CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #3

Fall 2019

Assignment Due: Friday, November 15, 2019 by 11:59 pm

Updates to the Document:

- 11/1/2019: changed first line of pseudocode in Part IX (if-statement)
- 10/29/2019: Under "Data Structures" I have clarified how the O-piece and I-piece should be represented when rotated.

Learning Outcomes

After completion of this programming project you should be able to:

- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly register conventions.
- Implement algorithms that process structs and 2D arrays of bytes.
- Read files from disk and build data structures based on their contents.

Getting Started

Visit Piazza and download the file `proj3.zip`. Decompress the file and then open `proj3.zip`. Fill in the following information at the top of `proj3.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `proj3.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `proj3.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `proj3.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- **You must use the Stony Brook version of MARS posted on Blackboard.** Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement the programming projects in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Do not submit a file with the function/label `main` defined. You are also not permitted to start your label names with two underscores (__). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- Each test case must execute in 1,000,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There really is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best, a form of laziness, so don't do it. We will comment out any such code we find.

How to Test Your Functions

To test your implementated functions, open the provided `main` files in MARS. Next, assemble the `main` file and run it. MARS will include the contents of any `.asm` files referenced with the `.include` directive(s) at the end of the file and then add the contents of your `proj3.asm` file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases. To test each of your functions thoroughly, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s). Do not submit your main files to Blackboard – we will delete them.

Again, any modifications to the `main` files will not be graded. You will submit only your `proj3.asm` for grading. Make sure that all code required for implementing your functions is included in the `proj3.asm` file. To make sure that your code is self-contained, try assembling your `proj3.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in a `main` file (e.g., a helper function) to `proj3.asm`.

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Welcome to MIPS Tetris!

In this assignment you will be implementing a collection of functions to support the implementation of a version of Tetris, the classic video game from the 1980s. These functions depend on two data structures, described next.

Data Structures

In this assignment, you will familiarize yourself with a feature of the C programming language known as the **struct**. A struct is a composite data type that is used to group variables under one name in a contiguous block of memory.

```
struct GameState {
    unsigned byte num_rows; // byte #0 of the struct
    unsigned byte num_cols; // byte #1 of the struct
    char[][] field;         // bytes #2, #3, ... of the struct
}

// represents a Tetris game piece -- always 8 bytes in size
struct Piece {
    unsigned byte num_rows; // byte #0 of the struct
    unsigned byte num_cols; // byte #1 of the struct
    char[][] blocks;        // bytes #2, #3, ... , #7 of the struct
}
```

The `GameState` struct represents the state of the game *field* during a particular snapshot in time. A capital letter `'O'` represents an occupied "slot" in the game field, whereas a period, `'.'` represents an empty slot. The attributes `num_rows` and `num_cols` record the dimensions of the field. Rows are numbered `0` through `num_rows-1`, where row #`0` is the topmost row in the field. Columns are numbered `0` through `num_cols-1`, left to right. The characters inside the `field` attribute are stored in row-major-order. For example, the 6×8 game field depicted below

```
........
...O...O
...OO.OO
```

```
.OOOO.OO
.OOOOOOO
OOOOOOOO
```

would be represented by the following 50-byte struct. Quotation marks around the `'O'` and `'.'` characters have been omitted for clarity:

```
6 8 ..........O...O...OO.OO.OOOO.OO.OOOOOOOOOOOOOOO
```

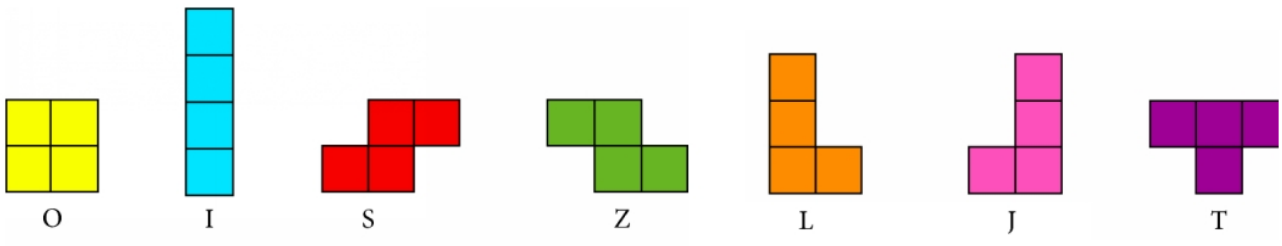
A Tetris game piece ("tetromino") is represented by the `Piece` data type and uses exactly 6 bytes for the `blocks` array. So, for example, the T-shaped piece depicted below

```
OOO
.O.
```

would be stored in the 8-byte struct as `2 3 'O' 'O' 'O' '.' 'O' '.'`. The blocks that make up the piece are stored in row-major-order inside the attribute called `blocks`.

