CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #3

Spring 2020

Assignment Due: Monday, April 13, 2020 by 11:59 pm

Updates to the Document:

- 4/4/2020: In Parts V, VI and VII, clarified the cases for which the function should return `-1`.

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 2D arrays of value.
- Read and write files to/from disk.

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 500,000 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 implemented 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 enqueue 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. Also please note that these testing main files will not be used in grading.

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!

Threes!

For this assignment you will be implementing a variant of the game Threes! Before reading any further, play the game for a few minutes to get a feel for it.

By pressing up, down, left or right, the player slides the tiles around the board. When two adjacent tiles cannot move in the desired direction (e.g., because they are against the border of the game board or are adjacent to other tiles that are against the border), the two tiles merge into one if:

- one tile contains a 1 and the other a 2
- the two tiles contain the same value, but that value must be at least 3

The value in the merged tile is the sum of the values from the original two tiles, which disappear from the board.

The location of the merged tile depends on the direction of the shift (slide):

- for an up-shift, the upper of the two tiles is replaced with the sum; the lower of the two is replaced with a 0
- for a down-shift, the lower of the two tiles is replaced with the sum; the upper of the two is replaced with a 0
- for a left-shift, the leftmost of the two tiles is replaced with the sum; the rightmost of the two is replaced with a 0
- for a right-shift, the rightmost of the two tiles is replaced with the sum; the leftmost of the two is replaced with a 0

In the visualizations below, a 0 signifies an empty tile, which is how we will represent an empty tile in our implementation. In our variant of the game, the game board can contain more than four rows and columns.

Shifting can be a little tricky to understand when merges are possible. Some examples are below.

First, note that only one pair of tiles can be merged as a result of a shift. For an up-shift, traverse the tiles top-to-bottom to shift the tiles into empty gaps. For a down-shift, traverse the tiles bottom-to-top. Likewise, for a left-shift, traverse the tiles left-to-right; for a right-shift, traverse the tiles right-to-left.

Examples of Up-Shifts

```
Before   After      Before   After      Before   After      Before   After
   2        2          3        6          2        2         24       24
   3        3          3        0          2        2          3        6
   0        1          0        2          6       12          3        3
   1        1          2        3          6        3          3        3
   1        2          3        6          3       12          3        3
   2        0          6        0         12        0          3        0
```

For sake of argument, suppose we are performing an up-shift. We start traversing the tiles at the top of the board in a given column. If we encounter a 0 before finding any tiles can be merged, this means we have a gap to shift tiles into and no merge will take place in the column. All tiles (and other gaps) below that 0 shift up by one row, thereby closing the gap. At the bottom-most row we shift in a 0. This situation is illustrated in the first example above.

Another possibility is that while traversing the tiles down a column and before finding a 0, we encounter two tiles that can be merged. The two tiles are merged, and all other tiles and gaps below the merge tile are shifted up by one row. Some different possible ways this could happen are shown in the other three examples given above. Note that these examples are not necessarily exhaustive of all possibilities.

Data Structures

The game board can be represented very easily. We will assume that the board will have at most 255 rows and 255 columns, meaning that each of these values can be represented by 1 unsigned byte. Furthermore, we will assume that the largest value in a tile is $49{,}152$, which is 3×2^{14}. If a player succeeds in creating a tile with that value in it, then the player has won the game. One unsigned, 16-bit half-word is sufficient to store a value in the range $[0, 49{,}152]$.

As an example, to store a game board of 4 rows and 6 columns, we would need $1 + 1 + 4 \times 6 \times 2 = 50$ total bytes. Byte 0 would contain 4, byte 1 would contain 6, and bytes 2 through 49 would contain the contents of the 2D board stored in row-major order. As an example, the board shown below:

```
  0     0     0    96    12    12
 24    24     0     0     0     3
  3     0     3     3     0     0
  1     0     0     0    48     1
```

would be represented in a MIPS `.data` section as:

```
.align 2
board:
.byte 4
.byte 6
.half 0 0 0 96 12 12 24 24 0 0 0 3 3 0 3 3 0 0 1 0 0 0 48 1
```

As shown above, we will assume that a game board data structure is word-aligned.

