CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #1
Total Points: 50

Spring 2021

Assignment Due: Sunday, Feb 28, 11:59 PM (EST)

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 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 the course website and download *MarsSpring2021.jar*. Do not download a MARS version from the web. The version of MARS we use is customized for this course. Next, use **hw1.asm** to write your code. This file also contains starter code (**hw1.asm**) that will help you get started. Do not change any part of the starter code. Submit **hw1.asm**.

How Your CSE 220 Assignments Will Be Graded

Your submission will be graded almost entirely by automated scrips. The script will check your program's output against expected outputs. Make sure that the values and messages you print to the console are exactly as specified in the document or the starter code. If you do not pay attention to this, then you will not get credit if a test case fails because you do not have the exact message or the expected value.

Some more criteria on which your program will be evaluated:

- Each test case must execute in 10,000 instructions or fewer. The no. of instructions you have in your program has a direct impact on the performance of the code. Hence, you are advised to pay attention to this detail. 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 adversely affect grading. Do not leave erroneous print-outs in your code.

- This HW doc has sample test cases that you should test your code with. However, we will evaluate your program with other test cases too. It is your responsibility to think about edge cases.
- The test cases and expected results used for testing will be released via comments on Blackboard.

Configuring MARS for Command-line Arguments

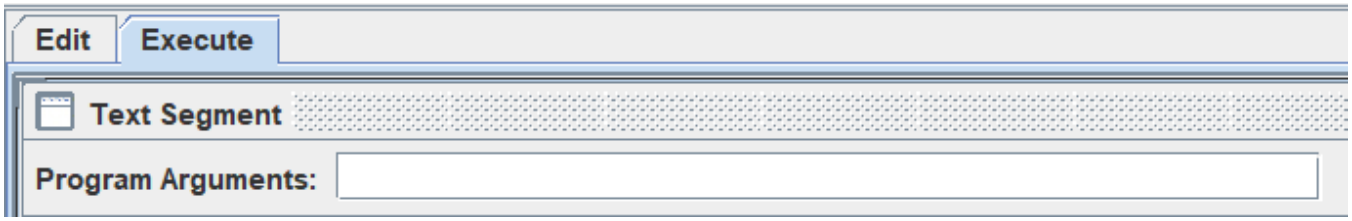
Your program is going to accept command-line arguments, which will be provided as input to the program. To tell MARS that we wish to accept command-line arguments, go to the Settings menu and check the box marked:

Program arguments provided to the MIPS program

Additionally, 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. 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. The starter file has code that stores the addresses of the command-line arguments at pre-defined, unique labels (e.g., `arg1_addr` and `arg2_addr`). You should use these labels. 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, arg1_addr`, followed by `lbu $t0, 4($s0)` would store the character located at index 4 in `$t0` (assuming 0-based indexes) of the 1st command line argument.

Running the Program

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 1: Validate the First Command-line Argument and the Number of Command-line Arguments (5 points)

For this assignment you will be implementing several operations that perform computations on a hexadecimal string.

In the starter code file provided to you (*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 provided must appear exactly as defined in the given file. Do not delete or rename these labels, as doing so will negatively impact grading.

The number of arguments is stored in memory at the label `num_args` in the starter code. 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 (http://courses.missouristate.edu/KenVollmar/Mars/Help/SyscallHelp.html).
You can also find the documentation for all instructions and supported system calls within MARS itself. Click the blue question mark in the right-hand end of the tool bar to open it.

The program takes an operation and a hexadecimal string as command-line arguments. The first argument is a character that indicates the operation to perform. Its address is `arg1_addr`. The second argument is the hexadecimal string on which the operation will be executed and will be stored in the address `arg2_addr`. In this first part of the assignment your program must verify that the arguments provided are valid. Perform the validations in the following order:

1. The program should accept *exactly two arguments*. If more or less than two arguments are provided, then print the error message found at label `WrongArgMsg`.

2. The first command-line argument must be a string of at least one character. The first character must be one of the following: ‘O’,’S’,’T’,’I’,’E’,’C’,’X’, or ‘M’. The characters must be in upper-case. If the first character is not one of the aforementioned characters, then print the string found at label `ErrMsg` and exit the program (via system call 10). If the string has more than one character, then only the first character should be considered and the remaining should be ignored.