There are seven tetromino shapes: O, I, S, Z, L, J, T. Shown below is how each is represented as a small 2D grid (with some rotated 90° clockwise one or more times).

```
OO    OOOO   .OO    OO.    ..O    O..    OOO
OO           OO.    .OO    OOO    OOO    .O.
```



Even though the O-piece and I-piece could conceivably be stored using only 6 bytes total per struct, our struct will always use 8 bytes to represent a game piece. Below are examples of how these structs are stored in MIPS:

```
# T piece
.byte 2
.byte 3
.ascii "OOO.O."
```

```
# O piece
.byte 2
.byte 2
.ascii "OOOO.."
```

Note that both data types (`GameState` and `Piece`) are fundamentally the same: two 1-byte integers followed by an array of characters. This means that for several of the functions below (specifically, `initialize`, `get_slot` and `set_slot`) you will see two function prototypes for each function, but keep in mind that you will implement only one version of each function that will work properly for both data types.

Added October 29, 2019:

Please note the following. The O-piece is always represented as follows, regardless of how many times it has been

rotated:

```
2 2 "OOOO.."
```

and I-piece has only two valid representations:

```
4 1 "OOOO.."
1 4 "OOOO.."
```

The following are **incorrect** representations of the I-piece:

```
4 1 "..OOOO"
1 4 "..OOOO"
```

Part I: Initialize the Game State

```
int, int initialize(GameState* state, int num_rows, int num_cols,
                    char character)
int, int initialize(Piece* Piece,     int num_rows, int num_cols,
                    char character)
```

This function takes the address of an uninitialized, but allocated, region of memory and sets the `num_rows` and `num_cols` fields of a struct that will be stored in that region of memory. Starting at byte #2 of the struct, it fills the following `num_rows*num_cols` bytes with the value `character`. The notation `GameState*` represents a *pointer* (i.e., address) to a `GameState` struct. We are borrowing this syntax from the C language.

The function may assume that sufficient memory has been allocated to store `num_rows*num_cols` instances of `character` in memory. In cases where an invalid argument has been provided (specifically, `num_rows` or `num_cols` is invalid), the function must make no changes whatsoever to memory.

The function takes the following arguments, in this order:

- `state` or `piece`: the address of a uninitialized `GameState` or `Piece` struct, respectively
- `num_rows`: the value to be written in the `num_rows` attribute of the struct referenced by the first argument
- `num_cols`: the value to be written in the `num_cols` attribute of the struct referenced by the first argument
- `character`: the value to be written in the `num_rows*num_cols` bytes starting at offset 2 with respect to the starting address of the struct

Returns in `$v0`:

- `num_rows` if `num_rows` > 0, or -1 on error

Returns in `$v1`:

- `num_cols` if `num_cols` > 0, or -1 on error

Additional requirements:

- The function must not write any changes to main memory except as required by the function.

Example #1 (initializing a `GameState` struct):

Function call: `initialize(pointer, 8, 6, 'x')`

Expected return values: `8, 6`

Expected contents of `field`: `"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"`

Example #2 (initializing a `Piece` struct):

Function call: `initialize(pointer, 2, 3, '.')`

Expected return values: `2, 3`

Expected contents of `blocks`: `"....."`

Example #3 (invalid arguments):

Function call: `initialize(pointer, -4, 7, 'A')`

Expected return values: `-1, -1`

Expected contents of `field/blocks`: unchanged from initial contents, whatever those happened to be

Part II: Load a Saved Game from Disk

`int, int load_game(GameState* state, string filename)`

The `load_game` function attempts to read the contents of the file named `filename`, formatted as described below, and uses the contents to initialize the given (uninitialized) `GameState` struct:

```
# of rows (always 1 or 2 digits)
# of columns (always 1 or 2 digits)
contents of the game field (O or . characters normally)
```

The number of rows and the number of columns will always be in the range $2-99$, given as one-digit or two-digit numbers, as appropriate (e.g., `8`, `24`). For values in the range $2-9$ *no leading zero is provided*. The number of characters for the game field will always be exactly `num_rows * num_cols`. The newline characters at the end of each line are not included in this count.

As an example, below is the contents of `game1.txt`. The game file contains 72 `'O'` characters and 5 invalid characters.

```
16
10
```

```
..........
..........
..........
..........
..........
..........
O.O.OO.OO.
OOOO.O..OO
OO.OOO.OCO
OO.OOOOO.O
OOAOOO.OOO
OO.OOO..OO
OO.OBOO.OO
ODFOOOOO.O
O.O.OOO.OO
OOOOO.OOO.
```

The function updates the attributes `num_rows` and `num_cols` based on what is given in the file. As the function reads the contents of the game field provided in the file, it should expect to see only `'O'` and `'.'` characters. Each such valid character is written directly into the `field` grid. Whenever the function encounters a character not equal to one of these, it writes a `'.'` into the corresponding spot instead of the invalid character. So, for example, for the sample file shown above, the 2D `state.field` would be initialized as follows:

```
..........
..........
..........
..........
..........
..........
O.O.OO.OO.
OOOO.O..OO
OO.OOO.O.O
OO.OOOOO.O
OO.OOO.OOO
OO.OOO..OO
OO.O.OO.OO
O..OOOOO.O
O.O.OOO.OO
OOOOO.OOO.
```

Note how the 5 invalid characters in the file have been replaced with periods in memory. Also, no newline characters are stored in `state.field` and no memory will be provided to store them. The function may assume that exactly `2 + num_rows * num_cols` bytes have been allocated to store the `GameState` struct. The two bytes that indicate the number of rows and columns *must* be stored at bytes 0 and 1 of the struct, respectively.

