CS 21 PROJECT 1
TETRISITO

Submitted by: Joshua L. Felipe
Section: CS21 Lab3

Introduction

This project entitled “Tetrisito” is a mini version of Tetris wherein the program must output “YES” if we can achieve the desired grid by using all or some of the pieces that will be inputted. Otherwise, it must output “NO”. It is important to note that the starting grid may begin as empty or with some pieces. The project must be implemented using MIPS Assembly Language and must pass specific implementation types. For implementation A, only one piece would be dropped to the initial grid.and must be able to determine if the the initial and final grids are equal. For implementation B, the program would be given 3 to 5 pieces and would be dropped in order. Lastly, for implementation C there are gonna be 3 to 6 pieces that would be dropped wherein order would not matter. Hence, to check if the grid is possible we must make use of recursion to check if the current state is equal to the final state, else, it should either move to the next state or previous state depending on the base case/s.

Implementation

Prior to implementing Tetrisito, a Python code was provided as a guide on how to implement the game in MIPS. It included several functions that would be discussed in detail in the next sections. The main function included several macros that served as function calls such as init_arr, get_input, read_file, init_chosen, get_pieces and backtrack. Code blocks for the MIPS and Python code would be provided as supplementary to the explanation. Also, for each macro being implemented, store and load words are immediately added to avoid any loss of data originally written in the registers.

As part of the implementation, some registers were marked as “untouchables” and should not be modified as much as possible as these registers would hold the original values or base addresses of the grids or array inputs. These registers and their corresponding values are listed below:

1. S0: base address of start_grid
2. S1: base address of final_grid
3. S2: number of pieces inputted by the user
4. S3: base address of chosen
5. S4: base address of pieces.

init_arr(address)

The first step in building tetrisito is to initialize the arrays wherein the pieces would be stored. Since MIPS does not have array functionalities, the program made use of the *.data* to store the

values of the pieces or dots. The program allocated 96 bytes in the data segment to store a 10x6 grid. It is important to note that an additional 2 columns would be stored also in the data segment, this would be discussed later, hence we would have 10x8 grid in total. The address of the empty grid would be passed to the macro init_arr wherein it would setup a 4x6 grid and place '.' to each byte. Additionally, it would add '\r\n' to each row to ensure word alignment and reduce the possibility of encountering issues once user inputs are added. This macro is equivalent to void init_arr(address) wherein we only update the values in the address.

Code Block 1: init_arr() implementation

```
      # ACTS AS VOID INIT_ARR(ADDRESS):
      # THIS MACRO INITIALIZES AN ARRAY OF 4 ROWS WITH 6 COLUMNS
      # IMPLEMENTATION: 4 ROWS, 8 COLUMNS TO CONSIDER THE \n\r VALUES
      # ONCE INPUTS ARE ADDED TO GRIDS.
      # THIS UPDATES THE ADDRESS VALUE OF %P

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)
      sw    $t4, 12($sp)

      # INITIALIZATION OF VARIABLES
      li    $t0, 0                        # r
      li    $t1, 0                        # c
      li    $t2, 6                        # cols
      li    $t3, 4                        # rows
      li    $t4, 0x2e               # ASCII '.'
init_loop:
      beq   $t0, $t3, end_init_loop       # r < rows
      beq   $t1, $t2, end_inner_init_loop # c < cols
      sb    $t4, 0(%p)              # store '.' into memory
      addi  %p, %p, 1               # increase value of base address
      addi  $t1, $t1, 1             # increment counter
      j     init_loop

end_inner_init_loop:
      li    $t5, 0x0d
      sb    $t5, 0(%p)              # store '\r' into memory
      addi  %p, %p, 1               # increase value of base address
```

```
	li	$t5, 0x0a
	sb	$t5, 0(%p)		# store '\n' into memory
	addi	%p, %p, 1		# increase value of base address

	li	$t1, 0			# set c = 0
	addi	$t0, $t0, 1		# r++
	j	init_loop

end_init_loop:
	##### POSTAMBLE #####
	lw	$t0, 28($sp)
	lw	$t1, 24($sp)
	lw	$t2, 20($sp)
	lw	$t3, 16($sp)
	lw	$t4, 12($sp)
	addi	$sp, $sp, 32
```

read_file(bytes, address)

This macro only reads the file inputs and adds the inputs to the array/address being passed onto. File input makes use of syscall 14, wherein we pass the buffer_addresss wherein we store the values, and the number of characters we will read from the file.

Code Block 2: read_file() implementation

```
	# THIS MACRO READS THE FILE INPUTS
	# AND RETURNS THE BASE ADDRESS

	li	$a0, 0			#  file  descriptor  for  stdin,
uncomment for stdin
	move	$a1, %add		# buffer address
	addi	$a2, $0 %bytes		# number of bytes to take

	addi	$v0, $0, 14		# read from file
	syscall
```

get_input(address)

This macro is made in partner with init_arr() and read_file(). This macro reads the input file that contains the user's start, final grids, number of pieces, and the pieces to be dropped. This macro acts as a void get_input(address) function as it only updates the values in the address. It

works by taking a file input and reading the first N characters in the file then attaching it to the end of the original grid (being pointed by address). After placing the inputs to the address it will now manipulate the elements. Part of get_input() is that it should convert all '#' that it reads from the inputs to 'X'. This means that the blocks are frozen and should not be moved or overwritten.

