# **Tetrisito:** A Tetris Puzzle Solver

CS 21 Machine Problem 1

AY 2022-2023 2nd Semester

Submitted by:

**BARCENAS, Marius Ulyzses**

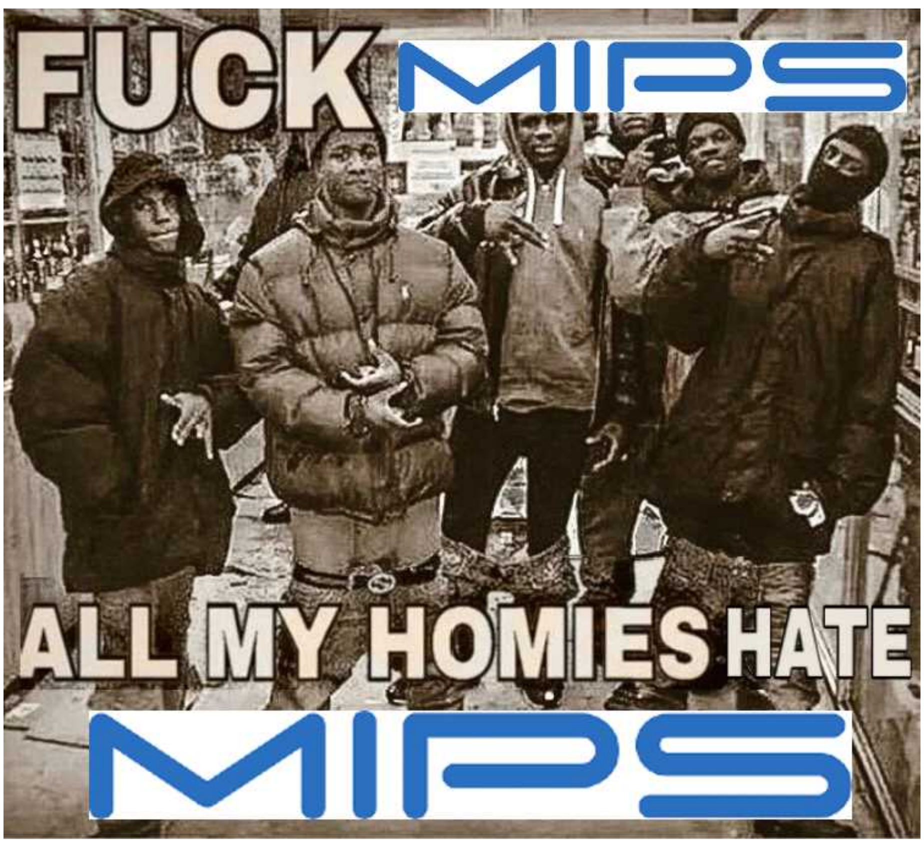
SN 2021-02371 — CS 21 Lab 4

21 May 2023

# Contents

# Foreword

Before beginning with the documentation, I would like to offer my sincerest emotions in the following messages:




if 0 ulit ako rito im gonna cry fr

# Chapter 1

# Introduction

## 1.1 Tetris®

Tetris® used to be a fun game.

It's a game created by a Soviet software engineer Alex Pajitnov in 1985[1]. It is a puzzle video game composed of a grid and a set of pieces called tetriminos[2] where the tetriminos drop from above to populate the grid. These falling tetriminos can be laterally moved or rotated by the player until they touch the bottom of the grid or another piece that was already placed. Each time a row is completed with blocks, it is cleared, and the blocks above it fall one row.

Clearing lines gives players points, and clearing the max of four lines (using a long bar) gives the most points and is called a Tetris. Classical versions of Tetris® is singleplayer with the objective of simply getting higher scores from line clears. However, some modern Tetris® variants like TETR.IO, Tetris® 99, and Puyo Puyo™ Tetris® are multiplayer and use line clears and special moves and combos (see T-spins) to send "garbage lines" to opponents. Their objectives then, is to knock opponents out by having them fill their board.

## 1.2 The Problem

This project, "Tetrisito", is a mini Tetris® puzzle solver. It answers the following question: "Given two $6 \times 6$ grids and a list of tetriminos, is it possible to get from one grid to the other via (mostly conventional) Tetris® rules?" If it is possible, the solver prints out `"YES"`, otherwise, it prints out `"NO"`. The primary differences from the conventional Tetris® are as follows:

[1] Lifted from Wikipedia: https://en.wikipedia.org/wiki/Tetris

[2] Tetrimino is the official trademarked name for these tetrominoes so that's what I'll continue to call them

- Pieces are not allowed to move laterally while falling
- Pieces are not allowed to rotate
- Complete lines are not cleared
- Dropped pieces which go out of the board's bounds are ignored

There are three levels of difficulty which we can choose from, depending on what we want our grade to be and how much we hate ourselves. The first one, Implementation A, only gives one tetrimino to drop. Meanwhile the second one, Implementation B, gives two to five tetriminos, dropped in the specific order it was given. Finally, Implementation C gives two to five tetriminos, dropped in any order.

Aside from these basic implementations, additional bonuses are offered for the project. Bonus 1 removes the limitations that pieces are not allowed to move laterally while falling. With this bonus, tetriminos are allowed to shift one column to the left or to the right every time it falls down one row. This allows for previously disallowed moves like tucking in pieces in blocked columns. Meanwhile, Bonus 2 removes the limitation of non-line clearing. With this, every time a row is completed, it is cleared and the rows above it are moved down by one row.

**The Problem** is that it has to be implemented in MIPS, an assembly language.

## 1.3 Solution Overview

In this documentation, I will be explaining the general solution for Implementation C, as that is what I did. For now, I will provide an overview to my implementation.

First is to parse the input. This involves mainly converting the given boards and pieces into their respective register representations and store them in their corresponding spaces in memory.

After the input is parsed, the recursive backtracking part begins. Each piece is dropped into the board, trying all different offsets. Then, once a piece is dropped, it recursively tries for another piece that has not yet been dropped. Doing so ensures that the pieces are dropped in all possible orders.

