CONTENTS

PREFACE 2

SECTION 1: PREPLANNING AND HOUSEKEEPING 2

SECTION 2: MAIN 4

SECTION 3: BACKTRACK FUNCTION 6

Section 3.1: Memory management and unique namespaces 6

Section 3.2: Calling is_equal_grid 8

Section 3.3: Outer for loop and calling max_x_of_piece 8

Section 3.4: For loop 2 9

SECTION 4: IS_FINAL_GRID FUNCTION 10

SECTION 5: GET_MAX_X_OF_PIECE FUNCTION 11

SECTION 6: DROP_PIECE_IN_GRID FUNCTION 12

Section 6.1: Memory management 13

Section 6.2: Function initialization 14

Section 6.3: The while() loop 15

Section 6.4: Returning values 16

SECTION 7: FREEZE_BLOCKS FUNCTION 17

SECTION 8: CLEARLINES FUNCTION 17

SECTION 9: AUXILLARY MARS DEFINITIONS 19

Section 9.1: Syscall Macros 19

Section 9.2: .data segment 20

FOREWORD AND ENDING THOUGHTS 21

PREFACE

This submission of Project 1 is a MIPS translation of a modified version of the provided mp1c.py file, supplied as mp1c_bonus2.py. For simplicity and conciseness, this file will be referred to as .py from here onwards. References to variable names, loops, functions, and terms associated with .py will be formatted in `green`. Please access .py as needed.

Likewise, references to any MIPS-related terms will be in `blue`. References to the provided submission cs21project1C.asm will be referred to as .asm from here onwards.

This documentation does not discuss the code line-by-line; if needed, line-by-line comments are available in the contents of .asm. This documentation instead discusses general ideas and considerations taken in the process of creating the submission, and discusses the code in chunks rather than lines, which is more seamless and less forced as a form of communicating ideas.

Additionally, while this documentation has been submitted for multiple implementations, it specifically tackles Project 1 implementation C, with bonus 2. For other submissions, the above formatting applies with different filenames (for example, .py will now refer to mp1c.py where relevant), and sections of this documentation will either be irrelevant (such as Section 8 for implementation C) or be different (such as Section 3 for implementation C). As the same documentation can be submitted regardless of implementation, these changes will no longer be discussed.

SECTION 1: PREPLANNING AND HOUSEKEEPING

MIPS arrays are difficult to deal with. To bypass having to write lines of code dedicated to allocating memory for input variables (in particular, `start_grid`, `final_grid`, `chosen`, and `converted_pieces`), these variables have been pre-assigned memory values through `.data` in MARS. These variables have also been initialized with relevant values: both `start_grid` and `final_grid` are initialized to all periods, while `chosen` and `converted_pieces` have been initialized to all zeroes.

.py has `start_grid` and `final_grid` both as 2-dimensional 10x6 arrays of characters, where accesses are done through two indices. These two indices also function as coordinates of a 10x6 grid. These grids are replicated in .asm through a 60-byte long single dimensional array, accessed through a single index. Conversions from 2-dimensions to 1 dimension and vice versa are done through the following calculations:

$$\texttt{start\_grid}[i][j] = \texttt{start\_grid}[6i + j]$$
$$\texttt{start\_grid}[k] = \texttt{start\_grid}[\lfloor k/6 \rfloor][k \,\%\, 6]$$

.py has `chosen` as a variable size array with length from 1 to 5. This is easily replicated in MIPS through an array, however note that 5 is an awkward number for MIPS's word-alignment. The minimum number of words which can hold 5 bytes is 2 words (totaling 8 bytes), thus .asm implements `chosen` as an 8-byte array. While space-inefficient, doing so allows for more convenience in terms of both memory visualization (arrays do not start mid-word when viewed in memory) and instruction management (can now compare and manipulate arrays through words instead of bytes). Additionally, another variable can be stored in this array because of the extra space: the 6th byte will contain the total number of input pieces.

.py has converted_pieces as a variable size 3-dimensional array, with minimum dimensions 1x4x2 and maximum dimensions 5x4x2, where accesses are done through 3 indices. Luckily, the maximum size of converted_pieces of 5x4x2=40 neatly falls into MIPS's 4-byte word alignment. Thus, .asm implements converted_pieces through a 40-byte long 1-dimensional array. Below is an example of how bytes are laid out in .py and .asm.

converted_pieces: [i][j][k]							
[0][0][0]	[0][0][1]	[0][1][0]	[0][1][1]	[0][2][0]	[0][2][1]	[0][3][0]	[0][3][1]
[1][0][0]	[1][0][1]	[1][1][0]	[1][1][1]	[1][2][0]	[1][2][1]	[1][3][0]	[1][3][1]
[2][0][0]	[2][0][1]	[2][1][0]	[2][1][1]	[2][2][0]	[2][2][1]	[2][3][0]	[2][3][1]
[3][0][0]	[3][0][1]	[3][1][0]	[3][1][1]	[3][2][0]	[3][2][1]	[3][3][0]	[3][3][1]
[4][0][0]	[4][0][1]	[4][1][0]	[4][1][1]	[4][2][0]	[4][2][1]	[4][3][0]	[4][3][1]

Addr.	converted_pieces: [index]			
0x36	39	38	37	36
0x32	35	34	33	32
0x28	31	30	29	28
0x24	27	26	25	24
0x20	23	22	21	20
0x16	19	18	17	16
0x12	15	14	13	12
0x8	11	10	9	8
0x4	7	6	5	4
0x0	3	2	1	0

$$\texttt{converted\_pieces}[i][j][k] = \texttt{converted\_pieces}[8i + 2j + k]$$

There is no instance in the submission where the single index is converted to 3 indices, thus that calculation is not included above.

Note that both chosen and converted_pieces are of variable size and length, however .asm implements them as hardcoded length. This is done for three reasons: 1) it is easier to have a constant array size in terms of memory management as well as writing instructions, 2) the code needs to be able to run with 5 input pieces anyway, so having the array sizes constantly at maximum simplifies the troubleshooting and testing process, and 3) no restrictions have been provided on either time and space complexity, and so long as .asm runs in reasonable time then there is little reason to pursue better metrics for either.