Code Block 3: get_input() implementation

```
      # ACTS AS VOID GET_INPUT(ADDRESS)
      # THIS MACRO TAKES IN A FILE INPUT AND TAKES THE FIRST N (HARDCODED
VALUE) CHARACTERS
      # THEN IT CONNECTS IT TO THE END OF THE ORIGINAL GRID
      # UPDATES THE GRID AND ADDRESS OF %P

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)
      sw    $t4, 12($sp)

read_lines:
      move  $t0, %p                     # get buffer address
      read_file(48, $t0)                #  reads  48  bytes  from  the  input
and saves it to the buffer address
      li    $t2, 0                      # counter = 0
      li    $t1, 48                     # numCols = 8
loop:
      li    $t4, 0x23               # t7 = '#'
      beq   $t2, $t1, end_loop            # counter < numCols
      lb    $t3, 0($t0)             # rows[counter] == '#' ?
      bne   $t3, $t4, go_back
      li    $t4, 0x58               # True: rows[counter] = 'X'
      sb    $t4, 0($t0)
go_back:
      addi  $t0, $t0, 1             # t0++
      addi  $t2, $t2, 1             # t2++
      j     loop

end_loop:
      ##### POSTAMBLE #####
      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
```

```
    lw    $t2, 20($sp)
    lw    $t3, 16($sp)
    lw    $t4, 12($sp)
    addi  $sp, $sp, 32
```

---

init_chosen(len, arr)

This macro initializes an empty array to an arrow of boolean values. The boolean values are initialized to 0, which correspond to False. This macro is equivalent to void init_chosen(len, address) wherein it only updates the values stored in the address and the value of len correspond to the number of pieces inputted by the user.

Code Block 4: init_chosen() implementation

```
    # ACTS AS VOID INIT_CHOSEN(LEN, ADDRESS)
    # THIS UPDATES AN EMPTY ARRAY TO AN ARRAY OF BOOLEANS.
    # BOOLEANS ARE INITIALIZED TO 0 (FALSE).
    # chosen = [False for _ in range(numPieces)]

    ##### PREAMBLE #####
    subi  $sp, $sp, 32
    sw    $t0, 28($sp)
    sw    $t1, 24($sp)
    sw    $t2, 20($sp)

    # tracks which piece has been used
    li    $t1, 0                        # initialize i
    li    $t2, 0                        # set value to False
loop:
    beq   %len, $t1, end                # if i == len
    sb    $t2, 0(%arr)                  # chosen[i] = False
    addi  %arr, %arr, 1
    addi  $t1, $t1, 1             # i++
    j     loop
end:
    ##### POSTAMBLE #####
    lw    $t0, 28($sp)
    lw    $t1, 24($sp)
    lw    $t2, 20($sp)
    addi  $sp, $sp, 32
```

---

get_pieces(len, address)

This macro similar to get_input() takes in user inputs, however, it takes the pieces that will be added to the current grid. It reads through the file using the read_file() macro then adds it to the pieceAscii array that is initialized inside the function. Afterwards, we convert the pieces to coordinates using the convert_pieces_to_pairs(), more on this later. Next, we store these array of coordinates to converted_pieces array that was provided as an argument. The result of this function is an array of array of coordinates. This macro is equivalent to void get_pieces(len, array) wherein it just updates the array.

Code Block 5: get_pieces() implementation

```
# ACTS AS VOID GET_PIECES(LEN, ADDRESS)
      # This code asks for user to input the pieces and returns
      # an array containing the piecePairs.
      # piecePairs is an array of coordinates

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)

      move  $t0, %arr
      li    $t1, 0                        # initialize i = 0
      la    $t2, pieceAscii               # pieceAscii = []

outer_loop:
      beq   $t1, %len, end_outer_loop

      # row = [character for character in line]
      move  $t3, $t2                # get buffer address
      read_file(24, $t3)                  # reads 24 bytes from the input
and saves it to the buffer address

      convert_pieces_to_pairs($t2, $t1)   #           piecePairs           =
convert_piece_to_pairs(pieceAscii)
      addi  $t1, $t1, 1
      sw    $v0, 0(%arr)                  #
converted_pieces.append(piecePairs)
      addi  %arr, %arr, 4                 # move to next piece
      addi  $t3, $t3, 4             # increment buffer address
```

```
	j	outer_loop

end_outer_loop:
	##### POSTAMBLE #####
	lw	$t0, 28($sp)
	lw	$t1, 24($sp)
	lw	$t2, 20($sp)
	lw	$t2, 16($sp)
	addi	$sp, $sp, 32
```

---

freeze_blocks(grid)

This macro converts all '#' in the grid to 'X'. An 'X' means that the block is frozen and cannot be moved. This ensures that when a piece is dropped onto the current grid, the initial pieces placed or dropped would not be overwritten. This function acts as a char freeze_blocks(grid) which returns the grid address. It works by iterating through the grid and checking if the piece is a '#', if so, convert it to an 'X'.

Code Block 6: freeze_blocks() implementation

```
# ACTS AS CHAR FREEZE_BLOCKS(GRID)
	# THIS ITERATES THROUGHT THE GRID AND CONVERTS EVERY '#' TO AN 'X'
	# AN 'X' MEANS THAT THE BLOCK IS FROZEN AND CANNOT BE MOVED
	# RETURNS THE GRID ADDRESS

	##### PREAMBLE #####
	subi	$sp, $sp, 32
	sw	$t0, 28($sp)
	sw	$t1, 24($sp)
	sw	$t2, 20($sp)
	sw	$t3, 16($sp)
	sw	$t4, 12($sp)
	sw	$t5, 8($sp)
	sw	$t6, 4($sp)

	# INITIALIZATION OF VARIABLES
	move	$t6, %grid		# only t6 should be updated. v0 and
grid are untouchables
	li	$t0, 0			# initialize i
	li	$t1, 0			# initialize j
	li	$t2, 10			# end of i
```

```
      li    $t3, 6                      # end of j
      li    $t5, 0x58               # HEX OF 'X'
loop1:
      beq   $t0, $t2, end_loop          # for i in range(6 + 4):
loop2:
      beq   $t1, $t3, end_loop2         # for j in range(6):
      lb    $t4, 0($t6)
      bne   $t4, 0x23, increment        # if grid[i][j] == '#':
      sb    $t5, 0($t6)             # grid[i][j] = 'X'
increment:
      addi  $t1, $t1, 1
      addi  $t6, $t6, 1
      j     loop2                   # back to j
end_loop2:
      addi  $t6, $t6, 2
      addi  $t0, $t0, 1
      li    $t1, 0                      # reset j = 0
      j     loop1                   # back to i

end_loop:
      move  $v0, %grid              # Returns grid

      ##### POSTAMBLE #####
      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
      lw    $t2, 20($sp)
      lw    $t3, 16($sp)
      lw    $t4, 12($sp)
      lw    $t5, 8($sp)
      lw    $t6, 4($sp)
      addi  $sp, $sp, 32
```

is_equal_grids(start, final)