If, at any point, the board matched the target board, it immediately returns to print `"YES"`. Otherwise, once it's exhausted all permutations and did not find a match, it returns to print `"NO"`.

# Chapter 2

# Memory Management

## 2.1 .data Directive

In making the program, I used the `.data` directive to store specific data that I used throughout.

```
.data
filename:       .asciiz "test.in"
yes:            .asciiz "YES"
no:             .asciiz "NO"
cont:           .align 2
                .space 256
```

The `filename` field contains the name of the input file that I used. This is used purely used during development and is not used in the final version of the assembly file. This field is what is fed to `syscall 13` during development for testing.

The `yes` and `no` fields contain the ASCII equivalent of the `"YES"` and `"NO"` strings respectively. These are what is printed at the end of the program as the output.

Finally, the `cont` field is the memory space where the output of the `read_file()` macro is stored. It is word aligned and contains 256 bytes in order to make sure that any length of the input file can be stored simultaneously. This field is only used during the input parsing session and any time it is used, it overwrites the previous content.

## 2.2 Grid

### 2.2.1 Register Representation

In my implementation, I decided to use only two registers to represent a board or piece. Each cell in the grid is represented by only one bit in one of the registers: `1` if it contains a block, `0` otherwise. This is so every time a grid is manipulated, only bit-wise operations are required between two registers.

The grid has 10 rows and 6 columns split into two height-wise, each containing three columns. The cells are arranged in column-major order, with the topmost row appearing in the rightmost bit in each column group and the first column from the left appearing in the left most column group in the register. That is, a grid is split into two similar parts and stored into registers in the format shown in Figure 2.1.

```
. . . || . . .
. . . || . . .
. . . || . . .
F L R || F L R
E K Q || E K Q
D J P || D J P
C I O || C I O
B H N || B H N
A G M || A G M
# # # || # # #
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | #ABCDEF... | #GHIJKL... | #MNOPQR... |
| reg2 | 00 | #ABCDEF... | #GHIJKL... | #MNOPQR... |

Figure 2.1: The register representations of a grid.

Since each column has 10 rows, only 30 bits are used for three columns. As such, a leading `00` is found from bits 30-31. The first column is then found in bits 20-29, the second column in bits 10-19, and the third column in bits 0-9.

Figure 2.2 shows an example grid layout with a line separating the two halves for better visualisation.

```
. . . || . . .
. . . || . . .
. . . || . . .
. . # || . . .
. . # || . . .
. . # || . . .
# # # || . . .
# # # || . # #
# # # || # # .
# # # || # # #
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 1111000000 | 1111000000 | 1111111000 |
| reg2 | 00 | 1100000000 | 1110000000 | 1010000000 |

Figure 2.2: A sample grid with its corresponding register representations.

### 2.2.2 Board

The playing area is only a 6 × 6 grid according to the specifications of the problem. These are bits 1-6 of each column (the alphabetical characters in my previous representation).

Below this playing area (bit 0) is one row of solid blocks (`"#"`) and above it (bits 7-9) are three rows of empty cells (`"."`). These extra rows are added for use by the `drop_piece` function and its exact use will be discussed in that section.

Figures 2.3–2.4 show example boards with their register representations. The playing areas are separated in the grids for better visualisation.

```
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

# # # # # #
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 1000000000 | 1000000000 | 1000000000 |
| reg2 | 00 | 1000000000 | 1000000000 | 1000000000 |

Figure 2.3: An empty board with its corresponding register representations.

```
. . . . . .
. . . . . .
. . . . . .

. . . . . .
. . . . . .
. . . . . .
. # # # # .
. . # # # #
# # # # . .

# # # # # #
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 1100000000 | 1101000000 | 1111000000 |
| reg2 | 00 | 1111000000 | 1011000000 | 1010000000 |

Figure 2.4: A non-empty board with its corresponding register representations.

### 2.2.3 Pieces

Each piece is a 4 × 4 grid anchored to the bottom left cell, according to the specifications of the problem. These are bits 6-9 of the first four columns in the entire grid. That is, it makes use of all collumns in the first register, and only the first column in the second register. This makes use of the empty rows above the board and unlike the board, a piece has no row of solid blocks on the bottommost row.

Figures 2.5–2.7 show example pieces with their corresponding register representations. The piece areas are separated in the grids for better visualisation.

```
. . . . . .
. . . . . .
# # . . . .
# # . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 0000001100 | 0000001100 | 0000000000 |
| reg2 | 00 | 0000000000 | 0000000000 | 0000000000 |

Figure 2.5: An O-tetrimino with its corresponding register representations.

```
. . . . . .
. # . . . .
# # . . . .
. # . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 0000000100 | 0000001110 | 0000000000 |
| reg2 | 00 | 0000000000 | 0000000000 | 0000000000 |

Figure 2.6: A T-tetrimino with its corresponding register representations.

```
# . . . . .
# . . . . .
# . . . . .
# . . . . .

. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 0000001111 | 0000000000 | 0000000000 |
| reg2 | 00 | 0000000000 | 0000000000 | 0000000000 |

Figure 2.7: An I-tetrimino with its corresponding register representations.

## 2.3 Global Frame Allocation

In order to save values used in the rest of the program, I made use of the global frame. The values used in here are word-aligned for consistency in accessing them.

The first value saved is the file descriptor returned by `syscall 13` during development. This is what's used as an argument by `syscall 14` to read the files. The next value saved is the number of pieces that the program is expected to receive. This is the parsed value of line 13 in the input files and dictates how much of the global frame is used.

The next four words in the global frame represent the boards passed from the input. The first two words are the representations of the initial board, while the latter two words are the representations of the final board that the program aims to achieve.

Finally, the pieces are stored. There are `2n` words used for this where `n` is the number of pieces. Each piece is represented by two words as discussed in the previous section.

The following table shows a summary of the offset, size, and description of each of the values stored in the global frame.

| **Offset** | **Size** | **Description** |
|---|---|---|
| 0 | 1 | File descriptor used by `syscall 14` in development |
| 4 | 1 | Number of pieces the program is expected to receive |
| 8 | 2 | Representation of the initial board |
| 16 | 2 | Representation of the final board |
| 24 | 2 | Representation of the first piece |
| 32+ | 2+ | Representation of the succeeding pieces |

Table 2.1: Allocation of memory in the global frame

# Chapter 3

# Algorithms

## 3.1 Macros

### 3.1.1 read_file

This is a macro I made to make it easier to read a specified number of bits from the input file. It uses `syscall 14` and a file descriptor of 0, the address of the content as specified in the `.data` derivative, and the number of bytes as specified by the input to the macro.

I chose to pass in a register `%register` and an integer `%bytes` value to the macro to specify the number of bytes to read instead of just one integer value as the input sometimes contained variable lengths (i.e. the grids and the pieces) and I am using the same function to parse both of these inputs.

