CADIANG, IVAN CASSIDY A.
2021-12575
LAB 1

CS 21 PROJECT 1
TETRISITO

UNIVERSITY OF THE PHILIPPINES – DILIMAN
COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

Table of Contents

1 An Introduction to the Chosen Implementation

For this Machine Problem, the student opted to use the Python code `mp1c.py` as a guide in completing the project. The Python implementation was converted to its equivalent version in MIPS (Microprocessor without Interlocked Pipeline Stages) and was further optimized by removing redundant and unnecessary procedures (e.g., breaking loops early) for the sake of better performance and ease of debugging in its MIPS equivalent. Hence, the succeeding sections will be heavily based on the names of the variables, functions, conditions, and loops in `mp1c.py`.

1.1 C: Several pieces fall, in any order

In the Python code provided, `mp1c.py`, the program/algorithm is capable of solving **Implementation C**. Thus, the student declares that the MIPS program created is for the **C version of Tetrisito**. This is where *"one to five pieces can be given, and the order in which the pieces are dropped can be altered."*

1.2 BONUS 2: Lines are completed and cleared

Furthermore, the student opted to implement **BONUS 2** together with the **C** version of Tetrisito. This was tested first with a high-level implementation in Python. As such, `mp1c.py` was slightly modified to support this feature. After that, it can be easily imported/translated to its MIPS implementation. To reiterate the problem specifications, this is where *"once a piece settles into place in the grid, if a row (or line) is fully occupied, that line gets cleared, and any blocks previously on top of the cleared line will descend until they hit other blocks or the bottom of the grid."*

2 Implementation Approach: Main

In the `main` section, most of the code concerns input and output handling. We will see how this was managed and explain the design decisions made by the student to successfully store and access the necessary inputs in memory.

2.1 Initializations and the `.data` Segment

```
.include "macros.asm"
.eqv numPieces $t9

.text
# ============== MAIN ============== #
main: li $t0, 0x10040000 # load heap memory address to $t0; this is to manually track memory allocation so that we can 'free' the heap later
sw $t0, 0($gp) # store the base address of the heap globally
sw $t0, 4($gp) # store the last allocated heap address (initial setup)
li $t0, 0 # sbrk_override_flag = False
sw $t0, 8($gp) # store sbrk_override_flag globally
la $t0, start_grid # get start_grid starting address
la $t2, final_grid # get final_grid starting address
li $t8, 0 # i = 0
li $t1, 6 # MAX_COLS_FOR_GRID = 6
addiu $t0, $t0, 24 # skip 4 empty rows for falling piece
addiu $t7, $t2, 24 # skip 4 empty rows for falling piece
```

Code Block 1: Initializations at the Start of the `.text` Segment

In Code Block 1, the `main` section begins with initializing values/addresses. The starting address of the heap, `0x10040000`, is saved globally (`$gp`). This is to manually track the heap later on when we need to dynamically allocate memory for `deepcopy_chosen()` and `deepcopy_grid()` alongside with `sbrk_override_flag`. These concepts/implementations are further expounded in the Auxiliary Functions Section. For now, we focus on the loading of the addresses of `start_grid` (`$t0`) and `final_grid` (`$t2`). To better understand this, we take a look at the `.data` segment in Code Block 2.

```
.data
# grids will be 10 rows by 6 columns, so we can put the piece at the top before letting it fall
start_grid: .byte '.', '.', '.', '.', '.', '.'
```

```
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'

final_grid: .byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
.byte '.', '.', '.', '.', '.', '.'
```

Code Block 2: .data Segment with start_grid and final_grid

Notice that we already have a dedicated space for two 10x6 grids (the extra 4x6 grid on top of the original 6x6 grid is for the pieces to have space when dropping them), namely start_grid and final_grid. This is to prevent having to track available memory when filling out the data since these inputs already have predefined sizes. Hence, we can reserve space early in the memory and call their corresponding labels to get each of their base addresses when needed.

```
allocate_bytes(line_grid, 7)
allocate_bytes(line_piece, 5)
allocate_bytes(chosen, 5) # 5 pieces; 1 byte for bool (0 or 1); tracks which piece has been
    used
allocate_bytes(converted_pieces, 40) # exactly 4 hashtags per piece; 5 pieces; 2 pos; 4*5*2
allocate_bytes(pieceAscii, 20) # 4x4 grid + 4 null terminators; (4*4)+4
allocate_bytes(pieceCoords, 8) # exactly 4 hashtags per piece; 2 pos; 4*2

newline: .asciiz "\n"
yes: .asciiz "YES"
no: .asciiz "NO"
hashtag: .byte  '#'
bigX: .byte 'X'
dot: .byte '.'
```

Code Block 3: .data Segment with allocate_bytes and String & Character Initializations

In Code Block 3, `allocate_bytes` is a macro-defined instruction that is further explained in the Macros Section. Basically, this code snippet shows the computation and allocation of bytes for the different sizes of input. Here, `line_grid` refers to the taking of the input rows of the grid (we process them row-per-row) and `line_piece` to the piece rows. Note that the null terminator '\0' was considered when taking the input for the rows of the grids and pieces. Thus, we reserve one additional byte for each row of the grid and piece respectively.

Furthermore, `chosen` is reserved with 5 bytes which is dictated by the maximum number of pieces possible that will be dropped in the grid (usage will be explained later). Next, we have `converted_pieces` that will store the row-column indices of each hashtag in the piece/s. Note that `converted_pieces` is dependent on `pieceCoords` as we process the input one piece at a time. Since we have exactly 4 '#' characters in every possible piece, we can reserve 40 bytes (4 hashtags $\times$ 2 coordinates $\times$ 5 pieces max).

Similarly, `pieceAscii` stores the 4x4 grid input which is dependent on `line_piece`. Note that `pieceAscii` can be initialized the same way we did with `start_grid` and `final_grid` and vice versa.

Lastly, the necessary strings (and characters) are preloaded as well. For strings, `.asciiz` was used. For characters, `.byte` was used to simply show that we do not need to allocate more than what is needed.

2.2 start_grid and final_grid Input Handling

```
31 loop_for_start_grid: read_str(line_grid, 7) # line = input()
32 la $t3, line_grid # get line base address
33 li $t2, 0 # counter = 0
34
35 start_loop_for_line: lbu $t4, hashtag # load '#' character
36 addu $t5, $t3, $t2 # target_index_on_row_array = line_base_addr + counter
37 lbu $t5, 0($t5) # load value of target index
38 bne $t5, $t4, start_increment_line_counter # check if target_index_value == '#'
39 replace_elem_in_grid_with_X($t0, bigX) # start_grid replacement X
40
41 start_increment_line_counter: addiu $t2, $t2, 1 # counter++
42 bne $t2, $t1, start_loop_for_line # character for character in line; while (counter != 6)
43 addiu $t8, $t8, 1 # i++
44 bne $t8, $t1, loop_for_start_grid # for _ in range(6)
```

Code Block 4: start_grid Input Handling through Nested Loops

To fill `start_grid` (`$t0`) (a.k.a. the initial grid) with arbitrary input from the user, we need to perform nested looping (i.e., perform 1 outer loop & 1 inner loop) as shown in Code Block 4. Nested looping is necessary so that we can access each row-column index of the grid (in similar terms, accessing a 2-D array).

Note that the guard for the outer loop is in Line *41*. Likewise, the guard for the inner loop is in Line *43*. The loops are designed this way since `start_grid` (`$t0`) has a definite size and more importantly to avoid relying too much on `j` instructions when doing iterations. Thus, in doing this, we can effectively utilize branch instructions in MIPS. This concept is applicable across the other implementations with similar looping logic. However, note that the accessing of the rows starts at row 3 (0-indexing) as a minor optimization to avoid looping through the excess 4x6 grid on top of `start_grid` (`$t0`). Prior initializations are shown in Code Block 1, specifically in Lines *24-29*.

