CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #3

Spring 2021

Assignment Due: Sunday, April 11, 2021

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 implement functions that implement the MIPS assembly function calling conventions.
- Implement algorithms that process 2D arrays of values.
- Read and Write files from disk using MIPS system calls.

Getting Started

Access the most up to date starting files here

Inside the downloaded folder you will find **hw3.asm** and numerous "test" files. Write your code in **hw3.asm**. The file contains several function stubs, which you will need to implement. These stubs contain only `jr $ra` instructions. Your job in this assignment is to implement all the functions as described below. Do not change the function names/labels since the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, and add them to **hw3.asm**.

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.

IMPORTANT: Do not define a `.data` section in your hw3.asm file. A submission that contains a `.data` section will most likely not integrate with our grading script. This may lead to you getting no credit.

Important Information about CSE 220 Programming Projects

- Read the entire project description document 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 the course website. 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 assignments 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 the assignment if you do this.

- Submit your final .asm file to the course website by the due date and time. Late work will be penalized as described in the course syllabus. 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

Your programming assignments will be graded almost entirely in an automated way. 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 assignment, your program will be generating output and your functions will be returning values that will be checked for exact matches by the grading scripts. It is your responsibility to output/return the expected values.

Some other items you should be aware of:

- Each test case must execute in 100000 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.

If you are using the command line to assemble and run your code, you can use the following command:

```
$ java -jar /path/to/MarsSpring2021.jar /path/to.asm --noGui -i -q
```

The path separators will be '\' instead of '/' if you are on a Windows machine.

- Any excess output from your program (debugging notes, etc.) will impact grading. Do not leave erroneous print-outs in your code.
- 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.

Unit-Testing Functions

To test your implemented functions, open the provided “test” files in MARS. Next, assemble the “test” 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 append the contents of your *hw3.asm* file before assembling the program.

Each “test” 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 “test” files. Your submission will not be graded using the examples provided in this document or using the provided “test” file(s). Do not submit your “test” files with your hw3.asm file – we will delete them.

Make sure that all code required for implementing your functions is included in the *hw3.asm* file.

Understanding The Game

For this assignment you will be creating a special implementation of Mancala in MIPS. This is a very simple game that will be formatted in a two dimensional array to represent the board state.

Here is an example of a board in progress in the 2D array in relation to a traditional board:

P2 Manc.	8	7	6	1	0	4	P1 Manc.
1	4	4	4	4	4	4	0

Please note the P2 Manc and P1 Manc are not part of this 2D array. Also notice that this board is assuming each player has 6 pockets (more on that later).

Where the red row belongs to player 2 (Opponent), and the cyan row belongs to player 1 (You). You can assume when moving that it is done in a “to the right” orientation.

This means that when the game state has P1 moving you would execute the move in coordination with their respective pit and move to the right.

Please watch this video to understand how this game works.
It’s only a mere minute of your time!

In addition to keeping track of the board you will also need to update predefined data sections of memory for things like whose turn it is, each player's score in Mancala, and the game status (if it is over).

Data Structures

In order to keep track of these items you will utilize a series of data fields that are addressable by a single pointer. It will be referred to as `GameState` from now on.
Here is the data that will makeup the `GameState` struct:

- `bot_mancala: # of stones in player 1's mancala (unsigned byte)`
- `top_mancala: # of stones in player 2's mancala (unsigned byte)`
- `bot_pockets: # of pockets for player 1's row (unsigned byte)`
- `top_pockets: # of pockets for player 2's row (unsigned byte)`
- `moves_executed: # of moves executed (unsigned byte)`
- `player_turn: which player is moving (ASCII Char / unsigned byte)`
  `{ 'D' == Game Done 'T' == P2, 'B' == P1 }`
- `game_board: contents of the game board (asciiz string)`

game_board format:

- First two characters are the mancala for player 2 (top mancala)
- The next {top_pockets * 2} characters will hold the ASCII numeric values of 0 - 99 or "00" - "99" in ASCII
- The next {bot_pockets * 2} characters will hold the ASCII numeric values of 0 - 99 or "00" - "99" in ASCII
- Last two characters are the mancala for player 1 (bottom mancala).

