

CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #1

Spring 2020

Assignment Due: Friday, February 21, 2020 by 11:59 pm

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`. (Not a typo! We are using the Fall 2019 version of MARS.) 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 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.
- 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:

- Each test case 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

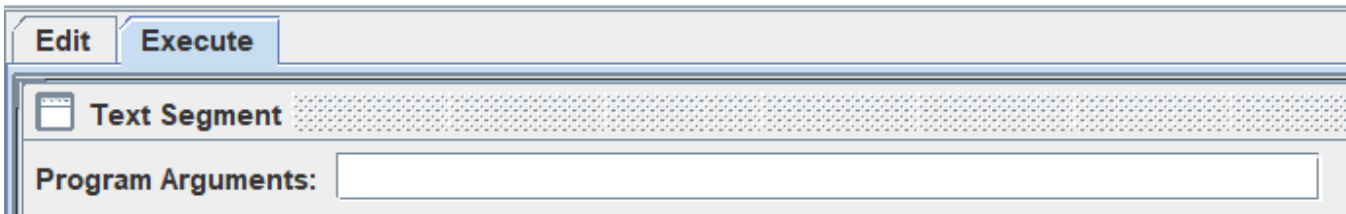
Your program is going to accept [command-line arguments](#), which will be provided as input to the program. To tell MARS that we wish to accept command-line arguments, we must go to the **Settings** menu and check the box marked:

☐ **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. For instance, `lw $s0, addr_arg2`, followed by `lbu $t0, 4($s0)` would store in `$t0` the fourth character (0-based index) of the *third* command line argument.

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 dequeue 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: B, C, D or E. 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. The B and D operations expect one additional argument. If the total number of command-line arguments for these commands 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.
3. The C operation expects three additional arguments. If the total number of command-line arguments for this command is not four, print the string found at label `invalid_args_error` and exit the program (via system call 10).
4. The E operation expects four additional arguments. If the total number of command-line arguments for this command is not five, 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. 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
Z 2 Hello	INVALID_OPERATION
Convert S 2 0x8000000A	INVALID_OPERATION
B ASTS7S6S2SQHJH9HKC4CKD9D3D UhOh	INVALID_ARGS
C 2 0x8000000A	INVALID_ARGS
C	INVALID_ARGS
D 0x84ADE23A 0x002938EF	INVALID_ARGS
E 12 07 09 -00004 1423 92	INVALID_ARGS

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.

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: Decode a String of Eight Hexadecimal Digits that Encode a MIPS I-type Instruction

First Argument: `D`

Second Argument: the characters `0x`, followed by exactly 8 hexadecimal digits

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

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

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

The `D` operation takes a command-line argument consisting of the characters `0x`, followed by exactly 8 hexadecimal digits that encode an I-type MIPS instruction and which prints out the fields as follows, formatted exactly like this:

```
opcode rs_field rt_field immediate_field
```

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

As an example, suppose the command-line argument were the string "0x30E9FFFC". The program must first convert the rightmost 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:

```
0011 0000 1110 1001 1111 1111 1111 1100
```

Now let's reorganize the values into the four fields of an I-type instruction:

```
001100 00111 01001 1111111111111100
```

These correspond with the decimal values

```
12 7 9 -4
```

which would be the output of the program in this case. Note that the 16-bit *immediate* field of a real I-type instruction is *usually* signed, but not always. For the purposes of this assignment we will always assume that the immediate field is always signed. (For example, with the `addiu` instruction, the immediate is unsigned. We will pretend that it's a signed immediate for this assignment.)

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 the seven-bit value `01000011`. Likewise, the character `'3'` has the 7-bit ASCII code `0110011`, not the four-bit value `0011`. Your code will need to deal with this discrepancy by doing some conversions. (Hint: the math behind those computations is pretty simple – see if you can figure it out!)

Input Validation:

The second command-line argument must consist of the characters `0x`, followed by exactly 8 hexadecimal digit characters (0–9, A–F), with uppercase letters only for digits A–F. If the command-line argument is not formatted in this way, print the string found at label `invalid_args_error` and exit the program (via system call 10). You may assume that the command-line argument is always exactly 10 characters in length.