To assist with reading and writing files, MARS has several system calls:

Service	Code in `$v0`	Arguments	Results
open file	13	`$a0` = address of null-terminated file-name string `$a1` = flags `$a2` = mode	`$v0` contains file descriptor (negative if error)
read from file	14	`$a0` = file descriptor `$a1` = address of input buffer `$a2` = maximum # of characters to read	`$v0` contains # of characters read (0 if end-of-file, negative if error)
close file	16	`$a0` = file descriptor	

Service 13: MARS implements three *flag* values: 0 for read-only, 1 for write-only with create, and 9 for write-only with create and append. It ignores *mode*. The returned file descriptor will be negative if the operation failed. MARS maintains file descriptors internally and allocates them starting with 3. File descriptors 0, 1 and 2 are always open for reading from standard input, writing to standard output, and writing to standard error, respectively. An example of how to use these syscalls can be found on the MARS syscall web page.

The function must assume that every line of the file ends only with a `'\n'` character, *not* the two-character combination `"\r\n"` employed in Microsoft Windows. If you create your own files for testing purposes, use MARS to edit the files. If you are developing on a Windows computer, do not use a regular text editor like Notepad. Such an editor will insert both endline characters. In contrast, MARS will insert only a `'\n'` at the end of each line, *so only use MARS to create custom game files*.

Read the contents of the file *one character at a time* using system call #14. This system call requires a memory buffer to hold the character read from disk. You should allocate one byte of memory on the stack (by adjusting `$sp`) to store that byte temporarily. Discard newline characters as you read them and do not store them in the `GameState` struct. Finally, remember to reset `$sp` once you have finished reading the file contents and to close the file with system call #16.

MARS can be a little buggy when it comes to opening files. Therefore, either:

- put all your .asm and game files in the same directory as the MARS .jar file, or
- use absolute path names when giving the filename in your testing mains.

The function takes the following arguments, in this order:

- `state`: a pointer to an uninitialized `GameState` struct that the function will initialize based on the contents of a saved-game file
- `filename`: the name of the file containing the state of a game of Tetris

Returns in `$v0`:

- the number of O's read from the file, or `-1` if a file of the given name does not exist

Returns in `$v1`:

- the number of invalid characters read from the file , or `-1` if a file of the given name does not exist

Additional requirements:

- The function must not write any changes to main memory except as required by the function.

Part III: Get the Character in a Particular Slot in the Game Field

```
char get_slot(GameState* state, int row, int col)
char get_slot(Piece* piece,     int row, int col)
```

This function returns the character at `state.field[row][col]` or `piece.blocks[row][col]`, depending on whether the first argument to the function points to a `GameState` or `Piece` struct. Note that because both structs have the same underlying structure, the function doesn't need to "know" which type of struct it is processing. If `row` is outside the valid range or `col` is outside the valid range for the given `GameState` or `Piece` struct, the function returns `-1`. For instance, for a `GameState` struct, the valid range for the `row` argument would be $[0, \texttt{state.num\_rows})$.

The function takes the following arguments, in this order:

- `state` or `piece`: the address of a `GameState` or `Piece` struct, respectively
- `row`: the row of the `field` array (for a `GameState` struct) or `blocks` array (for a `Piece` struct) from where we want to read a character
- `col`: the column of the `field` array (for a `GameState` struct) or `blocks` array (for a `Piece` struct) from where we want to read a character

Returns in `$v0`:

- the character located at `state.field[row][col]` or `piece.blocks[row][col]`, as appropriate, or `-1` for the error condition explained above

Additional requirements:

- The function must not write any changes to main memory.

Example #1 (getting a value from a `GameState` struct):

Function arguments:

Struct loaded from `game2.txt`:

```
6
10
..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO

row = 3
```

```
col = 4
```

Return value: `46` (ASCII value of the period character)

Example #2 (invalid arguments: `row` $\geq$ `num_rows`):

Function arguments:

Struct loaded from `game2.txt`:

```
6
10
..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO

row = 3
col = 11
```

Return value: `-1`

Example #3 (invalid arguments: `row` $\geq$ `num_rows`):

Function arguments:

O-piece (visualized in 1-D):

```
2
2
OOOO..

row = 0
col = 2
```

Return value: `-1`

Part IV: Set the Character in a Particular Slot in the Game Field

```
char set_slot(GameState* state, int row, int col, char character)
char set_slot(Piece* piece,     int row, int col, char character)
```

This function changes the character at `state.field[row][col]` or `piece.blocks[row][col]` to `character`, depending on whether the first argument to the function is a `GameState` or `Piece` struct. Note that because both structs have the same underlying structure, the function doesn't need to "know" which type of

struct it is processing. If `row` is outside the valid range or `col` is outside the valid range for the given `GameState` or `Piece` struct, the function returns `-1` and makes no changes to main memory. For instance, for a `GameState` struct, the valid range for the `row` argument would be $[0, \texttt{state.num\_rows})$. Otherwise, the function returns `character`. The function does not validate the value of `character` (i.e., it should not check if the character is `'.'` or `'O'` before writing it).