```
.macro  read_file(%register, %bytes)
        addi    $a0, $0, 0
        la      $a1, cont
        addi    $a2, %register, %bytes

        addi    $v0, $0, 14
        syscall
.end_macro
```

### 3.1.2 save

This is made for register housekeeping and is inserted at the beginning of every function. It saves all the `$s` registers (except `$s7` due to stack size concerns) as well as the `$ra` register into the stack.

Due to the non-saving of the `$s7` register, a conscious design choice throughout the writing of the program was to assume that usage of that register will be erratic and can be overwritten anytime, similar to a `$t` register.

```
.macro  save()
        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)
        sw      $s5, 24($sp)
        sw      $s6, 28($sp)
.end_macro
```

### 3.1.3 return

The pair of the `save()` macro, this is inserted at the end of every function. It retrieves all the `$s` registers (except `$s7` due to stack size concerns) as well as the `$ra` register back from the stack. Then, it calls `jr $ra` to return the control back to the caller.

```
.macro  return()
        lw      $s6, 28($sp)
        lw      $s5, 24($sp)
        lw      $s4, 20($sp)
        lw      $s3, 16($sp)
        lw      $s2, 12($sp)
        lw      $s1, 8($sp)
        lw      $s0, 4($sp)
        lw      $ra, 0($sp)
        addi    $sp, $sp, 32

        jr      $ra
.end_macro
```

### 3.1.4 save_temp

Similar to the `save()` macro, this saves register values into the stack. However, this saves all the `$t` registers instead (except for $t8-$`t9` due to stack size concerns).

Rather than at the start of the callee function call, I use this macro before a function is called inside the caller to indicate that it is specifically the caller that requires `$t` registers to be saved, since saving temporary registers is not generally part of function housekeeping. Further, similar to `$s6` in the `save()` macro, due to the non-saving of the `$t8-$t9` registers, a conscious design choice throughout the writing of the functions which use this macro was to assume that usage of those registers will be erratic and can be overwritten anytime.

```
.macro  save_temp()
        addi    $sp, $sp, -32
        sw      $t0, 0($sp)
        sw      $t1, 4($sp)
        sw      $t2, 8($sp)
        sw      $t3, 12($sp)
        sw      $t4, 16($sp)
        sw      $t5, 20($sp)
        sw      $t6, 24($sp)
        sw      $t7, 28($sp)
.end_macro
```

The pair of the `save_temp()` macro, this is inserted after the end of a function call, inside the caller function. It retrieves all the `$t` registers (except for `$t8-$t9` due to stack size concerns).

### 3.1.5 return_temp

```
.macro  return_temp()
        lw      $t7, 28($sp)
        lw      $t6, 24($sp)
        lw      $t5, 20($sp)
        lw      $t4, 16($sp)
        lw      $t3, 12($sp)
        lw      $t2, 8($sp)
        lw      $t1, 4($sp)
        lw      $t0, 0($sp)
        addi    $sp, $sp, 32
.end_macro
```

## 3.2 Functions

### 3.2.1 build_row

The purpose of this function is to add a row to the grid. It takes in two arguments: the registers representing the grid, and a shift amount which indicates which row it's supposed to be included in. Additionally, it assumes that the row to include is already loaded into the input buffer address `cont`. After the function is finished, it returns two registers for the representation of the new grid with the row added.

The first part of this function is the housekeeping. It first stores the previous state of `$s` registers into the stack via the `save()` macro. Then, it loads the buffer address, shift amount, and board registers into their respective `$s` registers.

```
5  build_row:
6          save()
7
8          la      $s0, cont       # buffer address
9          move    $s1, $a0        # shift amount
10         move    $s2, $a1        # reg 1
11         move    $s3, $a2        # reg 2
```

Next, it loops through each item in the input buffer. It converts all the `"#"` characters into `1` and everything else into `0`.

As seen in the code, the number 6 is hardcoded in line 26. For board grids, this is self-explanatory as each row has exactly 6 cells. However, pieces only have 4 characters in each row. Despite that, this still works due to the fact that MIPS uses `"\r\n"` for newlines. As such, since neither `\r` nor `\n` are equal to `"#"`, they are still parsed as `0` which is the target behaviour.

```
13         addi    $t0, $0, 0      # offset
14 br_loop:                        # do {
15         add     $t1, $s0, $t0   # add offset
16         lb      $t2, 0($t1)     # get character at offset
17
18         bne     $t2, 0x23, br_isdot     # if '#', proceed
19         addi    $t2, $0, 1
20         j       br_isdot_
21 br_isdot:
22         addi    $t2, $0, 0      # if not '#'
23 br_isdot_:
24         sb      $t2, 0($t1)     # return parsed to memory
25         addi    $t0, $t0, 1     # increment offset
26         blt     $t0, 6, br_loop # } while (offset < 6)
```

Once the inputs are parsed to their bit equivalents, it is now time to add the row to the boards. This is manually done so that the first bit is on bit 0, the second bit is on bit 10, and the third bit is on bit 20 (refer to Section 2.2.1).

Two of these code blocks are found in the function, one for each half of the grid.