During the execution of the outer loop, we first get the row input from the user by calling a macro-defined instruction `read_str`. The said instruction will store the input in `line_grid`. Then, we access each element of `line_grid` and check if there is a '`#`' character. If so, we call another macro-defined instruction `replace_elem_in_grid_with_X` to replace '`#`' with '`X`' to distinguish already placed blocks in the grid with the pieces to be dropped later. The reading and storing logic of the arrays are further explained in the Macros Section. For now, we assume these operations on arrays work as intended.

```
31 li $t8, 0 # i = 0
32 loop_for_final_grid: read_str(line_grid, 7) # line = input()
33 la $t3, line_grid # get line base address
34 li $t2, 0 # counter = 0
35
36 final_loop_for_line: lbu $t4, hashtag # load '#' character
37 addu $t5, $t3, $t2 # target_index_on_row_array = line_base_addr + counter
38 lbu $t5, 0($t5) # load value of target index
39 bne $t5, $t4, final_increment_line_counter # check if target_index_value == '#'
40 replace_elem_in_grid_with_X($t7, bigX) # final_grid replacement X
41
42 final_increment_line_counter: addiu $t2, $t2, 1 # counter++
43 bne $t2, $t1, final_loop_for_line # character for character in line; while (counter != 6)
44 addiu $t8, $t8, 1 # i++
45 bne $t8, $t1, loop_for_final_grid # for _ in range(6)
```

Code Block 5: final_grid Input Handling through a Single Loop

Likewise, for `final_grid` (`$t7`) (a.k.a. the goal grid), the exact same logic applies as shown in Code Block 5. However, note that the offsetted address (skipped 4x6 grid on top) of `final_grid` is now in the register `$t7` as shown in Line *29* of Code Block 1. Furthermore, since we are using the same registers, we need to ensure first that they are resetted to their base values to perform the loop properly. An example of this is shown in Line *31* since we reused `$t8` for the counter of the outer loop. We do the same concept for the other essential registers as well.

2.3 numPieces Input Handling and the Initialization of Boolean Values for chosen

```
62 li $t8, 0 # i = 0
63 get_int_user_input(numPieces) # numPieces = int(input())
64 sw numPieces, 12($gp) # numPieces (global)
65 la $t4, chosen
66
67 loop_for_chosen: addu $t3, $t4, $t8 # target_index_on_chosen_array = chosen_base_addr + counter
68 sb $0, 0($t3) # False
69 addiu $t8, $t8, 1 # i++
70 bne $t8, numPieces, loop_for_chosen # False for _ in range(numPieces)
```

Code Block 6: `numPieces` Input Handling & Initializing Boolean Values to `chosen`

Observing Code Block 6, in Line *63*, we called a macro-defined instruction `get_int_user_input` to read and store the arbitrary integer input of the user to `numPieces` (a.k.a. the number of tetrominoes). Note that `numPieces` is only an equivalent name to the register `$t9` as illustrated in Code Block 1 in Line *15*. Since we will need the value of `numPieces` across other functions, we can store it globally through offsetting the base address of `$gp`.

In Lines *67-70*, `chosen` (`$t4`) array is initialized/filled with `0`'s (`False`) using a loop which depends on the number of tetriminoes to be dropped (`numPieces`). The purpose of `chosen` (`$t4`) is to keep track of the pieces that have successfully (or unsuccessfully) dropped in the grid. This is particularly useful in ensuring that we are able to monitor and cover every possible order of the pieces when we perform backtracking later on.

2.4 pieceAscii Input Handling and Conversion to Row-Column Coordinates

```
72 li $t8, 0 # i = 0
73 li $t7, 0 # j = 0
74 li $t1, 4 # MAX_COLS_FOR_PIECE = 4
75 la $t4, pieceAscii
76 li $t0, 0 # track consumed bytes of converted pieces
77
78 loop_for_pieceAscii_grid: read_str(line_piece, 5) # line = input()
79 la $t3, line_piece # get line base address
80 li $t2, 0 # counter = 0
81 li $t5, 1 # temp = 1
82 bgt $t5, $t7, skip_offset # manage offset of base address of pieceAscii for the succeeding iterations (<=1)
83 addiu $t4, $t4, 5 # offset by 5 base address of pieceAscii (null terminator included)
84
85 skip_offset: addiu $t7, $t7, 1 # j++
86
87 pieceAscii_loop_for_line: addu $t5, $t3, $t2 # target_index_on_row_array = line_base_addr + counter
88 lbu $t5, 0($t5) # load value of target index
89 addu $t6, $t4, $t2 # target_index_on_pieceAscii_array = pieceAscii_base_addr + counter
90 sb $t5, 0($t6) # store character of line in pieceAscii
91 addiu $t2, $t2, 1 # counter++
92 bne $t2, $t1, pieceAscii_loop_for_line # # character for character in line; while (counter != 4)
93 bne $t7, $t1, loop_for_pieceAscii_grid # for _ in range(4)
94
95 prepare_for_convertion: la $t5, pieceAscii
96 move $a0, $t5 # pieceAscii (passing to function)
97 jal convert_piece_to_pairs # convert_piece_to_pairs(pieceAscii)
98 move $t5, $v0 # piecePairs = convert_piece_to_pairs(pieceAscii)
99 move $t3, $v1 # index tracker
```

Code Block 7: `pieceAscii` Input Handling through Nested Loops

For Code Block 7, we first focus on Lines *72-76*. We initialize the necessary values to perform nested iterations later for each of the input pieces (tetrominoes). Note that the guard for the outer loop can be found at Line *113* which dictates how many 4x4 grids are we expecting before we end the loop. Similarly, the guard for the inner loop is found at Line *93*, which controls the iteration on each of the input rows that are dictated by the user. Note also that we have another nested loop in the inner loop (guard at Line *92*) that stores each character of the row input in `pieceAscii` ($t4).

Now, at execution, the program asks first the user to input a row, which defines partially the shape of the piece. A macro-defined instruction `read_str` is called to take a row input from the user similar to how `start_grid` and `final_grid` inputs are taken. Then, the row input is stored in `line_piece` ($t3). Now, we access each character of the string in `line_piece` ($t3) and store them to `pieceAscii` ($t4). These steps are done through Lines *78-93*. However, note that we need to handle the extra null terminator at the end of the input and do proper offsetting on the base address of `pieceAscii` ($t4) since we process the grid row-wise (as seen in Lines *82-85*).

After one 4x4 grid (`pieceAscii`) is completed, we now call the first function `convert_piece_to_pairs()`, which its implementation is explained in the Functions Section. For now, we assume that the function works as intended. The idea behind `convert_piece_to_pairs()` is that it returns a list of row-column coordinates of each `'#'` character in `pieceAscii`. This is to ensure that we can keep track of the location of the 'form' of the piece when we drop it later in the grid and handle collisions properly. Note that the return value of `convert_piece_to_pairs()` is stored in the register $t5 (namely, `piecePairs`). These are handled through Lines *95-99* of the code.

```
101 append_to_converted_pieces: la $t6, converted_pieces
102 addu $t6, $t6, $t0 # adjust base address of converted_pieces
103 lbu $t4, 0($t5) # load i from temp
104 sb $t4, 0($t6) # store i to converted_pieces
105 lbu $t4, 1($t5) # load j from temp
106 sb $t4, 1($t6) # store j to converted_pieces
107 addiu $t0, $t0, 2 # consumed bytes = consumed bytes + 2
108 addiu $t5, $t5, 2 # adjust base address of piecePairs
109 bne $t5, $t3, append_to_converted_pieces
110 li $t7, 0 # j = 0
111 la $t4, pieceAscii # reset offset base address of pieceAscii
112 addiu $t8, $t8, 1 # i++
113 bne $t8, numPieces, loop_for_pieceAscii_grid # for _ in range(numPieces)
```

Code Block 8: Convert the Tetrominoes into Coordinates

Finally, we append the list of row-column coordinates in the array `converted_pieces` ($t6) as seen in Lines *101-109* of Code Block 8. Note that since we are loading two elements simultaneously in one iteration, the loop has been set with a step value of 2 to properly track and access the correct addresses of `converted_pieces` ($t6) and `piecePairs` ($t5) in each iteration.