Examples:

Command-line Arguments	Expected Output
0x30E9FFFC	12 7 9 -4
0x60303A98	24 1 16 15000
0x97B78000	37 29 23 -32768
0X8675309A	INVALID_ARGS
0x8675Z09A	INVALID_ARGS

(Does not start with `0x`.)

(Contains an invalid character.)

Part III: Encode Four Numerical Fields as an I-Type MIPS Instruction

First Argument:	E
Second Argument:	a string that encodes a positive, two-digit decimal integer in the range $[0, 63]$ (the <i>opcode</i> field). A leading zero is provided if needed.
Third Argument:	a string that encodes a positive, two-digit decimal integer in the range $[0, 31]$ (the <i>rs</i> field). A leading zero is provided if needed.
Fourth Argument:	a string that encodes a positive, two-digit decimal integer in the range $[0, 31]$ (the <i>rt</i> field). A leading zero is provided if needed.
Fifth Argument:	a string that encodes a five-digit decimal integer in the range $[-2^{15}, 2^{15} - 1]$ (the <i>immediate</i> field). Leading zeroes are provided as needed to pad the number to five digits. A minus sign begins the number if it is negative. A positive number is not preceded by a plus sign.

The `E` operation takes four decimal values, treats them as the four fields of a MIPS I-type instruction, and, using shifting and masking operations, combines the four values into a single 32-bit integer that represents an I-type instruction. The program then prints this value in hexadecimal using `syscall 34`. An example will help to clarify this process.

Suppose the command-line arguments were the strings

"E" "12" "07" "09" "-00004"

The program must process each numerical argument in turn, converting each of the four values into an integer:

12 07 09 -4

Remember that all data in a computer is stored in binary, so these values will be stored as the 32-bit values

[illegible]

We treat these four values as the *opcode*, *rs*, *rt* and *immediate* fields. Next, using bitwise operations, we need to concatenate the six rightmost bits of the opcode, five rightmost bits of the *rs* and *rt* fields, and 16 rightmost bits of the immediate field:

001100 00111 01001 111111111111111100

When printed in hexadecimal, the output value (produced by syscall 34) will be

```
0x30e9fffc
```

Input Validation:

You may assume that the second, third and fourth command-line arguments always consist of exactly (and only) two decimal digit characters that encode a positive integer. You may also assume that the fifth argument consists of exactly (and only) five decimal digit characters with an optional negative sign on the left. However, you may not assume that the values are in the legal ranges specified above. If any command-line argument is outside its legal range, print the string found at label `invalid_args_error` and exit the program (via system call 10).

Examples:

In the examples below, hexadecimal digits in the sample output are given in uppercase, but MARS produces lowercase letters. The grading scripts will account for this difference. Your code does not have to produce uppercase letters.

Command-line Arguments	Expected Output
12 07 09 -00004	0x30E9FFFC
24 01 16 15000	0x60303A98
63 31 31 32767	0xFFFF7FFF
02 11 22 -67989	INVALID_ARGS
33 52 14 -00412	INVALID_ARGS

(-67989 is out of range.)

(52 is out of range.)

Part IV: Convert from One Integer Representation to Another

First Argument: C

Second Argument: the character 1, 2 or S, which represents the integer representation the function is converting *from* (the *source* representation scheme)

Third Argument: the character 1, 2 or S, which represents the integer representation the function is converting *to* (the *destination* representation scheme)

Fourth Argument: the characters 0x, followed by exactly 8 hexadecimal digits

The C operation converts an integer represented in one integer representation (the *source* representation scheme) and prints that same integer in binary as it would be represented in the *destination* representation scheme.

