

# CSE 220: Systems Fundamentals I

## Stony Brook University

### Programming Project #1

Fall 2019

Assignment Due: Friday, September 27, 2019 by 11:59 pm

#### Updates to the Document:

- 9/13/2019: added note in Part II to clarify that for the M operation you may assume that exactly 8 characters are given and that they are chosen only from ABCDEF0123456789

#### Learning Outcomes

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

- Use system calls to print values to the screen in different formats.
- Design and implement algorithms in MIPS assembly that involve if-statements and counter-driven loops.
- Read and write values stored in a MIPS assembly `.data` section.
- Use bitwise operations to perform simple computations in MIPS assembly.

#### Getting Started

Visit [Blackboard](#) and download the files `proj1.asm` and `MarsFall2019.jar`. Fill in the following information at the top of `proj1.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 `proj1.asm` you will find some code to start with. Your job in this assignment is implement all the operations as specified below. If you are having difficulty implementing these operations, write out pseudocode or implement the algorithms in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS assembly code.

#### 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 [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 will 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 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.
- 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 programming assignments 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 Programming Project #1, your program will be generating output that will be checked for *exact matches* by the grading scripts. Therefore, it is [imperative](#) that you implement the “print statements” exactly as specified in the assignment.

Some other items you should be aware of:

- All test cases must execute in 100,000 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.) will 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.

## Configuring MARS for Command-line Arguments

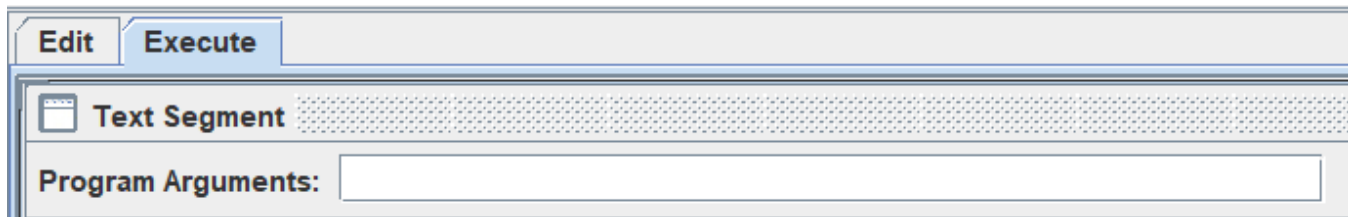
Your program is going to accept [command-line arguments](#), which will be provided as input to the program. To tell MARS that we wish to accept command-line arguments, we must go to the **Settings** menu and check the box marked:

☐ **Program arguments provided to the MIPS program**

While you’re at it, also check the box marked:

## ❑ Initialize Program Counter to global 'main' if defined

After assembling your program, in the **Execute** tab you should see a text box where you can type in your command-line arguments before running the program:



The command-line arguments must be separated by spaces. Note that your program must always be run with at least one command-line argument. When your program is assembled and then run, the arguments to your program are placed in main memory before execution. Information about the arguments is then provided to your code using the argument registers, `$a0` and `$a1`. The `$a0` register contains the number of arguments passed to your program. The `$a1` register contains the starting address of an array of strings. Each element in the array is the starting address of the argument specified on the command-line.

All arguments are saved in memory as ASCII character strings, terminated by the null character (ASCII 0, which is denoted by the character `'\0'` in assembly code). So, for example, if we want to read an integer argument on the command-line, we actually must take it as a string of digit characters (e.g., `"2034"`) and then convert it to an integer ourselves in assembly code. We have provided code for you that stores the addresses of the command-line arguments at pre-defined, unique labels (e.g., `addr_arg0`, `addr_arg1`, etc.) Note that the strings themselves are not stored at these labels. Rather, the *starting addresses* of the strings are stored at those labels. You will need to use load instructions to obtain the contents of these strings stored at the addresses: `lw` to load the address of a string, then multiple `lbu` instructions to get the characters.