SECTION 2: MAIN

```
main:
### PROCESSING START_GRID ###   #need to have a per-row for loop:
                la $t4, start_grid #$t4 stores the start_grid
                addi $t4, $t4, 24 #skip to the 5th row
                addi $t0, $t4, 36 #this is the end counter for the for loop, that is, we've reached the final byte in start_grid
                addi $t6, $0, 0x23 #$t6 stores the ASCII code for '#'.
                addi $t1, $0, 0x2E #$t1 stores the ASCII code for '.'.
start_row_loop: #need a place in memory to store 6 bytes for the first row. why not use $sp?
                add $t2, $0, $sp #original sp, as a for loop counter
                addi $sp, $sp, -6 #allocate 6 bytes in memory for the input row
                input_grid_row($sp) #receive input
                #iterate through all the characters, freeze if needed
                addi $t3, $0, 6 #for loop 2 end
start_bytes:    lb $t5, 0($sp) #t5 stores the current row character
                sb $t1, 0($t4) #store a period
                bne $t5, $t6, no_freeze_start #branch if current character is NOT #
                addi $t5, $0, 0x58 #replace the current character with X
                sb $t5, 0($t4) #store X in place of the current character
no_freeze_start:addi $sp, $sp, 1 #next byte, also appends counter
                addi $t4, $t4, 1 #next byte for start_grid
                bne $sp, $t2, start_bytes #continue start_bytes loop
                bne $t0, $t4, start_row_loop #continue start_row_loop
```

The above code handles receiving inputs for the start grid, equivalent to lines 110 to 116 of .py. This is done by first receiving a row of input, then checking the input byte-by-byte. It contains 2 for loops: one to iterate from the first input row to the last, and another to cycle through the bytes of the input row. If the current byte `$t5` is equivalent to the ASCII code for '#', then an ASCII 'X' is stored for that byte. Otherwise, an ASCII '.' is stored. ASCII values are stored to the memory assigned to `start_grid` through `.data`.

```
### PROCESSING FINAL_GRID ###
                #this is the same as above but for the final grid
                #need to have a per-row for loop:
                la $t4, final_grid #$t4 stores the final_grid
                addi $t4, $t4, 24 #skip to the 5th row
                addi $t0, $t4, 36 #this is the end counter for the for loop, that is, we've reached the final byte in final_grid
                addi $t6, $0, 0x23 #$t6 stores the ASCII code for '#'.
                addi $t1, $0, 0x2E #$t1 stores the ASCII code for '.'.
final_row_loop: #need a place in memory to store 6 bytes for the first row. why not use $sp?
                add $t2, $0, $sp #original sp, as a for loop counter
                addi $sp, $sp, -6 #allocate 6 bytes in memory for the input row
                input_grid_row($sp) #receive input
                #iterate through all the characters, freeze if needed
                addi $t3, $0, 6 #for loop 2 end
final_bytes:    lb $t5, 0($sp) #t5 stores the current row character
                sb $t1, 0($t4) #store a period
                bne $t5, $t6, no_freeze_final #branch if current character is NOT #
                addi $t5, $0, 0x58 #replace the current character with X
                sb $t5, 0($t4) #store X in place of the current character
no_freeze_final:addi $sp, $sp, 1 #next byte, also appends counter
                addi $t4, $t4, 1 #next byte for final_grid
                bne $sp, $t2, final_bytes #continue final_bytes loop
                bne $t0, $t4, final_row_loop #continue final_row_loop
```

The above snippet is a duplicate of the prior snippet, except it now manages inputs for `final_grid`. It is equivalent to lines 118 to 124 of .py.

```
### PROCESSING INPUT PIECES ###
                input_int($t0) #this is the number of input pieces
                la $t1, chosen #load the chosen array from static memory
                sb $t0, 5($t1) #too many bytes assigned to chosen anyway, so just set the 5th byte to number of input pieces
                addi $t1, $0, 0 #counter for a for loop
                addi $t8, $0, 0x23 #stores the ASCII code for #
                la $t6, converted_pieces #load the converted_pieces array from static memory
pieces_loop:    addi $t2, $sp, -16 #allocate 16 bytes for an input piece
                input_piece_row($t2) #receive the first 4 bytes
                addi $t2, $t2, 4 #place in the lowest numbered address
                input_piece_row($t2) #next 4 bytes
                addi $t2, $t2, 4 #next lowest numbered address
                input_piece_row($t2) #next 4 bytes
                addi $t2, $t2, 4 #next lowest numbered address
                input_piece_row($t2) #next 4 bytes
                addi $t2, $t2, -12 #return $sp to the start address of the input piece array
                #need to scroll through until we find all the #s:
                addi $t5, $0, -1 #count the number of loops until the #
                addi $t7, $0, 16 #end loop when at the last byte of the input piece
pieces_charloop:addi $t5, $t5, 1 #why start at offset -1? add here to start at 0, also loads next byte
                lb $t4, 0($t2) #t4 stores the current character
                addi $t2, $t2, 1 #next character
                bne $t4, $t8, notablock #if current char is not a #, move to the next
                addi $t3, $0, 4 #save 4
                div $t5, $t3 #divide the index by 4
                mflo $t3 #i = floor(index/4)
                sb $t3, 0($t6) #store the row i
                mfhi $t3 #j = index mod 4
                sb $t3, 1($t6) #store the column j
                addi $t6, $t6, 2 #move 2 bytes forward in converted_pieces
notablock:      bne $t5, $t7, pieces_charloop #continue cycling through bytes of piece
                addi $t1, $t1, 1 #move to next input piece
                bne $t0, $t1, pieces_loop #continue cycling through input pieces
```

The above code snippet manages the input of pieces, equivalent to lines `126` to `136` of .py. After storing the number of input pieces, a for loop is to parse the first input piece to the last. All rows of the input piece are received, then scrolled through to find the coordinates of '#' characters (which is managed through another for loop), similar to `convert_piece_to_pairs()`. The coordinates of '#' characters are saved in `converted_pieces` (defined in `.data`), as seen through the two bottom `sb` commands.

Notice the use of `$sp` despite not being in a function. While `$sp` is meant to be used to indicate the end location of the stack, it is also useful as a marker for allocating arrays. This is akin to using `sbrk` to allocate memory through the heap, except much more flexible as it is easy to manipulate the stack pointer. More examples of using `$sp` to dynamically allocate memory will appear throughout this documentation.