You can safely assume the following
You will never use more than 99 stones in the **entire** board
You will never use more than 98 pockets in the **entire** board

The space provided to you at GameState's pointer will adequately hold the struct's size including the game_board string.

In MIPS GameState will have an overall structure, bear in mind that the values will vary for tests:

```
.align 2         # state is aligned on a word boundary
state:      # name of the data structure
    .byte 0      # bot_mancala       (byte #0)
    .byte 1      # top_mancala       (byte #1)
    .byte 6      # bot_pockets       (byte #2)
    .byte 6      # top_pockets       (byte #3)
    .byte 2      # moves_executed    (byte #4)
    .byte 'B'    # player_turn       (byte #5)
    # game_board                     (bytes #6-end)
    .asciiz
    "0108070601000404040404040400"
```

Part 1: Load a Game Board from Disk

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

The `load_game` function reads the contents of a file that defines a game board and initializes the referenced `GameState` data structure. The notation `GameState*` indicates that `state` is the starting address of a `GameState` struct. You may assume that `state` points to a block of memory large enough to store the struct represented inside the file. If the file exists, you may assume that the file's contents are valid. If the file does not exist, the function simply returns -1, -1.

The file format is very simple:

```
# of stones in top Mancala
# of stones in bot Mancala
# of pockets in each row
Contents of top row where every two characters is a pocket
Contents of bot row where every two characters is a pocket
```

Important note: on Microsoft Windows, a line ends with the character combination `\r\n`, whereas on Mac and other Unix-like operating systems like Linux, a line ends only with `\n`. Your code must be able to handle both line-ending styles. All lines of the file, including the final row of the grid, are guaranteed to end with `\n` or `\r\n`. As an example, the above game grid would be saved in a Microsoft Windows-generated file as:

```
1\r\n
0\r\n
6\r\n
080706010004\r\n
040404040404\r\n
```

Again, your code must be able to contend with `\r\n` line-endings and with `\n` line-endings.

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 (negative if error) $a2 = maximum # of characters to write	$v0 contains # of characters written
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.

Some advice: 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 the disk. You should allocate four bytes of memory on the stack (by adjusting $sp) to store that byte temporarily. Discard newline characters (both `\r` and `\n`) 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 .txt 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 `load_game` takes the following arguments, in this order:

- `state`: a *pointer* to (i.e., starting memory address of) an uninitialized `GameState` struct large enough to hold the contents of the game represented by file to be loaded. 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:

- -1 if the input file does not exist; otherwise: 0 if there is too many stones within the game including those in the Mancala. 1 if the stones abide by the limit

Returns in $v1:

- -1 if the input file does not exist; otherwise: the number of pockets total.
- If the number of pockets exceeds the limit (98) return 0.

Example #1:
Board_filename = “game01.txt”
Returns in
$v0 = 1
$v1 = 12
game01.txt has the default Mancala layout.

Example #2:
Board_filename = “gameE1.txt”
Returns in
$v0 = 0
$v1 = 12

Example #3:
Board_filename = “gameE3.txt”
Returns in
$v0 = 0
$v1 = 0

Example #4:
Board_filename = “not_real_file.txt”
Returns in
$v0 = -1
$v1 = -1

Part 2: Get Quantity of Stones in the pocket

```
int get_pocket(GameState* state, byte player, byte distance)
```

The function `get_pocket` returns the number of stones inside the pocket for the given player in the `GameState` grid. The argument player will determine which player's pockets we want to examine. Then the distance is the quantity of pockets away from the same respective player's Mancala. For instance, if I provided a player value of 1 for the bottom row and gave a distance with the value 0 then I would be providing the pocket quantity of the one right next to the Mancala for player 1 (the bottom row).

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `player`: designates which row's you are examining. This is an 8-bit, two's complement value.
- `distance`: the pocket from where we want to read a value. This is an 8-bit, two's complement value.

