

CSE 220: Systems Fundamentals I

Stony Brook University

Homework Assignment #1

Fall 2018

Assignment Due: September 21, 2018 by 11:59 pm

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.
- 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 homework 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.
- You must use the Stony Brook version of MARS posted on Piazza. 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.
- For Homework Assignments #2 and later, do not submit a file with the functions/labels `main` or `_start` 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 Homework Assignment #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. For Homework Assignments #2 and later 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:

- All test cases must execute in 10,000 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.
- 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.

Assignment Objectives

The primary objective of this homework assignment is for you to become familiar with the basics of MIPS assembly language, including bitwise operators, conditionals, loops and system calls.

Getting Started

Visit Piazza and download the “bare bones” file `hw1.asm`. Open the file and fill in the following information at the top:

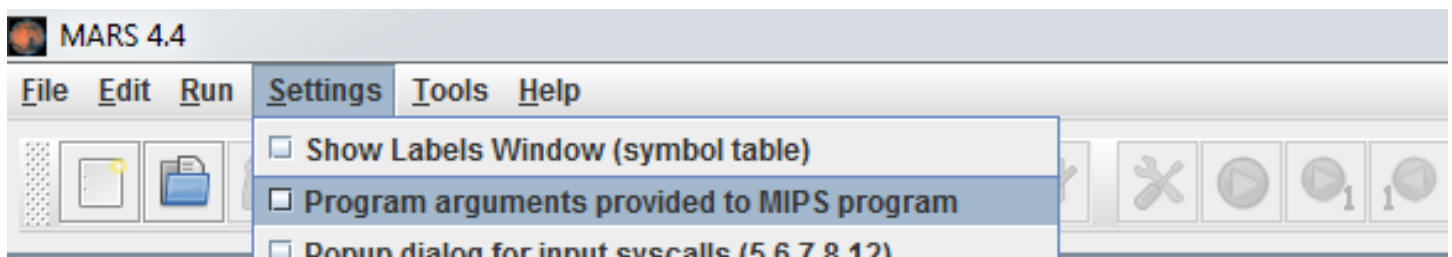
1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

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.

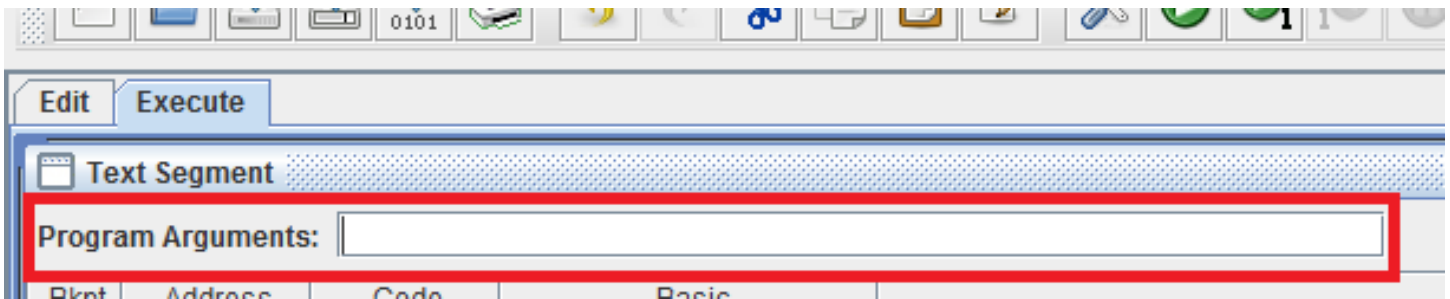
Configuring MARS for Command-line Arguments

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 select:

Program arguments provided to the MIPS program.



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.