Note that inputs are received line-by-line (as seen through the macros `input_grid_row($sp)` and `input_piece_row($t2)`) because of the formatting of the input file; that is, even though all inputs in the file are visible, parsing multiple lines at once does not seem to work from testing.

```
### PREPROCESSING DONE. TIME TO BACKTRACK ###
                la $a0, start_grid #first argument: start address of start_grid
                la $a1, chosen #second argument: start address of chosen
                la $a2, converted_pieces #third argument: start address of converted_pieces
                jal backtrack #call backtrack

                beq $v0, $0, notpossible #v0 is 0 if false, 1 if true. 1 prints yes.
                addi $sp, $sp, -4 #allocate 4 bytes in memory for the string (3 chars + end char)
                li $t0, 0x00534559 #store an ASCII 'YES'
                j end_program #print the
notpossible:    li $t0, 0x00004F4E #store an ASCII 'NO'
end_program:    sw $t0, 0($sp) #store the string into memory
                add $a0, $0, $sp #put the start address of string into a0
                addi $v0, $0, 4 #syscall 4
                syscall #print the string in a0
                addi $sp, $sp, 4 #return stack pointer to original place
                exit() #syscall 10
```

The above code snippet contains the remainder of the code in main, equivalent to lines `138` to `142` of .py. It assigns the input arguments of `backtrack`, calls `backtrack`, then prints NO if `backtrack` returns `0` (or `False`) and YES otherwise.

SECTION 3: BACKTRACK FUNCTION

Section 3 discusses `backtrack`, the equivalent to the `backtrack()` function from .py. backtrack() has three input arguments: `currGrid`, `chosen`, and `pieces`. The .asm equivalents of these arguments are the starting address of an array `currgrid`, the starting address of the `chosen` array, and the starting address of an array `converted_pieces`. `backtrack` returns 1 if `currgrid` and `final_grid` are equal, and 0 otherwise.

Because `backtrack` is such a large function, it will be tackled in subsections.

Section 3.1: Memory management and unique namespaces

```
### FUNCTION START: BACKTRACK ###
backtrack:      addi $sp, $sp, -144
                sw $ra, 140($sp)
                sw $s0, 136($sp)
                sw $s1, 132($sp)
                sw $s2, 128($sp)
                sw $s3, 124($sp)
                sw $s4, 120($sp)
                sw $s5, 116($sp)
                sw $s6, 112($sp)
                sw $s7, 108($sp)
```

```
end_backtrack:  lw $ra, 140($sp)
                lw $s0, 136($sp)
                lw $s1, 132($sp)
                lw $s2, 128($sp)
                lw $s3, 124($sp)
                lw $s4, 120($sp)
                lw $s5, 116($sp)
                lw $s6, 112($sp)
                lw $s7, 108($sp)
                addi $sp, $sp, 144
                jr $ra
### FUNCTION END: BACKTRACK ###
```

The above snippets contain the memory allocation and management for backtrack. Note how `$sp` is reduced by 144, which is far larger than the 36 bytes needed to allocate for the registers. This is because of unique namespaces; that is, calling `backtrack` with some `grid` then modifying `currgrid` within the function should not modify `grid`. Thus, a copy of the input arguments should be made localized within `backtrack`. These copies will be stored in memory, and can be accessed through offsets from `$sp`.

$$60 \text{ bytes from } \texttt{currGrid} + 8 \text{ bytes from } \texttt{chosen} + 40 \text{ bytes from } \texttt{converted\_pieces} + 36 \text{ for registers} = 144 \text{ bytes}$$

```
                #firstly, need to duplicate input arguments.
                #start by duplicating curr_grid:
                add $s0, $a0, $0 #a0 stores the address of currgrid
                addi $s1, $s0, 60 #need 60 bytes
currgriddupeloop:
                lw $s2, 0($s0) #get the argument row
                sw $s2, 0($sp) #duplicate into stack memory
                addi $s0, $s0, 4 #next row
                addi $sp, $sp, 4 #next row in memory
                bne $s0, $s1, currgriddupeloop #loop through rows
                addi $s0, $s0, -60 #$s0 stores the address of currgrid
                #then duplicate chosen
                add $s1, $a1, $0 #s1 stores the address of chosen
                lw $s2, 0($s1) #get first 4 bytes of chosen
                sw $s2, 0($sp) #place first 4 bytes into stack memory
                lw $s2, 4($s1) #get next 4 bytes
                sw $s2, 4($sp) #place in memory
                addi $sp, $sp, 8 #adjust $sp for next for loop
                #then duplicate pieces
                add $s2, $a2, $0 #s2 stores the address of pieces
                addi $s3, $s2, 40 #pieces array has 40 bytes
piecesdupeloop: lw $s4, 0($s2) #load row of pieces
                sw $s4, 0($sp) #store row of pieces into memory
                addi $s2, $s2, 4 #next row
                addi $sp, $sp, 4 #next row in memory
                bne $s2, $s3, piecesdupeloop #loop through rows
                addi $s2, $s2, -40 #s2 now stores address of pieces
                addi $sp, $sp, -108 #return $sp to the bottom of the stack frame
```

The above snippet handles duplication and localization of the input arguments. This is done through loops for large arrays like `currgrid` and `pieces` and lines for a small array like `chosen`.

Section 3.2: Calling is_equal_grid

```
addi $s1, $0, 0 #result = false

#check if current grid is already right
add $a0, $s0, $0 #argument: currGrid address
jal is_final_grid #call function
bne $v0, $0, backtrack_true #if is_final_grid returns true, then end backtrack and return true
backtrack_true: addi $v0, $0, 1 #return true, end backtrack
```

The above snippets are equivalent to lines 85 to 87 of .py. The input argument of is_final_grid is placed in $a0 (note that $s0 at this point is the address of currgrid) and is_final_grid is called. If is_final_grid returns 1, then backtrack returns 1. Otherwise, processing of backtrack continues.