Returns in $v0:

- the integer value of the # of stones located at the designated pocket, or
- -1 for the error condition explained below

Errors:

`player` is not valid (Only values 0 or 1 are valid)
`distance` is not valid (offset from the row's Mancala not possible for this board)

Additional requirements:

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

Example #1:
state contains the following pockets
040404040404
200102400005
Given player = 'T'
Given distance = 3
$v0 = 4

Example #2:
state contains the following pockets
040404040404
200102400005
Given player = 'B'
Given distance = 3
$v0 = 2

Example #3:
state contains the following pockets
040404040404
200102400005
Given player = ‘D’
Given distance = 3
$v0 = -1

Part 3: Set the Character Stored in the Game Grid

```
int set_pocket(GameState* state, byte player, byte distance, int
size)
```

The function `set_pocket` sets the value stored in the pocket desired by the arguments to `size` in the form of an insertion of two ASCII chars. A recommendation: use `sb` to write a character in the game board.
You are not expected to validate if this set_pocket violates the max size through the entire board. Only if the value itself is violating the bounds [0 - 99].

The function takes the following arguments, in this order:
- `state`: a pointer to a valid `GameState` struct
- `player`: what row is being reassigned on the board. This is an 8-bit, two’s complement value.
- `distance`: specifies the pocket to be reassigned a value This is an 8-bit, two’s complement value.
- `size`: the value to write in ASCII in the designated pocket

Returns in $v0:
- `size`, provided that `player` and `distance` are both valid, or
- -1 if player or distance is not valid for this game board
- -2 the value given in size is beyond size 99 or below 0.

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

Example #1:
state contains the following pockets
040404040404
200102400005
Given player = ‘T’
Given distance = 3
Given size = 16
After returning
$v0 = 16
state contains the following pockets
040404160404
200102400005

Example #2:
state contains the following pockets
040404040404
200102400005
Given player = ‘B’
Given distance = 0
Given size = 1
After returning
$v0 = 1
state contains the following pockets
040404160404
200102400001

Example #3:
state contains the following pockets
040404040404
200102400005
Given player = ‘B’
Given distance = 10
Given size = 1
After returning
$v0 = -1
state contains the following pockets
040404040404
200102400005

Example #4:
state contains the following pockets
040404040404
200102400005
Given player = ‘B’
Given distance = 1
Given size = 101
After returning
$v0 = -2
state contains the following pockets
040404040404
200102400005

Part 4: Add Stones to Mancala

`int collect_stones(GameState* state, byte player, int stones)`

The function `collect_stones` will increment the indicated `player` mancala value by the provided quantity in `stones`.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct, *this must be updated accordingly*
- `player`: what player's mancala is being given the stones
- `stones`: quantity to add to the specified mancala

Returns in $v0:

- `stones`, provided that `player` and `distance` are both valid

Follow the error cases in the order presented to you here

- -1 the byte given to specify player is invalid.
- -2 the value given in stones is less than or equal to zero.

Additional requirements:

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

Example #1:

state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 0
Given player = 'T'
Given stones = 2
After returning
$v0 = 2
state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 2

Example #2:

state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 0
Given player = 'A"
Given stones = 2
After returning
$v0 = -1
state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 0

Example #3:
state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 0
Given player = 'T'
Given stones = -2
After returning
$v0 = -2
state contains the following values in the mancala of P1 & P2:
P1: 4
P2: 0

Part 5: Verify Move
`int verify_move(GameState* state, byte origin_pocket, byte distance)`
The function `verify_move` will verify if the move provided by arguments `origin_pocket` and `distance` can occur without violating the game rules. There is one special case, when distance is equal to 99 you will modify the GameState to be the other turn. You can ignore validating origin_pocket.
The function takes the following arguments, in this order:
- `state`: a pointer to a valid `GameState` struct
- `origin_pocket`: # of pockets away from the mancala of the *current player's turn*
- `distance`: # of pockets to move from origin

Returns in $v0:
- 2 if distance is equal to 99
- 1 if move is legal

Follow the error cases in the order presented to you here
- 0 if `origin_pocket` has zero stones
- -1 the `origin_pocket` is invalid for the row size
- -2 the `distance` is zero, or not equal to the stones in the `origin_pocket`

Additional requirements:
- The function must not write any changes to main memory except for a turn transition.