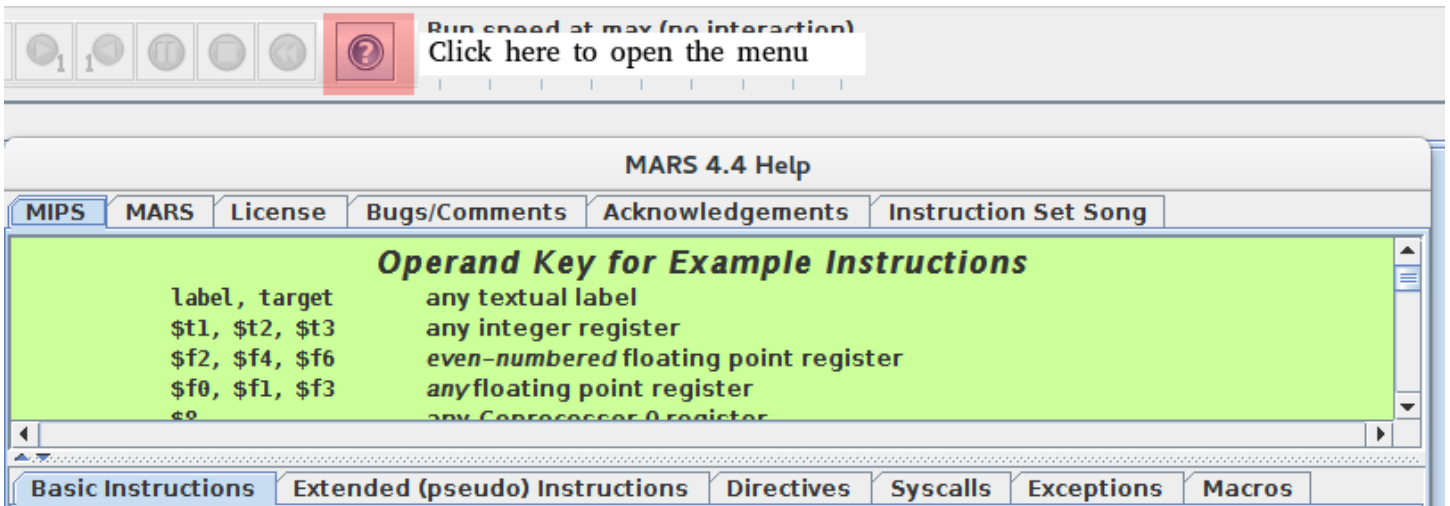
Part I: Validate the First Command-line Argument and the Number of Command-line Arguments

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

In `hw1.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 specified must appear exactly as defined in the provided 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 integers and strings 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 tool bar to open it:



The following 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.). Not all operations expect the same number of parameters. 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 length one that consists of one of the following characters: F, C or 2. If the argument is a letter, it must be given in uppercase. If the argument is not one of these strings, print the string found at label `invalid_operation_error` and exit the program (via system call 10).
2. If the first command-line argument is F or 2, then there must be exactly 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).
3. If the first command-line argument is C, then there must be exactly three other arguments. If the total number of command-line arguments is not four, print the string found at label `invalid_args_error` and exit the program (via system call 10).

Important: You must use the provided `invalid_operation_error` and `invalid_args_error` strings when printing error messages. There are also five other strings marked with the comment `# Output strings`, which you must use for generating the output of your program. 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 1 for more details about how your work will be graded.

Examples:

<i>Command-line Arguments</i>	<i>Expected Output</i>
DB abc	INVALID_OPERATION
f xyz	INVALID_OPERATION
1	INVALID_OPERATION
F abc def	INVALID_ARGS
2	INVALID_ARGS
C 145 6842	INVALID_ARGS
C 1 2 3 4	INVALID_ARGS

Review of 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)`. Since most of the command-line arguments for this program can be of arbitrary length, you will probably find that you must use the first approach in most cases. Also 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 `length()` function or method in assembly to tell us how long a string is. Therefore, 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.

If your program determines that the first command-line argument is a valid command and that it has been given a correct number of additional arguments, your 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 0s and 1s as a Two's Complement Number

First Argument: 2

Second Argument: a two's complement value (given as a string of 0 and 1 characters)

The 2 operation treats the second command-line argument as a string of 0 and 1 characters that represent a two's-complement number. The leftmost character is the most significant bit of the integer. The argument might contain fewer than 32 characters. The operation converts the string into a two's-complement value and prints it in decimal to the screen, with a newline (`'\n'`) at the end.

Input Validation:

The second command-line argument must consist only of at most 32 0 and 1 characters. If the argument contains invalid characters or more than 32 characters, print the string found at label `invalid_args_error` and exit the program (via system call 10).

You may assume that valid, converted values can fit within a single 32-bit register.