The above data structure can be treated as a **struct**, which is something like an object, but has no methods:

```
struct GameBoard {
    unsigned byte num_rows;     // offset 0 from starting address
    unsigned byte num_cols;     // offset 1 from starting address
    unsigned short[][] tiles;   // offset 2 from starting address
                                // a short is a 16-bit half-word
}
```

The topmost row of the game board is row #0 and the bottom-most row is `num_rows-1`. The leftmost column is #0. So, our sample game board's tiles would be represented as the following 2D array, stored in memory in row-major-order, with the 2D array index of each value given in brackets:

0 [0][0]	**0** [0][1]	**0** [0][2]	**96** [0][3]	**12** [0][4]	**12** [0][5]
24 [1][0]	**24** [1][1]	**0** [1][2]	**0** [1][3]	**0** [1][4]	**3** [1][5]
3 [2][0]	**0** [2][1]	**3** [2][2]	**3** [2][3]	**0** [2][4]	**0** [2][5]
1 [3][0]	**0** [3][1]	**0** [3][2]	**0** [3][3]	**48** [3][4]	**1** [3][5]

A game board can also be stored as a plain text file. As an example, this game board would be stored in a file as:

```
4 6 0 0 0 96 12 12 24 24 0 0 0 3 3 0 3 3 0 0 1 0 0 0 48 1
```

Note that values are not padded with zeroes.

Part I: Load a Game Board from Disk

`int load_game_file(GameBoard* board, string filename)`

This function reads the contents of a file that defines a game board and initializes the referenced `GameBoard` data structure. The notation `GameBoard*` indicates that `board` is the starting address of a `GameBoard` struct. You may assume that `board` points to a block of memory large enough to store the game board represented inside the file.

The function takes the following arguments, in this order:

- `board`: a pointer to (i.e., starting memory address of) an *uninitialized* `GameBoard` struct. Assume that the contents of the struct are filled with random garbage.
- `filename`: a string containing the filename to open and read the contents of

Returns in `$v0`:

- the number of non-zero game tiles read from the file, or
- `-1` if any error is encountered while reading the file. You may assume that if the file exists, that it is properly formatted.

Additional requirements:

- The function must not write any changes to main memory except as necessary to initialize the struct.

To test whether your code correctly loads the file's contents into memory, write some code that iterates over the `tiles` array and prints it one half-word at a time. Don't forget to check that `num_rows` and `num_cols` are also set correctly.

During grading of other parts of the assignment, your `load_game_file` function will **not** be used to initialize `GameBoard` structs.

Examples:

`filename` Argument	**Return Value**
`board1.txt`	`46`
`board2.txt`	`15`
`junk.txt`	`-1`

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

Service	**Code in `$v0`**	**Arguments**	**Results**
open file	13	`$a0` = address of null-terminated filename 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	`$v0` contains # of characters read (0 if end-of-file, negative if error)
write to file	15	`$a0` = file descriptor `$a1` = address of output buffer `$a2` = maximum # of characters to write	`$v0` contains # of characters written (negative if error)
close file	16	`$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 `$sp`) to store that byte temporarily. Discard newline characters as you read them and do not store them in the `GameBoard` struct. Finally, remember to reset `$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 file (*surprise!*). 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.

Part II: Save a Game Board to Disk

```
int save_game_file(GameBoard* board, string filename)
```

This function writes the contents of the referenced, valid `board` struct to disk in a file named `filename`. The format of the file must be as described earlier in the assignment:

```
num_rows SPACE num_cols NEWLINE
tiles[0][0] SPACE tiles[0][1] SPACE ... SPACE tiles[0][num_cols-1]
tiles[1][0] SPACE tiles[1][1] SPACE ... SPACE tiles[1][num_cols-1]
.
.
.
tiles[num_rows-1][0] SPACE ... SPACE tiles[num_rows-1][num_cols-1]
```

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `filename`: the name of the file where the game board is to be saved

Returns in `$v0`:

- `0` if the file was written successful; or
- `-1` for any error while attempting to open the file for writing

Additional requirements:

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

The easiest way to test your implementation of this function is to load a game board from disk using your `load_game_file` function and then write it to a different file. Consider using a tool like Linux's and MacOS's `diff` command or WinMerge under Windows.

Part III: Get the Value Stored in a Tile

```
int get_tile(GameBoard* board, int row, int col)
```

This function returns the value stored in the tile at `board.tiles[row][col]`. A recommendation: use `lhu` to read tile values from the game board. If `row` is outside the valid range or `col` is outside the valid range for the given `GameBoard`, the function returns `-1`. For instance, the valid range for the `row` argument is [0, `board.num_rows-1`].