As an example, the command-line arguments

```
"C" "2" "S" "0xFFFFFFFFD"
```

indicate that the value 0xFFFFFFFFD is an integer represented in two's complement and that the program should output the signed/value representation of this value. Let's write 0xFFFFFFFFD in binary:

```
1111 1111 1111 1111 1111 1111 1111 1101
```

We are told that this is a two's complement value, so the leading 1 indicates that this is a negative value. To represent it in signed/magnitude, we will need to take the two's complement of that value, which yields:

```
0000 0000 0000 0000 0000 0000 0000 0011
```

Because the number is negative, we set the sign bit to 1:

```
10000000000000000000000000000011
```

which would be the output of the program in this case. You can use syscall 35 to print a register's contents in binary.

Valid second arguments to the operation are 1, 2 or S, which indicate the one's complement, two's complement and sign/magnitude representation schemes, respectively. It is possible that the second and third arguments have the same value, in which case no conversion is needed.

Please try to always keep in mind that a computer represents all data in binary and, in the case of the MIPS architecture, integers are represented in (binary) two's complement notation. We as human beings can think in terms of decimal arithmetic and decimal values, but the computer represents *everything* in binary.

When converting “negative zero” *from* one’s complement or signed/magnitude representation *to* one’s complement or signed/magnitude representation, always write the result as “positive zero”, i.e., 32 binary zeroes. If the source and destination representations are the same, then leave the bits unchanged (e.g., we want to “convert” negative zero from one’s complement into one’s complement; in this case just write the representation of negative zero). “Positive zero” should always be represented in the destination representation as 32 binary zeroes, regardless of which representation is used.

Input Validation:

You may assume that the input value can be represented in the target representation. (This is only really an issue for -2^{31} .) You may also assume that the provided hexadecimal number is properly formatting, beginning with `0x` and consisting of exactly 8 hexadecimal digits, 0 – 9 and A–F (uppercase letters only). However, the source and destination representation codes could be invalid. If the source or destination is not one of 1, 2 or S, print the string found at label `invalid_args_error` and exit the program (via system call 10).

Examples:

Please note that the examples given below are in no way comprehensive!

[illegible]

Part V: Score a Hand of Cards from the Card Game *Bridge*

First Argument: B

Second Argument: a string of 26 characters that encode a 13-card hand of Bridge

The B input accepts a string of 26 characters that encode a 13-card hand of Bridge and scores the hand according to simplified rules laid out below. The operation prints the score in decimal.

Each card is represented by a two-character code: the rank, followed by the suit. The ranks 2–10, Jack, Queen, King and Ace are represented by the characters 2, 3, . . . , 9, T, J, Q, K and A, respectively. The suits Hearts, Diamonds, Clubs and Spades are represented by the characters H, D, C and S, respectively.

There are two aspects to computing the score of the hand:

- Each Ace is worth four points, each King three points, each Queen two points, and each Jack one point. The other cards contribute no points.
- The distribution of cards in the hand also contribute to the score. Specifically, if the hand has only two cards of a particular suit, then the hand is worth an extra point. If the hand has only one card of a particular suit, then the hand is worth two extra points. If the hand has no cards of a particular suit, then the hand is worth three extra points.

Let's consider the input 6DQH2S3SJS6S6H5H5DAH8SJD8D. To make it easier to understand how to score this hand, let's pull it apart and rearrange the cards. Note that these steps are not needed to implement the operation.

First isolate the cards:

6D QH 2S 3S JS 6S 6H 5H 5D AH 8S JD 8D

Next group by suit:

5D 6D 8D JD 5H 6H QH AH 2S 3S 6S 8S JS

We see that we have 2 Jacks, 1 Queen and 1 Ace, which together contribute $(2 \times 1) + (1 \times 2) + (1 \times 4) = 8$ points. Additionally, there are no Clubs in the hand, which adds 3 more points, bringing the total score of the hand to 11 points.

Input Validation:

You may assume that all input is valid. Every card will be unique in the 13-card hand.

Examples:

Command-line Arguments	Expected Output
KSQH5SJH9DAD7DJD9SKC6H9CJC	15
ASTS7S6S2SQHJH9HKC4CKD9D3D	14
3H7DKS3STCQD8D9S4STD7C4D2S	8
6DKDQHQD2HJH5D3HTH2DAH8D7H	18
TS8SJS7S6S5S3SASQS9S4S2SKS	19

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 `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.