```
28         lb      $t2, 0($s0)
29         sll     $t3, $t2, 10    # mask 1: x
30         lb      $t2, 1($s0)
31         or      $t3, $t3, $t2   # x000000000x
32         sll     $t3, $t3, 10
33         lb      $t2, 2($s0)
34         or      $t3, $t3, $t2   # x000000000x000000000x
35         sllv    $t3, $t3, $s1   # shift based on argument
36         or      $v0, $t3, $s2   # add row to board
```

Finally, the previous `$s` register values are retrieved from the stack and the completed board is returned by the `return()`.

Overall, the entire function looks like the following.

```
# Adds a row to the board
# Arguments: shift amount, reg 1, reg 2
# Returns: the board registers with added row

build_row:
        save()

        la      $s0, cont        # buffer address
        move    $s1, $a0         # shift amount
        move    $s2, $a1         # reg 1
        move    $s3, $a2         # reg 2

        addi    $t0, $0, 0       # offset
br_loop:                         # do {
        add     $t1, $s0, $t0    # add offset
        lb      $t2, 0($t1)      # get character at offset

        bne     $t2, 0x23, br_isdot     # if '#', proceed
        addi    $t2, $0, 1
        j       br_isdot_
br_isdot:
        addi    $t2, $0, 0       # if not '#'
br_isdot_:
        sb      $t2, 0($t1)      # return parsed to memory
        addi    $t0, $t0, 1      # increment offset
        blt     $t0, 6, br_loop  # } while (offset < 6)

        lb      $t2, 0($s0)
        sll     $t3, $t2, 10     # mask 1: x
        lb      $t2, 1($s0)
        or      $t3, $t3, $t2    # x000000000x
        sll     $t3, $t3, 10
        lb      $t2, 2($s0)
        or      $t3, $t3, $t2    # x000000000x000000000x
        sllv    $t3, $t3, $s1    # shift based on argument
        or      $v0, $t3, $s2    # add row to board

        lb      $t2, 3($s0)
        sll     $t4, $t2, 10     # mask 2: x
        lb      $t2, 4($s0)
        or      $t4, $t4, $t2    # x000000000x
        sll     $t4, $t4, 10
        lb      $t2, 5($s0)
        or      $t4, $t4, $t2    # x000000000x000000000x
        sllv    $t4, $t4, $s1    # shift based on argument
        or      $v1, $t4, $s3    # add row to board

        return()
```

### 3.2.2 create_board

The purpose of this function is to create a grid. This is used to parse the inputs for the boards and the registers and convert them into a form usable by the rest of the program. It takes in just one argument, the board size (in this case either 6 or 4), and returns the final grid registers.

It starts off by first saving old `$s` register values to the stack and saving the board size to `$s0`.

```
create_board:
        save()

        move    $s0, $a0        # board size
```

For the first row, it reads `size + 2` bytes from the input file (`+2` for the `\r\n` newline). It sets the initial values for the grid which is empty at first and the shift counter which starts at `0` (the topmost row). Then, it calls `build_row` to build this first row.

```
        read_file($s0, 2)       # get a row from the input

        addi    $a0, $0, 0      # initial values are 0 for row 0
        addi    $a1, $0, 0
        addi    $a2, $0, 0
        jal     build_row       # build the first row
```

Afterwards, a loops is utilised for the rest of the board. In each iteration of the loop, the shift counter is incremented, and a new row is taken from the input file. The register outputs returned by `build_row` become the new inputs. This repeats until the counter exceeds the given size of the grid.

```
        addi    $s1, $0, 0      # counter
create_loop:                    # do {
        addi    $s1, $s1, 1     # increment counter for shifting
        move    $t0, $v0
        move    $t1, $v1

        read_file($s0, 2)       # get a row from the input

        move    $a0, $s1
        move    $a1, $t0        # use previous outputs as inputs
        move    $a2, $t1
        jal     build_row       # build a row using the new values

        addi    $t2, $s0, -1
        blt     $s1, $t2, create_loop   # } (while counter < size)
```

When this is all done, the final grid values are stored inside `$v0` and `$v1` from the `build_row` function. Old `$s` register values are retrieved from the stack and the final board is returned by the `return()`.

Overall, the entire function looks like the following.

```
# Creates a board
# Arguments: board size
# Returns: the board registers (implicitly via build_board_row)

create_board:
        save()

        move    $s0, $a0        # board size

        read_file($s0, 2)       # get a row from the input

        addi    $a0, $0, 0      # initial values are 0 for row 0
        addi    $a1, $0, 0
        addi    $a2, $0, 0
        jal     build_row       # build the first row

        addi    $s1, $0, 0      # counter
create_loop:                    # do {
        addi    $s1, $s1, 1     # increment counter for shifting
        move    $t0, $v0
        move    $t1, $v1

        read_file($s0, 2)       # get a row from the input

        move    $a0, $s1
        move    $a1, $t0        # use previous outputs as inputs
        move    $a2, $t1
        jal     build_row       # build a row using the new values

        addi    $t2, $s0, -1
        blt     $s1, $t2, create_loop   # } (while counter < size)

        return()
```

### 3.2.3 drop_piece

The purpose of this function is to drop a single piece into the board. It is used by the main recursive function to generate new boards. It takes in registers which represent the board and registers which represent the piece to drop. After the function, it returns a new board, with the piece (if it dropped) included.

The beginning of this function is for housekeeping and I will skip its explanation. The first crucial part is the declaration of the counter and the top mask. The counter counts the number of times the piece has dropped and is initialised to zero. Meanwhile, the top mask will be used later to determine whether a piece is out of bounds. A visualisation of this top mask is shown in Figure 3.1.

```
        addi    $t2, $0, 0        # number of rows dropped
        li      $t3, 0x00701C07   # top mask
```

```
# # # # # #
# # # # # #
# # # # # #
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 0000000111 | 0000000111 | 0000000111 |
| reg2 | 00 | 0000000111 | 0000000111 | 0000000111 |

Figure 3.1: The top mask `0x00701C07` defined in `drop_piece`.