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `row`: the row of the `tiles` array from where we want to read a value
- `col`: the column of the `tiles` array from where we want to read a value

Returns in `$v0`:

- the value located at `board.tiles[row][col]`, or
- `-1` for the error condition explained above

Additional requirements:

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

Examples:

Contents of game board loaded from `board1.txt`:

```
 6     1     6     2     3     3     6
 0     1     2     0     0     0     0
 0     2     0     1     2     3     6
 6    48    12     1    24     3     3
 2    12     3     0     1     3     0
 3     6     3     6     3     6     3
12    12    12    12    12    12    12
24     3     2     6     1     0     3
```

Name of File That `board` was Loaded From	**`row, col`**	**Return Value**
`board1.txt`	`3, 1`	`48`
`board1.txt`	`0, 3`	`2`
`board1.txt`	`7, 2`	`2`
`board1.txt`	`3, 0`	`6`
`board1.txt`	`6, 4`	`12`
`board1.txt`	`200, 1`	`-1`
`board1.txt`	`3, -7`	`-1`

Part IV: Set the Value Stored in a Tile

```
int set_tile(GameBoard* board, int row, int col, int value)
```

This function sets the value stored in the tile at `board.tiles[row][col]` to `value`. A recommendation: use `sh` to write tile values in the game board. The valid range for `value` is $[0, 49{,}152]$. If `value` is outside that range, the function returns `-1`. Also, `row` is outside the valid range or `col` is outside the valid range for the given `GameBoard`, the function returns `-1`. Otherwise, if `value`, `row` and `col` are all valid, the function returns `value`.

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `row`: the row of the `tiles` array from where we want to write a value
- `col`: the column of the `tiles` array from where we want to write a value
- `value`: the value to write at `board.tiles[row][col]`

Returns in `$v0`:

- `value`, provided that `value`, `row` and `col` are all valid; or
- `-1` for the error condition explained above

Additional requirements:

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

Initial contents of game board loaded from `board1.txt`:

```
     6     1     6     2     3     3     6
     0     1     2     0     0     0     0
     0     2     0     1     2     3     6
     6    48    12     1    24     3     3
     2    12     3     0     1     3     0
     3     6     3     6     3     6     3
    12    12    12    12    12    12    12
    24     3     2     6     1     0     3
```

Example #1. Set value at a valid tile in the middle of the board. Board loaded from `board1.txt`.

Arguments: `row = 3, col = 1, value = 96.`

Return value: `96`

State of `board` after function call:

```
     6     1     6     2     3     3     6
     0     1     2     0     0     0     0
     0     2     0     1     2     3     6
     6    96    12     1    24     3     3
     2    12     3     0     1     3     0
     3     6     3     6     3     6     3
    12    12    12    12    12    12    12
    24     3     2     6     1     0     3
```

Example #2. Set a value at a valid tile in the topmost row. Board loaded from `board1.txt`.

Arguments: `row = 0, col = 3, value = 192.`

Return value: `192`

State of `board` after function call:

```
     6      1      6    192      3      3      6
     0      1      2      0      0      0      0
     0      2      0      1      2      3      6
     6     48     12      1     24      3      3
     2     12      3      0      1      3      0
     3      6      3      6      3      6      3
    12     12     12     12     12     12     12
    24      3      2      6      1      0      3
```

Example #3. Set a value at a valid tile in the bottommost row. Board loaded from `board1.txt`.

Arguments: `row = 7, col = 1, value = 96`.

Return value: `96`

State of `board` after function call:

```
     6      1      6      2      3      3      6
     0      1      2      0      0      0      0
     0      2      0      1      2      3      6
     6     48     12      1     24      3      3
     2     12      3      0      1      3      0
     3      6      3      6      3      6      3
    12     12     12     12     12     12     12
    24     96      2      6      1      0      3
```

Example #4. Set a value at a valid tile in the leftmost column. Board loaded from `board1.txt`.

Arguments: `row = 2, col = 0, value = 24`.

Return value: `24`

State of `board` after function call:

```
     6      1      6      2      3      3      6
     0      1      2      0      0      0      0
    24      2      0      1      2      3      6
     6     48     12      1     24      3      3
     2     12      3      0      1      3      0
     3      6      3      6      3      6      3
    12     12     12     12     12     12     12
    24      3      2      6      1      0      3
```

Example #5. Set a value at a valid tile in the rightmost column. Board loaded from `board1.txt`.