The function takes the following arguments, in this order:

- `state` or `piece`: the address of a `GameState` or `Piece` struct, respectively
- `row`: the row of the `field` array (for a `GameState` struct) or `blocks` array (for a `Piece` struct) where we want to write a character
- `col`: the column of the `field` array (for a `GameState` struct) or `blocks` array (for a `Piece` struct) where we want to write a character
- `character`: the character to be written

Returns in `$v0`:

- `character` if `row` is a valid row number for the given `GameState` or `Piece` struct and `col` is a valid column number for the struct. Otherwise, `-1` for the error condition explained above.

Additional requirements:

- The function must not write any changes to main memory except as required by the function.

Example #1 (getting a value from a `GameState` struct):

Function arguments:

Struct loaded from `game2.txt`:

```
6
10
..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO
```

```
row = 3
col = 4
character = 'X'
```

Return value: `88` (ASCII value of the `X` character)

Updated state of the struct:

```
6
10
..........
..........
OOO.......
OOOOX....O
OOOOO....O
OOOOOO..OO
```

Example #2 (invalid arguments: `row` $\geq$ `num_rows`):

Function arguments:

Struct loaded from `game2.txt`:

```
6
10
..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO

row = 9
col = 4
character = 'X'
```

Return value: `-1`. The contents of the struct remain unchanged.

Part V: Rotate a Game Piece

```
int rotate(Piece* piece, int rotation, Piece* rotated_piece)
```

This function rotates a game piece, given in the argument `piece`, saving the rotated piece in `rotated_piece`. The original value of `piece` remains unchanged. The value `rotation` indicates the number of times that the piece should be rotated $90°$ clockwise. The function returns `rotation` if the rotation was successful, and returns `-1` if `rotation` is negative. Examples below will help to clarify what the function does.

The function takes the following arguments, in this order:

- `piece`: the address of a `Piece` struct
- `rotation`: the number of $90°$ clockwise turns that should be performed
- `rotated_piece`: the address of a buffer to write the rotated version of `piece`

Returns in `$v0`:

- `rotation` if `rotation` ≥ 0, or `-1` on error

Additional requirements:

- The function must call `initialize`, `get_slot` and `set_slot`.
- The function must not write any changes to main memory except as required by the function.

Example #1 (rotating a T piece $90°$ clockwise one time):

Function arguments:

```
piece = 2 3 "OOO.O."
rotation = 1
rotated_piece = "HUJ2na6x"   (8 bytes of uninitialized memory)
```

`piece.blocks` visualized as a 2D structure:

```
OOO
.O.
```

Contents of `rotated_piece` after the function call:

```
3 2 ".OOO.O"
```

`rotated_piece.blocks` visualized as a 2D structure:

```
.O
OO
.O
```

Return value: `1`

Example #2 (rotating an I piece $90°$ clockwise three times):

Function arguments:

```
piece = 1 4 "OOOO.."
rotation = 3
rotated_piece = "LKJmy4h2"   (8 bytes of uninitialized memory)
```

`piece.blocks` visualized as a "2D" structure:

```
OOOO..
```

Contents of `rotated_piece` after the function call:

```
4 1 "OOOO.."
```

`rotated_piece.blocks` visualized as a 2D structure:

```
O
O
O
O
```

Return value: `3`

Example #3 (rotating a J piece 90° clockwise 13 times):

Function arguments:

```
piece = 3 2 ".O.OOO"
rotation = 13
rotated_piece = "97h;12!f"   (8 bytes of uninitialized memory)
```

`piece.blocks` visualized as a 2D structure:

```
.O
.O
OO
```

Contents of `rotated_piece` after the function call:

```
2 3 "O..OOO"
```

`rotated_piece.blocks` visualized as a 2D structure:

```
O..
OOO
```

Return value: `13`

Example #4 (invalid value for `rotation`):

Function arguments:

```
piece = 3 2 ".O.OOO"
rotation = -5
rotated_piece = "GB8uJnHG"   (8 bytes of uninitialized memory)
```

`piece.blocks` visualized as a 2D structure:

```
.O
.O
OO
```

Contents of `rotated_piece` after the function call (unchanged):

```
71 66 "8uJnHG" (71 and 66 are the ASCII codes of G and B)
```

Return value: `-1`

Part VI: Count How Many Blocks Overlap Between a Game Piece and the Game Field

`int count_overlaps(GameState* state, int row, int col, Piece* piece)`

This function takes a game piece and determines how many of its non-empty blocks overlap with non-empty blocks in `state.field`. The overall purpose of the function is to help determine whether the game piece is in a valid position within the game field. This function will be used later to help simulate dropping a piece into the game field and to determine when it will stop (i.e., it has collided with pieces already in the game board). The indices `row` and `col` provide the the position of the upper-left corner of `piece.blocks` inside of `state.field`. If these coordinates place any element of the `piece.blocks` array outside the bounds of the game field, the function returns `-1`. Otherwise, the function returns the number of blocks of `piece.blocks` that overlap with `state.field`. This number must be in the range $0 - 4$. Note that the contents of `state` and `piece` must remain unchanged.