## Running the Program

Running the provided `proj1.asm` file is pretty simple. Hit F3 on the keyboard or press the button shown below to assemble your code:



If your code has any syntax errors, MARS will report them in the **Mars Messages** panel at the bottom of the window. Fix any syntax errors you may have. Then press F5 or hit the Run button shown below to run your program:




Any output generated by your program will appear in the **Run I/O** panel at the bottom of the window.

## Part I: Validate the First Command-line Argument and the Number of Command-line Arguments

For this assignment you will be implementing several operations that perform computations and do data manipulation.

In `proj1.asm`, begin writing your program immediately after the label called `start_coding_here`.

You may declare more items in the `.data` section after the provided code. Any code that has already been provided must appear exactly as defined in the given file. Do not remove or rename these labels, as doing so will negatively impact grading.

The number of arguments is stored in memory at the label `num_args` by code already provided for you. You will need to use various system calls to print values that your code generates. For example, to print a string in MIPS, you need to use system call 4. You can find a listing of all the official MARS system calls on the [MARS website](#). You can also find the documentation for all instructions and supported system calls within MARS itself. Click the  in the right-hand end of tool bar to open it.

Later in the document is a list of the operations that your program will execute. Each operation is identified by a single character. The character is given by the first command-line argument (whose address is `addr_arg0`). The parameter(s) to each argument are given as the remaining command-line arguments (located at addresses `addr_arg1`, `addr_arg2`, etc.). In this first part of the assignment your program must make sure that each operation is valid and has been given the correct number of parameters. Perform the validations in the following order:

1. The first command-line argument must be a string of `strlen` one that consists of one of the following characters: F, M or P. If the argument is a letter, it must be given in uppercase. If the argument is not one of these strings or if the argument contains more than one character, print the string found at label `invalid_operation_error` and exit the program (via system call #10). This string contains a newline character at the end, so you do not need to provide your own.
2. Each of these commands expects one other argument. If the total number of command-line arguments is not two, print the string found at label `invalid_args_error` and exit the program (via system call #10). This string contains a newline character at the end, so you do not need to provide your own.

**Important:** You must use the provided `invalid_operation_error` and `invalid_args_error` strings when printing error messages. Do not create your own labels for printing output to the screen. If your output is marked as incorrect by the grading scripts because of typos, then it is your fault for not using the provided strings, and you will lose all credit for those test cases. See page 2 for more details about how your work will be graded.

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 for later assignments will fill the registers and/or main memory with random values before executing your code.

The following pages will explain how to validate the arguments for each operation.

### Examples:

Command-line Arguments	Expected Output
F DC9A F311	INVALID_ARGS
FB F311	INVALID_OPERATION
P	INVALID_ARGS
PP 5HQD2D9C8C	INVALID_OPERATION
q	INVALID_OPERATION

## Character Strings in MIPS Assembly

In assembly, a string is a one-dimensional array of unsigned bytes. Therefore, to read each character of the string we typically need to use a loop. Suppose that `$s0` contains the **base address** of the string (that is, the address of the first character of the string). We could use the instruction `lbu $t0, 0($s0)` to copy the first character of the string into `$t0`. To get the next character of the string, we have two options: (i) add 1 to the contents of `$s0` and then execute `lbu $t0, 0($s0)` again, or (ii) leave the contents of `$s0` alone and execute `lbu $t0, 1($s0)`. Generally speaking, the first approach is easier and simpler to use. Note that syntax like `lbu $t0, $t1($s0)` is not valid; an immediate value (a constant) must be given outside the parentheses.

There is no `strlen()` function or method in assembly to tell us how long a string is. Therefore, if we don't know the number of characters in a string, we need a loop which traverses a string until it reads the null character, ASCII 0 (`'\0'`). The null character denotes the end of the string in memory. For this assignment we will always assume that the second command-line argument (`args[1]`) is always present.

### Next: Process the Input

If the program determines that the first command-line argument is a valid operation and that it has been given a correct number of additional arguments, the program continues by executing the appropriate operation as specified below. Note that you are permitted to add code to the `.data` section as necessary to implement these operations.