Arguments: `row = 4, col = 6, value = 12`.

Return value: `12`

State of `board` after function call:

```
   6     1     6     2     3     3     6
   0     1     2     0     0     0     0
   0     2     0     1     2     3     6
   6    48    12     1    24     3     3
   2    12     3     0     1     3    12
   3     6     3     6     3     6     3
  12    12    12    12    12    12    12
  24     3     2     6     1     0     3
```

Example #6. Attempt to set a value at an invalid row #. Board loaded from `board1.txt`.

Arguments: `row = 8, col = 3, value = 96`.

Return value: `-1`

State of `board` after function call:

```
   6     1     6     2     3     3     6
   0     1     2     0     0     0     0
   0     2     0     1     2     3     6
   6    48    12     1    24     3     3
   2    12     3     0     1     3     0
   3     6     3     6     3     6     3
  12    12    12    12    12    12    12
  24     3     2     6     1     0     3
```

Example #7. Attempt to set a value at an invalid column #. Board loaded from `board1.txt`.

Arguments: `row = 3, col = -6, value = 96`.

Return value: `-1`

State of `board` after function call:

```
   6     1     6     2     3     3     6
   0     1     2     0     0     0     0
   0     2     0     1     2     3     6
   6    48    12     1    24     3     3
   2    12     3     0     1     3     0
   3     6     3     6     3     6     3
  12    12    12    12    12    12    12
  24     3     2     6     1     0     3
```

Example #8. Attempt to set an invalid value at a valid row and column. Board loaded from `board1.txt`.

Arguments: `row = 3, col = 1, value = 96000`.

Return value: `-1`

State of `board` after function call:

```
     6     1     6     2     3     3     6
     0     1     2     0     0     0     0
     0     2     0     1     2     3     6
     6    48    12     1    24     3     3
     2    12     3     0     1     3     0
     3     6     3     6     3     6     3
    12    12    12    12    12    12    12
    24     3     2     6     1     0     3
```

Part V: Determine If Two Tiles Can Be Merged

`int can_be_merged(GameBoard* board, int row1, int col1, int row2, int col2)`

This function determines whether the two tiles `board.tiles[row1][col1]` and `board.tiles[row2][col2]` can be merged into one tile based on the locations of the tiles in the game board and their respective values. For instance, two tiles which are not adjacent horizontally or vertically cannot, by definition, be merged. Two adjacent tiles with "incompatible" values cannot be merged, either. Compatible values would be a 1 and a 2, or two identical values ≥ 3.

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `row1`: the row index of the first tile we want to check for a possible merge
- `col1`: the column index of the first tile we want to check for a possible merge
- `row2`: the row index of the second tile we want to check for a possible merge
- `col2`: the column index of the second tile we want to check for a possible merge. This argument will be pushed onto the stack by the caller and will be available at `0($sp)`. `can_be_merged` must leave this argument in place, as the function calling conventions stipulate that the caller must pop that value off the stack.

Returns in `$v0`:

- `get_tile(board, row1, col1) + get_tile(board, row2, col2)` if the two tiles can be merge; or
- `-1` the two tiles are non-adjacent or if the two tiles are "incompatible" or if any of `row1`, `col1`, `row2` or `col2` is outside its valid range.

Additional requirements:

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

Examples:

Contents of game board loaded from `merge1.txt`:

```
     2      3      3      6     12      48     1      2
     1      6      0      0      1      2      1     24
     0      0      1      2     12     96      3      3
    12     12     24      0     12      1      2      3
```

Function Arguments	Return Value
`0, 1, 0, 1`	`-1`
`0, 6, 1, 6`	`-1`
`0, 4, 3, 6`	`-1`
`0, 7, 1, 0`	`-1`
`2, 3, 2, 4`	`-1`
`1, 2, 1, 3`	`-1`
`2, 2, 2, 3`	`3`
`2, 4, 3, 4`	`24`
`2, 6, 2, 7`	`6`

Part VI: Slide a Single Row of the Game Board Left or Right

```
int slide_row(GameBoard* board, int row, int direction)
```

This function slides row #`row` of `board.tiles` to the left if `direction` is `-1`, or to the right if `direction` is `1`. The mechanics of shifting a row or column of values is laid out in pages 4–5. The examples below will also help to explain the shifting process, including special or peculiar cases. **Be sure to ask for clarification on Piazza if you do not understand how to shift rows or columns in this game.**

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `row`: the particular row in `board.tiles` to shift left or right
- `direction`: the direction to shift the row (`-1` for left, `1` for right)

Returns in `$v0`:

- `1` if two tiles were merged as a result of this function call; or
- `0` if no tiles were merged; or
- `-1` if either of `row` or `direction` is an illegal value

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `get_tile`, `set_tile` and `can_be_merged`.