Section 3.3: Outer for loop and calling max_x_of_piece

```
		#if not, go crazy
		#firstly, some inventory management.
		add $s0, $sp, $0 #store $sp into s0.
		#this allows us to access currGrid, chosen, and pieces with 1 variable. 0(s0) is currgrid, 60(s0) is chosen, 68(s0) is pieces.

		la $at, 60($s0) #60(s0) is start address of chosen
		add $s3, $at, $0 #s3 is the counter for the bigloop, starts from address of first byte of chosen to address of last byte of chosen
		la $s4, chosen #load the start address of chosen
		lb $s4, 5($s4) #number of pieces
		add $s4, $s3, $s4 #break bigloop when the finished iterating through [chosen]
backtrack_bigloop:
		lb $s5, 0($s3)  #s5 is chosen[i]
		bne $s5, $0, continue_bigloop #if true, skip

		#now, need to make a copy of chosen. can allocate more memory here:
		addi $sp, $sp, -8 #8 bytes because chosen is 4 words long
		#chosencopy can be accessed through -8($s0)
		lw $at, 60($s0) #60(s0) is start address of chosen
		sw $at, 0($sp) #copy first word of chosen to memory
		lw $at, 64($s0) #get next word of chosen
		sw $at, 4($sp) #store next word, chosen now copied

		#next, solve for max_x_of_piece
		la $at, 60($s0) #60(s0) is start address of chosen
		sub $a0, $s3, $at #index of current chosen
		sll $a0, $a0, 3 #multiply by 8 because of how pieces is stored (indices are adjusted by 8)
		la $at, 68($s0) #68(s0) is start address of pieces
		add $a0, $a0, $at #argument: offset + start address of pieces
		jal get_max_x_of_piece #call function
continue_bigloop: #this continue is found prior to the allocation of chosenCopy, so deallocation should not happen
		addi $s3, $s3, 1 #go to next byte in chosen
		bne $s3, $s4, backtrack_bigloop #continue cycling through chosen
		addi $v0, $s1, 0 #result = false, a jump needs to occur for result to be true
		j end_backtrack #ends backtrack while returning false
```

The above snippets are equivalent to lines 88 to 91 and line 100 of .py. Note that as this point, the various input arguments are stored in registers $s0, $s1, and $s2. To save on register use, $sp is saved into $s0 instead, and the input arguments can be accessed through specific offsets of $s0. backtrack_bigloop represents the first for loop of backtrack(), and iterates through chosen by byte. If the current byte is 0 (for False), bigloop is continued. Otherwise, chosen is copied (equivalent to deepcopy(chosen)) through an extra allocation of memory with $sp. This memory will be deallocated at a later subsection.

Then, the input arguments of get_max_x_of_piece are managed. After placing the starting address of the relevant word in pieces, get_max_x_of_piece is called.

Note that if bigloop ends naturally then backtrack returns 0. A branch must occur for backtrack to return 1.

Section 3.4: For loop 2

```
                #start small loop
                add $s5, $0, $0 #s5 is now counter for small loop, also offset
                add $s6, $0, 6 #s6 = 6
                sub $s6, $s6, $v0 #end loop when s5 goes from 0 to 6-max_x_of_piece
backtrack_smallloop:
                add $a0, $s0, $0 #first argument: address of currgrid
                la $at, 60($s0) #60(s0) is start address of chosen
                sub $a1, $s3, $at #index of current chosen
                sll $a1, $a1, 3 #multiply by 8 because of how pieces is stored
                la $at, 68($s0) #68(s0) is start address of pieces
                add $a1, $a1, $at #second argument: start address of relevant piece
                add $a2, $s5, $0 #third argument: offset

                #note that we need to store nextgrid somewhere.
                addi $sp, $sp, -60 #allocate space for nextgrid. can be accessed through -68($s0)
                addi $a3, $sp, 0 #fourth argument: where to place nextgrid
                jal drop_piece_in_grid #call function

                beq $v1, $0, continue_smallloop #if success is false, continue
                #at this point, no longer need success.
                la $at, 60($s0) #load address of chosen
                sub $s7, $s3, $at #s7 = i
                la $at, -8($s0) #-8($s0) is the start address of chosencopy
                add $s7, $s7, $at #s7 is the address of chosencopy[i]
                addi $at, $0, 1 #store a true
                sb $at, 0($s7) #chosencopy[i] = true

                #call clearlines
                la $a0, -68($s0) #-68($s0) is the start address of nextgrid
                jal clearlines #call function

                beq $at, $s1, dealloc #if result: return true. this is placed before to function as an OR

                la $a0, -68($s0) #first argument: start address of nextgrid
                la $a1, -8($s0) #second argument: start address of chosencopy
                la $a2, 68($s0) #third argument: start address of pieces
                jal backtrack #call function
                add $s1, $v0, $0 #result = backtrack

                addi $at, $0, 1 #store a true to compare to
                beq $at, $s1, dealloc #if result: return true

continue_smallloop:
                add $s5, $s5, 1 #append to loop counter
                addi $sp, $sp, 60 #deallocate nextGrid at the end of smallloop
                bne $s5, $s6, backtrack_smallloop #next xoffset

                add $sp, $sp, 8 #deallocate chosenCopy at the end of bigloop
dealloc:        addi $sp, $sp, 68 #deallocate memory, return true, end backtrack
backtrack_true: addi $v0, $0, 1 #return true, end backtrack
```

The snippets above represent lines `92` to `99` of .py. A for loop is created with `backtrack_smallloop` which cycles from `$s5` = 0 up to 6 minus the return value of `get_max_x_of_piece` from the previous subsection. For every cycle, `drop_piece_in_grid` is called. Doing so is not as simple as just applying input arguments, as `drop_piece_in_grid` returns an array. As such, memory for the array must be allocated before the function call, as seen through `addi $sp, $sp, -60`. This memory is deallocated through `addi $sp, $sp, -60` or if needed, through a jump to dealloc which deallocates both `nextGrid` is the copy of `chosen`, then has `backtrack` return 1.

Once all arguments of drop_piece_in_grid have been assigned and the function has been called, the relevant byte of the copy of chosen (chosenCopy[i] in .py) is set to 1 (True) and the function clearlines is called. Note that clearlines does not return anything; it simply modifies the addresses themselves (akin to receiving a pointer and editing the values of the pointee).