This macro returns a boolean value if the start and final grids are equal. Since MIPS can compare words, this macro makes use of that functionality. It iterates through the grids and compares if the word of each grid are equal. If at least one is not equal to the other, it should immediately return false. Otherwise, continue until it reaches the end. Initially, the return value is set to true. This macro is equivalent to saying bool is_equal_grid(start, final).

Code Block 7: is_equal_grid() implementation

```
# ACTS AS BOOL IS_EQUAL_GRIDS(START, FINAL)
# COMPARES WORD BY WORD THE VALUES OF START AND FINAL
# RETURNS A BOOL IF THE GRIDS ARE EQUAL
##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)
      sw    $t4, 12($sp)

      move  $t0, %start
      move  $t1, %final

      li    $t2, 0                      # initialize i
      li    $v0, 1                      # return true

loop:
      beq   $t2, 20, end_loop       # checks if i reaches end of grid
      lw    $t3, 0($t0)             # takes start[i]
      lw    $t4, 0($t1)             # takes final[i]
      bne   $t3, $t4, false               # if false, return 0
      addi  $t0, $t0, 4             # update address of start
      addi  $t1, $t1, 4             # update address of final
      addi  $t2, $t2, 1             # i++
      j     loop
false:      li    $v0, 0                      # return false
end_loop:
      ##### POSTAMBLE #####
      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
      lw    $t2, 20($sp)
      lw    $t3, 16($sp)
      lw    $t4, 12($sp)
      addi  $sp, $sp, 32
```

---

not_fit(start, final)

To reduce the number of iterations for backtracking, this macro returns a boolean value if the pieces being dropped does not fit with the final configuration of the grid. This function iterates through the pieces in each grid and checks if the piece in start does not pair with the piece in final grid. This function checks if start[i][j] == ‘X’ and end[i][j] != ‘X’, if this is true it means the

state does not fit and continue to next offset instead of dropping the next piece immediately. A sample python code is provided as reference.

Code Block 8: python implementation of not_fit()

```
def not_fit(gridOne, gridTwo):
    for i in range(6 + 4):
        for j in range(6):
            if (gridOne[i][j] == 'X' and gridTwo[i][j] != 'X'):
                return True
    return False
```

Code Block 9: not_fit() implementation

```
# ACTS AS BOOL NOT_FIT(START, FINAL)
# COMPARES BYTE BY BYTE THE VALUES OF START AND FINAL
# CHECKS IF THE PIECE IN START DOES NOT PAIR/FIT WITH THE PIECE IN FINAL GRID
# RETURNS A BOOL IF THE POSITION/TYPE OF THE PIECES ARE EQUAL

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)
      sw    $t4, 12($sp)

      move  $t0, %start
      move  $t1, %final

      li    $t2, 0                      # initialize i
      li    $v0, 0                      # return false

loop:
      beq   $t2, 80, end_loop       # checks if it reaches the end of grid
      lb    $t3, 0($t0)             # takes start[i][j]
      bne   $t3, 'X', go_back       # start[i][j] == 'X'
      lb    $t4, 0($t1)             # takes final[i][j]
      bne   $t4, 'X', true                # final[i][j] != 'X'
go_back:
```

```
        addi  $t0, $t0, 1
        addi  $t1, $t1, 1
        addi  $t2, $t2, 1
        j     loop
true:   li    $v0, 1                    # return true
end_loop:
        ##### POSTAMBLE #####
        lw    $t0, 28($sp)
        lw    $t1, 24($sp)
        lw    $t2, 20($sp)
        lw    $t3, 16($sp)
        lw    $t4, 12($sp)
        addi  $sp, $sp, 32
```

---

convert_pieces_to_pairs(address, increment)

This macro returns an array of tuples of int coordinates where '#' can be found. This works by iterating through the array address and checking if the piece being read is a '#' or not. If it is it must calculate the y and x coordinates of the piece using the for loop counters i and j. These coordinates are being stored to a word wherein the byte 0 stores the y coordinate and byte 2 stores the x coordinate, then the addresses are being updated for the next element. To reduce memory space, we skip checking the \r\n elements. Getting the coordinates are useful when we begin adding and dropping the pieces in our current grid.