Contents of game board loaded from `slide1.txt`:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #1. Shifting to the left causes some tiles to move left. No merges take place. Board loaded from `slide1.txt`.

Arguments: `row = 0, direction = -1`

Return value: `0`

State of `board` after function call:

```
1     6     3     3    48     6     0
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #2. Shifting to the left causes some tiles to move left. Two tiles are merged. Board loaded from `slide1.txt`.

Arguments: `row = 1, direction = -1`

Return value: `1`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    24     0     6     3     6     0
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #3. Shifting to the right causes some tiles to move right. No merges take place. Board loaded from `slide1.txt`.

Arguments: `row = 1, direction = 1`

Return value: `0`

State of `board` after function call:

```
1     6     0     3     3    48     6
0     2    12    12     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #4. Shifting to the right causes some tiles to move right. Two tiles are merged. Board loaded from `slide1.txt`.

Arguments: `row = 2, direction = 1`

Return value: `1`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     0     3     3    12     0    48
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #5. Attempting to shifting to the left causes no changes to the board. Board loaded from `slide1.txt`.

Arguments: `row = 3, direction = -1`

Return value: `0`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #6. Attempting to shifting to the right causes no changes to the board. Board loaded from `slide1.txt`.

Arguments: `row = 3, direction = 1`

Return value: `0`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #7. Shifting to the right causes a merge in the middle of a row. Board loaded from `slide1.txt`.

Arguments: `row = 4, direction = 1`

Return value: `1`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
0     6    12     3     3    48    24
```

Example #8. Invalid value for the direction argument Board loaded from `slide1.txt`.

Arguments: `row = 3, direction = 0`

Return value: `-1`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Example #9. Invalid value for the row argument Board loaded from `slide1.txt`.

Arguments: `row = 7, direction = 1`

Return value: `-1`

State of `board` after function call:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Part VII: Slide a Single Column of the Game Board Up or Down

```
int slide_col(GameBoard* board, int col, int direction)
```

This function slides column #`col` of `board.tiles` up if `direction` is `-1`, or down if `direction` is `1`. The mechanics of shifting a row or column of values is laid out in pages 4–5. The examples below will also help to explain the shifting process, including special or peculiar cases. **Be sure to ask for clarification on Piazza if you do not understand how to shift rows or columns in this game.**

The function takes the following arguments, in this order:

- `board`: a pointer to a valid `GameBoard` struct
- `col`: the particular column in `board.tiles` to shift up or down
- `direction`: the direction to shift the column (`-1` for up, `1` for down)

Returns in `$v0`:

- `1` if two tiles were merged as a result of this function call; or
- `0` if no tiles were merged; or
- `-1` if either of `col` or `direction` is an illegal value

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `get_tile`, `set_tile` and `can_be_merged`.

Contents of game board loaded from `slide2.txt`:

```
 1     2     0     1     6
 6    12     3     3    12
 0    12     3     2     1
 3     0    12     6     2
 3     6     0     1     3
48     3    24    48    48
 6     6    24    12    24
```

Example #1. Shifting up causes some tiles to move up. No merges take place. Board loaded from `slide2.txt`.

Arguments: `col = 0, direction = -1`

Return value: `0`

State of `board` after function call:

```
 1     2     0     1     6
 6    12     3     3    12
 3    12     3     2     1
 3     0    12     6     2
48     6     0     1     3
 6     3    24    48    48
 0     6    24    12    24
```

Example #2. Shifting up causes some tiles to move up. Two tiles are merged. Board loaded from `slide2.txt`.

Arguments: `col = 1, direction = -1`

Return value: `1`

State of `board` after function call:

```
 1     2     0     1     6
 6    24     3     3    12
 0     0     3     2     1
 3     6    12     6     2
```