## Part II: Interpret a String of Eight Hexadecimal Digits as a MIPS R-type Instruction

**First Argument:** M

**Second Argument:** a string of exactly 8 hexadecimal digits

Recall the instruction format for a MIPS R-type instruction:

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

(You might wish to review the [R-type instruction format](#) and a [MIPS instruction reference](#).)

The M operation takes a command-line argument consisting of exactly 8 hexadecimal digits that encode an R-type MIPS instruction and prints out the fields as follows, formatted exactly like this:

```
opcode rs_field rt_field rd_field shamt_field funct_field
```

with the six values extracted from the encoded instruction all printed in base 10.

As an example, suppose the command-line argument were the string "033DB82A". The program must first

convert this sequence of 8 ASCII characters into a 32-bit value that represents a 32-bit MIPS machine instruction. In this example, the 32-bit value would be:

```
0000 0011 0011 1101 1011 1000 0010 1010
```

Now let's reorganize the values into the six fields of an R-type instruction:

```
000000 11001 11101 10111 00000 101010
```

These correspond with the decimal values

```
0 25 29 23 0 42
```

which would be the output of the program in this case.

The program must print exactly one space between adjacent fields, ending with a newline character.

Remember that the command-line argument consists of ASCII characters. So for instance, the character 'C' is intended to encode the four-bit value 1100, but the binary representation of the letter 'C' is actually 01000011. Your code will need to deal with this discrepancy using arithmetic involving ASCII codes.

### Input Validation:

The second command-line argument must consist of exactly 8 hexadecimal digit characters (0–9, A–F), with uppercase letters only for digits A–F. **You may assume for this part of the assignment that exactly 8 valid hexadecimal digit characters given in uppercase are provided.** However, you may not assume that the instruction itself is an R-type instruction. (Recall that for an R-type instruction the opcode must be 0.) If the given opcode field is not zero, then print the string found at label `invalidargs_error` and exit the program (via system call #10).

### Examples:

Command-line Arguments	Expected Output
M 00622020	0 3 2 4 0 32
M 00155242	0 0 21 10 9 2
M 033DB82A	0 25 29 23 0 42
M 00881024	0 4 8 2 0 36
M 03050018	0 24 5 0 0 24
M 099FC62A	INVALID_ARGS
M ABCDEF12	INVALID_ARGS

## Part III: Interpret a String of Four Hexadecimal Digits as a 16-bit Floating-point Number

**First Argument:** F

**Second Argument:** a string of exactly 4 hexadecimal digits

The F operation treats the second command-line argument as a string of exactly 4 hexadecimal digit characters, 0–9, A–F (uppercase letters only), that represent a 16-bit floating-point number given in [16-bit IEEE 754 notation](#). The leftmost two characters together give the most significant byte of the 16-bit floating-point representation,

whereas the rightmost two characters provide the least significant byte. You may assume that when the F operation has been provided, the second argument always contains exactly 4 characters.

16-bit IEEE 754 floating-point numbers are formatted as follows:

	sign		exponent					fraction									
bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

The exponent is represented using *excess-15* notation. That is, after the 5-bit unsigned exponent field has been extracted from the number, 15 must be subtracted from the value to obtain the actual exponent.

When the input is not a special value (zero, infinity or NaN), the operation extracts the sign bit, exponent and fraction from the input. The operation then prints the number in the following format:

```
1.fraction_2*2^exponent
```

The fraction is printed in binary using exactly 10 bits to the right of the radix point. The exponent is printed in decimal after subtracting the bias of 15. The program must not print any leading zeroes for the exponent.

If the number is negative, then print a minus sign in front of the value. Here's an example:

```
-1.0010111100_2*2^7
```

Likewise, if the exponent is negative, print a minus sign in front of it:

```
1.0010111100_2*2^-9
```

If the entire number and/or exponent is positive, do not print a plus sign.

After printing the number in this format, print a newline character ( `'\n'` ) on the output.

When the input is a special value, print the corresponding string as indicated in the table of examples below. Do not create your own strings in the `.data` section to print these outputs. Rather, use the strings found at the labels `zero_str`, `neg_infinity_str`, `pos_infinity_str`, `NaN_str` and `floating_point_str` at the top of the provided `.asm` file.