Code Block 10: convert_pieces_to_pairs() implementation

```
# ACTS AS INT PIECE_TO_PAIRS(ADDRESS, I)
# RETURNS AN ARRAY OF TUPLE OF INT COORDINATES OF WHERE '#' WERE FOUND
# THIS ARRAY WOULD LATER BE STORED TO CONVERTED_PIECES[]

        ##### PREAMBLE #####
        subi  $sp, $sp, 32
        sw    $s0, 28($sp)
        sw    $t0, 24($sp)
        sw    $t1, 20($sp)
        sw    $s1, 16($sp)
        sw    $t2, 12($sp)
        sw    $t3, 8($sp)
        sw    $t4, 4($sp)

        # INITIALIZATION
```

```
      move  $s1, %arr
      la    $s0, pieceCoords

      move  $t4, %inc
      sll   $t4, $t4, 4
      add   $s0, $s0, $t4               # pieces[i] : p + r * 16
      move  $t2, $s0
      li    $t0, 0                      # initialize i
      li    $t1, 0                      # initialize j

loop_i:
      blt   $t0, 4, loop_j              # for i in range(4)
      j     end_i

loop_j:
      bge   $t1, 4, end_j               # for j in range(4)
      lb    $t3, 0($s1)             # load value of piecegrid[i][j]
      beq   $t3, 0x23, piece_found        # piecegrid[i][j] == '#'
      addi  $s1, $s1, 1
      addi  $t1, $t1, 1             # j++
      j     loop_j

piece_found:
      sb    $t0, 0($t2)             # store y-coordinate to pieceCoords[i]
      sb    $t1, 2($t2)             # store x-coordinate to pieceCoords[i]
      addi  $t1, $t1, 1             # j++
      addi  $t2, $t2, 4
      addi  $s1, $s1, 1             # update address
      j     loop_j                      # back to j

end_j:
      addi  $t0, $t0, 1             # i++
      li    $t1, 0                      # reset j = 0
      addi  $s1, $s1, 2             # update/skip '\r\n'
      j     loop_i                      # back to i

end_i:
      move  $v0, $s0                # return pieceCoords address

      ##### POSTAMBLE #####
      lw    $s0, 28($sp)
      lw    $t0, 24($sp)
      lw    $t1, 20($sp)
```

```
    lw    $s1, 16($sp)
    lw    $t2, 12($sp)
    lw    $t3, 8($sp)
    lw    $t4, 4($sp)
    addi  $sp, $sp, 32
```

---

get_max_x_of_piece(piece, position)

This function returns the coordinate of the rightmost element of a piece. This helps in checking if the piece would hit the right border of the playing grid. This function is equivalent to writing it as int get_max_of_piece(piece, position). This macro iterates through the array of piecePairs and gets the maximum value of the x coordinate. Afterwards, it returns the value of the max_x.

Code Block 11: get_max_x_of_piece() implementation

```
# ACTS AS INT GET_MAX_X_OF_PIECES(PIECE, I)
# RETURNS THE COORDINATE OF THE RIGHT MOST PIECE
# WILL HELP IN CHECKING IF PIECE HITS RIGHTMOST BORDER OF PLAYING GRID

    ##### PREAMBLE #####
    subi  $sp, $sp, 32
    sw    $t0, 24($sp)
    sw    $t1, 20($sp)
    sw    $t2, 16($sp)
    sw    $t3, 12($sp)
    sw    $t4, 8($sp)

    move  $t0, %pos
    sll   $t0, $t0, 4            # initialize i
    addi  $t1, $t0, 4            # initialize i + 4
    move  $t3, %piece            # # piece[i] : p + r * cols + c
    add   $t3, $t3, $t0
    li    $t2, -1                     # max_x = -1

loop:
    bge   $t0, $t1, end               # i < i + 4 ?
    lb    $t4, 2($t3)            # block[1]
    addi  $t3, $t3, 4
    addi  $t0, $t0, 1
    blt   $t2, $t4, block_is_greater    # max(max_x, block[1])
    j     loop
```

```
block_is_greater:
      move  $t2, $t4                # max = block[1]
      j     loop
end:
      move  $v0, $t2                # returns max_x

      ##### POSTAMBLE #####
      lw    $t0, 24($sp)
      lw    $t1, 20($sp)
      lw    $t2, 16($sp)
      lw    $t3, 12($sp)
      lw    $t4, 8($sp)
      addi  $sp, $sp, 32
```

---

deepcopy(array, length)

This macro serves one (1) purpose to two (2) different values. It can deepcopy a grid or an array based on the length which acts as a boolean on what to deepcopy. If length is 10 it would deepcopy a grid, otherwise, it is certain to deepcopy an array of boolean. This function returns the address to the array it has deepcopied. After checking what to copy, it allocates 96 bytes to the heap memory for a grid or 8 bytes using the sbrk() functionality of mips. After getting the address to the allocated memory space, it takes the values of each word from the array argument and stores it word-by-word to the allocated memory space, hence deepcopy of the array. Afterwards, it returns the address of the memory space of the deepcopied array.

Code Block 12: deepcopy() implementation

```
# DEEPCOPY A GRID
      li    $t0, 96                       # allocate 24 bytes for the copy
      li    $v0, 9                  # system call for memory allocation
      move  $a0, $t0
      syscall
      move  $t3, $v0                #  save  the  address  of  the  allocated
memory to $t3

      li    $t4, 0
      li    $t5, 0
grid_copy_loop:
      lw    $t4, 0($t1)
      sw    $t4, 0($t3)
      addi  $t1, $t1, 4
```

```
      addi  $t3, $t3, 4
      addi  $t5, $t5, 1
      blt   $t5, 20, grid_copy_loop
      j     end_grid

deep_copy_chosen:
      # DEEPCOPY CHOSEN
      li    $t0, 8                        # allocate 8 bytes for the copy
      li    $v0, 9                  # system call for memory allocation
      move  $a0, $t0
      syscall
      move  $t3, $v0                #  save  the  address  of  the  allocated
memory to $t3

      li    $t4, 0
      li    $t5, 0
chosen_copy_loop:
      lw    $t4, 0($t1)
      sw    $t4, 0($t3)
      addi  $t1, $t1, 4
      addi  $t3, $t3, 4
      addi  $t5, $t5, 1
      blt   $t5, 2, chosen_copy_loop

end_chosen:
      move  $v0, $t3                # returns address of deepcopy chosen
      j     end

end_grid:
      move  $v0, $t3                # returns address of deepcopy grid

end:
      ##### POSTAMBLE #####
      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
      lw    $t2, 20($sp)
      lw    $t3, 16($sp)
      lw    $t4, 12($sp)
      addi  $sp, $sp, 32
```

---

drop_piece_in_grid(grid, piece, offset)

This macro is equivalent to grid drop_piece_in_grid(grid, piece, offset) that returns two values, a grid and a boolean. This function drops each piece from top to bottom once the offset has been added. Once it is placed inside the backtrack function it moves the pieces down and to the right by an offset. It works by first converting the coordinates in pieces into '#' and places them onto the current grid. Afterwards, it iterates bringing the pieces down until it either hits an 'X' or reaches the bottom. Most of the code written for this function was translated from the python code provided. Relevant lines of code were provided comments on their purpose for the function.