3. The second command-line argument must be a string of at least 10 characters. The first two characters should be ‘0’ followed by ‘x’ (should be lower case). The following eight characters should be either a digit [0-9] or an uppercase letter [A-F]. In other words, the second argument should be a valid 32-bit hexadecimal string. If the string has more than 10 characters, then all characters after the first 10 should be ignored.

Important: You must use the strings provided in labels `WrongArgMsg` and `ErrMsg` 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 you will lose all credit for those test cases.

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 garbage values before executing your code.

Sample Test Cases

The meaning of each valid operation is described in subsequent parts.

Command-line Arguments	**Label for String to Print**	**Expected Output**
`T`	`WrongArgMsg`	`You must provide exactly two arguments`
`g 12345678`	`ErrMsg`	`Invalid Argument`
`T 12345678`	`ErrMsg`	`Invalid Argument`

Tip:
Processing 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 need to iterate over the contents of the string. Let us assume that `$s0` contains the base address of the string (i.e., 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 `$t0`. To get the next character of the string, we have two options: (i) increment the address in `$s0` by 1 and then execute `lbu $t0, 0($s0)` again, or (ii) leave the contents of `$s0` alone and execute `lbu $t0, 1($s0`. Either option is valid. You should make an appropriate choice based on the context and convenience (e.g., length of the string). Syntax like `lbu $t0, $t1($s0)` is invalid; the offset (i.e., the value outside the parenthesis) must be an immediate value (a constant).

If the program determines that the first command-line argument is a valid operation and that it has been given a valid 32-bit hex representation as the second argument, 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 2: Identify Components of an I-type Instruction (25 points)

An I-type instruction in MIPS has the following format:

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

Assume that the string provided as the second argument to our program is a hexadecimal encoding of an I-type instruction. Our goal is to extract the fields, **opcode**, **rs**, **rt**, and **immediate**, of an I-type instruction from given encoding. For example, assume that the second argument is the hexadecimal string "`0x30E9FFFC`". The program must first convert the string to a 32-bit binary representation. For the string "`0x30E9FFFC`", the 32-bit binary value will be:

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

Next, the program should reorganize the bits as per the operation provided in the first argument

Part 2A: Operation 'O' (6 points)

If the first argument is the character 'O', then program should print the **opcode** field. In the example that we are discussing, the opcode field is the first 6 bits of the binary value (read left to right) since the number of bits in the opcode field of an I-type instruction is 6. Hence, the opcode should be `001100`, which is `12` in decimal. The program should print this decimal value. You can assume that the opcode will always be *unsigned*.

Sample Test Cases

Command-line Arguments	**Expected Output**
`O 0x30E9FFFC`	`12`
`O 0x10E9FFFC`	`4`
`O 0x30e9fffc`	`Invalid Argument`

Part 2B: Operation 'S' (6 points)

If the first argument is the character 'S', then the program should print the **rs** field. In the running example, the rs field is 5 bits of the binary value starting from 7th bit to the 11th bit (inclusive; read left to

right). Hence, the rs field should be `00111`, which is `7` in decimal. The program should print this decimal value. You can assume that the rs field will always be *unsigned*.

Sample Test Cases

Command-line Arguments	**Expected Output**
`S 0x30E9FFFC`	`7`
`S 0x1029FFFC`	`1`
`S 0x30e9fffc`	`Invalid Argument`

Part 2C: Operation 'T' (6 points)

If the first argument is the character 'T', then the program should print the **rt** field. So for the hexadecimal encoding `0x30E9FFFC`, the rt field is 5 bits of the corresponding binary value starting from 12th bit to the 16th bit (inclusive; read left to right). Hence, the rt field should be `01001`, which is `9` in decimal. The program should print this decimal value. You can assume that the rt field will always be *unsigned*.

Sample Test Cases

Command-line Arguments	**Expected Output**
`T 0x30E9FFFC`	`9`
`T 0x10A6FFFC`	`6`
`T 0x30e9fffc`	`Invalid Argument`

Part 2D: Operation 'I' (7 points)

If the first argument is the character 'I', then the program should print the **immediate** field. Assuming that the hexadecimal encoding provided as the second argument is `0x30E9FFFC`, the **immediate** field is the last 16 bits of the corresponding binary value. Hence, the immediate should be `1111 1111 1111 1100`. We will assume that the 16-bit immediate will always be *signed*. Since in this example, the MSB of the immediate is 1, the corresponding decimal value will be negative. In this case, it will be -4.