Then, the program runs through a loop. In each step of the loop, it does three things. The first is to check for a collision. Since both board and piece have the same bitwise grid format, this step can be done by a simple bitwise AND of the corresponding registers. If it results to anything other than a `0` for either register, it means a collision has occurred. In that case, stop dropping and exit the loop.

On the other hand, if there are no collisions, drop the tetrimino one row. Since the grid is stored in column-major fashion, this is simply a task of shifting the piece register once to the left. Finally, increment the counter and then restart the loop.

This checking for collision logic to stop the loop is the design choice behind the row of `"#"` on bits 9, 19, and 29 of the board. Without this extra row, the pieces would drop ad infinitum. Adding this row solved that issue by essentially adding blocks at the bottom that pieces cannot pass through.

```
dp_loop:
        and     $t0, $s0, $s2     # check for collision
        and     $t1, $s1, $s3

        bnez    $t0, dp_stop      # stop if there's a collision
        bnez    $t1, dp_stop

        sll     $s2, $s2, 1       # drop the tetrimino one row
        sll     $s3, $s3, 1
        addi    $t2, $t2, 1
        j       dp_loop           # repeat until collision

dp_stop:
```

After the loop terminates, the next thing that is done is to check whether the tetrimino dropped at all. This is done by checking the counter against `0`. If it's `0`, it means the piece didn't drop. In that case, move the control to `dp_no_drop`. Otherwise, since the piece is still colliding with the board, move it up one row, which can be done by shifting the piece register once to the right.

```
dp_stop:
        beqz    $t2, dp_no_drop # if it didn't drop, return board
        srl     $s2, $s2, 1     # else move it back
        srl     $s3, $s3, 1
```

If the piece dropped, next thing to check is if it's out of the board. This is done by doing a bitwise AND with the top mask that was declared at the beginning of the function. If it results to anything other than a `0` for either register, it means the piece went out of the bounds of the board. In that case, move the control to `dp_no_drop`.

```
        and     $t4, $t3, $s2   # check for out of board bounds
        and     $t5, $t3, $s3

        bnez    $t4, dp_no_drop # if out of board, return board
        bnez    $t5, dp_no_drop
```

If the piece dropped and is not out of bounds, then the function was successful. Add the piece to the board and return from the function. Adding the piece to the board can be done by a bitwise OR, since we've already checked for collisions earlier.

```
        or      $v0, $s0, $s2   # return board with piece included
        or      $v1, $s1, $s3

        j       dp_exit
```

Meanwhile, if the piece didn't drop or is out of bounds, then the boards will be returned as they were.

```
dp_no_drop:
        move    $v0, $s0
        move    $v1, $s1
```

Overall, the entire function looks like the following. A simple step by step-by-step example of what this function does is found on Figures 3.2–3.3

```
# Drops a piece until it can no longer drop
# Arguments: board registers, block registers
# Returns: updated board registers

drop_piece:
        save()

        move    $s0, $a0        # board reg 1
        move    $s1, $a1        # board reg 2
        move    $s2, $a2        # block reg 1
        move    $s3, $a3        # block reg 2

        addi    $t2, $0, 0      # number of rows dropped
        li      $t3, 0x00701C07 # top mask
dp_loop:
        and     $t0, $s0, $s2   # check for collision
        and     $t1, $s1, $s3

        bnez    $t0, dp_stop    # stop if there's a collision
        bnez    $t1, dp_stop

        sll     $s2, $s2, 1     # drop the tetrimino one row
        sll     $s3, $s3, 1
        addi    $t2, $t2, 1
        j       dp_loop         # repeat until collision

dp_stop:
        beqz    $t2, dp_no_drop # if it didn't drop, return board
        srl     $s2, $s2, 1     # else move it back
        srl     $s3, $s3, 1

        and     $t4, $t3, $s2   # check for out of board bounds
        and     $t5, $t3, $s3

        bnez    $t4, dp_no_drop # if out of board, return board
        bnez    $t5, dp_no_drop

        or      $v0, $s0, $s2   # return board with piece included
        or      $v1, $s1, $s3

        j       dp_exit
dp_no_drop:
        move    $v0, $s0
        move    $v1, $s1
dp_exit:
        return()
```

### 3.2.4 shift_piece

Rather than have my offset code in the `drop_piece` function, I opted to have it as a separate function. This function takes in a piece and returns a grid of it shifted one column to the right.

```
. . . . . .       . . . . . .       . . . . . .
. . . . . .       . . . . . .       . . . . . .
@ @ . . . .       . . . . . .       . . . . . .

@ @ . . . .       @ @ . . . .       . . . . . .
. . . . . .       @ @ . . . .       @ @ . . . .
. . . . . .   →   . . . . . .   →   @ @ . . . .
# # . . . .       # # . . . .       # # . . . .
# . . . . .       # . . . . .       # . . . . .
# . . . . .       # . . . . .       # . . . . .

# # # # # #       # # # # # #       # # # # # #

. . . . . .       . . . . . .       . . . . . .
. . . . . .       . . . . . .       . . . . . .
. . . . . .       . . . . . .       . . . . . .

. . . . . .       . . . . . .       . . . . . .
. . . . . .       @ @ . . . .       # # . . . .
@ @ . . . .   →   @ @ . . . .   →   # # . . . .
X X . . . .       # # . . . .       # # . . . .
# . . . . .       # . . . . .       # . . . . .
# . . . . .       # . . . . .       # . . . . .

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

Figure 3.2: A successful `drop_piece`. It drops the O tetrimino until it collides, then moves it back up once.

After the housekeeping part, it starts by declaring a last column mask. As the name suggests, this creates a mask which has a `1` for bits in the last column. A visualisation of this mask is shown in Figure 3.4.

This mask is used in order to get the last column of the left register. Once that column is taken via bit-wise AND with the mask, it is moved to be in first column position by shifting left twice by 10 bits each time. This is because it will eventually become the first column of the right register.