Code Block 13: convert_pieces_to_pairs() python implementation

```
def drop_piece_in_grid(grid, piece, yOffset):
    gridCopy = deepcopy(grid)
    maxY = 9
    for block in piece:
        gridCopy[block[0]][block[1] + yOffset] = '#'  # put piece in grid
    #  only active blocks are '#'; frozen blocks are 'X'
    while True:
        canStillGoDown = True
        for i in range(4 + 6):
            for j in range(6):
                  if gridCopy[i][j] == '#' and (i + 1 == 10 or gridCopy[i +
1][j] == 'X'):
                    canStillGoDown = False

        if canStillGoDown:
                 for i in range(8, -1, -1):   # move cells of piece down,
starting from bottom cells
                for j in range(6):
                    if gridCopy[i][j] == '#':  # move cells down one space
                        gridCopy[i + 1][j] = '#'
                        gridCopy[i][j] = '.'
        else:
            break

    for i in range(4 + 6):
        for j in range(6):
            if gridCopy[i][j] == '#':
                maxY = min(maxY, i)

    if maxY <= 3:  # piece protrudes from top of 6x6 grid
```

```
        return grid, False
    else:
        return freeze_blocks(gridCopy), True
```

Code Block 14: drop_piece_in_grid() implementation

```
# ACTS AS GRID DROP_PIECE_IN_GRID(GRID, PIECE, OFFSET)
# THIS FUNCTION DROPS EACH PIECE FROM TOP TO BOTTOM
# WITH REGARDS TO THE XOFFSET. ONCE PLACED WITH THE BACKTRACK FUNCTION
# IT MOVES THE PIECES DOWN AND TO THE RIGHT BY OFFSET.
# IT FIRSTS CONVERT THE COORDINATES IN PIECES INTO '#' AND PLACES THEM
# IN THE BOARD. AFTERWARDS, IT ITERATES BRINGING THE PIECES DOWN TILL IT
# EITHER HITS AN 'X' OR REACHES THE BOTTOM.
#  MOST  OF  THE  CODE  WRITTEN  HERE  WERE  TRANSLATED  FROM  THE  PYTHON  CODE
PROVIDED
# THIS FUNCTION RETURNS TWO VALUES: V0: GRID, V1: BOOLEAN

    ##### PREAMBLE #####
    subi  $sp, $sp, 64
    sw    $s5, 60($sp)
    sw    $s6, 56($sp)
    sw    $s7, 52($sp)
    sw    $t0, 48($sp)
    sw    $t1, 44($sp)
    sw    $t2, 40($sp)
    sw    $t3, 36($sp)
    sw    $t4, 32($sp)
    sw    $t5, 28($sp)
    sw    $t6, 24($sp)
    sw    $t7, 20($sp)
    sw    $t8, 16($sp)
    sw    $t9, 12($sp)

    # INITIALIZATION
    move  $s5, %grid
    move  $s6, %piece
    move  $s7, %offset

    # Create a deepcopy of the grid such that original grid would
    # not be manipulated.
    deepcopy($s5, 10)
```

```
      subi  $v0, $v0, 0x50
      move  $t0, $v0                # gridCopy

      # THIS CODE READS THROUGH PIECES AND ADDS #' TO THE BOARD
      # WITH REGARDS TO PIECE'S COORDINATES

      # for block in piece:
      li    $t7, 0
      subi  $sp, $sp, 4
      sw    $s6, 0($sp)
loop_block:
      # gridCopy[block[0]][block[1] + yOffset] = '#'
      subi  $sp, $sp, 4             # Save i in memory
      sw    $t0, 0($sp)

      lb    $t4, 0($s6)             # block[0]
      lb    $t5, 2($s6)             # block[1]
      add   $t5, $t5, $s7                 # block[1] + yoffset

      # p + r * cols + c
      li    $t6, 8
      mult  $t4, $t6
      mflo  $t4                     # offset to convert 2D -> 1D array

      add   $t4, $t4, $t5
      add   $t0, $t0, $t4

      li    $t6, 0x23
      sb    $t6, 0($t0)             # stores '#' to coordinate in grid

      addi  $s6, $s6, 4
      addi  $t7, $t7, 1

      lw    $t0, 0($sp)
      addi  $sp, $sp, 4
      bne   $t7, 4, loop_block

      lw    $s6, 0($sp)
      addi  $sp, $sp, 4

loop_forever:

      li    $t1, 0x1                # canStillGoDown
```

```
      li    $t2, 0                        # initialize i

loop_i:
      beq   $t2, 10, end_loop_i           # for i in range(4 + 6)
      li    $t3, 0                        # initialize j

loop_j:
      beq   $t3, 6, end_loop_j            # for j in range(6):
      subi  $sp, $sp, 4
      sw    $t0, 0($sp)             # t0 is gridCopy

      move  $t4, $t2                # i
      move  $t5, $t3                # j

      # p + i * 8 + j
      sll   $t4, $t4, 3
      add   $t4, $t4, $t5
      add   $t0, $t0, $t4                 # gridCopy[i][j]
      lb    $t9, 0($t0)
      bne   $t9, '#', go_back_to_j        # if gridCopy[i][j] == '#':