Note that there exists a result or backtrack() in .py. This is replicated in .asm by checking the value of result (or $s1 in .asm) twice: once before the backtrack call and again after it. backtrack is called with the arguments of the return array of drop_piece_in_grid after being managed by clearlines, the copy of chosen, and converted_pieces.

At this point, the inner call of backtrack will either return a 1 or a 0. If it returns 1, then backtrack (the current function call) will also return 1 through the jump to dealloc. If the jump to dealloc is never called, then nextGrid is deallocated at the end of smallloop, and chosenCopy is deallocated right before the end of bigloop.

SECTION 4: IS_FINAL_GRID FUNCTION

This section tackles the function is_final_grid, somewhat equivalent to is_equal_grid() in .py. While is_equal_grid() takes in two arguments, the only times it is used in .py is to compare currGrid to final_grid in backtrack(), thus the second argument is removed in is_final_grid. The first argument is the starting argument of some grid to compare to final_grid. is_final_grid returns 1 if the argument grid is equal to the final_grid, and 0 otherwise.

```
### FUNCTION START: IS_FINAL_GRID ###
is_final_grid:  addi $sp, $sp, -32
                sw $ra, 0($sp)
                sw $s0, 4($sp)
                sw $s1, 8($sp)
                sw $s2, 12($sp)
                sw $s3, 16($sp)
                sw $s4, 20($sp)
                add $s0, $a0, $0 #save argument: start address of grid1
                la $s1, final_grid #final_grid is grid 2
                add $s2, $s0, 60 #60 bytes per grid
is_equal_loop:  lw $s3, 0($s0) #get a word in grid1
                lw $s4, 0($s1) #and a word in grid2
                bne $s3, $s4, is_not_equal #need to be equal. if not, end function, return false
                addi $s0, $s0, 4 #next word in grid1
                addi $s1, $s1, 4 #next word in grid2
                bne $s0, $s2, is_equal_loop #end loop when at the end of the grids
                addi $v0, $0, 1 #no jumps? return true
                j is_equal_end #end function
is_not_equal:   addi $v0, $0, 0 #jumped? return false
is_equal_end:   lw $ra, 0($sp)
                lw $s0, 4($sp)
                lw $s1, 8($sp)
                lw $s2, 12($sp)
                lw $s3, 16($sp)
                lw $s4, 20($sp)
                addi $sp, $sp, 32
                jr $ra
### FUNCTION END: IS_FINAL_GRID ###
```

After the preamble, `is_final_grid` words by getting the address of both `grid` and `final_grid` and iterating through both arrays word-by-word, checking equality. If at any point the words are not equal, the for loop breaks and `is_final_grid` returns 0. Otherwise, `is_final_grid` returns 1.

SECTION 5: GET_MAX_X_OF_PIECE FUNCTION

This section tackles the `get_max_x_of_piece` function, equivalent to `get_max_x_of_piece()`. The argument of `get_max_x_of_piece` is the starting address of a specific word in `converted_pieces`. The function returns some `max_x`, which is the maximum x coordinate of the blocks of a piece.

```
### FUNCTION START: GET_MAX_X_OF_PIECE ###
get_max_x_of_piece:
                addi $sp, $sp, -32
                sw $ra, 0($sp)
                sw $s0, 4($sp)
                sw $s1, 8($sp)
                sw $s2, 12($sp)
                add $s0, $a0, $0 #save argument: start address of piece
                addi $v0, $0, -1 #max_x = -1
                addi $s1, $s0, 8 #always 4 blocks in a piece, always 2 coords per block, total 8 coords, break loop at end of coords
get_max_loop:   lb $s2, 1($s0) #load the second coordinate of the piece
                ble $s2, $v0, get_max_next #if second coordinate <= max_x, do nothing
                add $v0, $s2, $0 #if s2 > v0, change max_x to second coordinate
get_max_next:   addi $s0, $s0, 2 #move to next block in piece
                bne $s0, $s1, get_max_loop #itearte until at last block of piece
                lw $ra, 0($sp)
                lw $s0, 4($sp)
                lw $s1, 8($sp)
                lw $s2, 12($sp)
                addi $sp, $sp, 32
                jr $ra
### FUNCTION END: GET_MAX_X_OF_PIECE ###
```

A loop is used to iterate through the values of the argument, and if the current value is more than `$v0`, `$v0` is updated to that value. Note that `max_x` is set to `$v0` from the start; this is doable because there are no function calls within the function body, so `$v0` will not be modified by a function outside of `get_max_x_of_piece`.

SECTION 6: DROP_PIECE_IN_GRID FUNCTION

This section tackles `drop_piece_in_grid`, equivalent to `drop_piece_in_grid()`. There are 4 input arguments: the first is the address of some `grid`, the second is `piece` (the start address of a specific word in `converted_pieces`), the third is some offset along the grid columns `offset`, and the fourth is a pointer to an area in memory reserved for `nextgrid`. `drop_piece_in_grid` modifies the memory pointed to by the fourth argument (`nextgrid`) and returns the start address of `nextgrid` as well as some Boolean set to 1 if `nextgrid` fits a 6x6 grid, and 0 otherwise.