2.5 Output Printing

```
115 la $a0, start_grid
116 la $a1, chosen
117 la $a2, converted_pieces
118 jal backtrack # backtrack(start_grid, chosen, converted_pieces)
119 move $t0, $v0 # answer = backtrack(start_grid, chosen, converted_pieces)
120 beq $t0, $0, answer_no
121 print_str(yes) # print("YES")
122 j terminate
123
124 answer_no: print_str(no) # print("NO")
125
126 terminate: exit() # syscall code 10
```

Code Block 9: Code Snippet for the Output

In Lines *115-119* of Code Block 9, we passed `start_grid` ($a0), `chosen` ($a1), and `converted_pieces` ($a2) to the function `backtrack()`. In simple terms, `backtrack()` is an exhaustive search algorithm that checks every possible scenario using the tetrominoes and initial grid given if the goal grid can be attained or not. Hence, `backtrack()` returns either `0` or `1` (`False` or `True`) ($t0). If `True`, then we output `YES` using a macro-defined instruction `print_str`. Otherwise, we output `NO` as shown in Lines *120-124*. Lastly, we terminate the program using a macro-defined instruction called `exit` as seen in Line *126*.

3 Implementation Approach: Functions

Note that for the given functions below, the **PREAMBLE** and **POSTAMBLE** were intentionally omitted for conciseness of the documentation. As such, we assume that the registers are properly saved and restored for each function call. Note also that `s` registers are the only registers being preserved across calls. Regardless, this can be verified through `cs21project1c.asm`.

Additionally, for the looping and branching design of most of the functions, the student opted to use `j` instructions to ensure the correctness of the implementation since there are several nested loops and conditional statements present in the functions. Furthermore, this might complicate the MIPS code and degrade its readability.

3.1 `convert_piece_to_pairs()`

```
137 li $t5, 4 # temp = 4
138 li $t6, 0 # i = 0
139 li $s2, 0 # j = 0
140 li $s0, 0 # pieceCoords_index_counter = 0
141
142 convert_piece_loop: access_2d_array($t6, $s2, $a0, $s1, 5) # pieceGrid[i][j]
143 lbu $t3, hashtag # load '#' character
144 bne $s1, $t3, skip_append_coords # if pieceGrid[i][j] == '#'
145 la $t3, pieceCoords
146 addu $t3, $t3, $s0 # adjust base address of pieceCoords
147 sb $t6, 0($t3) # i
148 sb $s2, 1($t3) # j
149 addiu $s0, $s0, 2 # offset += 2
150
151 skip_append_coords: addiu $s2, $s2, 1 # j++
152 bne $s2, $t5, convert_piece_loop # for j in range(4)
153 addiu $t6, $t6, 1 # i++
154 li $s2, 0
155 bne $t6, $t5, convert_piece_loop # for i in range(4)
156 la $s1, pieceCoords
157 addu $s0, $s1, $s0 # signal end of array
158 move $v0, $s1 # return pieceCoords
159 move $v1, $s0 # return pieceCoords array index termination
```

Code Block 10: Code Snippet for `convert_piece_to_pairs()`

In Code Block 10, the function `convert_piece_to_pairs()` takes in a 4x4 grid, which contains the piece denoted by '`#`' characters. Furthermore, it returns a list of the row-column coordinates of each '`#`' character in the 4x4 grid and the last index processed (note here that we have a contiguous memory) since we need a guard in the loop when the list of coordinates is appended to `converted_pieces`. The coordinates represent the actual 'form' of the piece so that we are able to identify the piece when we start to drop it in the grid. These also help in collision checking with other blocks or pieces, monitor the piece when it is potentially going beyond grid borders, and dropping logic.

Taking a look at Lines *137-159*. Observe that the implementation consists of nested iterations on `pieceAscii` and a macro-defined instruction `access_2d_array` is called to extract the element given the row-column indices of the array. Then, if an element is a '`#`', then we save the current row-column indices in `pieceCoords` (`$s1`). This is repeated until the whole 4x4 grid is explored completely.

3.2 `backtrack()`

Since the function `backtrack()` is the main algorithm of Tetrisito, we divide `backtrack()` into Code Blocks 11 and 12 to explain each part of the algorithm with clarity.

```
lw numPieces, 12($gp) # get numPieces from global
jal is_equal_grids # is_equal_grids(currGrid, final_grid)
move $s2, $v0
li $s1, 0 # result = False
li $t2, 1 # True
```

```
188 beq $s2, $t2, backtrack_return_True # if is_equal_grids(currGrid, final_grid)
189 li $s2, 0 # i = 0
190
191 backtrack_outer_loop: beq $s2, numPieces, backtrack_return_result # for i in range(len(chosen))
192 move $s0, $a0 # save contents of $a0 (currGrid) in preparation for possible recursion or to make space
193 move $s5, $a1 # save contents of $a1 (chosen) to make space
194 access_1d_array($s2, $s5, $s4) # chosen[i]
195 li $t2, 1 # True
196 beq $s4, $t2, backtrack_increment_i # if not chosen[i]:
197 move $a0, $s2 # pass value of i to get_max_x_of_piece
198 jal get_max_x_of_piece # get_max_x_of_piece(pieces, i)
199 move $s3, $v0 # max_x_of_piece = get_max_x_of_piece(pieces, i)
200 jal deepcopy_chosen # deepcopy(chosen)
201 move $s6, $v0 # chosenCopy = deepcopy(chosen)
202 move $a0, $s0 # move currGrid back to $a0
203 li $a3, 0 # offset
204 li $t0, 6
205 subu $s3, $t0, $s3 # range(6 - max_x_of_piece)
```

Code Block 11: Code Snippet for `backtrack()`: Outer Loop

In Code Block **11**, the algorithm starts with checking the most simple case, which is the current grid and goal grid are the already equal. Note that we used the term 'current grid' for the supposedly 'initial grid' since `backtrack()` is recursively defined as we will see later. In Line *184*, it calls the function `is_equal_grids()` and checks if they are equal or not. If the return value of `is_equal_grids()` is 1 (`True`), then we immediately return `True` ($v0 → $s2) and end `backtrack()` skipping most of the algorithm proper (Lines *185-188*; *232-233* in Code Block **12**). Otherwise, if the return value is `False` ($v0 → $s2) , we go to Lines *189-201* for the first half of the `backtrack()` algorithm.

The outer loop manages the number of pieces to be dropped in the grid. To ensure that we do not drop the same piece (assume uniqueness for identical pieces), we check if `chosen[i]` ($s4), where `i` denotes the index to be accessed, is `True` or `False` (accessing logic is done by the macro-defined instruction `access_1d_array`). If `chosen[i] = True`, then the corresponding piece has already been dropped in the current grid. Thus, we check the next piece and so forth. Otherwise, we call the function `get_max_x_of_piece()` to properly move the piece horizontally when capturing other dropping points for the piece. This is shown in Line *198-199*.

After calling `get_max_x_of_piece()` and storing its return value to the register $s3 (`max_x_of_piece`), we call another function `deepcopy_chosen()`. `deepcopy_chosen()` allocates a portion of heap memory to create an exact copy of the contents of `chosen`. This is vital in the program execution as we need to preserve the pending states of the pieces for one recursive instance to properly exhaust all potential solutions.