```
        li      $s2, 0x3FF         # last column mask

        and     $t0, $s0, $s2      # get the last column of reg 1
        sll     $t0, $t0, 20       # shift it to be the first column
```

Then, both registers are moved once to the right. This is done by shifting them right once by 10 bits.

```
        srl     $s0, $s0, 10       # move one column over
        srl     $s1, $s1, 10
```

```
@ . . . . .      . . . . . .      . . . . . .
@ . . . . .      @ . . . . .      . . . . . .
@ . . . . .      @ . . . . .      @ . . . . .

@ . . . . .      @ . . . . .      @ . . . . .
. . . . . .      @ . . . . .      @ . . . . .
. . . . . .  →   . . . . . .  →   @ . . . . .
# # . . . .      # # . . . .      # # . . . .
# . . . . .      # . . . . .      # . . . . .
# . . . . .      # . . . . .      # . . . . .

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

```
. . . . . .      . . . . . .      . . . . . .
. . . . . .      . . . . . .      . . . . . .
. . . . . .      @ . . . . .      . . . . . .

@ . . . . .      @ . . . . .      . . . . . .
@ . . . . .      @ . . . . .      . . . . . .
@ . . . . .  →   @ . . . . .  →   . . . . . .
X # . . . .      # # . . . .      # # . . . .
# . . . . .      # . . . . .      # . . . . .
# . . . . .      # . . . . .      # . . . . .

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

Figure 3.3: An unsuccessful `drop_piece`. It drops the long bar until it collides, then moves it back up once. However, it goes out of bounds so the original board is returned

```
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
. . # . . .
```

| | | | | |
|---|---|---|---|---|
| `reg1` | 00 | 0000000000 | 0000000000 | 1111111111 |
| `reg2` | 00 | 0000000000 | 0000000000 | 0000000000 |

Figure 3.4: The column mask defined in `drop_piece`. `reg1` contains the last column mask.

Finally, the last column of the left register which we extracted earlier is added as the first column of the right register completing the intended function. The new registers are returned.

Old `$s` values in the stack are retrieved, and control is given back to the caller via the `return()` macro.

```
        or      $s1, $s1, $t0   # add last column of reg 1
                                # to first column of reg 2

        move    $v0, $s0
        move    $v1, $s1

        return()
```

Overall, the entire function looks like the following. A simple step by step-by-step example of what this function does is found on Figure 3.5

```
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. # # | # . .    . # @ | # . .    . . # | . # .    . . # | # # .
. . . | # . .    . . . | # . .    . . . | . # .    . . . | . # .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
. . . | . . .    . . . | . . .    . . . | . . .    . . . | . . .
```

Figure 3.5: `shift_piece` in action.

```
# Shifts a piece once to the right
# Arguments: block registers
# Returns: shifted block registers

shift_piece:
        save()

        move    $s0, $a0        # block reg 1
        move    $s1, $a1        # block reg 2

        li      $s2, 0x3FF      # last column mask

        and     $t0, $s0, $s2   # get last column of reg 1
        sll     $t0, $t0, 20    # shift it to be first column

        srl     $s0, $s0, 10    # move one column over
        srl     $s1, $s1, 10

        or      $s1, $s1, $t0   # add last column of reg 1
                                # to first column of reg 2

        move    $v0, $s0
        move    $v1, $s1

        return()
```

Fun fact: I wrote this function while waiting for HONNE May 2023 Asia Tour to start last May 10.

### 3.2.5 mama_mo_backtrack

The longest and quite possibly the most important function in the entire program, `mama_mo_backtrack`[1] ensures that all the permutations of dropping and offsets are tested. It takes in the board registers, as well as a chosen *array* (it's a register). After the whole lot of recursion, it returns a boolean whether the final board can be reached or not.

The first part of this function is once again housekeeping. It puts `$s` registers into the stack frame and saves the arguments into their respective `$s` registers. It also loads the final board representations and the total number of blocks from the global frame.

```
mama_mo_backtrack:
        save()

        move    $s0, $a0        # board reg 1
        move    $s1, $a1        # board reg 2
        move    $s2, $a2        # chosen registers

        lw      $s3, 16($gp)    # final board reg 1
        lw      $s4, 20($gp)    # final board reg 2
        lw      $s5, 4($gp)     # number of blocks
```

After retrieving the relevant values, it computes for the address of the last piece saved in the global frame. This is because the counter to be used later on will be the address in the global frame of the current piece. This part also sets an initial value to `$s6` (the return value) to `0`.

```
        sll     $s5, $s5, 3     # compute for the end of the
                                # blocks for use as counter
        addi    $s5, $s5, 24    # ($s5 * 8) + 24

        addi    $s6, $0, 0      # result
```

Before beginning any recursion, however, we first need to check whether the passed board is already equal to the final board, which is done by the following lines. If they are equal, automatically return `1` and exit.

```
        bne     $s0, $s3, bt_ne1        # if not equal, go to the
        bne     $s1, $s4, bt_ne1        # rest of the function
        addi    $v0, $0, 1              # else, return 1
        j       bt_exit
bt_ne1:
```

[1] Mind not the name but the essence of the function

Now we can begin initialising the first loop. In this case `$t0` is the chosen registers tracker. It acts as a mask which points to the current register in the chosen register array. If the bit it points to is `0`, then that specific register has not been added to the board yet. Meanwhile `$t1` is the offset for global pointer. This stores the amount of offset for the current piece to drop.

```
        addi    $t0, $0, 1      # chosen registers tracker mask
        addi    $t1, $0, 24     # gp offset for blocks
```

Then, we enter the main loop of the function. The purpose of this loop is to run through each of the pieces still available and set them as the current piece to drop. It first checks whether a register's already been chosen. If it hasn't, then the offset from `$t1` is added to `$gp` to retrieve the corresponding registers.