The function takes the following arguments, in this order:

- `state`: the address of a `GameState` struct
- `row`: the row of the topmost block of the piece
- `col`: the column of the leftmost block of the piece
- `piece`: the address of a `Piece` struct

Returns in `$v0`:

- The number of blocks of `piece.blocks` that overlap with `state.field`, or `-1` on error

Additional requirements:

- The function must call `get_slot`.
- The function must not write any changes to main memory.

Examples:

In the figures below, an `X` is used to visualize those positions where the game piece overlaps with the game field.

Example #1 (no overlapping blocks):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
```

```
  OOOOOO..OO

row = 1
col = 4

piece.blocks:
  OOO
  O..
```

Visualization of the piece placed into the game field:

```
  ..........
  ....OOO...
  OOO.O.....
  OOOO.....O
  OOOOO....O
  OOOOOO..OO
```

Return value: `0`

Example #2 (some overlapping blocks):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = 3
col = 2

piece.blocks:
  OOO
  O..
```

Visualization of the piece placed into the game field:

```
  ..........
  ..........
  OOO.......
  OOXXO....O
  OOXOO....O
  OOOOOO..OO
```

Return value: `3`

Example #3 (some overlapping blocks):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = 4
col = 0

piece.blocks:
  OOO
  O..
```

Visualization of the piece placed into the game field:

```
  ..........
  ..........
  OOO.......
  OOOO.....O
  XXXOO....O
  XOOOOO..OO
```

Return value: `4`

Example #4 (invalid placement: part of the piece is outside the bounds of the game field):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = 5
col = 2
```

```
piece.blocks:
  OOO
  O..
```

Return value: `-1`

Example #5 (invalid placement: part of the piece is outside the bounds of the game field):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = 2
col = 8

piece.blocks:
  .OO
  OO.
```

Return value: `-1`

Example #6 (invalid row number):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = -3
col = 2

piece.blocks:
  OOO
  O..
```

Return value: `-1`

Example #7 (invalid column number):

Function arguments (`GameState` struct loaded from `game2.txt`):

```
state.field:
  ..........
  ..........
  OOO.......
  OOOO.....O
  OOOOO....O
  OOOOOO..OO

row = 3
col = 19

piece.blocks:
  OOO
  O..
```

Return value: `-1`

Part VII: Drop a Piece into the Game Field

```
int drop_piece(GameState* state, int col, Piece* piece, int rotation,
               Piece* rotated_piece)
```

This function drops a rotated piece into a game field, updating the game field when the piece stops falling. If the value of `rotation` is negative, the `drop_piece` function returns `-2`. Similarly, if `col` negative or `col` $\geq$ `state.num_cols`, the function returns `-2`.

If the arguments pass these validation checks, the function rotates the piece 90° clockwise `rotation` times by calling the `rotate` function with suitable arguments so that the rotated piece is stored in `rotated_piece`. The original contents of `piece` remain unchanged.

The `col` argument provides the position of the rotated game piece relative to the game field. For example, `col = 4` indicates that the leftmost non-empty block of rotated piece is aligned with column #4 of the game field . Below is shown an example.

Original `piece`, before rotation:

```
.O
OO
.O
```

Suppose `rotation = 3`. The rotated piece, stored in the `rotated_piece` argument, will contain:

```
OOO
.O.
```

Suppose that `col = 2`. The rotated piece will be aligned with the game field (from `game2.txt`) as shown below:

```
  OOO
  .O.

..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO
```

Now, it is possible that rotation of a piece causes it to "poke out" the right-hand side of the game board, which means that it cannot be dropped into the game field. An example of this is shown below.

Original `piece`, before rotation (two additional bytes of the `blocks` attribute omitted for clarity):

```
O
O
O
O
```

Suppose `rotation = 1`. The rotated piece, stored in the `rotated_piece` argument, will contain:

```
OOOO
```

Suppose that `col = 8`. The rotated piece will be aligned with the game field (from `game2.txt`) as shown below:

```
        OOOO

..........
..........
OOO.......
OOOO.....O
OOOOO....O
OOOOOO..OO
```

The piece cannot be dropped into the game field because it will be partially out of bounds. Note that because `col` gives the left/right position of the *rotated* piece, it is not possible for a rotated piece to poke out of the left-hand side of the game field. If a rotated piece cannot be dropped because it is out of bounds of the game field, the function returns `-3`.