Section 6.1: Memory management

```
### FUNCTION START: DROP_PIECE_IN_GRID ###
#first argument: address of currgrid
#second argument: start address of relevant piece
#third argument: offset
drop_piece_in_grid:
                addi $sp, $sp, -104 #need to create a copy of the grid and piece, so minimum 60 + 8 = 68 bytes in memory
                #also use 8 + 1 registers, so store another 36 bytes
                #totalling 104
                sw $ra, 100($sp)
                sw $s0, 96($sp)
                sw $s1, 92($sp)
                sw $s2, 88($sp)
                sw $s3, 84($sp)
                sw $s4, 80($sp)
                sw $s5, 76($sp)
                sw $s6, 72($sp)
                sw $s7, 68($sp)

                #first, duplicate grid
                add $s0, $a0, $0 #s0 stores the address of grid
                add $s3, $a3, $0 #s3 stores the address of nextgrid. by default, set nextgrid to be grid
                addi $s1, $s0, 60 #60 bytes in a grid
griddupeloop:   lw $s2, 0($s0) #load a word from grid
                sw $s2, 0($s3) #store that word into nextgrid (nextgrid starts as grid)
                sw $s2, 0($sp) #also store that word into memory (deepcopy of grid)
                addi $s0, $s0, 4 #next word in grid
                addi $s3, $s3, 4 #next word in nextgrid
                addi $sp, $sp, 4 #next word in deepcopy(grid)
                bne $s0, $s1, griddupeloop #iterate through bytes in grid
                addi $s0, $sp, -60 #s0 now stores the address of gridcopy

                #next, duplicate piece. piece is 2 words long
                add $s1, $a1, $0 #s1 stores the address of piece
                lw $s2, 0($s1) #load first word of piece
                sw $s2, 0($sp) #store in memory
                lw $s2, 4($s1) #second word loaded
                sw $s2, 4($sp) #store in memory
                addi $sp, $sp, -60 #return $sp to start of gridcopy
end_drop_piece: lw $ra, 100($sp)
                lw $s0, 96($sp)
                lw $s1, 92($sp)
                lw $s2, 88($sp)
                lw $s3, 84($sp)
                lw $s4, 80($sp)
                lw $s5, 76($sp)
                lw $s6, 72($sp)
                lw $s7, 68($sp)
                addi $sp, $sp, 104
                jr $ra
### FUNCTION END: DROP_PIECE_IN_GRID ###
```

104 bytes are allocated to every call of `drop_piece_in_grid`. This is because of `deepcopy(grid)`, as well as the creation of a unique namespace for the input array `piece`.

$$60 \text{ bytes for } \texttt{gridCopy} + 8 \text{ bytes from } \texttt{piece} + 36 \text{ for registers} = 104 \text{ bytes}$$

The above snippets include the preamble, register restoration, as well as routines for copying `grid` to create `gridCopy`, as well as creating a local copy of `piece`. `grid` is copied by a for loop cycling from the first word of `grid` until the last word, copying each word into an allocated space in memory. The same is done with `piece`, however it is duplicated by immediate offset instead of through a loop as it only has a length of two words.

Section 6.2: Function initialization

```
                addi $s2, $0, 100 #maxY = 100
                add $s3, $s1, $0 #counter for blocks in piece
                add $s4, $s3, 8 #end loop when iterated through all blocks in piece
drop_block_loop: lb $s5, 0($s3) #s5 = block[0]
                addi $s6, $0, 6 #store 6
                mult $s5, $s6 #block[0] * 6
                mflo $s5 #s5 = 6 * block[0]
                lb $s6, 1($s3) #s6 = block[1]
                add $s5, $s5, $s6 #s5 = 6*block[0] + block[1]
                add $s5, $s5, $a2 #s5 = 6*block[0] + block[1] + yoffset
                add $s5, $s0, $s5 #s5 is now the address of the associated piece to put in
                addi $s6, $0, 0x23 #ASCII code for #
                sb $s6, 0($s5) #current piece is now a #
                add $s3, $s3, 2 #append loop counter; move to next block
                bne $s3, $s4, drop_block_loop #not yet last block in piece? continue loop
```

The above snippet is equivalent to lines `33` to `35` of .py. After setting the register `$s2` to contain the value of `maxY`, a loop is used to cycle through every block in `piece` and place that block in `gridCopy`. Note that indices are translated from 2D to 1D through the calculation discussed in Section 1.

Section 6.3: The while() loop

```
drop_while_loop:addi $s3, $0, 1 #canStillGoDown = True
                add $s4, $s0, $0 #save the start address of gridcopy, this is a loop counter
                add $s5, $s4, 60 #iterate through all 60 bytes of gridcopy
goDown_check_loop:
                lb $s7, 0($s4) #s7 is gridcopy[i][j]
                addi $s6, $0, 0x23 #ASCII code for #
                bne $s7, $s6, continue_check_loop #if gridcopy[i][j] != #: continue the loop
                sub $s6, $s4, $s0 #current index
                addi $s7, $0, 6 #store 6
                div $s6, $s7 #index/6
                mflo $s6 #s6 = index/6, which is i
                mfhi $at #at = index%6, which is j
                addi $s6, $s6, 1 #s6 = i + 1
                addi $s7, $0, 10 #store 10
                beq $s6, $s7, goDown_false #at this point, gridcopy[i][j] == '#' and i+1 == 10, set to false
                addi $s7, $0, 6 #s7 = 6
                mult $s6, $s7 #[i+1] * 6
                mflo $s6 #s6 = [i+1] * 6
                add $s6, $s6, $at #s6 = (i+1)*6 + j, which is the index
                add $s6, $s6, $s0 #address of gridCopy[i+1][j]
                lb $s6, 0($s6) #s6 = gridcopy[i+1][j]
                addi $s7, $0, 0x58 #stores the ascii code for X
                beq $s6, $s7, goDown_false #at this point, gridcopy[i][j] == '#' and gridcopy[i+1][j] == 'X', set to false
                j continue_check_loop #coninue loop
goDown_false:   addi $s3, $0, 0 #set canStillGoDown to false
continue_check_loop:
                add $s4, $s4, 1 #next byte in gridcopy
                bne $s4, $s5, goDown_check_loop #continue loop if not done with iterating through gridcopy
```

```
                beq $s3, $0, drop_while_end #if canStillGoDown is False, break
                addi $s3, $s3, 53 #s3 = 8*6 + 5 = 53, which is the index for [8,5]
                add $s3, $s3, $s0 #s3 is now the address for gridcopy[8,5]
                addi $s4, $s0, -1 #stop after reaching index -1
move_down_loop: lb $s5, 0($s3) #gridcopy[i][j]
                addi $s6, $0, 0x23 #stores the ASCII code for #
                bne $s5, $s6, continue_move_down
                addi $s6, $0, 0x2E #stores the ASCII code for .
                sb $s6, 0($s3) #gridcopy[i][j] = '.'
                sub $s5, $s3, $s0 #s5 = index
                addi $s6, $0, 6 #store a 6
                div $s5, $s6 #index/6
                mflo $s5 #s5 = index/6, which is i
                mfhi $s7 #s7 = index%6, which is j
                addi $s5, $s5, 1 #s5 = i + 1
                mult $s5, $s6 #(i+1)*6
                mflo $s5 #s5 = (i+1)*6
                add $s5, $s5, $s7 #s5 = (i+1)*6 + j, which is the index
                add $s5, $s0, $s5 #s5 is now the address of gridcopy[i+1][j]
                addi $s6, $0, 0x23 #stores the ASCII code for '#'
                sb $s6, 0($s5) #gridcopy[i+1][j] = '#'
continue_move_down:
                addi $s3, $s3, -1 #go to the left and up gridwise
                bne $s3, $s4, move_down_loop #not done moving down rows? continue
                j drop_while_loop #continue while(true), a branch will break it if needed
drop_while_end:
```