```
bt_main_loop:
        and     $t2, $t0, $s2   # continue if already chosen
        bnez    $t2, bt_continue

        add     $t3, $gp, $t1   # gp location for block regs

        lw      $t4, 0($t3)     # get block reg 1
        lw      $t5, 4($t3)     # get block reg 2
```

After a piece is selected, we enter another loop, this time for the possible offsets of the piece in the grid.

The first part of this loop drops the current piece with its offset. It requires the use of the `save_temp()` and `return_temp()` macros as we need the values of the `$t` registers to persist after the call.

```
bt_shift_loop:
        move    $a0, $s0        # drop the current piece
        move    $a1, $s1
        move    $a2, $t4
        move    $a3, $t5
        save_temp()
        jal     drop_piece
        return_temp()
```

The returned grid with the dropped piece is then checked against the previous board (before it dropped). If they match, it means the piece didn't drop. So, we continue to the next piece. Otherwise, it enters the recursive phase.

```
        bne     $v0, $s0, bt_recurse    # if next grid != curr grid,
        bne     $v1, $s1, bt_recurse    # continue with recursion
        j       bt_sl_continue          # else, continue
```

The recursive phase starts by marking the register as already chosen. This is done using a bit-wise OR with the register mask. Then, we feed the updated board and chosen array to another call of `mama_mo_backtrack`. It requires the use of the `save_temp()` and `return_temp()` macros as we need the values of the `$t` registers to persist after the call.

```
bt_recurse:
        or      $t2, $t0, $s2   # add register to chosen

        move    $a0, $v0        # recurse with the next grid
        move    $a1, $v1        # and chosen registers
        move    $a2, $t2
        save_temp()
        jal     mama_mo_backtrack
        return_temp()
```

Once control returns back to the current function, we check whether any state of the board which spanned from the current state matched the final board. If it did, automatically return `1` and exit. Otherwise, continue to the next offset.

```
        or      $s6, $s6, $v0           # result or backtrack()
        bnez    $s6, bt_return_true     # if result, return true
        j       bt_sl_continue          # else, continue

bt_return_true:
        addi    $v0, $0, 1      # return true
        j       bt_exit
```

If the current piece with the current offset did not span a match, then we need to check the next offset. This is initiated by first declaring another column mask and determining whether the current piece can still shift. If the bit-wise AND of the mask with the right register is not `0`, then there is a block in the last column and the piece can no longer shift, which ends this offset loop.

```
bt_sl_continue:
        andi    $t6, $t5, 0x3FF # last column mask

        bnez    $t6, bt_continue# if can no longer shift, end loop
```

If, however, the piece can still be shifted, it gets shifted once to the right via `shift_piece`. It requires the use of the `save_temp()` and `return_temp()` macros as we need the values of the `$t` registers to persist after the call. The shifted piece will now become the new current piece. Jump back to the start of the offset loop.

```
        move    $a0, $t4        # shift once
        move    $a1, $t5
        save_temp()
        jal     shift_piece
        return_temp()
        move    $t4, $v0
        move    $t5, $v1

        j       bt_shift_loop
```

Once all offsets of the current piece is exhausted, we can now move on to the next piece. This is done by shifting the chosen register tracker once to the left and by adding 2 words to the `$gp` offset. If the pieces haven't all been visited yet, loop from the very beginning. Otherwise, continue to exit.

```
bt_continue:
        sll     $t0, $t0, 1     # move to next register
        addi    $t1, $t1, 8
        bne     $t1, $s5, bt_main_loop
```

If all pieces and possible offsets have been exhausted, then the function terminates. A `0` is returned, signifying that the function did not find anything that matches the final board.

```
        addi    $v0, $0, 0
bt_exit:
        return()
```

Taken altogether, the following code block becomes the backbone of this program.

```
# The bread and butter of this whole program
# Arguments: board registers, chosen
# Returns: boolean if board can be reached

mama_mo_backtrack:
        save()

        move    $s0, $a0        # board reg 1
        move    $s1, $a1        # board reg 2
        move    $s2, $a2        # chosen registers

        lw      $s3, 16($gp)    # final board reg 1
```

```
        lw      $s4, 20($gp)    # final board reg 2
        lw      $s5, 4($gp)     # number of blocks

        sll     $s5, $s5, 3     # compute for the end of the
                                # blocks for use as counter
        addi    $s5, $s5, 24    # ($s5 * 8) + 24

        addi    $s6, $0, 0      # result

        bne     $s0, $s3, bt_ne1        # if not equal, go to the
        bne     $s1, $s4, bt_ne1        # rest of the function
        addi    $v0, $0, 1              # else, return 1
        j       bt_exit
bt_ne1:
        addi    $t0, $0, 1      # chosen registers tracker mask
        addi    $t1, $0, 24     # gp offset for blocks
bt_main_loop:
        and     $t2, $t0, $s2   # continue if already chosen
        bnez    $t2, bt_continue

        add     $t3, $gp, $t1   # gp location for block regs

        lw      $t4, 0($t3)     # get block reg 1
        lw      $t5, 4($t3)     # get block reg 2

bt_shift_loop:
        move    $a0, $s0        # drop the current piece
        move    $a1, $s1
        move    $a2, $t4
        move    $a3, $t5
        save_temp()
        jal     drop_piece
        return_temp()

        bne     $v0, $s0, bt_recurse    # if next grid != curr grid,
        bne     $v1, $s1, bt_recurse    # continue with recursion
        j       bt_sl_continue          # else, continue

bt_recurse:
        or      $t2, $t0, $s2   # add register to chosen

        move    $a0, $v0        # recurse with the next grid
        move    $a1, $v1        # and chosen registers
        move    $a2, $t2
        save_temp()
        jal     mama_mo_backtrack
        return_temp()

        or      $s6, $s6, $v0           # result or backtrack()
        bnez    $s6, bt_return_true     # if result, return true
        j       bt_sl_continue          # else, continue

bt_return_true:
        addi    $v0, $0, 1      # return true
        j       bt_exit