      move  $t4, $t2                # copy of orig i
      move  $t5, $t3                # copy of orig j
      addi  $t4, $t4, 1
      bne   $t4, 10, check_next           # i + 1 == 10, if false check next
condition
      j     skip_check              # already true since OR

check_next:
      # gridCopy[i + 1][j] == 'X'
      # p + (i + 1) * 8 + j
      sll   $t4, $t4, 3
      add   $t4, $t4, $t5
      lw    $t0, 0($sp)
      add   $t0, $t0, $t4
      lb    $t9, 0($t0)
      bne   $t9, 'X', go_back_to_j

skip_check:
      lw    $t0, 0($sp)             # load back t0
      addi  $sp, $sp, 4
      li    $t1, 0x0                # canStillGoDown = False
      j     end_loop_j              # go to increment i
```

```
go_back_to_j:
      addi  $t3, $t3, 1
      lw    $t0, 0($sp)
      addi  $sp, $sp, 4
      j     loop_j                          # increment j

end_loop_j:
      addi  $t2, $t2, 1
      li    $t3, 0
      beqz  $t1, end_loop_i                 # if not canStillGoDown:
      j     loop_i

end_loop_i:
      beqz  $t1, end_loop_forever

      # if canStillGoDown:
      li    $t8, 8                          # i
loop_down_i:
      li    $t2, 0                          # j
loop_down_j:
      subi  $sp, $sp, 4
      sw    $t0, 0($sp)

      move  $t3, $t8
      move  $t4, $t2
      move  $t6, $t0

      # p + i * 8 + j
      # gridCopy[i][j] == '#'
      sll   $t3, $t3, 3
      add   $t3, $t4, $t3
      add   $t6, $t6, $t3
      lb    $t5, 0($t6)
      bne   $t5, 0x23, skip                 # if gridCopy[i][j] == '#':

      # gridCopy[i][j] = '.'
      li    $t7, 0x2e
      sb    $t7, 0($t6)

      # gridCopy[i + 1][j] = '#'
      move  $t3, $t8
      move  $t6, $t0
```

```
      addi  $t3, $t3, 1
      sll   $t3, $t3, 3
      add   $t3, $t4, $t3
      add   $t6, $t6, $t3

      li    $t7, 0x23
      sb    $t7, 0($t6)

skip:
      lw    $t0, 0($sp)
      addi  $sp, $sp, 4
      addi  $t2, $t2, 1
      bne   $t2, 6, loop_down_j          # loop back to down j
      subi  $t8, $t8, 1
      bne   $t8, -1, loop_down_i         # # loop back to down i

      j     loop_forever

end_loop_forever:
      # for i in range(4 + 6):
      li    $t1, 9                       # maxY
      li    $t2, 0                       # i

loop_outside_i:
      li    $t3, 0                       # j
loop_outside_j:
      subi  $sp, $sp, 4
      sw    $t0, 0($sp)

      move  $t4, $t2
      move  $t5, $t3
      move  $t6, $t0

      # p + i * 8 + j
      # gridCopy[i][j] == '#'
      sll   $t4, $t4, 3
      add   $t4, $t4, $t5
      add   $t6, $t6, $t4
      lb    $t7, 0($t6)
      bne   $t7, 0x23, skip_outside

      # maxY = min(maxY, i)
      blt   $t1, $t2, skip_outside       # i < maxY, maxY
```

```
      move  $t1, $t2                # maxY = i

skip_outside:
      lw    $t0, 0($sp)             # load back t0
      addi  $sp, $sp, 4
      addi  $t3, $t3, 1
      bne   $t3, 6, loop_outside_j        # loop back to outside j
      addi  $t2, $t2, 1
      bne   $t2, 10, loop_outside_i       # loop back to outside i

      ble   $t1, 3, ret_grid_F            # maxY <= 3? if True, return grid,
False
      freeze_blocks($t0)
      move  $v0, $t0                # return freeze(gridCopy)
      li    $v1, 0x1                # return True
      j     end_of_piece_drop
ret_grid_F:
      move  $v0, $s5                # return grid
      li    $v1, 0x0                # return False
end_of_piece_drop:

      ##### POSTAMBLE #####
      lw    $s5, 60($sp)
      lw    $s6, 56($sp)
      lw    $s7, 52($sp)
      lw    $t0, 48($sp)
      lw    $t1, 44($sp)
      lw    $t2, 40($sp)
      lw    $t3, 36($sp)
      lw    $t4, 32($sp)
      lw    $t5, 28($sp)
      lw    $t6, 24($sp)
      lw    $t7, 20($sp)
      lw    $t8, 16($sp)
      lw    $t9, 12($sp)
      addi  $sp, $sp, 64
```

---

counter(address)

This macro counts the number of pieces being used in the grid. It returns an int on the number of ‘X’s in the grid. This will help in determining if the final grid is possible given a specific number

of pieces and number of ‘X’s in the current grid. It works by iterating through the grid and when it detects an ‘X’ it increments the counter.