Example #1:
state contains the following data:

0	(bot_mancala)
0	(top_mancala)
6	(bot_pockets)
6	(top_pockets)
0	(moves_executed)
B	(player_turn)
0004040404040404040404040400	(game_board)

origin_pocket = 03
distance = 04
Returns in
$v0 = 1

Example #2:
state contains the following data:

0	(bot_mancala)
0	(top_mancala)
6	(bot_pockets)
6	(top_pockets)
0	(moves_executed)
B	(player_turn)
0004040404040404040404040400	(game_board)

origin_pocket = 03
distance = 99
Returns in
$v0 = 2

Example #3:
state contains the following data:

0	(bot_mancala)
0	(top_mancala)
6	(bot_pockets)
6	(top_pockets)
0	(moves_executed)
B	(player_turn)
0004040404040404040004040400	(game_board)

origin_pocket = 03
distance = 92
Returns in
$v0 = 0

Example #4:
state contains the following data:

0	(bot_mancala)
0	(top_mancala)
6	(bot_pockets)
6	(top_pockets)
0	(moves_executed)
B	(player_turn)
0004040404040404040104040400	(game_board)

origin_pocket = 33
distance = 92
Returns in
$v0 = -1

Example #5:
state contains the following data:

0	(bot_mancala)
0	(top_mancala)
6	(bot_pockets)
6	(top_pockets)
0	(moves_executed)
B	(player_turn)
0004040404040404040104040400	(game_board)

origin_pocket = 3
distance = 3
Returns in
$v0 = -2

Part 6: Execute Move

`int, int execute_move(GameState* state, byte origin_pocket)`

The function `execute_move` will be called after successful verification from `verify_move`. It will proceed to move the stones like in a typical game of Mancala (watch the video again). We are only executing ONE move.

Keep in mind that:

- Assume the # of stones in origin_pocket is identical to distance (already verified)
- You skip the opponent's Mancala, and only count the deposit in your mancala as a decrement to the # of stones left to deposit
- You are expected to modify the GameState according to the move's circumstances (i.e. increase score, change turn, etc.)

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `origin_pocket`: # of pockets away from the mancala of the *current player's turn*

Returns in $v0:

- Number of stones added to the mancala

Returns in $v1:

- 2 if last deposit was in the Mancala
- 1 if last deposit was in the player's row and was empty before the last deposit
- 0 if last deposit was anywhere else

Additional requirements:

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

Part 7: Steal Execute

`int steal(GameState* state, byte destination_pocket)`

Steal will be utilized during the event that execute_move returns the value 1 in $v1. In this instance the player who just moved (may be different from the current status) will receive any stones in that aligned column.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct
- `destination_pocket`: # of pockets away from the mancala of the player from the former turn

Returns in $v0:

- Number of stones added to the mancala

Additional requirements:

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

Part 8: Check Row

`int check_row(GameState* state)`

Called after `execute_move` to check if either row is currently empty. If this is the case, the other non empty side will receive the remaining stones in that player's respective mancala. Remember that the game ends if this happens and you must modify GameState accordingly.

The function takes the following arguments, in this order:

- `state`: a pointer to a valid `GameState` struct

Returns in $v0:

- 1 if a row was found to be empty (Game is over)
- 0 if both rows are not empty

Returns in $v1:

- 1 if player 1 has higher # of stones in mancala
- 2 if player 2 has higher # of stones in mancala

Additional requirements:

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

Part 9: Load Moves from Disk

`int load_moves(byte[] moves, string filename)`

The `load_moves` function will operate in a similar fashion to `load_game`.

Here is the correct layout of a move file:

```
# Quantity of columns in in move array
# Quantity of rows in move array
# The moves where every two characters is a move
```

Sample move file in MacOS (notice the lack of \r you have to handle both file encodings)

```
4\n
4\n
030400000304000005000000005000000\n
```

Take notice that there may be invalid moves in your sample move files

Do not worry about validation of the move file, you can assume it will be valid other than some possibly invalid moves that can be skipped over and will be discussed in Part 10.

