_grid function. We set the other return value $v1 to TRUE/1.

XI. backtrack

```
backtrack:      # Start exhaustively searching if final_grid is possible from start_grid and input pieces
        ####preamble####
        addi    $sp, $sp, -48
        sw      $ra, 44($sp)
        sw      currGrid, 40($sp)
        sw      pieces, 36($sp)
        sw      i, 32($sp)
        sw      $s3, 28($sp)
        sw      $s4, 24($sp)
        sw      $s5, 20($sp)
        sw      $s6, 16($sp)
        sw      $s7, 12($sp)
        sw      $t0, 8($sp)
        sw      $t1, 4($sp)
        ####preamble####
        move    currGrid, $a0           # Store addr[$a0] to currGrid
        move    pieces, $a1             # Store addr[$a1] to pieces
        lw      $a1, 4($gp)             # Store addr[final_grid] to $a1
        jal     is_equal_grids
        beq     $v0, 1, exit_backtrack  # return True

        move    i, $a2          # Store $a2 to i
        lw      $t0, 8($gp)     # $t0 = numPieces
        bge     $a2, $t0, backtrack_return_false        # Exit when i >= len(pieces)

        move    $s3, pieces             # Store addr[pieces] to $s3
        move    $t0, i                  # $t0 = i
        sll     $t0, $t0, 3             # $t0 = 8 * i
        add     $s3, $s3, $t0           # $s3 = pieces[i]
        move    $a0, $s3                # Store pieces[i] to $a0
        jal     get_max_x_of_piece      # $v0 = get_max_x_of_piece(pieces[i])
        move    $s4, $v0                # max_x_of_piece = get_max_x_of_piece(pieces[i])

        addi    $s5, $0, 0              # offset = 0
        addi    $s6, $0, 6              # 6
        sub     $s6, $s6, $s4           # 6 - max_x_of_piece
        addi    $s7, $0, 0              # unsuccessful = 0
        backtrack_loop: # for offset in range(6 - max_x_of_piece)
                beq     $s5, $s6, backtrack_return_false        # Exit if offset == 6 - max_x_of_piece
                move    $a0, currGrid   # $a0 = currGrid
                move    $a1, $s3        # $a1 = pieces[i]
                move    $a2, $s5        # $a2 = offset
                jal     drop_piece_in_grid      # nextGrid, success = drop_piece_in_grid(currGrid, pieces[i],
                move    $t0, $v0        # Move nextGrid to $t0
                move    $t1, $v1        # Move success to $t5
                move    $a0, $t0        # Move nextGrid to $a0
                jal     line_clearing   # line_clearing(nextGrid)
                move    $t0, $v0        # Move $v0 to $t0
                beq     $t1, 0, else_not_success
                move    $a0, $t0        # $a0 = nextGrid
                move    $a1, pieces     # $a1 = pieces
                addi    $a2, i, 1       # $a2 = i + 1
                jal     backtrack       # backtrack(nextGrid, pieces, i + 1)
                beq     $v0, 1, exit_backtrack  # Return True
                j       backtrack_loop_increment
                else_not_success:
                addi    $t2, $s6, -1    # 6 - max_x_of_piece - 1
                bne     $s7, $t2, increment_unsuccessful        # Proceed if (unsuccessful == 6 - max_x_of_pi
                move    $a0, $t0        # $a0 = nextGrid
                move    $a1, pieces     # $a1 = pieces
                addi    $a2, i, 1       # $a2 = i + 1
                jal     backtrack               # backtrack(nextGrid, pieces, i + 1)
                beq     $v0, 1, exit_backtrack  # Return True
                increment_unsuccessful:
                addi    $s7, $s7, 1     # Increment unsuccessful by 1
                backtrack_loop_increment:
                addi    $s5, $s5, 1     # Increment offset
                j       backtrack_loop
        backtrack_return_false:
                addi    $v0, $0, 0      # Return false
        exit_backtrack:
        ####end####
        lw      $ra, 44($sp)
        lw      currGrid, 40($sp)
        lw      pieces, 36($sp)
        lw      i, 32($sp)
        lw      $s3, 28($sp)
        lw      $s4, 24($sp)
        lw      $s5, 20($sp)
        lw      $s6, 16($sp)
        lw      $s7, 12($sp)
        lw      $t0, 8($sp)
        lw      $t1, 4($sp)
        addi    $sp, $sp, 48
        ####end####
        jr      $ra
```

This function is called to exhaustively check if the final gird is possible from the given start grid and input pieces. It stores the address of the current grid from $a0 to the register currGrid. It stores the address of the converted pieces from $a1 to the register pieces. Then, the function loads the address of the final grid to $a1. Backtrack then calls is_equal_grids to check whether currGrid is equal to the final grid. It stores the result in $v0, and the backtrack function returns 1 if they are equal. Otherwise, the function continues. The input parameter $a2 is stored to the register i. Then, the number of pieces the user has input is loaded from $gp and stored in $t0. If $a2 is greater than or equal to the number of pieces, then the backtrack function returns false, or sets the value of $v0 to 0.