Assuming that the rotated piece is fully within the bounds of the game field, the `drop_piece` function next attempts to drop the piece into the game field as far as the piece can travel, stopping when a collision occurs. The algorithm proceeds in iterative fashion, starting with an attempt to position the piece at row #0. The function calls `count_overlaps` to see if the `rotated_piece` could theoretically be placed into the game field. Remember that the `row` argument of `count_overlaps` provides the position of the *topmost* row of the piece in the game

field. So, for instance, the O-piece could not be dropped into the game field as shown in the example below:

```
  OO
  OO

OOO....OOO
OOOOO.OOOO
OOOOO.OOOO
OOOOO.OOOO
```

but a properly rotated and horizontally aligned I-piece *could* be dropped in:

```
 OOOO

OOO....OOO
OOOOO.OOOO
OOOOO.OOOO
OOOOO.OOOO
```

The return value of `count_overlaps` can be interpreted to tell whether or not a piece can be placed at a given row and column. For instance, if `count_overlaps` returns `0`, this indicates that no blocks from the piece overlap with blocks from the game field. That is, the piece is floating down into the game field.

In the event that the rotated piece could not be placed inside the game field at column `col` and row #0 because there is not enough room (like the example with the O-piece above not fitting), the `drop_piece` function returns `-1`. Otherwise, it must be that `count_overlaps` returned `0`, and so the algorithm continues to iterate. Specifically, `count_overlaps` attempts to place the piece at row #1, then row #2, etc., until the piece finally comes to rest either because (a) it hit the bottom of the game field, or (b) it collided with one or more blocks already in the game field. Once the piece has finally come to rest, the `drop_piece` function updates the contents of `state.field` and returns the row # inside of `state.field` where the rotated game piece came to rest. This row # will be the row # of the *topmost* block of the rotated game piece.

The function takes the following arguments, in this order:

- `state`: the address of a `GameState` struct
- `col`: the column of the leftmost block of the rotated game piece
- `piece`: the address of a `Piece` struct
- `rotation`: the number of 90° clockwise turns that must be performed on the game piece before it is dropped into the game field
- `rotated_piece`; the address of a buffer to temporarily store the rotated version of `piece`, which is what will be dropped into the game field (Note: during grading, the contents of this array will not be checked or graded.)

Returns in `$v0`:

- the row # inside of `state.field` where the rotated game piece came to rest if the piece was successfully dropped into the game field; or

- `-2` if `col` is negative or `col` $\geq$ `state.num_cols`; or
- `-3` if the rotated game piece overlaps the rightmost edge of the game field (i.e., the game piece "pokes out" of the game field); or
- `-1` if the rotated game piece could not be dropped into the game field at the desired column due to a collision with blocks already in the game field

The error codes are listed above in a peculiar order because they reflect the order in which the error-checks should be performed.

Additional requirements:

- The function must call `get_slot`, `set_slot`, `rotate` and `count_overlaps`.
- The function must not write any changes to main memory except as required by the function.

Example #1 (piece drops a few rows into the game field):

Function arguments:

```
state.field:
  ..........
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

col = 4

piece.blocks:
  OOO
  .O.

rotation = 3
```

After function call:

```
state.field:
  ..........
  .O........
  .O........
  .O..O...O.
  OO..OO.OOO
  OOOOO..OOO
  OOOOO.OOOO
```

```
  OOOOO.OOOO

rotated_piece.blocks:
  O.
  OO
  O.
```

Return value: `3`

Example #2 (piece fits in the game field but cannot move past the top row):

Function arguments:

```
state.field:
  ..........
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

col = 1

piece.blocks:
  OOO
  .O.

rotation = 0
```

After function call:

```
state.field:
  .OOO......
  .OO.......
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

rotated_piece.blocks:
  OOO
  .O.
```

Return value: `0`

Example #3 (piece cannot enter the game field at the given column because of a collision):

Function arguments:

```
state.field:
  ..........
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

col = 0

piece.blocks:
  .O
  OO
  O.

rotation = 1
```

After function call:

```
state.field:
  ..........       (unchanged)
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

rotated_piece.blocks:
  OO.
  .OO
```

Return value: `-1`

Example #4 (invalid value for `col`):

Function arguments:

```
state.field:
  ..........
  .O........
```

```
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

col = 26

piece.blocks:
  .O
  OO
  O.

rotation = 0
```

After function call:

```
state.field:
  ..........      (unchanged)
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

rotated_piece.blocks:
   (undefined)
```

Return value: `-2`

Example #5 (rotated piece overlaps right-hand boundary of game field):

Function arguments:

```
state.field:
  ..........
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO
```

```
col = 8

piece.blocks:
  OO.
  .OO

rotation = 2
```

After function call:

```
state.field:
  ..........       (unchanged)
  .O........
  .O........
  .O......O.
  OO.....OOO
  OOOO...OOO
  OOOOO.OOOO
  OOOOO.OOOO

rotated_piece.blocks:
  OO.
  .OO
```

Return value: `-3`

Part VIII: Check if a Row in the Game Field Can Be Cleared

```
int check_row_clear(GameState* state, int row)
```

This function checks whether row #`row` of `state.field` is filled entirely with `'O'` characters and, if so, it shifts all of the rows above row #`row` down one row and then fills the "top" row (row #0) of the game field with `'.'` characters. Examples below help to clarify how the function works. If row #`row` of `state.field` is not filled entirely with `'O'` characters, the function makes no changes to the contents of `state.field`. Return values are specified below.