Finally, after we have copied `chosen` and store the return value ($v0) (base address of the copy) in the register $s6 (`chosenCopy`), we can now proceed to the inner loop that contains the recursive call to `backtrack()`.

```
207 backtrack_inner_loop: beq $a3, $s3, backtrack_increment_i # for offset in range(6 - max_x_of_piece)
208 move $a1, $s2 # copy contents of i to $a1 to get pieces[i] for drop_piece_in_grid
209 jal drop_piece_in_grid # drop_piece_in_grid(currGrid, i, pieces, offset)
210 move $a0, $v0 # nextGrid
211 move $t0, $v1 # success
212 move $a1, $s6 # store chosenCopy to $a1
213 beq $t0, $0, backtrack_increment_offset # if success
214 li $t2, 1 # True
215 store_in_1d_array($s2, $a1, $t2) # chosenCopy[i] = True
216 move $s7, $a3 # save the value of offset temporarily
217 jal backtrack # backtrack(nextGrid, chosenCopy, pieces)
218 move $s1, $v0 # result = backtrack(nextGrid, chosenCopy, pieces)
219 move $a3, $s7 # store back the value of offset back to $a3
220 beq $s1, $0, backtrack_increment_offset # if result
221 j backtrack_return_True
222
```

```
223 backtrack_increment_i: move $a1, $s5 # move chosen back to $a1
224 addiu $s2, $s2, 1
225 j backtrack_outer_loop
226
227 backtrack_increment_offset: move $a0, $s0 # move currGrid back to $a0
228 jal free # free(nextGrid); if nextGrid is currGrid, no deallocation will happen
229 addiu $a3, $a3, 1 # offset++
230 j backtrack_inner_loop
231
232 backtrack_return_True: li $v0, 1 # return True
233 j backtrack_done
234
235 backtrack_return_result: move $v0, $s1 # return result
```

Code Block 12: Code Snippet for `backtrack()`: Inner Loop and Return Value

For Code Block **12**, in Line *207*, the inner loop controls the `offset` of the piece starting from the leftmost side of the grid. For each iteration, we offset the piece by 1 to the right until the rightmost side of the piece is about to breach the right border of the current grid (refer to Lines *203-205* in Code Block **11**).

Now, we call on the function `drop_piece_in_grid()` to execute the dropping of the piece in the current grid. `drop_piece_in_grid()` has two return values, namely `nextGrid` (`$v0` → `$a0`) and `success` (`$v1` → `$t0`) as shown in Lines *209-211*. If the piece was successfully dropped in the grid, the function returns the grid with the dropped piece and `success = True`. Otherwise, it returns the same grid before the function was called and `success = False`.

If `success = True`, we update `chosenCopy[i] = True` and recursively call `backtrack()` which takes in `nextGrid` (`$a0`), `chosenCopy` (`$a1`), and `converted_pieces` (`$a2`). Otherwise, we increment `offset` (`$a3`) in the inner loop and make an another attempt to drop the piece in a different location. These are all evident in Lines *213-219*; *227-230*. Note that the function `free()` in Line *228* will be explained in another section.

Now, after the recursive call on `backtrack()`, we check if the return value `result` (`$v0`) is `True` or `False`. If `result = True`, it means that after n recursive calls, we found a solution to the goal grid. Therefore, we return `True` and end `backtrack()`. Otherwise, we loop back and attempt to drop the piece in a different location. Likewise, these are all shown in Lines *220-221; 227-233*

After exhausting all possible locations for a piece, we increment the counter for the outer loop and repeat the process for the other pieces (if there are any). If the outer loop managed to finish, it means that exhausted all possible scenarios and the goal grid is yet to be achieved. Therefore, we return `result = False` and end `backtrack()` (Lines *223-225*; *235*).

3.3 `is_equal_grids()`

```
li $s0, 1 # result = True
li $t2, 4 # i = 4; skip first 4 rows since they are only for dropping pieces
li $t6, 0 # j = 0
li $t4, 10 # range(6 + 4)
li $t5, 6 # range(6)
la $t7, final_grid # get final_grid starting address

is_equal_grids_loop: access_2d_array($t2, $t6, $a0, $t0, 6)
access_2d_array($t2, $t6, $t7, $t1, 6)
beq $t0, $t1, is_equal_grids_skip_set_to_false # gridOne[i][j] == gridTwo[i][j]
li $s0, 0 # result = False;
j is_equal_grids_return # exit the loop since one element is not equal; thus, the grids are not equal.

is_equal_grids_skip_set_to_false: addiu $t6, $t6, 1 # j++
bne $t6, $t5, is_equal_grids_loop # for j in range(6)
li $t6, 0 # j = 0
addiu $t2, $t2, 1 # i++
bne $t2, $t4, is_equal_grids_loop # for i in range(6 + 4)
```

```
275
276 is_equal_grids_return: move $v0, $s0 # return result
```

Code Block 13: Code Snippet for `is_equal_grids()`

The function `is_equal_grids()` that is shown in Code Block 13 takes in the current grid (`currGrid`) (`$a0`) that is passed to the function `backtrack()`. Note that the goal grid (`final_grid`) (loaded in `$t7`) is global.

`is_equal_grids()` also uses similar concepts to access the grids. Thus, it utilizes nested loops and calls the macro-defined instruction `access_2d_array` as well. However, there are minor optimizations made to decrease the number of instructions executed. It is important to keep in mind that `is_equal_grids()` will be also called for each recursive call of `backtrack()`. Thus, optimizations are necessary to prevent useless and redundant calculations as much as possible. The following improvements are made to the said function:

- Outer loop iteration starts at `i = 4` (`$t2`). This is to skip again the extra 4x6 grid on top of the 6x6 grid since these will be always equal regardless (Lines *258* & *274*).
- One mismatched element when comparing the grids makes both grids unequal. Hence, if we detect a mismatch, we immediately break from the loop and return `result = False` (Lines *266-268*).

3.4 `get_max_x_of_piece()`

```
290 li $t4, -1 # max_x = -1
291 li $t5, 1 # index = 1
292 mul $t7, $a0, 8 # offset by i * 8
293 addu $t5, $t5, $t7 # index = index + offset
294 addiu $t6, $t5, 8 # max index for a piece
295
296 get_max_x_of_piece_loop: beq $t5, $t6, get_max_x_of_piece_return # for block in piece
297 access_1d_array($t5, $a2, $t7) # block[1]
298 blt $t4, $t7, get_max_x_of_piece_change_max_val # max(max_x, block[1])
299 j get_max_x_of_piece_increment_index
300
301 get_max_x_of_piece_change_max_val: move $t4, $t7 # max_x = max(max_x, block[1])
302
303 get_max_x_of_piece_increment_index: addiu $t5, $t5, 2 # index = index + 2
304 j get_max_x_of_piece_loop
305
306 get_max_x_of_piece_return: move $v0, $t4 # return max_x
```

Code Block 14: Code Snippet for `get_max_x_of_piece()`

The `get_max_x_of_piece()` function in Code Block 14 is similar to the `max()` function. It takes in a list of coordinates of a piece (`converted_pieces[i]`) (`i` → `$a0`; `converted_pieces` → `$a2`) and accesses each `x`-coordinate of the piece using the macro-defined instruction `access_1d_array`. If the succeeding `x`-coordinate is greater than the previous `x`-coordinate, then we change the max value `max_x` (`$t4`). Note that by default, `max_x = -1` (`$t4`) as shown in Line *290*.

Additionally, since we have contiguous memory, we need to set delimiters to access the correct list of coordinates of a piece. Hence, given an index `i`, we multiple this by 8 (recall that we have exactly 2 coordinates for each of the 4 hashtags) to serve as our offset to get the 'base' address of the list of coordinates we need. Then, we add 1 since the `x`-coordinates are stored on odd number indices (Lines *292-294*). Lastly, we add this by 8 so we have a guard condition (i.e., signal that this is the end of the list) for the loop (Line *296*).