Code Block 15: counter() implementation

```
# THIS FUNCTION COUNTS THE NUMBER OF PIECES BEING USED IN THE GRID
# IT RETURNS AN INT ON THE NUMBER OF 'X'
# THIS WILL HELP IN DETERMINING IF THE FINAL GRID IS POSSIBLE GIVEN
# A SPECIFIC NUMBER OF PIECES AND NUMBER OF 'X' IN CURRENT GRID

      subi  $sp, $sp, 32
      sw    $t0, 28($sp)
      sw    $t1, 24($sp)
      sw    $t2, 20($sp)
      sw    $t3, 16($sp)

      li    $t0, 0                      # i
      move  $t1, %add             # arr
      li    $t2, 0                      # counter

loop:
      lb    $t3, 0($t1)           # arr[i][j]
      beq   $t3, 'X', increment         # if arr[i][j] == 'X', t2++
      j     next
increment:
      addi  $t2, $t2, 1           # t2++
next:
      addi  $t1, $t1, 1           # arr++
      addi  $t0, $t0, 1           # i++
      blt   $t0, 80, loop
      move  $v0, $t2              # return counter

      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
      lw    $t2, 20($sp)
      lw    $t3, 16($sp)
      addi  $sp, $sp, 32
```

backtrack(grid, chosen, pieces)

This function by the name itself is the recursive function where most of the above macros were used. It works by first checking if the pieces fit in the final grid to reduce the number of

iterations. Afterwards, it checks if the current and final grids are equal if so it must return true and go back to the function call. If both functions fail, it would begin the bulk of the function. It first creates a deepcopy of the chosen array then takes an element in chosen. It then calculates for the maximum x coordinate of a piece using get_max_x_of_piece. This would ensure that the itearations would not go beyond the board through the offset. After calculating the offset, we can now drop the piece using the macro implemented earlier and its return values would be stored to $v0 and $v1. If success, means if the piece dropped was valid and it can be recused to use a different piece and state. Once backtracking is done we finish all remaining iterations and return a boolean value that would be used for printing. Further explanation of the code were commented below

Code Block 16: backtrack implementation

```
##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $s0, 28($sp)                # currGrid
      sw    $s3, 24($sp)                # chosen
      sw    $s4, 20($sp)                # pieces
      sw    $t1, 16($sp)                # i
      sw    $t5, 12($sp)                # offset
      sw    $s5, 8($sp)             # chosen_copy
      sw    $ra, 4($sp)             # ra
      sw    $t3, 0($sp)             # max_x_piece


      move  $s0, $a0                # address of currGrid -> nextGrid
($s0)
      move  $s3, $a1                # address of chosen -> chosen_copy
($s5)
      move  $s4, $a2                # address of pieces -> pieces
($s4)

      not_fit($s0, $s1)             # Checks if the piece fits in the final
grid. Used for optimization
      beq   $v0, 1, return_false          # If 0, return false. Go back to
recursive call

      is_equal_grids($s0, $s1)            # Checks if the grids are equal.
      bnez  $v0, end                # If 1, stop recursion.

else:
      deepcopy($s3, 2)
      move  $s5, $v0                # chosen_copy = chosen[:]
```

```
      subi  $s5, $s5, 8            # Subtracts 8 since for some reason it
increments the final address by 8.

      li    $t1, 0                      # initialize i
      move  $s6, $s3               #  created  a  copy  of  the  address  of
chosen
      move  $s7, $s5               #  created  a  copy  of  the  address  of
chosen_copy
loop_i:
      lb    $t2, 0($s6)            # chosen[i]
      bnez  $t2, loop_back_to_i          # if not chosen[i]:

      get_max_x_of_piece($s4, $t1)       # get_max_x_of_piece(pieces[i])
      move  $t3, $v0               #  max_x_of_piece

      #  for offset in range(6 - max_x_of_piece):
      li    $t5, 6
      sub   $t3, $t5, $t3                # range(6 - max_x_of_piece)
      li    $t5, 0                       # offset

loop_offset:
      # nextGrid, success = drop_piece_in_grid(currGrid, pieces[i], offset)

      subi  $sp, $sp, 4            # Preamble
      sw    $t1, 0($sp)            #  Due  to  lack  of  registers  need  to
store t1 to memory and return value after

      sll   $t1, $t1, 4
      add   $t1, $t1, $s4                # pieces[i]

      drop_piece_in_grid($s0, $t1, $t5)   #     drop_piece_in_grid(currGrid,
pieces[i], offset)
      # return values: $v0 = nextGrid & $v1 = success

      lw    $t1, 0($sp)
      addi  $sp, $sp, 4            # Postamble

      beqz  $v1, loop_back_to_offset     # if success:
      li    $t0, 1
      sb    $t0, 0($s7)            # chosen_copy[i] = True

      # Moving values for recursive call
      move  $a0, $v0               # nextGrid
```

```
      move  $a1, $s5              # chosen_copy
      move  $a2, $s4              # pieces
      jal   backtrack             #    backtrack(nextGrid,    chosen_copy,
pieces)


      bnez  $v0, end              #  if  backtrack(nextGrid,  chosenCopy,
pieces):
      li    $t0, 0
      add   $s7, $s5, $t1
      sb    $t0, 0($s7)           # chosen_copy[i] = False

loop_back_to_offset:
      addi  $t5, $t5, 1           # offset++
      blt   $t5, $t3, loop_offset       # go back to loop_offset

loop_back_to_i:
      addi  $t1, $t1, 1           # i++
      add   $s6, $s3, $t1               # chosen++
      add   $s7, $s5, $t1               # chosen_copy++
      blt   $t1, $s2, loop_i      # go back to loop_i

return_false:
      li    $v0, 0

end:
      ##### POSTAMBLE #####
      lw    $s0, 28($sp)                # currGrid
      lw    $s3, 24($sp)                # chosen
      lw    $s4, 20($sp)                # pieces
      lw    $t1, 16($sp)                # i
      lw    $t5, 12($sp)                # offset
      lw    $s5, 8($sp)           # chosen_copy
      lw    $ra, 4($sp)           # ra
      lw    $t3, 0($sp)           # max_x_piece
      addi  $sp, $sp, 32
      jr    $ra                   # return to last recursive call
```

---

main()

The main function contains all of the function calls, backtrack, and printing of the output. In order to properly output the string, it checks if $v0 or the return value from backtrack or counter is 1 or 0. If 1 it should output YES, otherwise NO. To end the program, syscall 10 was implemented.