The two long snippets above represent lines `37` to `56` of .py. First, `$s3` is set to hold the value for `canStillGoDown`. Next, a for loop is created which iterates through all bytes found in `grid`. A long conditional then takes place, which starts with the current byte being checked against '#' (which is `bne $s7, $s6, continue_check_loop`). If that is true, the first index of the index *i* (note that steps are taken to translate the single index into a double-index format) is checked against a register which contains the integer 10 (which is `beq $s6, $s7, goDown_false`). Because the second part of the conditional is an `or` statement, if the conditional fails there is still another opportunity for a branch to `goDown_false` to occur: if the bottom byte of the current byte is equal to 'X' (which is `beq $s6, $s7, goDown_false`). These checks are performed for every byte, where a jump to `goDown_false` sets `$s3` to 0.

The while loop is broken if `$s3` is 0 (as seen in `beq $s3, $0, drop_while_end`). If the while loop is not broken, then another for loop is created which cycles from the last byte of `grid` to the byte representing the start of the 5th row, and checks for blocks (or when the byte contains an ASCII '#'). If a byte is a block, then it is moved down by one, and the loop continues. Otherwise, the loop just continues.

Note that the while loop ends with a `j` rather than with a `bne`. This is because the conditional is `while(true)`, and so a branch out of the loop has to occur for the loop to stop. Such a branch occurs when `$s3` becomes 0 (as seen through `beq $s3, $0, drop_while_end`).

Section 6.4: Returning values

```
                add $s3, $s0, $0 #store the start address of gridcopy
                addi $s4, $s0, 60 #iterate through all of gridcopy
                addi $s6, $0, 0x23 #stores the ASCII code for #
maxY_loop:      lb $s5, 0($s3) #s5 = gridcopy[i][j]
                bne $s5, $s6, continue_maxY #continue if gridcopy[i][j] != '#'
                sub $s5, $s3, $s0 #s5 = index
                addi $s6, $0, 6 #store 6
                div $s5, $s6 #index/6
                mflo $s5 #s5 = index/6, which is i
                bge $s5, $s2, continue_maxY #do nothing if i >= maxY
                add $s2, $s5, $0 #if i < maxY, maxY = i
continue_maxY:  addi $s3, $s3, 1 #next byte
                bne $s3, $s4, maxY_loop #continue until done with gridcopy

                addi $s3, $0, 3 #store a 3
                ble $s2, $s3, drop_piece_fail #if max<=3, branch to end
                addi $a0, $s0, 0 #first argument: start address of gridcopy
                jal freeze_blocks #call function
                addi $v1, $0, 1 #v1 to true

                #need to copy $v0 to nextgrid.
                add $s0, $a3, $0 #start address of nextgrid, which is an argument
                add $s1, $v0, $0 #v0 is the address of freeze_blocks(gridCopy)
                addi $s2, $a3, 60 #scroll through all of nextgrid
copy_nextgrid:  lw $at, 0($s1) #get a word in freeze_blocks(gridCopy)
                sw $at, 0($s0) #place as appropriate nextgrid word
                addi $s0, $s0, 4 #next word in freeze_blocks(gridcopy)
                addi $s1, $s1, 4 #next word in nextgrid
                bne $s0, $s2, copy_nextgrid #not done copying? continue
                j end_drop_piece #end function
drop_piece_fail:addi $v1, $0, 0 #return false, end function
```

The above snippet is equivalent to lines 53 to 61 of .py. First, a for loop is created which iterates through the entirety of `gridCopy`. The lowest value of i (noting again the 1D index to 2D index conversion) is found, and is set to `$s2`, the register representing `maxY`. If `$s2` ≤ 3 then the function simply returns a 0 (note that at this point the values pointing to `nextGrid` are set to `grid` by default, which is the behavior described in .py). Otherwise, the function uses a for loop to copy `gridCopy` to `nextGrid` (as prior to copying `nextGrid` is still just `grid`) then calls the function `freeze_blocks` on the address for `nextGrid`. The start address of `nextGrid` is set as the first return value, and the second return value is set to 1.

SECTION 7: FREEZE_BLOCKS FUNCTION

This section tackles the `freeze_blocks` function, equivalent to `freeze_blocks()`. The argument of `freeze_blocks` is the starting address of some `grid`. `freeze_blocks` returns the start address of the now-frozen grid.

```
### FUNCTION START: FREEZE_BLOCKS ###
freeze_blocks:  addi $sp, $sp, -32
                sw $ra, 0($sp)
                sw $s0, 4($sp)
                sw $s1, 8($sp)
                sw $s2, 12($sp)
                sw $s3, 16($sp)
                addi $s0, $a0, 24 #only need to freeze 36 bottom blocks
                addi $s1, $s0, 60 #until the last block
freeze_loop:    addi $s2, $0, 0x23 #stores the ASCII code for '#'
                lb $s3, 0($s0) #load a byte from the grid
                bne $s2, $s3, dont_freeze #if byte != '#', don't freeze it
                addi $s2, $0, 0x58 #stores the ASCII code for 'X'
                sb $s2, 0($s0) #store an 'X' in place of '#'
dont_freeze:    addi $s0, $s0, 1 #next byte
                bne $s0, $s1, freeze_loop #not done freezing? continue the loop
                add $v0, $a0, $0 #return the start address of the grid
                lw $ra, 0($sp)
                lw $s0, 4($sp)
                lw $s1, 8($sp)
                lw $s2, 12($sp)
                lw $s3, 16($sp)
                addi $sp, $sp, 32
                jr $ra
### FUNCTION END: FREEZE_BLOCKS ###
```

After the preamble, the function iterates from the first byte of the fourth row of `grid` and iterates through `grid` byte-by-byte. If the current byte is a '#', it gets replaced with a 'X'.