3.5 `drop_piece_in_grid()`

For the function `drop_piece_in_grid()`, it will be divided into 5 parts to take into account each step done to move the piece downwards in the grid and check if it is valid.

```
move $s4, $a1 # store i temporarily in $s4 to prepare for malloc()
jal deepcopy_grid # deepcopy(grid)
move $a1, $s4 # bring back i to $a1
```

```
327 move $s0, $v0 # gridCopy = deepcopy(grid)
328 mul $t1, $a1, 8 # for each piece[i], we have exactly 4 '#'s. Thus, 4 * 2 = 8 coords and counter = i * 8
329 addiu $t2, $t1, 8 # range for end of loop (array end)
330 move $t4, $a2 # temp = base address of pieces
331 move $t7, $s0 # temp = base address of gridCopy
332 lbu $s1, hashtag # load '#' character
333 lbu $s2, bigX # load 'X' character
334 lbu $s3, dot # load '.' character
335
336 drop_piece_in_grid_block_piece_loop: beq $t1, $t2, drop_piece_in_grid_while_True_loop # for block in piece
337 addu $t4, $t4, $t1 # block[0]
338 lbu $t5, 0($t4) # load value of block[0]
339 addiu $t4, $t4, 1 # block[1]
340 lbu $t6, 0($t4) # load value of block[1]
341 addu $t6, $t6, $a3 # col = block[1] + yOffset
342 store_in_2d_array($t5, $t6, $t7, $s1, 6) # gridCopy[block[0]][block[1] + yOffset] = '#'; put piece in grid
343 move $t4, $a2 # reset base address of pieces
344 move $t7, $s0 # reset base address of gridCopy
345 addiu $t1, $t1, 2 # counter += 2
346 j drop_piece_in_grid_block_piece_loop
```

Code Block 15: Code Snippet for `drop_piece_in_grid()`: Placing the Piece in the Grid

The function `drop_piece_in_grid()` takes in the values `currGrid` ($a0), `converted_pieces[i]` (`converted_pieces` → $a2; `i` → $a1), and `yOffset` ($a3). Now, before we start placing the piece on the top of the 10x6 grid, we need to create a copy of the current grid to prevent the manipulations being done insinde the function from mutating it. It is possible that we will still need the original grid when the dropped piece is invalid (e.g., piece protrudes on top of the 6x6 grid). To perform this, we call the function `deepcopy_grid()` as shown in Line *325* of Code Block **15**.

After we have a copy of the grid (`gridCopy`) in $v0, we moved the value to the register $s0 (Line *327*), we can now place the piece on top of the grid using the coordinates from `converted_pieces`. Using the same accessing logic for the coordinates described in the previous subsection, we get the base address of `converted_pieces[i]` and iterate over each `x` and `y` coordinates of the '`#`' of the piece. The accessing logic and prior initializations are shown in Lines *328-334*.

Once we have accessed to some `x` and `y` coordinate, namely `block[1]` ($t6) and `block[0]` ($t5) respectively, we can now place them on the grid. To do this, we execute a macro-defined instruction `store_in_2d_array`, which accesses `gridCopy` ($s0 → $t7) and replaces the element with the portion of the piece as defined by its coordinates. We repeat this step until the piece is completely placed on top of the 6x6 grid. Note that we also need to consider here the current offset (`yOffset`) ($a3) value for the piece. Hence, we add `yOffset` ($a3) to `block[1]` ($t6) to offset the piece horizontally to the right. These are evident in Lines *336-346*.

```
348 drop_piece_in_grid_while_True_loop: li $t1, 0 # i = 0; while True; only active blocks are '#'; frozen blocks are 'X'
349 li $t2, 0 # j = 0
350 li $t4, 10 # range(4 + 6)
351 li $t5, 6 # range(6)
352 li $t7, 0 # flag_one = False
353 li $t8, 0 # flag_two = False
354
355 drop_piece_in_grid_outer_loop: beq $t1, $t4, drop_piece_in_grid_canStillGoDown # for i in range(4 + 6)
356
357 drop_piece_in_grid_inner_loop: beq $t2, $t5, drop_piece_in_grid_increment_i # for j in range (6)
358 access_2d_array($t1, $t2, $s0, $t3, 6) # access gridCopy[i][j]
359 beq $t3, $s1, flag_one_True # if gridCopy[i][j] == '#'
360 j drop_piece_in_grid_increment_j # first condition failed; hence, we can safely increment j
361 check_other_conditions: addiu $t6, $t1, 1 # i + 1
362 beq $t6, $t4, flag_two_True # i + 1 == 10
363 access_2d_array($t6, $t2, $s0, $t3, 6) # access gridCopy[i + 1][j]
```

```
364 beq $t3, $s2, flag_two_True # if gridCopy[i + 1][j] == 'X'
365 j drop_piece_in_grid_increment_j # all conditions failed (False and False); we can safely
        increment j
366
367 flag_one_True: li $t7, 1 # flag_one = True
368 j check_other_conditions
369
370 flag_two_True: li $t8, 1 # flag_two = True
371 j break_all_loops # canStillGoDown = False; we can safely break from the while loop since we
        have (True and True) since next branch would fail regardless
372
373 drop_piece_in_grid_increment_i: addiu $t1, $t1, 1 # i++
374 li $t2, 0 # j = 0
375 j drop_piece_in_grid_outer_loop
376
377 drop_piece_in_grid_increment_j: addiu $t2, $t2, 1 # j++
378 j drop_piece_in_grid_inner_loop
```

Code Block 16: Code Snippet for `drop_piece_in_grid()`: Collision Checking Below the Piece

At this point, we are ready to drop the piece in the grid. However, before dropping the piece, we need to check first if it is possible to drop the piece; that is, we consider if the piece will overlap with a frozen block denoted by the `'X'` character or the piece is already at the bottom of the grid. To do this, we first traverse the grid and confirm the current location of the piece (denoted by `'#'` characters). After confirming its location, we check one space down relative to the piece. This implementation is shown in Code Block **16**.

The `while True` loop in Line *348* ensures that we keep dropping the piece one row down until the piece collides with a frozen block or is already resting at the bottom of the grid. The nested loops (Lines *355-357*) are already standard procedure for grid traversal. Observe that in Lines *359-371*, we only break from the `while True` loop when one of the two fail conditions are satisfied.

Notice that after getting the element in `gridCopy[i][j]` through `access_2d_array`, we check first if the element is a `'#'`. If not, then we immediately increment the inner loop to check the next element of the grid. Otherwise, we change the value of the register $t7 (`flag_one`) to `True` (Lines *359*; *367-368*). This means that we have found the location of a portion of the piece. Thus, we check the two fail conditions if they are satisfied or not. If the current location of the `'#'` is at row 9, then the whole piece cannot be dropped anymore since it reached the bottom of the grid (Line *364*). Otherwise, we check if the the row just below the `'#'` is a `'X'`. If it is, we change the value of the register $t8 (`flag_two`) to `True` and break from the `while True` loop (Lines *362*; *370-371*). Now, if the grid is completely searched and the two fail conditions are yet to be satisfied, we can drop the piece one row down.

```
380 drop_piece_in_grid_canStillGoDown: li $t0, -1 # if canStillGoDown; range(8, -1, -1); move
        cells of piece down, starting from bottom cells
381 li $t1, 8 # i = 8
382 li $t2, 0 # j = 0
383 li $t4, 6 # range(6)
384
385 drop_piece_in_grid_canStillGoDown_outer_loop: beq $t1, $t0, drop_piece_in_grid_while_True_loop
         # for i in range(8, -1, -1)
386
387 drop_piece_in_grid_canStillGoDown_inner_loop: beq $t2, $t4,
        drop_piece_in_grid_canStillGoDown_decrement_i # for j in range(6)
388 access_2d_array($t1, $t2, $s0, $t3, 6) # access gridCopy[i][j]
389 bne $t3, $s1, drop_piece_in_grid_canStillGoDown_increment_j # if gridCopy[i][j] == '#'; move
        cells down one space
390 addiu $t5, $t1, 1 # i + 1
391 store_in_2d_array($t5, $t2, $s0, $s1, 6) # gridCopy[i + 1][j] = '#'
392 store_in_2d_array($t1, $t2, $s0, $s3, 6) # gridCopy[i][j] = '.'
393 j drop_piece_in_grid_canStillGoDown_increment_j
394
395 drop_piece_in_grid_canStillGoDown_decrement_i: subiu $t1, $t1, 1 # i--
396 li $t2, 0 # j = 0
397 j drop_piece_in_grid_canStillGoDown_outer_loop
398
399 drop_piece_in_grid_canStillGoDown_increment_j: addiu $t2, $t2, 1 # j++
```

```
400 j drop_piece_in_grid_canStillGoDown_inner_loop
```

Code Block 17: Code Snippet for `drop_piece_in_grid()`: Moving the Piece Downwards