Note that you are not required to handle *subnormal* values, i.e., those representation with an exponent field of 00000.

To implement this operation you can (and must) use only integer registers. MIPS instructions that use the floating-point registers have been disabled in the CSE 220 version of MARS.

### Input Validation:

The second command-line argument must consist of exactly 4 hexadecimal digit characters (0–9, A–F), with uppercase letters only for digits A–F. If the argument contains invalid characters print the string found at label `invalid_args_error` and exit the program (via system call #10). You may assume that the string always contains exactly 4 printable characters.

### Examples:

Command-line Arguments	Expected Output
F G13F	INVALID_ARGS
F 0000 or F 8000	Zero
F FC00	-Inf
F 7C00	+Inf
F FC01 through F FFFF	NaN
F 7C01 through F 7FFF	NaN
F 7BFA	$1.1111111010_2 \cdot 2^{15}$
F 843C	$-1.0000111100_2 \cdot 2^{-14}$
F D39D	$-1.1110011101_2 \cdot 2^5$

## Part IV: Identify a Five-card Hand from Draw Poker

**First Argument:** P

**Second Argument:** a string of exactly 10 characters

The P operation treats the second command-line argument as a string of exactly 10 characters that encode a five-card hand from draw poker. The program must be able to identify some of the [top-ranked hands](#) of five-card draw poker and print a message indicating which one it found. If the hand can be matched with two or more hands, the card of highest rank is printed. For example, a royal flush is a special kind of straight flush, so the program would produce ROYAL\_FLUSH as the output, not STRAIGHT\_FLUSH. If none of these six top-ranked hands is identified, the program prints HIGH\_CARD. You may assume that when the P operation has been provided, the second argument always contains exactly 10 characters.

The hand ranks from highest-to-lowest, along with the relevant MIPS string to print, are:

Rank	Hand Name	Label for String to Print
1	royal flush	royal_flush_str
2	straight flush	straight_flush_str
3	four of a kind	four_of_a_kind_str
4	full house	full_house_str
5	flush	simple_flush_str
6	straight	simple_straight_str
7	everything else	high_card_str

For the purposes of this assignment (namely, in straights) we will always treat Ace as a high card and never as a low card.

Playing cards are denoted using a two-character string, with the rank preceding the suit.

Rank	Character
2 through 9	'2' through '9'
10	'T'
Jack	'J'
Queen	'Q'
King	'K'
Ace	'A'



Suit	Character
Spades	\S'
Clubs	\C'
Hearts	\H'
Diamonds	\D'

For instance, the 10 of Hearts would be encoded as "TH"; the 2 of Spades is "2S".

Five such pairs of characters encode a five-card hand. For example, "7H3H5H4H6H" encodes a straight flush consisting of the 3 of Hearts through 7 of Hearts. It is possible for the same card to appear multiple times in a single string. For example, "5H7D5H5C5H" contains three instances of the 5 of Hearts and one instance of the 5 of Clubs that form a valid four-of-a-kind.

### Input Validation:

The second command-line argument must consist of exactly 10 characters that encode valid cards. If the argument contains invalid characters and/or card encodings, print the string found at label `invalid_args_error` and exit the program (via system call #10). You may assume that the string always contains exactly 10 printable characters.

### Examples:

Command-line Arguments	Expected Output
P KSQSTSJSAS	ROYAL_FLUSH
P KSQSTSJS2S	SIMPLE_FLUSH
P JSQSTSJSQS	SIMPLE_FLUSH
P 7H3H5H4H6H	STRAIGHT_FLUSH
P 2H3H5H4S6H	SIMPLE_STRAIGHT
P 2H5S5D5S2C	FULL_HOUSE
P 5H7D5D5C5S	FOUR_OF_A_KIND
P 5H7D5H5C5H	FOUR_OF_A_KIND
P KH7D2D4C5S	HIGH_CARD
P KH7D2D4C5F	INVALID_ARGS
P KH1D2D4C5S	INVALID_ARGS
P KH7D24DC5S	INVALID_ARGS

## 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 im-

portance 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 `proj1.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 `proj1.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.