bt_sl_continue:
```

```
        andi    $t6, $t5, 0x3FF # last column mask

        bnez    $t6, bt_continue# if can no longer shift, end loop

        move    $a0, $t4        # shift once
        move    $a1, $t5
        save_temp()
        jal     shift_piece
        return_temp()
        move    $t4, $v0
        move    $t5, $v1

        j       bt_shift_loop
bt_continue:
        sll     $t0, $t0, 1     # move to next register
        addi    $t1, $t1, 8
        bne     $t1, $s5, bt_main_loop

        addi    $v0, $0, 0
bt_exit:
        return()
```

### 3.2.6 main

This is the entry point to the program. As such, it does not require any housekeeping for `$s` registers.

It begins by first storing two important registers: the buffer address and the bottom mask. The buffer address is where `syscall 14` stores the values that it reads from the input file. The bottom mask is a mask which, as the name explains, contains the bits which make up the bottom row. This will be used later to populate the bottom row with blocks for the board. A visualisation of this bottom mask is shown in Figure 3.6.

```
main:
        la      $s0, cont       # save the buffer address
        li      $s6, 0x20080200 # magic number
```

Inputs are then read. The first ones are the initial and the final boards. A size of `6` is fed into the `create_board` function. Then, the resulting registers are shifted thrice to the left to add the extra three rows found at the top of each board.

Afterwards, the bottom row of blocks is added via bit-wise OR of the bottom mask. Finally, the resulting registers are stored into their respective address in the global frame. For the initial board, these offsets are `8` and `12`, while they are `16` and `20` for the final board.

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
# # # # # #
```

| | | | | |
|---|---|---|---|---|
| reg1 | 00 | 1000000000 | 1000000000 | 1000000000 |
| reg2 | 00 | 1000000000 | 1000000000 | 1000000000 |

Figure 3.6: The bottom mask `0x20080200` defined in `drop_piece`.

```
        addi    $a0, $0, 6
        jal     create_board

        sll     $v0, $v0, 3
        sll     $v1, $v1, 3
        or      $v0, $v0, $s6
        or      $v1, $v1, $s6
        sw      $v0, 8($gp)     # initial board reg 1
        sw      $v1, 12($gp)    # initial board reg 2
```

After the boards, the number of pieces is taken. Three bytes are read from the input file for this: the ASCII value of the number followed by `\r\n`. The ASCII value is converted to its numerical value by subtracting 48, and this value is stored to its allocated address in the global frame.

```
        read_file($0, 3)
        lb      $s5, 0($s0)     # get number of blocks
        addi    $s5, $s5, -48   # ascii to num conversion
        sw      $s5, 4($gp)
```

Then, we enter a loop. This loops runs for as many times as there are pieces according to the previous input. In each iteration, `create_board` is called with size `4`. The resulting registers are then saved to their respective addresses in the global frame.

```
m_block_loop:
        addi    $s5, $s5, -1    # counter for blocks

        addi    $a0, $0, 4
        jal     create_board
        add     $t0, $gp, $s6   # global address for block
        sw      $v0, ($t0)      # save block reg 1 in global data
        addi    $t0, $t0, 4
        sw      $v1, ($t0)      # save block reg 2 in global data
```

```

        addi    $s6, $s6, 8      # increment global address

        bnez    $s5, m_block_loop
```

Once all the inputs have been parsed, it's time to actually begin the solver. We call `mama_mo_backtrack` and feed it the initial board and the initial chosen array, which is `0`.

```
        lw      $a0, 8($gp)
        lw      $a1, 12($gp)
        move    $a2, $0

        jal     mama_mo_backtrack
```

After backtracking, the output is ready to be printed. It checks the return value against `0`. If it's `0`, then the recursion terminated without finding a match. It then loads `"NO"` from memory to be printed by **`syscall 4`**. Otherwise, the recursion found a match and loads **`"YES"`** from memory to be printed by the same syscall.

The program finally terminates.

```
        bnez    $v0, m_yes
        la      $a0, no
        j       m_print
m_yes:
        la      $a0, yes
m_print:
        addi    $v0, $0, 4
        syscall
exit:
        addi    $v0, $0, 10
        syscall          # exit
```

Overall, the main entry point is the following code block.

```
main:
        la      $s0, cont        # save the buffer address
        li      $s6, 0x20080200  # magic number

        # INPUTS #
        addi    $a0, $0, 6
        jal     create_board

        sll     $v0, $v0, 3
        sll     $v1, $v1, 3
        or      $v0, $v0, $s6
        or      $v1, $v1, $s6
```

```
        sw      $v0, 8($gp)     # initial board reg 1
        sw      $v1, 12($gp)    # initial board reg 2

        addi    $a0, $0, 6
        jal     create_board

        sll     $v0, $v0, 3
        sll     $v1, $v1, 3
        or      $v0, $v0, $s6
        or      $v1, $v1, $s6
        sw      $v0, 16($gp)    # final board reg 1
        sw      $v1, 20($gp)    # final board reg 2

        read_file($0, 3)
        lb      $s5, 0($s0)     # get number of blocks
        addi    $s5, $s5, -48   # ascii to num conversion
        sw      $s5, 4($gp)

        addi    $s6, $0, 24     # gp offset for blocks
m_block_loop:
        addi    $s5, $s5, -1    # counter for blocks

        addi    $a0, $0, 4
        jal     create_board
        add     $t0, $gp, $s6   # global address for block
        sw      $v0, ($t0)      # save block reg 1 in global data
        addi    $t0, $t0, 4
        sw      $v1, ($t0)      # save block reg 2 in global data

        addi    $s6, $s6, 8     # increment global address

        bnez    $s5, m_block_loop

        # START BACKTRACK
        lw      $a0, 8($gp)
        lw      $a1, 12($gp)
        move    $a2, $0

        jal     mama_mo_backtrack

        # OUTPUT
        bnez    $v0, m_yes
        la      $a0, no
        j       m_print
m_yes:
        la      $a0, yes
m_print:
        addi    $v0, $0, 4
        syscall
exit:
        addi    $v0, $0, 10
        syscall         # exit
```