The function takes the following arguments, in this order:

- `state`: the address of a `GameState` struct
- `row`: the row of `state.field` to check to see if it can be "cleared"

Returns in `$v0`:

- `-1` if `row` is invalid, or
- `1` if row #`row` of `state.field` is filled entirely with `'O'` characters, or
- `0` if row #`row` of `state.field` is not filled entirely with `'O'` characters

Additional requirements:

- The function must call `get_slot` and `set_slot`.
- The function must not write any changes to main memory except as required by the function.

Example #1: (specified row is cleared; game field is updated)

Function arguments:

```
state.field:
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
  OOOOOO

row = 5
```

After function call:

```
state.field:
  ......
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
```

Return value: `1`

Example #2: (specified row is not cleared)

Function arguments:

```
state.field:
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
```

```
  OOOOOO

row = 3
```

After function call:

```
state.field:
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
  OOOOOO
```

Return value: `0`

Example #3: (`row` argument is invalid)

Function arguments:

```
state.field:
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
  OOOOOO

row = 19
```

After function call:

```
state.field:
  ...O..
  ...OO.
  .OOOO.
  .OOOO.
  OOOOOO
  OOOOOO
  OOOOOO
  OOOOOO
```

Return value: `-1`

Part IX: Simulate a Game of Tetris

```
int, int simulate_game(GameState* state, string filename, string moves,
                       Piece* rotated_piece, int num_pieces_to_drop,
                       Piece[] pieces_array)
```

This function simulates a game of Tetris according to the algorithm explained below. **Do not deviate at all from this algorithm. If you are unclear about what to implement, ask about it on Piazza.** Your function's return values and final game field contents will be tested against what is produced by the algorithm below. Any printed output generated by your code will not be graded.

The function takes the following arguments, in this order:

- `state`: a pointer to an uninitialized `GameState` struct that this function will initialize via a call to `load_game`
- `filename`: the name of the file containing the state of a game of Tetris that `load_game` will use to initialize `state`
- `moves`: a non-empty, null-terminated string that encodes a series of game pieces dropped into the game field
- `rotated_piece`: the address of a buffer to temporarily store the rotated version of `piece`, which is what will be dropped into the game field (Note: during grading the contents of this array will not be checked or graded.)
- `num_pieces_to_drop`: the number of pieces that are *validly* dropped into the game field before the simulation must stop. This argument will be available at `0($sp)`.
- `pieces_array`: the starting address of an array that encodes the seven tetromino game pieces (see below). This argument will be available at `4($sp)`.

Returns in `$v0`:

- the number of pieces that were successfully dropped into the game field

Returns in `$v1`:

- the final score of the game (see below)

The `moves` string encodes game pieces that the function simulates dropping into the game field. The string's length is guaranteed to be a multiple of 4. Each group of four characters is formatted as follows:

```
[1-character piece] [1-character rotation] [2-digit column]
```

For example, `J612` indicates that a J-piece should be rotated 6 times clockwise and then dropped into column 12. The substring `T103` indicates that a T-piece should be rotated once and then dropped into column 3. The seven standard game pieces are stored in an array passed as the argument `pieces_array`. The `pieces_array` will *always* have the contents shown below. **However**, your function has no guarantee whatsoever that the array will

always by named with the label "pieces_array" in the main file. In fact, during grading, the array will *not* have this name! Therefore, you should only access the pieces_array via the function argument at 4($sp).

```
# T piece
.byte 2
.byte 3
.ascii "OOO.O."
# J piece
.byte 2
.byte 3
.ascii "OOO..O"
# Z piece
.byte 2
.byte 3
.ascii "OO..OO"
# O piece
.byte 2
.byte 2
.ascii "OOOO.."
# S piece
.byte 2
.byte 3
.ascii ".OOOO."
# L piece
.byte 2
.byte 3
.ascii "OOOO.."
# I piece
.byte 1
.byte 4
.ascii "OOOO.."
```

It is possible that a four-character substring inside moves represents an invalid move. For instance, suppose the game field has 8 columns and the moves string contains the move I006. This piece cannot be dropped into the game field because it will overlap the right-hand boundary of the field. The function may otherwise assume that the string is well-formatted (i.e., the 0th character of each 4-character substring is one of [T,J,Z,O,S,L,I] and that the 1st, 2nd and 3rd characters are digit characters). The function keeps track of the number of pieces successfully (i.e., validly) dropped into the game field and returns this count in $v0.