```
     3      3      0      1      3
    48      6     24     48     48
     6      0     24     12     24
```

Example #3. Shifting down causes some tiles to move down. No merges take place. Board loaded from `slide2.txt`.

Arguments: `col = 1, direction = 1`

Return value: `0`

State of `board` after function call:

```
     1      0      0      1      6
     6      2      3      3     12
     0     12      3      2      1
     3     12     12      6      2
     3      6      0      1      3
    48      3     24     48     48
     6      6     24     12     24
```

Example #4. Shifting down causes some tiles to move down. Two tiles are merged. Board loaded from `slide2.txt`.

Arguments: `col = 2, direction = 1`

Return value: `1`

State of `board` after function call:

```
     1      2      0      1      6
     6     12      0      3     12
     0     12      3      2      1
     3      0      3      6      2
     3      6     12      1      3
    48      3      0     48     48
     6      6     48     12     24
```

Example #5. Attempting to shifting up causes no changes to the board. Board loaded from `slide2.txt`.

Arguments: `col = 3, direction = -1`

Return value: `0`

State of `board` after function call:

```
     1      2      0      1      6
     6     12      3      3     12
     0     12      3      2      1
     3      0     12      6      2
```

```
 3     6     0     1     3
48     3    24    48    48
 6     6    24    12    24
```

Example #6. Attempting to shifting down causes no changes to the board. Board loaded from `slide2.txt`.

Arguments: `col = 3, direction = 1`

Return value: `0`

State of `board` after function call:

```
 1     2     0     1     6
 6    12     3     3    12
 0    12     3     2     1
 3     0    12     6     2
 3     6     0     1     3
48     3    24    48    48
 6     6    24    12    24
```

Example #7. Shifting down causes a merge in the middle of a column. Board loaded from `slide2.txt`.

Arguments: `col = 4, direction = 1`

Return value: `1`

State of `board` after function call:

```
 1     2     0     1     0
 6    12     3     3     6
 0    12     3     2    12
 3     0    12     6     3
 3     6     0     1     3
48     3    24    48    48
 6     6    24    12    24
```

Example #8. Invalid value for the direction argument Board loaded from `slide2.txt`.

Arguments: `col = 3, direction = -3`

Return value: `-1`

State of `board` after function call:

```
 1     2     0     1     6
 6    12     3     3    12
 0    12     3     2     1
 3     0    12     6     2
 3     6     0     1     3
48     3    24    48    48
```

```
     6     6    24    12    24
```

Example #9. Invalid value for the col argument Board loaded from `slide2.txt`.

Arguments: `col = 5, direction = 1`

Return value: `-1`

State of `board` after function call:

```
     1     2     0     1     6
     6    12     3     3    12
     0    12     3     2     1
     3     0    12     6     2
     3     6     0     1     3
    48     3    24    48    48
     6     6    24    12    24
```

Part VIII: Slide the Entire Game Board to the Left

`int slide_board_left(GameBoard* board)`

This function simply calls `slide_row` on every row of a given game board to slide each row to the left.

The function takes one argument:

- `board`: a pointer to a valid `GameBoard` struct

Returns in `$v0`:

- the sum of return values from the series of calls to `slide_row`

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `slide_row`.

Example:

Contents of game board loaded from `slide1.txt`:

```
     1     6     0     3     3    48     6
     2    12    12     0     6     3     6
     0     3     3    12     0    24    24
     1     3     2     6     1    48    12
     6    12     1     2     3    48    24
```

Return value: `2`

State of `board` after function call:

```
1     6     3     3    48     6     0
2    24     0     6     3     6     0
3     3    12     0    24    24     0
1     3     2     6     1    48    12
6    12     3     3    48    24     0
```

Part IX: Slide the Entire Game Board to the Right

`int slide_board_right(GameBoard* board)`

This function simply calls `slide_row` on every row of a given game board to slide each row to the right.

The function takes one argument:

- `board`: a pointer to a valid `GameBoard` struct

Returns in `$v0`:

- the sum of return values from the series of calls to `slide_row`

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `slide_row`.

Example:

Contents of game board loaded from `slide1.txt`:

```
1     6     0     3     3    48     6
2    12    12     0     6     3     6
0     3     3    12     0    24    24
1     3     2     6     1    48    12
6    12     1     2     3    48    24
```

Return value: `3`

State of `board` after function call:

```
0     1     6     0     6    48     6
0     2    12    12     6     3     6
0     0     3     3    12     0    48
1     3     2     6     1    48    12
0     6    12     3     3    48    24
```

Part X: Slide the Entire Game Board Upwards

`int slide_board_up(GameBoard* board)`

This function simply calls `slide_col` on every column of a given game board to slide each column upwards.

The function takes one argument:

- `board`: a pointer to a valid `GameBoard` struct

Returns in `$v0`:

- the sum of return values from the series of calls to `slide_col`

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `slide_col`.

Example:

Contents of game board loaded from `slide2.txt`:

```
   1     2     0     1     6
   6    12     3     3    12
   0    12     3     2     1
   3     0    12     6     2
   3     6     0     1     3
  48     3    24    48    48
   6     6    24    12    24
```

Return value: `2`

State of `board` after function call:

```
   1     2     3     1     6
   6    24     3     3    12
   3     0    12     2     3
   3     6     0     6     3
  48     3    24     1    48
   6     6    24    48    24
   0     0     0    12     0
```

Part XI: Slide the Entire Game Board Downwards

`int slide_board_down(GameBoard* board)`

This function simply calls `slide_col` on every column of a given game board to slide each column downwards.

The function takes one argument:

- `board`: a pointer to a valid `GameBoard` struct

Returns in `$v0`:

- the sum of return values from the series of calls to `slide_col`

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `slide_col`.

Example:

Contents of game board loaded from `slide2.txt`:

```
    1     2     0     1     6
    6    12     3     3    12
    0    12     3     2     1
    3     0    12     6     2
    3     6     0     1     3
   48     3    24    48    48
    6     6    24    12    24
```

Return value: `3`

State of `board` after function call:

```
    0     0     0     1     0
    1     2     0     3     6
    6    12     3     2    12
    0    12     3     6     3
    6     6    12     1     3
   48     3     0    48    48
    6     6    48    12    24
```

Part XII: Check the Game's Status

`int game_status(GameBoard* board)`

This function returns an integer code that encodes the state of an on-going game of Threes.

The function takes one argument:

- `board`: a pointer to a valid `GameBoard` struct

Returns in `$v0` and `$v1`, respectively:

- `-2, -2`, if the number 49,152 appears anywhere inside `board.tiles`; or

- `-1, -1`, if no moves are possible because there are no gaps (i.e., zeroes) in the board or because the board contains only non-zero tiles and none of those tiles can be merged; or
- `sum1, sum2`, where `sum1` is the sum of the number of rows that can be shifted left or right, and `sum2` the number of columns that can be shifted up or down. A row (column) that can be shifted left or right (up or down) should not be double-counted.

Additional requirements:

- The function must not write any changes to main memory.
- The function must call `get_tile` directly or via a helper function.

Example #1. Game over (win).

Contents of game board loaded from `win.txt`:

```
    1      2      3     12      0
   24     96  49152      2      1
  192    192    384      0      0
    3      3      6     12     12
```

Return Values: `-2, -2`

Example #2. Game over (loss).

Contents of game board loaded from `loss.txt`:

```
    3      2      3      2      3
    6      3      6      3      6
    1      1      1      1      1
    3      6     12      3      6
```

Return Values: `-1, -1`

Example #3. Game can continue (test #1).

Contents of game board loaded from `random1.txt`:

```
    1      3      1     48
    3      1      1     24
  768    768      6      6
    1      2      2      6
  768      6      3      2
    0     48      1     24
```

Return Values: `3, 2`

Example #4. Game can continue (test #2).

Contents of game board loaded from `random2.txt`:

```
   1     0     6    96     1     1     1     2
   2     0     3    48  1536   192     3   192
   1     3     6   384   192     2   384  1536
   6     0    24   384    48   768   192   384
  48    12   384     1     2     1   192     3
```

Return Values: `4, 4`

Example #5. Game can continue (test #3).

Contents of game board loaded from `random3.txt`:

```
   2     6     6   768    24     1    96    24  1536     6     0     2
1536     1    12     0    12     3    24   768   384     1     3   192
   3   768   192     2     2   768    96    24    96     1     0     2
  96     2     0     3    12     3    48    48     0     1     1   384
  12     1     1    48     0   768    24     1   768    24     0     6
   3   192     1   768     6    48    12     0     2    48     1     2
  12     3     1     1   384    24   768     3     2     3    24    12
```

Return Values: `6, 7`

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.