Sample Test Cases

Command-line Arguments	Expected Output
`I 0x30E9FFFC`	`-4`
`I 0x10A6FFFF`	`-1`
`I 0x30e9fffc`	`Invalid Argument`

Part 3: Odd/Even (3 points)

The purpose of the 'E' operation is to determine if a given hexadecimal string is even or odd. In this part, the first argument is the operation 'E' and the second argument is a hexadecimal string. Just like in the previous part, you should first obtain the binary number representation of the hexadecimal string. You should print the message "Even" if the number is even and the message "Odd" if the number is odd. You can use the labels `EvenMsg` and `OddMsg` in the .data section of the starter code.

Sample Test Cases

Command-line Arguments	Expected Output
`E 0x30E9FFFC`	`Even`
`E 0x10A6FFFF`	`Odd`
`E 0x30e9fffc`	`Invalid Argument`

Part 4: Counting (4 points)

Operation 'C' will be used to count the no. of 1s in the binary representation of a hex string. In this part, the first argument will be the operation 'C' and the second argument will be a valid hex string. You should translate the hex string to a binary number and compute the no. of occurrences of 1 in the binary number. You should print out the no. of occurrences of 1. For example, if the first argument is 'C' and the second argument is `0x30E9FFFC`, then the no. of occurrences of 1 in the corresponding binary representation is 21.

Command-line Arguments	Expected Output
`C 0x30E9FFFC`	`21`
`C 0xFFFFFFFF`	`32`
`C 0x30e9fffc`	`Invalid Argument`

Part 5: Floating-point Exponent (13 points)

Consider the IEEE 754 floating point format

1 bit	8 bits	23 bits
sign	exponent	mantissa

Assume that the hexadecimal string provided as the second argument is an encoding of the IEEE 754 floating point format. For instance, if the hexadecimal string is "`0x30E9FFFC`", then the corresponding binary number in the IEEE 754 floating-point format will be:

```
0 01100001 11010011111111111111100
```

Part 5A: Exponent (7 points)

The exponent of an IEEE 754 floating point number is encoded in 127-excess form to allow for signed exponents. The 127-excess representation is obtained by adding 127 to the actual value. For example, in the above example the exponent `01100001` is in 127-excess form. The decimal value of the exponent in 127-excess form is 97. This was obtained by adding 127 to the actual value -30.

The operation 'X' should extract the actual decimal value of the exponent from a hexadecimal encoding of an IEEE 754 floating point number. Assuming that the first argument to the program is 'X' and the second argument is the hexadecimal string "`0x30E9FFFC`", the program should print -30 since -30 is the actual decimal value of the exponent.

Command-line Arguments	Expected Output
`X 0x30E9FFFC`	`-30`
`X 0xFFFFFFFF`	`128`
`X 0x30e9fffc`	`Invalid Argument`

Part 5B: Mantissa (6 points)

The mantissa of an IEEE 754 floating point number is a 23-bit binary number as shown above. To increase precision, the 23-bit binary mantissa is always normalized. Hence, we prepend `1.` to the 23-bit mantissa to obtain the actual number. For example, the mantissa for the hexadecimal string "`0x30E9FFFC`" is `11010011111111111111100`. But, the actual number is `1.11010011111111111111100`.

Assuming that the first argument to the program is the character 'M' and the second argument is a hexadecimal string, the program should print a string that denotes the actual number in binary. For example, if the hexadecimal string is "`0x30E9FFFC`", then the program should print `1.11010011111111111111100000000000`. Notice how nine 0s have been appended to the 23-bit mantissa to make it 32-bit. In total, the printed string should have 34 characters.

Command-line Arguments	**Expected Output**
`M 0x30E9FFFC`	`1.11010011111111111111100000000000`
`M 0xFFFFFFFF`	`1.11111111111111111111111000000000`
`M 0x30e9fffc`	`Invalid Argument`

How to Submit Your Work for Grading

Submit your **hw1.asm** file to Blackboard under *Assignments -> Assignment 1*.

1. Make sure the name of the file is as given (**hw1.asm**).
2. Make sure that you have used the right labels while printing output.
3. Do not leave additional print statements meant for debugging in your code.
4. Verify that the no. of instructions in your program does not exceed 10,000.

You can submit any number of times till the due date. We will grade only your last submission.