Examples:

<i>Command-line Arguments</i>	<i>Expected Output</i>
2 11111111111111111111111111111111	INVALID_ARGS
2 000000000000	0
2 0101	5
2 0101101	45
2 01111111111111111111111111111111	2147483647
2 1101110	-18
2 111111111	-1
2 10000000000000000000000000000000	-2147483648

Part III: Interpret a String of Hexadecimal Digits as a 32-bit Floating-point Number

First Argument: F

Second Argument: a string of exactly 8 hexadecimal digits

The F operation treats the second command-line argument as a string of exactly 8 hexadecimal digit characters, 0–9, A–F (uppercase letters only), that represent a 32-bit floating-point number given in IEEE 754 notation. The first two characters together give the most significant byte of the 32-bit floating-point representation, whereas the last two characters provide the least significant byte. 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:

$$1.\text{fraction}_2 * 2^{\text{exponent}}$$

The fraction is printed in binary using exactly 23 bits to the right of the radix point. The exponent is printed in decimal after subtracting the bias of 127. 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:

$$-1.00110111101000010100000_2 * 2^{15}$$

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

$$1.00110111101000010100000_2 * 2^{-15}$$

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. A string at label `nl` has been provided in the `.data` section for you that contains the newline character.

When the input is a special value, print the corresponding string as indicated in the table 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.

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 8 hexadecimal digit characters (0–9, A–F), with uppercase letters only for digits A–F. If the argument contains invalid characters and/or is of the wrong length, print the string found at label `invalid_args_error` and exit the program (via system call 10).

Examples:

<i>Command-line Arguments</i>	<i>Expected Output</i>
F 1234567	INVALID_ARGS
F G1231234	INVALID_ARGS
F 6318675309	INVALID_ARGS
F 00000000 or F 80000000	Zero
F FF800000	-Inf
F 7F800000	+Inf
F 7F800001 thru F 7FFFFFFF	NaN
F FF800001 thru F FFFFFFFF	NaN
F 42864000	$1.000011001000000000000000_2 \cdot 2^6$
F 14483B47	$1.10010000011101101000111_2 \cdot 2^{-87}$
F 94483B47	$-1.10010000011101101000111_2 \cdot 2^{-87}$
F C46AB32C	$-1.11010101011001100101100_2 \cdot 2^9$

Part IV: Convert a Positive Integer from One Base to Another Base

First Argument: C
Second Argument: a string of decimal digits representing a number in a base
Third Argument: a string of decimal digits representing the base we are converting from
Fourth Argument: a string of decimal digits representing the base we are converting to

This operation takes a number given in one base (the “from” base), converts it to another base (the “to” base) and prints out the number in the other base, with a newline (‘\n’) at the end. For example, C 165 9 7 means that we want the computer to print the value of 165_9 as it would be represented in base 7 (i.e., 260_7).

The SBU version of MARS contains a special system call (#84) for converting a string of ASCII digit characters into a decimal integer. You are welcome to use that system call for converting the third and fourth arguments to the “C” operation into integers. For example, suppose `$s0` contained the starting address of a string of digit characters we wanted to convert into an integer. We would write this code to convert it into a number:

```
move $a0, $s0
li $v0, 84
syscall
```

The resulting integer would be available in `$v0`.

Input Validation:

You may assume that only digits appear in the second, third and fourth command-line arguments. You may assume that the largest value we might need to convert is $2^{31} - 1$. You may assume that the third and fourth command-line arguments represent integers in the range $[2, 10]$. However, the program must check that the input number (second argument) is a valid representation of a number in the “from” base (third argument). If this is not the case, the operation prints the string found at label `invalid_args_error` and exits the program (via system call #10).

Examples:

<i>Command-line Arguments</i>	<i>Expected Output</i>
C 8154 8 7	INVALID_ARGS
C 000 5 10	0
C 23568 9 10	15776
C 0101 3 8	12
C 4123 5 5	4123
C 111102365 7 2	11001101001100001100000

Academic Honesty Policy

Be aware that before being able to submit your homework, you will be required on Blackboard to 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 `hw1.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 find the link for the Academic Honesty Statement for this homework assignment.
3. Hit “Begin” to view the Academic Honesty Statement. Read the statement. If you agree to the statement, enter YES in the text box and hit the “Save and Submit” button. If you do not agree to the statement, speak with the instructor.
4. Click on “Assignments” again in the left-hand menu and click the link to submit this assignment, which should be now visible.
5. Click the “Browse My Computer” button and locate the `hw1.asm` file. Submit only that one `.asm` file.
6. 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 steps 4–6 again. We will grade only your last submission.