Now, we need to move the piece one row downwards. This implementation is shown in Code Block **17**. Note that the implementation is simply using nested loops to access and store/replace elements in the grid as we did before. However, there is one notable change. Instead of traversing the grid the usual way (i.e., top to bottom), we traverse the grid through a bottom-up approach. This is to avoid the `'#'` characters from being replaced with a `'.'` character (in reality, this is the natural way to implement the algorithm). When Lines *389-392* are executed, it checks first if the current element is a `'#'`. If it is, we replace the space (element) below it with `'#'` and the current element with `'.'` to create the effect of moving the piece downward in the grid. Otherwise, we keep traversing the grid so that we can cover the whole piece.

```
402 break_all_loops: li $t0, 100 # maxY = 100
403 li $t1, 0 # i = 0
404 li $t2, 0 # j = 0
405 li $t4, 10 # range(10)
406 li $t5, 6 # range(6)
407
408 drop_piece_in_grid_maxY_outer_loop: beq $t1, $t4, check_piece_protrudes # for i in range(4 + 6)
409
410 drop_piece_in_grid_maxY_inner_loop: beq $t2, $t5, drop_piece_in_grid_maxY_increment_i # for j in range(6)
411 access_2d_array($t1, $t2, $s0, $t3, 6) # access gridCopy[i][j]
412 bne $t3, $s1, drop_piece_in_grid_maxY_increment_j # if gridCopy[i][j] == '#'
413 ble $t0, $t1, drop_piece_in_grid_maxY_increment_j # compares if maxY <= i; if false, then we update maxY
414 move $t0, $t1 # maxY = min(maxY, i)
415 j drop_piece_in_grid_maxY_increment_j
416
417 drop_piece_in_grid_maxY_increment_i: addiu $t1, $t1, 1 # i++
418 li $t2, 0 # j = 0
419 j drop_piece_in_grid_maxY_outer_loop
420
421 drop_piece_in_grid_maxY_increment_j: addiu $t2, $t2, 1 # j++
422 j drop_piece_in_grid_maxY_inner_loop
```

Code Block 18: Code Snippet for `drop_piece_in_grid()`: Grid Border Checking for the Piece

At some point in the execution of the `while True` loop (recall in Line *348* in Code Block **16**), it will inevitably exit the loop once the piece reaches one of the fail conditions as mentioned earlier. After it exits the loop, the program will go to Line *402* of Code Block **18**. In this part, we need to check if the rested piece is within the 6x6 grid. Thus, we need to check the height of the piece. Since the `y`-coordinate of the grid grows downward (i.e., row 0 of the 10x6 grid is the max `y`), we need to find the minimum `y`-coordinate of the piece. This is done through Lines *408-422* using nested loops to traverse the grid and check for `'#'` once more. Note that we set an arbitrary initial amount using the register `$t0` (`maxY = 100`) as shown in Line *402* for the first comparison that will be done.

```
424 check_piece_protrudes: li $t1, 3 # $t1 = 3
425 bgt $t0, $t1, piece_protrudes_else # if maxY <= 3; piece protrudes from top of 6x6 grid
426 jal free # free(gridCopy)
427 move $v0, $a0 # return grid
428 li $v1, 0 # return False
429 j drop_piece_in_grid_done
430
431 piece_protrudes_else: move $a0, $s0 # move to $a1 contents of gridCopy
432 jal check_line_clear # check_line_clear(gridCopy)
433 move $a0, $v0 # gridCopy = check_line_clear(gridCopy)
434 jal freeze_blocks # freeze_blocks(gridCopy)
435 li $v1, 1 # return freeze_blocks(gridCopy), true; (note: contents of $v0 is the return value of freeze_blocks)
```

Code Block 19: Code Snippet for `drop_piece_in_grid()`: Return Values

Once we get the height of the piece, we first check if the piece protrudes from the top of the 6x6 grid. In other words, $0 \leq$ `maxY` ≤ 3. If the piece protrudes, we return the original grid (the state of the grid wherein the piece is not yet dropped) and also return `False` as shown in Lines *424-429* of Code Block **19**. Note that the function `free()` found in Line *426* will be explained in the Auxiliary Functions Section.

Otherwise, we call on the bonus implementation function `check_line_clear()` (Line *432*). This function checks whether we have successfully filled one or more rows with '#'. Then, it clears the rows and descend the blocks above it one row down. Now, after `check_line_clear()` returns the (possibly) modified grid (`gridCopy`), we call the function `freeze_blocks()` to replace the '#' characters with 'X' to indicate and distinguish that these are already inactive blocks (Line *434*). After `freeze_blocks()` returns the modified grid (`gridCopy`), we finally return `gridCopy` ($v0) and also return `True` ($v1). These are all shown in Lines *431-435*.

3.6 `freeze_blocks()`

```
455 lbu $s0, hashtag # load '#' character
456 lbu $s1, bigX # load 'X' character
457 li $t0, 0 # i = 0
458 li $t1, 0 # j = 0
459 li $t2, 10 # range(4 + 6)
460 li $t4, 6 # range(6)
461
462 freeze_blocks_loop: access_2d_array($t0, $t1, $a0, $t3, 6) # access grid[i][j]
463 bne $t3, $s0, freeze_blocks_skip # if grid[i][j] == '#'
464 store_in_2d_array($t0, $t1, $a0, $s1, 6) # grid[i][j] = 'X'
465 freeze_blocks_skip: addiu $t1, $t1, 1 # j++
466 bne $t1, $t4, freeze_blocks_loop # for j in range(6)
467 addiu $t0, $t0, 1 # i++
468 li $t1, 0 # j = 0
469 bne $t0, $t2, freeze_blocks_loop # for i in range(4 + 6)
470 move $v0, $a0 # return grid
```

Code Block 20: Code Snippet for `freeze_blocks()`

For the function `freeze_blocks()`, the implementation is straightforward as it only requires again nested loops for grid accessing/storing as shown in Code Block **20**. The function takes in a grid and searches for '#' characters. If found, we replace it with 'X'. Likewise, these are done through the macro-defined instructions `access_2d_array` and `store_in_2d_array`. Note that the outer loop is found at Line *469* and the inner loop in Line *466*. After the grid is exhausted, we return the new grid.

3.7 `check_line_clear()` (Bonus Implementation)

To explain better the implementation, the function is divided into Code Blocks **21** and **22**.

```
li $t0, 9 # i = 9
li $t1, 3 # guard = 3
li $t2, 0 # j = 0
li $t4, 6 # range(6)
li $t5, 2 # guard_inner_j = 2
li $t6, 0 # k = 0
li $t8, 10 # 10
lbu $s1, dot # load '.' character
check_line_clear_while_loop: beq $t0, $t1, check_line_clear_return # while i != 3

check_line_clear_outer_j_loop: beq $t2, $t4, if_line_clear # for j in range(6)
access_2d_array($t0, $t2, $a0, $s0, 6) # access grid[i][j]
bne $s0, $s1, check_line_clear_increment_outer_j # if grid[i][j] == '.'
subiu $t0, $t0, 1 # i--
li $t2, 0 # j = 0; reset j for next while loop iteration
j check_line_clear_while_loop

check_line_clear_increment_outer_j: addiu $t2, $t2, 1 # j++
j check_line_clear_outer_j_loop
```

Code Block 21: Code Snippet for `check_line_clear()`: Conditions for Line Clear

For the line clearing feature, the function `check_line_clear` takes in the current grid (`grid`) (`$a0`). In constructing this, the student considered first the conditions for a successful line clear and then proceeded to implement the solution through code. The following considerations made are as follows:

- Algorithm must be able to determine rows filled with `'#'` or `'X'` characters **only**.
- Algorithm must be able to actually clear the row and move the blocks above it one row down.
- Algorithm must be able to handle cases where consecutive line clears are possible (e.g., a Tetris), line clears that are separated (e.g., assuming 0-indexed, rows 7 and 9 can be cleared but row 8 cannot be cleared) and a combination of both (e.g., assuming 0-indexed, rows 6, 8, and 9 can be cleared but row 7 cannot be cleared).

Now, in the implementation, we first observe Lines *495-505* in Code Block **21**. The `while` loop in Line *495* handles the traversal for each row. Note that we are starting at the bottommost row (`grid[i], i = 9` (`$t0`)) similar to the approach on moving the piece downward in `drop_piece_in_grid()`. Furthermore, the guard condition for the `while` loop is when the next iteration is for row 3 (0-indexed). The extra 4x6 grid on top would be all `'.'` regardless if we traverse them or not.