Every time you go to the next row you must insert a “99” move, the only exception is the last row. This means that our example move file would have 3 insertions of 99. You insert at the end of each row in preparation for the following one.

Remember do not add 99 as the final move of the file. The last row does not add 99.

The function takes the following arguments, in this order:

- `moves`: a label to the starting address of an array big enough to hold the moves read in the file (you can assume this)
- `filename`**:** The file to read the moves from

Returns in $v0:

- If the file exists return how many moves were in the file (include invalids + added “99”s)
- -1 if the there is an error accessing the file

Additional requirements:

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

Part 10: Play Game

`int, int play_game (string moves_filename, string board_filename, GameState* state, byte[] moves, int num_moves_to_execute)`

This is the main method that utilizes all the methods you've previously created in order to facilitate the game. The only new argument is `num_moves_to_execute` and that is just the number of moves that will be executed. You will skip over any move that is not valid, this will not count as an executed move.

The function takes the following arguments, in this order:

- `moves_filename`: The file to read the moves
- `board_filename`: The file to read the initial board
- `state`: The pointer to the GameState struct
- `moves`: The array that will hold the moves read from the file
- `num_moves_to_execute`: This is the number of moves that will be executed. You are not to exceed this limit if the moves array size is larger. This will be in the stack pointer as indicated by the `play_game_test.asm` file.

Returns in $v0:

- -1 if the there is an error accessing any file
- 0 if there is no error but nobody won
- 1 if player 1 is the winner
- 2 if player 2 is the winner

Returns in $v1:

- -1 if the there is an error accessing any file
- # of valid moves executed

Part 11: Print Board

`void print_board(GameState* state)`

In `print_board` you will be simply printing the current state's board string with a slight twist. There will be two lines printed before the two rows. Each signifying the top row player's mancala (Player 2) and the bottom row player's mancala (Player 1). You are expected to print this to the console exactly as defined.

Sample `print_board` output.

```
01\n
00\n
080706010004\n
040404040404\n
```

You can assume that you will be given a pointer to a valid and instantiated state struct.
You will not return anything, but keep in mind that console output is expected.

Part 12: Write Game Board to File

`int write_board(GameState* state)`

In `write_board` you will do the exact same procedure as print board except now you will be writing to a text file called output.txt.

Here is a short tutorial that will help you understand how to write to a file within your MIPS program:

```
# Sample MIPS program that writes to a new file.
#   by Kenneth Vollmar and Pete Sanderson

        .data
fout:   .asciiz "output.txt"      # filename for output
buffer: .asciiz "The quick brown fox jumps over the lazy dog."
        .text
  ###############################################################
  # Open (for writing) a file that does not exist
  li   $v0, 13       # system call for open file
  la   $a0, fout     # output file name
  li   $a1, 1        # Open for writing (flags are 0: read, 1: write)
  li   $a2, 0        # mode is ignored
  syscall            # open a file (file descriptor returned in $v0)
  move $s6, $v0      # save the file descriptor
  ###############################################################
  # Write to file just opened
  li   $v0, 15       # system call for write to file
  move $a0, $s6      # file descriptor
  la   $a1, buffer   # address of buffer from which to write
  li   $a2, 44       # hardcoded buffer length
  syscall            # write to file
  ###############################################################
  # Close the file
  li   $v0, 16       # system call for close file
  move $a0, $s6      # file descriptor to close
  syscall            # close file
  ###############################################################
```

Returns in $v0:

- 1 if the game board was written to the file “output.txt” without any issues
- -1 if there was any kind of error with the syscalls

How to Submit Your Work for Grading

Submit **hw3.asm.** This should contain all the function implementations of the respective parts outlined above. Do not include the main files. They will be deleted before grading if found.

Go to **Assignments->Assignment 3** on **Blackboard** and submit **hw3.asm**.

You can submit multiple times. We will grade your most recent submission before the due date.