If none of the above conditions are satisfied, then the function continues. We first calculate the address of pieces[i], or the current piece to be dropped, and store it in $s3. Next thing to do is to calculate the maximum x coordinate of the current piece (pieces[i]) and store the return value (maximum x coordinate), which is in $v0 after calling the get_max_x_of_piece function, in register $s4. We then initialize the counter of the for loop to 0 in $s5. Then, we calculate 6 – max_x_of_piece and store the result in $s6. We also initialize an unsuccessful counter to 0 and store it in $s7. It is then used in a for loop that exits when $s5 is equal to $s6.

Inside the for loop, we first move to $a0 the address of currGrid, to $a1 the address of current piece/pieces[i], and to $a2 the value of the loop counter (offset) from $s5. We then call the drop_piece_in_grid function. We store $v0 to $t0/nextGrid, and $v1 to $t1/success.

(In the implementation with line clearing, we move $t0 to $a0 and call the line_clearing function, which will be expounded on later).

We then check if $t1 is TRUE/1. If $t1 is TRUE/1, we recursively call backtrack. Its input parameters are as follows: $a0 – nextGrid, $a1 – pieces, $a2 – i + 1. If after recursively calling backtrack returns TRUE, then this conditional returns TRUE as well. Otherwise, we increment offset/$s5 and jump back to the start of the for loop.

On the other hand, if $t1 is FALSE/0, we then check if unsuccessful/$s7 is equal to 6 – max_x_of_piece – 1 or ($s6 – 1). If it is equal, then we recursively call backtrack. Its input parameters are as follows: $a0 – nextGrid, $a1 – pieces, $a2 – i + 1. If after recursively calling backtrack returns True, then this conditional returns TRUE as well. Otherwise, we increment offset/$s5, increment the unsuccessful counter ($s7), and jump back to the start of the for loop.

It is important to keep track of the unsuccessful counter because it counts how many attempts to drop the given piece are unsuccessful. If it is unsuccessful, we still have to check if the piece can be dropped by offsetting it along the x axis. If it still is unsuccessful when $s7 = $s6 – 1, then we skip to the next input piece. This means that we cannot drop the piece no matter how many times we offset it along the x axis in the grid.

XII (Bonus 2). line_clearing

```
line_clearing:
        ####preamble####
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      nextGrid, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        ####preamble####
        move    nextGrid, $a0   # nextGrid = $a0
        move    $t0, nextGrid   # $t0 = nextGrid for use in function
        addi    $s1, $0, 9      # $s1 = i = 9
        line_clearing_i:
                beq     $s1, 3, exit_line_clearing      # Exit when i = 3
        check_if_clearable:
                addi    $s2, $0, 1      # $s2 = Check if line can be cleared
                addi    $s3, $0, 0      # $s3 = j
                line_clearing_j:
                        beq     $s3, 6, exit_lc_j       # Exit when j = 6
                        sll     $t1, $s1, 3             # 8i
                        add     $t1, $t1, $s3           # 8i + j
                        add     $t0, $t0, $t1           # nextGrid[i][j]
                        lb      $t1, 0($t0)             # $t1 = nextGrid[i][j]
                        seq     $t1, $t1, dot           # nextGrid[i][j] == '.'? If yes, don't clear
                        beq     $t1, 0, continue_lc_j   # nextGrid[i][j] != '.', continue looping
                        addi    $s2, $0, 0              # $s2 = Line can't be cleared
                        move    $t0, nextGrid           # Restore address of nextGrid to $t0
                        j exit_lc_j                     # Exit the loop
                        continue_lc_j:
                        addi    $s3, $s3, 1             # Increment j
                        move    $t0, nextGrid           # Restore address of nextGrid to $t0
                        j       line_clearing_j
                exit_lc_j:
                beq     $s2, 1, shift_rows      # If line can be cleared, shift rows downwards
                addi    $s1, $s1, -1            # Decrement i
                j       line_clearing_i
                shift_rows:
                move    $t0, nextGrid
                addi    $t2, $s1, 0     # $t2 = $s1 = x
                shift_rows_out:
                        beq     $t2, 3, check_if_clearable      # Exit when x = 3
                        addi    $t3, $0, 0      # $t3 = y
                        shift_rows_inner:
                                beq     $t3, 6, exit sri        # Exit when y = 6
                                addi    $t4, $t2, -1    # x-1
                                sll     $t4, $t4, 3     # 8(x-1)
                                add     $t4, $t4, $t3   # 8(x-1) + y
                                add     $t0, $t0, $t4   # address of nextGrid[x-1][y]
                                lb      $t4, 0($t0)     # $t4 = nextGrid[x-1][y]
                                sb      $t4, 8($t0)     # nextGrid[x][y] = $t4
                                move    $t0, nextGrid   # Restore address of nextGrid to $t0
                                addi    $t3, $t3, 1     # Increment y by 1
                                j       shift_rows_inner
                        exit_sri:
                        addi    $t2, $t2, -1    # Decrement x
                        j       shift_rows_out
        exit_line_clearing:
        move    $v0, nextGrid
        ####end####
        lw      $ra, 28($sp)
        lw      nextGrid, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        addi    $sp, $sp, 32
        ####end####
        jr      $ra
```

This function is called inside the backtrack function, right after the drop_piece_in_grid function. After dropping the piece in the grid, it checks whether there are lines/rows in the grid that can be cleared. A row can be cleared if it has 6 Xs. The implementation of this function works by checking each row from the bottommost to the topmost of the 6x6 grid.