Code Block 17: main() implementation

```
main:
	###### DO NOT MODIFY THESE REGISTERS ######
	# CANNOT BE MODIFIED ONCE THEY HOLD THE ORIGINAL ADDRESSES/VALUES
	# S0 base address of start grid
	# S1 base address of final grid
	# S2 number of pieces
	# S3 base address for chosen
	# S4 base address of pieces
	###### DO NOT MODIFY THESE REGISTERS ######

	# Initialize grids to have 4 empty rows and 6 columns
	la    $s0, start_grid              # $s0 = address of start_grid
	la    $s1, final_grid              # $s1 = address of final_grid
	move  $s2, $s0               # S2 copy of base address of S0
	move  $s3, $s1               # S3 copy of base address of S1


	##### GET STARTING AND FINAL GRID #####
	init_arr($s2)
	init_arr($s3)
	get_input($s2)
	get_input($s3)

	##### GET INTEGER INPUT #####
	la    $t0, int               # get buffer address
	read_file(3, $t0)            #  reads  3  bytes  from  the  input  and
saves it to the buffer address
	lb    $s2, 0($t0)
	subi  $s2, $s2, 0x30               # removes the extra

	##### CHECK IF NUM_PIECES ARE VALID #####

	subi  $sp, $sp, 32
	sw    $t0, 28($sp)
	sw    $t1, 24($sp)
```

```
      sw    $t2, 20($sp)

      counter($s0)
      move  $t0, $v0
      counter($s1)
      move  $t1, $v0

      move  $t2, $s2
      sll   $t2, $t2, 2
      add   $t0, $t0, $t2

      sgt   $t2, $t1, $t0
      xori  $v0, $t2, 1

      lw    $t0, 28($sp)
      lw    $t1, 24($sp)
      lw    $t2, 20($sp)
      addi  $sp, $sp, 32
      beqz  $v0, end_program

      ##### INITIALIZE CHOSEN PIECES ARRAY #####
      la    $s3, chosen
      move  $t0, $s3              # make copy of address of s3
      init_chosen($s2, $t0)

      ##### GET PIECES INPUT #####
      la    $s4, converted_pieces
      move  $t0, $s4              # make copy of address of s4
      get_pieces($s2, $t0)              #  returns  converted_pieces  with
array of coordinates
      lw    $s4, 0($s4)

      ##### START OF BACKTRACKING #####
      # def backtrack(currGrid, chosen, pieces)

      move  $a0, $s0              # grid
      move  $a1, $s3              # chosen
      move  $a2, $s4              # pieces

      jal   backtrack
      j     end_program
##### PRINTING OF YES OR NO #####
end_program:
```

```
      move  $t0, $v0                # v0 return value of backtrack
      beqz  $t0, print_no                 # if v0 = 0, print no
      li    $v0, 4
      la    $a0, yes                # print yes
      syscall
      j     stop_program

print_no:
      li    $v0, 4
      la    $a0, no                       # print no
      syscall


##### END OF PROGRAM #####
stop_program:
      li $v0, 10                          # exit program
      syscall
```

Supplementary Macros

Code Block 18: print()

```
.macro print(%add)
      # THIS IS MACRO USED FOR TESTING ONLY
      # PRINTS OUT EACH WORD/ROW IN THE ARRAY

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t1, 0($sp)
      sw    $t2, 4($sp)
      sw    $t3, 8($sp)
      li    $t1, 0
      move  $t3, %add
loop:
      lb    $t2, ($t3)
      move  $a0, $t2
      li    $v0, 11
      syscall

      addi  $t1, $t1, 1
      addi  $t3, $t3, 1
      blt   $t1, 80, loop

      la    $a0, newline
      li    $v0, 4
      syscall

      ##### POSTAMBLE #####
      lw    $t1, 0($sp)
      lw    $t2, 4($sp)
      lw    $t3, 8($sp)
      addi  $sp, $sp, 32
.end_macro
```

Code Block 19: print_chosen()

```
.macro print_chosen(%add)
      # USED ONLY FOR TESTING
      # THIS PRINTS OUT THE ELEMENTS IN CHOSEN ARRAY

      ##### PREAMBLE #####
      subi  $sp, $sp, 32
      sw    $t1, 0($sp)
      sw    $t2, 4($sp)
      sw    $t3, 8($sp)

      li    $t1, 0
      move  $t3, %add
loop:
      lb    $t2, 0($t3)

      move  $a0, $t2
      li    $v0, 1
      syscall

      addi  $t1, $t1, 1
      addi  $t3, $t3, 2
      blt   $t1, 6, loop

      la    $a0, newline
      li    $v0, 4
      syscall

      ##### POSTAMBLE #####
      lw    $t1, 0($sp)
      lw    $t2, 4($sp)
      lw    $t3, 8($sp)
      addi  $sp, $sp, 32
.end_macro
```

Supplementary Information

Code Block 20: .data in MIPS

```
##### DATA #####
.data
start_grid: .space 96              # allocate space for a 4x6x4 char array
final_grid: .space 96              # allocate space for a 4x6x4 char array
int:        .space 8               # int input variable
chosen:           .space 24              #  allocate  space  for  4x6  bool
array
converted_pieces:.space 24               # allocate space for 4x6 converted
pieces array
pieceAscii: .space 96              # allocate 4x4x6 for pieceAscii
pieceCoords:      .space 96              # allocate 4x4x6 for pieceCoords
yes:        .asciiz "YES"
no:         .asciiz "NO"
```

Code Block 21: Logic in extracting an element in a 2D array through 1D array

```
##### 2D ARRAY -> 1D ARRAY ELEMENT EXTRACTION LOGIC #####

# 0  1  2  3  4  5
# 6  7  8  9  10 11
# 12 13 14 15 16 17
# 18 19 20 21 22 23

# p + r * cols + c

#############################################################
```

References

MIPS syscall functions available in MARS. (2023). Missouristate.edu. https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html