In Line *497*, the inner loop handles the traversal for each element in the row. Using the macro-defined instruction `access_2d_array`, we can check the current element if it is a `'.'` character or not. If it is, that means the row is not suitable for a line clear. Thus, we decrement `i` and proceed to the next iteration of the `while` loop.

Note that prior initializations were made for the implementation above as shown in Lines *487-490; 494*.

```
507 if_line_clear: subiu $t2, $t0, 1 # j = i - 1
508
509 if_line_clear_inner_j_loop: beq $t2, $t5, if_i_not_ten # for j in range(i - 1, 2, -1)
510
511 if_line_clear_k_loop: beq $t6, $t4, if_line_clear_decrement_inner_j # for k in range(6)
512 access_2d_array($t2, $t6, $a0, $s0, 6) # access grid[j][k]
513 addiu $t7, $t2, 1 # j + 1
514 store_in_2d_array($t7, $t6, $a0, $s0, 6) # grid[j + 1][k] = grid[j][k]; move each row below
515 j if_line_clear_increment_k
516
517 if_line_clear_decrement_inner_j: subiu $t2, $t2, 1 # j--
518 li $t6, 0 # k = 0
519 j if_line_clear_inner_j_loop
520
521 if_line_clear_increment_k: addiu $t6, $t6, 1 # k++
522 j if_line_clear_k_loop
523
524 if_i_not_ten: li $t2, 0 # j = 0; reset j for next while loop iteration
525 addiu $t7, $t0, 1 # i + 1
526 beq $t7, $t8, check_line_clear_while_loop # if i + 1 != 10
527 addiu $t0, $t0, 1 # i++
528 j check_line_clear_while_loop
529
530 check_line_clear_return: move $v0, $a0 # return grid
```

Code Block 22: Code Snippet for `check_line_clear()`: Grid Manipulation to Perform Line Clear

In the case that a `'.'` character was not found in the row, we proceed to Line *507* in Code Block **22** for the actual line clearing. To perform this, we access the row above the current row (note that the current row contains the line to be cleared). Then, for each element from the row above, we copy each of the element and replace the elements on the current row (to access the row above, we simply decrement the current row index by 1). This creates the effect that we 'cleared' the row and move the row above it one row down simultaneously. This concept can be extended to the other rows through a simple loop with bottom-up approach as shown in Line *509*. Then, we use another inner loop (as seen in Line *511*) to traverse each element of the row. After that, we use the elements to replace the contents of the row below them (to access the row below, we simply increment the row index by 1). These are all done with the macro-defined instructions `access_2d_array` and `store_in_2d_array`. Lastly, the whole implementation explained above is evident in Lines *507-522*.

For cases that contain consecutive line clears (as shown in Lines *524-528*), these were handled by:

- Before the next iteration of the `while` loop, we increment the row index by 1. This is to ensure that we did not miss a row suitable for line clear since they were brought one row down earlier.
- A condition is added as well to catch the case where the line clear performed was at the bottom of the grid. In other words, it is unnecessary to check the row below since we are already at the bottom of the grid. Furthermore, this might as well cause an issue as we are attempting to access a non-existent value beyond the grid if left unnoticed.

Similarly, note that prior initializations were made as well (Lines *491-493* in Code Block **21**). Finally, once the `while` loop ends, the function returns the modified grid as shown in Line *530*.

4 Implementation Approach: Auxiliary Functions

Note that for the given auxiliary functions below, the **PREAMBLE** and **POSTAMBLE** were intentionally omitted for conciseness of the documentation. As such, we assume that the registers are properly saved and restored for each function call. Note also that `s` registers are the only registers being preserved across calls. Regardless, this can be verified through `cs21project1c.asm`.

4.1 `deepcopy_chosen()`

```
lw numPieces, 12($gp) # get numPieces from global
move $t6, $a1 # store base address of chosen temporarily
li $a1, 6 # allocate bytes for chosen (max number of Pieces possible)
jal malloc # malloc(sizeof(chosen))
move $t1, $v0
move $a1, $t6 # bring back chosen to $a1
li $t2, 0 # i = 0

deepcopy_chosen_loop: access_1d_array($t2, $a1, $t3) # $t3 = chosen[i]; assume all arrays are
    1D at this point (contiguous memory)
addu $t4, $t1, $t2 # offset for copy; base address + offset
sb $t3, 0($t4) # store chosen[i] to copy[i]
addiu $t2, $t2, 1 # i++
bne $t2, numPieces, deepcopy_chosen_loop # for i in range(numPieces)
move $v0, $t1 # return base address of copy
```

Code Block 23: Code Snippet for `deepcopy_chosen()`

As the name implies, `deepcopy_chosen()` is an auxiliary function to create a new copy of the `chosen` array and returns the new copy of the array by saving its base address as shown in Code Block **23**. Now, before we can copy its contents, we need to reserve memory for the copy. Thus, we call on the auxiliary function `malloc()` to reserve n bytes, where n is the number of pieces to be dropped. After `malloc()` reserves n bytes and returns the base address of the new allocated memory, we execute a single loop to iterate over all of the contents of `chosen` and store them to the copy. Once finished, we return the base address of the copy.

4.2 `deepcopy_grid()`

```
li $a1, 60 # 10 x 6 grid = 60 characters; len(grid)
move $t8, $a0 # save contents of $a0 to temp
jal malloc # malloc(sizeof(grid))
move $t1, $v0
move $a0, $t8 # move grid back to $a0
li $t2, 0 # i = 0

deepcopy_grid_loop: access_1d_array($t2, $a0, $t3) # $t3 = grid[i]; assume all arrays are 1D
    at this point (contiguous memory)
addu $t4, $t1, $t2 # offset for copy; base address + offset
sb $t3, 0($t4) # store grid[i] to copy[i]
addiu $t2, $t2, 1 # i++
bne $t2, $a1, deepcopy_grid_loop # for i in range(len(grid))
move $v0, $t1 # return base address of copy
```

Code Block 24: Code Snippet for `deepcopy_grid()`

Similar to the previous auxiliary function, `deepcopy_grid()` creates a new copy of the current grid and returns the new copy of the grid by saving its base address as shown in Code Block **24**. Now, before we can copy its contents, we need to reserve memory for the copy. Thus, we call on the auxiliary function `malloc()` to reserve exactly 60 bytes. After `malloc()` reserves 60 bytes and returns the base address of the new allocated memory, we execute a single loop to iterate over all of the contents of the current grid and store them to the copy. Note that since we are operating on contiguous memory and the row-column indices of the grid does not concern us, we can use a single loop to traverse the elements of the grid. Once finished, we return the base address of the copy.

4.3 `malloc()`

```
487 lw $s1, 4($gp) # get address of available heap
488 move $s0, $s1 # update base address of current heap
489 addu $s1, $s1, $a1 # allocate n number of bytes
490 li $t1, 0x103FFFFC # max allowed heap address by sbrk
491 bgt $s1, $t1, override_sbrk # if current allocation is more than the maximum allowed heap,
        override to use more than sbrk allows.
492 sw $s0, 0($gp) # update base address of current heap
493 sw $s1, 4($gp) # store the new allocated memory globally
494 move $v0, $s0 # return the base address of the new allocated heap
495 j malloc_done
496
497 override_sbrk: lw $s0, 8($gp) # check sbrk_override_flag
498 beq $s0, 0, malloc_flag_true # if False, sbrk_override_flag = True
499 j malloc_skip
500
501 malloc_flag_true: li $s0, 1 # sbrk_override_flag = True
502 sw $s0, 8($gp) # store new value of sbrk_override_flag globally
503 li $s1, 0x7FC00000 # new initial address allocation; (note: 0x10400000 to 0x7FBFFFFC is
        restricted in MARS)
504 sw $s1, 0($gp) # update base address of current heap
505 sw $s1, 4($gp) # store the new allocated memory globally
506
507 malloc_skip: lw $s1, 4($gp) # get address of available heap
508 move $s0, $s1 # update base address of current heap
509 addu $s1, $s1, $a1 # allocate n number of bytes
510 sw $s0, 0($gp) # update base address of current heap
511 sw $s1, 4($gp) # store the new allocated memory globally
512 move $v0, $s0 # return the base address of the new allocated heap
```