The address of the grid is in $a0 and is stored in the register nextGrid. We make a temporary copy of this address by moving it to $t0. We initialize a counter for the for loop to 9, which is stored in $s1. The outer for loop is for iterating through each row of the 6x6 grid (from bottom to top), so it exists when $s1 = 3. We initialize a boolean counter to 1, which denotes if the given row can be cleared, and store the value to $s2. Next, the inner for loop is for iterating through each column of the row. We also initialize a counter for this inner for loop to 0, which is stored in $s3. We exit out of the for loop when $s3 is 6.

Now, in the innermost for loop, we iterate through each element of the row. We'll keep on iterating until we encounter a '.'. Once the program encounters a '.' in the row, it sets $s2 to FALSE/0, and exits out of the loop. Otherwise, it keeps on iterating through each column.

If $s2 is 1 after iterating through each column of a row (i.e., the row can be cleared), then we shift all the rows above it downwards. However, before we move on to the next row, we have to check again if there are rows that can be cleared. We keep on repeating this until there are no more rows that can be cleared before we move on to check the next row (go back to the outer loop).

The loops keep running until after the topmost row of the 6x6 grid has been checked for full rows. The address of nextGrid (grid with cleared lines) is then moved to $v0, then the function exits.

MAIN FUNCTION

```
main:
        initialize_empty_rows           # Initialize first 4 empty rows of start grid
        sw      $v0, 0($gp)             # Store address of start_grid in $gp + 0
        jal     input_6x6

        initialize_empty_rows           # Initialize first 4 empty rows of final grid
        sw      $v0, 4($gp)             # Store address of final_grid in $gp + 4
        jal     input_6x6

        get_numPieces:
        do_syscall(5)                   # Get input for number of pieces
        sw      $v0, 8($gp)             # Store numPieces in $gp + 8

        jal     get_input_pieces        # Asks the user to input pieces
        sw      $v0, 12($gp)            # Store address[piecesAscii] to $gp + 12

        jal     convert_piece_to_pairs  # Converts the input pieces to (row, col) coordinates
        sw      $v0, 16($gp)            # Store address[converted_pieces]

        lw      $a0, 0($gp)             # $a0 = start_grid
        lw      $a1, 16($gp)            # $a1 = converted_pieces
        addi    $a2, $0, 0              # $a2 = 0
        jal     backtrack               # backtrack(start_grid, converted_pieces, 0)

        beq     $v0, 0, return_no       # $v0 = TRUE? Otherwise, branch
        la      $a0, yes
        do_syscall(4)                   # Print 'YES'
        j       exit

        return_no:
        la      $a0, no
        do_syscall(4)                   # Print 'YES'

exit:   exit_program                    # Terminate the program
```

The first thing done in the main function is to initialize the first 4 rows of start_grid to all dots. Recall that we can use the macro initialize_empty_rows to make this easier for us. The address of start_grid is stored in $v0, which is then stored to the address $gp + 0. Then, we call the function input_6x6 to ask the user to input the start grid.

After that, we initialize the first 4 rows of final_grid to all dots. Again, we can utilize the macro initialize_empty_rows to make this easier. The address of final_grid is stored in $v0, which is then stored to the address $gp + 4. Then, we call the function input_6x6, this time to ask the user to input the final grid.

We then proceed to the label get_numPieces, which asks the user how many pieces they want to input. The value the user inputs is stored in $v0, which is then stored to the address $gp + 8. Right after that, the user is asked to input the pieces by calling the get_input_pieces function. The address that points to the start of the input pieces in the heap is stored in $v0, which is then stored in $gp + 12.

Then, we get the (row, col) coordinates of the input pieces and store them in heap. We call the convert_piece_to_pairs function. The address that points to the start of the coordinates of all the input pieces in the heap is stored in $v0, which is then stored in $gp + 16.

We load the address of the start grid from $gp + 0 to $a0, load the address of the converted pieces from $gp + 16, and load 0 to $a2. We then call the backtrack function, which starts the exhaustive search and determines whether the final grid is possible from the start grid and input pieces. After calling the backtrack function, the result is stored in \$v0. If \$v0 is 1, then we print 'YES', denoting that it is possible to reach the final grid given the start grid and input pieces. Otherwise, we print 'NO', denoting that it is not possible to reach the final grid given the start grid and input pieces.