SECTION 8: CLEARLINES FUNCTION

This section tackles the bonus function, `clearlines`, equivalent to `clearlines()`. The argument of `clearlines` is the starting address for some `grid` to clear. `clearlines` does not return anything.

```
### FUNCTION START: CLEARLINES ###
clearlines:     addi $sp, $sp, -32
                sw $ra, 0($sp)
                sw $s0, 4($sp)
                sw $s1, 8($sp)
                sw $s2, 12($sp)
                sw $s3, 16($sp)
                sw $s4, 20($sp)
                add $s0, $a0, $0 #s0 stores the start address of the grid
                addi $s0, $s0, 59 #move to the last byte
                addi $s1, $a0, 23 #stop when reaching 24($s0), which is the last byte of the 4th row.
clearlines_loop:addi $s2, $s0, -6 #start a for loop to check the row byte by byte if all are x
                addi $s3, $0, 1 #all X? assume true
                addi $at, $0, 0x58 #ASCII code for 'X'
clearlines_check_loop:
                lb $s4, 0($s0) #load a byte for the row
                beq $s4, $at, clearlines_stillX #keep allX true as long as the current char is still X
                add $s3, $0, $0 #all X? now false, didn't branch
clearlines_stillX:
                addi $s0, $s0, -1 #move to the previous byte (iterating down remember)
                bne $s0, $s2, clearlines_check_loop
                #at this point, done checking the current row.

                beq $s3, $0, continue_clearlines_loop #if allX is false, continue the loop, or move to the next row
                #otherwise, go crazy.
                #need a new loop to iterate through every row
                add $s2, $s0, $0 #save s0
                addi $s0, $s0, 6 #return $s0 to the start of the current row
                addi $s3, $a0, 23 #stop when reaching 24($s0), which is the last byte of the 4th row.
move_down_rows: lb $at, -6($s0) #get the byte above the current byte
                sb $at, 0($s0) #replace the current byte with the byte above it
                addi $s0, $s0, -1 #previous byte
                bne $s0, $s3, move_down_rows
                addi $s0, $s2, 6 #restore s0 back to the start of current row. this allows checking for consecutive rows
continue_clearlines_loop:
                bne $s0, $s1, clearlines_loop #not done with all the lines in the grid? continue
                lw $ra, 0($sp)
                lw $s0, 4($sp)
                lw $s1, 8($sp)
                lw $s2, 12($sp)
                lw $s3, 16($sp)
                lw $s4, 20($sp)
                addi $sp, $sp, 32
                jr $ra
### FUNCTION END: CLEARLINES ###
```

After the preamble, two loops are created. There is one big loop which cycles from the last byte of the grid, byte 59, until the first byte of the 4th row, byte 24, going downwards in byte number. Another for loop is created to compare the bytes of a row (every 6 bytes) byte-by-byte to the ASCII code of 'X'. If any of the 6 bytes are not 'X', the big loop continues. Otherwise, the row must be cleared.

The row is cleared by another small for loop, this time iterating from the first byte of the row to be cleared until the first byte of the 4th row. Because of the way `grid` is stored, accessing the byte above a byte in the grid is as easy as accessing the address with an offset -6 away from the current byte address. The bytes above a byte are moved down for every byte. Note that the 3rd row is duplicated and becomes the 4th row, and because the 3rd row is always blank, this means that no extra steps are needed to adjust `grid`.

While `clearlines()` does comparisons row-wise and moves rows row-wise, `clearlines` compares byte-wise and moves rows by byte. This is intentional, as a row is 6 bytes long, and storing 6 bytes on word sizes of 4 is awkward to code for in MIPS.

SECTION 9: AUXILLARY MARS DEFINITIONS

This section discusses the various MARS-specific functions used in the submission.

Section 9.1: Syscall Macros

```
.macro input_grid_row(%address)         #syscall code 8, receives a string.
        add $a0, %address, $0   #%address is a register containing the memory for the row
        li $a1 7                #receive specifically 6 characters, or one row.
        li $v0 8
        syscall
.end_macro
.macro input_piece_row(%address)        #syscall code 8, receives a string.
        add $a0, %address, $0   #%address is a register containing the memory for the row
        li $a1 5                #receive specifically 4 characters, or one piece row.
        li $v0 8
        syscall
.end_macro
.macro input_int(%register)     #syscall code 5, receives an integer to place in %register
        li $v0 5
        syscall
        add %register, $v0, $0
.end_macro
.macro exit()                   #syscall code 10
        li $v0 10
        syscall
.end_macro
```

The above snippet shows all the macros used in the submission. Macros are used whenever `syscall` is called.

Section 9.2: .data segment

```
start_grid: #need at least 60 bytes for the start grid. 60 bytes is 15 words minimum
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E


final_grid:
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
.word 0x2E2E2E2E
```

```
#note that input pieces are a maximum of 5, so we can just allocate the max memory already
#chosen will require at most 5 bytes, so 2 words:
chosen:
.word 0x00000000
.word 0x00000000

#every piece contains 4 blocks. each array contains 4*2 = 8 bytes.
#maximum of 5 pieces, so 5*8 = 40 bytes, or 10 words.
converted_pieces:
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
```

The above snippets show the `.data` segment of the submission. These have been previously explained under Section 1.

FOREWORD AND ENDING THOUGHTS

The submission can still be optimized. Easy suggestions include adding loop breaks (for example in `drop_piece_in_grid` there is an easy loop break) and removing unnecessary arguments (for example, `backtrack` is only ever called with `converted_pieces` as an argument, and thus `converted_pieces` can be turned into a global variable and no longer turned into an argument. However, as stated, there is no practical reason to pursue better metrics as the specifications have no minimum, thus implementing further optimizations remains outside the scope of this submission.

This documentation is written in (mostly) third person. This paragraph will be written in first person. About halfway through writing the various sections above, I realized that it may seem like I did not write the associated code. For clarity: I refer to my implementation as ".asm implements"; please do not be fooled, I am not describing self-writing code or someone else's implementation of .asm, I just dislike writing papers in first person. You can double-check this through my prior project submissions—they are also in third person.