Code Block 25: Code Snippet for `malloc()`

The implementation of `malloc()` on Code Block **25** is a mimic and improved version of the system call code 9 (`sbrk`) of MARS 4.5, which dynamically allocates memory using the heap. The `sbrk` functionality on MARS 4.5 does not support a negative offset on the heap (i.e., freeing the heap). Furthermore, the heap capacity is limited to ∼4 MB (`0x10040000` to `0x103FFFFF`) [Source: **Github**]. If we try to allocate memory beyond `0x103FFFFF`, MARS 4.5 outputs a runtime exception. The student discovered that MARS 4.5 restricts the user from storing anything from `0x10400000` to `0x7FBFFFFC`. To remedy this, recall the initializations made in Lines *19-23* of Code Block **1**. To implement a custom `sbrk` we need the following:

- **Base Address of Current Heap.** In Lines *19-20*, we initialized and stored the initial/starting address of heap memory in MIPS (`0x10040000`) globally. This is to keep track of the base address of the allocated memory whenever we call `malloc`.
- **Address of Available Heap Memory.** In Line *21*, the same starting address `0x10040000` is stored globally. However, once we call `malloc` to reserve memory, this address increments depending on the amount of bytes being reserved. This is to ensure that we do not overwrite the contents of occupied heap memory. These are shown in Lines *487-494* and *507-512*.
- **Heap Memory Limitation Bypass.** In Lines *22-23*, we initialized and stored a flag value set initially to `False` called `sbrk_override_flag` globally. Recall that in the **MIPS Green Sheet**, the heap and stack

have the same segment in memory allocation. The heap grows upward and the stack grows downward. This implies that heap and stack memory share the same space in memory. However, this is restricted by MARS 4.5. To bypass this, when `malloc` is called and the heap allocation already exceeds `0x103FFFFC`, we set `sbrk_override_flag = True` and use the stack's memory starting from `0x7FC00000` instead. This is shown in Lines *490* and *497-505*.

4.4 `free()`

```
lw $s0, 0($gp) # get the base address of current heap
sw $s0, 4($gp) # free allocated heap
```

Code Block 26: Code Snippet for `free()`

To prevent `malloc()` from exceeding the initial limit as much as possible, we need to deallocate or 'free' the heap as shown in Code Block **26**. However, the implementation `free()` is severely limited only to the most recent allocation to the heap. This is the reason why we call `free()` immediately in Line *228* of Code Block **12** and Line *426* of Code Block **19** when we surely know that the copy of the grid for that instance has no use anymore.

5 Implementation Approach: Macros

The macros for `cs21project1c.asm` are found in `macros.asm` as stated in Line *14* of Code Block **1**.

5.1 `do_syscall`

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

Code Block 27: Macro Instruction: `do_syscall`

The macro `do_syscall` in Code Block **27** is a shortcut that simply takes in an integer and loads the integer into the register `$v0` to issue the `syscall` instruction.

5.2 `exit`

```
.macro exit
do_syscall(10)
.end_macro
```

Code Block 28: Macro Instruction: `exit`

The macro `exit` in Code Block **28** is a shortcut to call syscall code 10 using the macro `do_syscall(10)`, which terminates the execution of the program.

5.3 `get_int_user_input`

```
.macro get_int_user_input(%store_reg)
do_syscall(5)
move %store_reg, $v0
.end_macro
```

Code Block 29: Macro Instruction: `get_int_user_input`

The macro `get_int_user_input` in Code Block **29** is a shortcut to call syscall code 5 through `do_syscall(5)` to read the integer input from the user and stores the value in `$v0` to an arbitrary register.

5.4 print_str

```
.macro print_str(%label)
la $a0, %label
do_syscall(4)
.end_macro
```

Code Block 30: Macro Instruction: print_str

The macro print_str in Code Block **30** is a shortcut to load a null-terminated string given a label to the register $a0 and calls syscall code 4 through do_syscall(4) to print the string in the console.

5.5 read_str

```
.macro read_str(%label, %len)
la $a0, %label
li $a1, %len
do_syscall(8)
.end_macro
```

Code Block 31: Macro Instruction: read_str

The macro read_str in Code Block **31** is a shortcut to call syscall code 8 through do_syscall(8) to read the string input from the user and store the string to the address (label) specified in the register $a0. Note that we also need to declare the string's expected length, which will be stored in $a1.

5.6 allocate_bytes

```
.macro allocate_bytes(%label, %n)
%label: .space %n
.end_macro
```

Code Block 32: Macro Instruction: allocate_bytes

The macro allocate_bytes in Code Block **32** is a shortcut to reserve n bytes worth of memory in the .data segment and tags the location (base address) with an arbitrary label.

5.7 replace_elem_in_grid_with_X

```
.macro replace_elem_in_grid_with_X(%base_addr, %label)
mul $t4, $t8, 6 # multiply row index by 6 to access desired row
addu $t4, $t4, $t2 # access desired col
addu $t4, $t4, %base_addr # target_index_grid = base_addr + offset
lbu $t5, %label # get 'X' character
sb $t5, 0($t4) # mark frozen blocks as 'X'
.end_macro
```

Code Block 33: Code Snippet for replace_elem_in_grid_with_X

The macro replace_elem_in_grid_with_X in Code Block 33 is a shortcut to replace an element in the 10x6 grid (or simply 6x6 grid) with 'X'. It takes in the base address of the 2-D array and the designated label (base address) of where the character 'X' is stored in the .data segment. Note that this macro is specifically tailored to support only the operations on the input setup on the initial grid (start_grid) and the goal grid (final_grid).

5.8 get_memory_address

```
.macro get_memory_address(%n, %m, %addr, %step)
mul $t3, %n, %step
addu $t3, $t3, %m
addu $t3, %addr, $t3
.end_macro
```

Code Block 34: Code Snippet for get_memory_address

The macro `get_memory_address` in Code Block **34** is a shortcut to get the memory address of a 2-D array given the row-column indices, base address, and number of columns. The number of columns act as the multiplier for the row index and the product is added to the column index to get the offset for the base address.

5.9 `access_2d_array`

```
.macro access_2d_array(%n, %m, %addr, %store_reg, %step)
get_memory_address(%n, %m, %addr, %step)
lbu %store_reg, 0($t3)
.end_macro
```

Code Block 35: Code Snippet for `access_2d_array`

The macro `access_2d_array` in Code Block **35** is a shortcut to access an element in a 2-D array. Note that this macro is macro-dependent on `get_memory_address` and the only additional step performed is to load the value to a designated register given the memory address computed.

5.10 `access_1d_array`

```
.macro access_1d_array(%n, %addr, %store_reg)
addu $t3, %addr, %n
lbu %store_reg 0($t3)
.end_macro
```

Code Block 36: Code Snippet for `access_1d_array`

The macro `access_1d_array` in Code Block **36** is a shortcut to access an element in a 1-D array. It simply gets the sum of the base address and offset to retrieve the final address of the element we want to access.

5.11 `store_in_2d_array`

```
.macro store_in_2d_array(%n, %m, %addr, %val, %step)
get_memory_address(%n, %m, %addr, %step)
sb %val, 0($t3)
.end_macro
```

Code Block 37: Code Snippet for `store_in_2d_array`

The macro `store_in_2d_array` in Code Block **37** is a shortcut to store or replace an element in a 2-D array. Note that this macro is macro-dependent on `get_memory_address` and the only additional step performed is to store the value to the computed memory address.

5.12 `store_in_1d_array`

```
.macro store_in_1d_array(%n, %addr, %val)
addu $t3, %addr, %n
sb %val, 0($t3)
.end_macro
```

Code Block 38: Code Snippet for `store_in_1d_array`

The macro `store_1d_array` in Code Block **38** is a shortcut to store or replace an element in a 1-D array. It gets the sum of the base address and offset to retrieve the final address of where we want to store the given value.

END OF DOCUMENTATION