Algorithm to implement:

```
if load_game(state, filename)[1] == (-1, -1) then  #
    return 0, 0

num_successful_drops = 0  # the number of pieces successfully dropped
move_number = 0  # the number of pieces we have attempted to drop so far
moves_length = len(moves) / 4  # the number of moves encoded in pieces string
game_over = False
```

```
score = 0

while not game_over and num_successful_drops < num_pieces_to_drop and
                        move_number < moves_length do
    # extract the next piece, column and rotation from the string
    piece_type = the game piece's shape (as a character)
    rotation = the game piece's rotation
    col = the game piece's column
    invalid = False

    if piece_type == 'T' then
        piece = pieces_array[0]
    elif piece_type == 'J' then
        piece = pieces_array[1]
    elif piece_type == 'Z' then
        piece = pieces_array[2]
    elif piece_type == 'O' then
        piece = pieces_array[3]
    elif piece_type == 'S' then
        piece = pieces_array[4]
    elif piece_type == 'L' then
        piece = pieces_array[5]
    else if piece_type == 'I' then
        piece = pieces_array[6]

    # attempt to drop the piece
    result = drop_piece(state, col, piece, rotation, rotated_piece)
    if result == -2 or result == -3 then
        invalid = True
    else if result == -1 then
        game_over = True

    if invalid == True then
        move_number += 1
        skip the rest of the loop body and continue to next iteration
           of the loop (i.e., a "continue" statement)

    # check for line clears by starting at the top of the game field and
    # working our way down
    count = 0  # number of lines cleared by dropping this piece
    r = state.num_rows - 1  # row counter
    while r >= 0 do
        if check_row_clear(state, r) == 1 then
            count += 1
        else
            r -= 1

    # update the score
    if count == 1 then
        score += 40
```

```
    else is count == 2 then
        score += 100
    else if count == 3 then
        score += 300
    else if count == 4 then
        score += 1200

    # Increment the counters
    move_number += 1
    num_successful_drops += 1

return num_successful_drops, score
```

Scoring: in a given round of play, it is possible that one, two, three or even four rows of blocks are cleared when a game piece is successfully dropped into the game field. Points are scored depending on how many rows are cleared in a single round:

- one-line clear: 40 points
- two-line clear: 100 points
- three-line clear: 300 points
- four-line clear: 1200 points

The total sum of points is returned in `$v1`.

Additional requirements:

- The function must call `load_game`, `drop_piece` and `check_row_clear`.
- The function must not write any changes to main memory except as required by the function.

Example #1: (8 I-pieces (and others) are dropped into an empty board to generate a score of 1200 points)

```
filename = "empty1.txt"
pieces = "I100I101I102I103J201I106I107I104I105O106"
num_pieces_to_drop = 15

final state.field:
  ........
  ........
  ........
  ........
  .O....OO
  .OOO..OO
```

Return values: `10, 1200`

Example #2: (several pieces are dropped into an empty game field to generate multiple line clears)

```
filename = "empty1.txt"
pieces = "T003O200I102O006L005O000T303"
num_pieces_to_drop = 7

final state.field:
  ........
  ........
  ........
  ........
  ........
  OOOO....
```

Return values: `7, 140`

Example #3: (several invalid moves (empty game field))

```
filename = "empty1.txt"
pieces = "I005Z005S001I104O605O000I009O202I107"
num_pieces_to_drop = 5

final state.field:
  ........
  ........
  ....OOO.
  OO..OOO.
  OOOOOOO.
  .OO.O.OO
```

Return values: `5, 0`

Example #4: (several invalid moves (non-empty game field))

```
filename = "game2.txt"
pieces = "I005Z005S001I104O605O000I009O202I107"
num_pieces_to_drop = 9

final state.field:
  ..OO......
  OOOO......
  OOOOO..O..
  .OO.OOOO..
  OOO.OOOO..
  OOOOOOOO.O
```

Return values: `8, 80`

Example #5: (several invalid moves (larger non-empty game field))

```
filename = "game5.txt"
pieces = "O500I212S916L508L310J607L609Z503S309Z719L418O619O806J511L506I61
          0O101O520Z514S512"
num_pieces_to_drop = 20

final state.field:
  ..................
  ......OO..OOOO....
  .......O....O.....
  .......O.O..O.....
  ......OO.OOOO.....
  ......OO..OO......
  .OO....O.OOO...O..
  .OO....OOO....OO..
  OO..O...OO....O...
  OO.OO....OO.OOOOO.
  O.OOOO.OOOO.O...OO
  OOOO.O..OOOOO..OOO
  OO.OOO.O.O.O..O.O.
  OO.OOOOO.OO.O.OO..
  OO.OOO.OOO..OO.O.O
  OO.OOO..OO....O.OO
  OO.O.OO.OO..OOO..O
  O..OOOOO.O...O..OO
  O.O.OOO.OO.......O
  OOOOO.OOO.O..O.O.O
```

Return values: `16, 0`

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss

material that I am willing to openly post on the discussion board.

6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `proj3.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on "Assignments" in the left-hand menu and click the link for this assignment.
3. Click the "Browse My Computer" button and locate the `proj3.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the steps again. We will grade only your last submission.