CSE 220: Systems Fundamentals I
Homework #1
Spring 2018

Assignment Due: Friday, Feb 16 by 11:59 pm via Sparky

READ THE WHOLE DOCUMENT TWICE BEFORE STARTING!

DO **NOT** COPY/SHARE CODE! We will check your assignments against this semester and previous semesters!

Download the Stony Brook version of MARS posted on Piazza. **DO NOT USE** the MARS available on the official webpage. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.

You personally must implement the assignment in MIPS Assembly language by yourself. You may not 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 assignment. You may also not write a code generator in MIPS Assembly that generates MIPS Assembly.

All test cases MUST execute in 5,000 instructions or less. Efficiency is an important aspect of programming.

Any excess output from your program (debugging notes, etc) WILL impact your grading. Do not leave erroneous printouts in your code!

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.

Introduction

The goal of this homework is to become familiar with basic MIPS instructions, syscalls, basic loops, conditional logic and memory representations.

In this assignment, you will familiarize yourself with a feature of the C programming language known as `structs`. A struct is a composite data type that is used to group variables under one name in a contiguous block of memory. You will be reading, populating and editing the fields of a struct. The struct you will be dealing with is depicted on the next page.

```
1 struct cse220_student {
2   int      id;          // size: 4b (b == bytes)
3   String   netid;       // size: 4b; starting address of the string
4   float    percentile;  // size: 4b; single precision IEEE-754
5   short    grade;       // size: 2b; byte[0] is letter and byte[1] is sign
6   nibble   recitation; // size: 4 bits; NOTE: nibble isn't a true C datatype
7   nibble   favtopics;  // size: 4 bits; bit vector
8 };
```

This struct is 15 bytes in size in total.

Line 1

This is the type declaration of the struct. This particular struct is named `cse220_student`. If you are interested in learning more about structs in C you can start here.

Line 2

This is the first field in the struct. This is an integer which represents the SBU ID of the student. A 32 bit integer takes up 4 bytes of memory.

Line 3

This field contains a "reference" to the NetID for the student. A string in MIPS is an array of characters (ASCII bytes). The struct holds the the starting address of the array of characters in memory.

Line 4

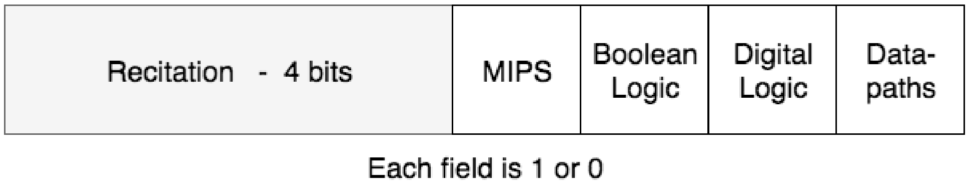
This field stores as a single precision float value. This float is stored in IEEE-754 single precision format. The float data type is 4 bytes in size.

Line 5

This is a 2 byte field that will hold the characters representing the letter grade of the student. Remember, ASCII characters are byte values.

Line 6 & 7

The `recitation` field is a 4 bit unsigned integer. The `favtopics` field is a 4 bit field which is a bit vector of the student's favorite topics. Combined, these fields are 1 byte in size.



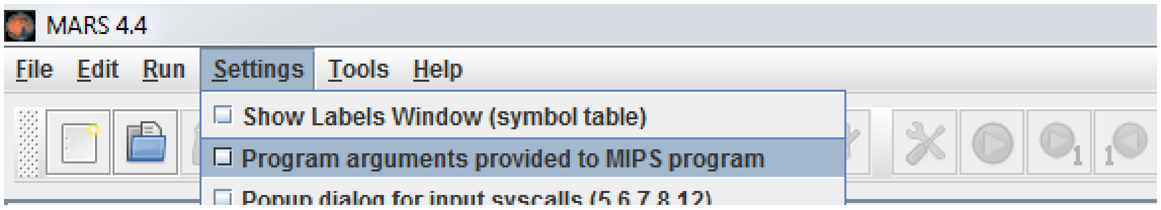
Part 1: Command-line arguments

In the first part of the assignment you will initialize your program and identify the different command line arguments.

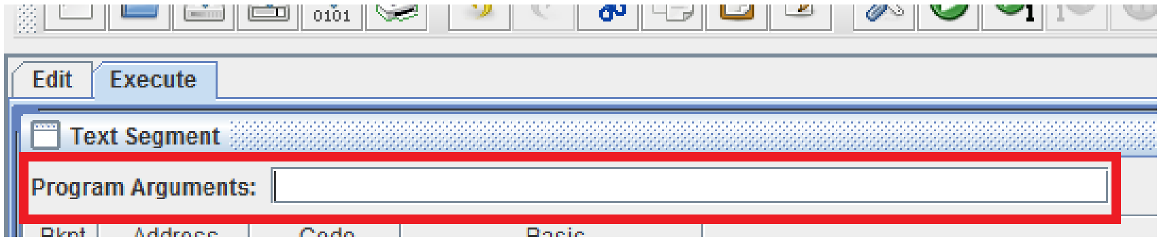
Configuring MARS for Command-line Arguments

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`.


MARS 4.4
File
Edit
Run
Settings
Tools
Help
Show Labels Window (symbol table)
Program arguments provided to MIPS program
Popup dialog for input syscalls (5,6,7,8,12)

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.



Each command line argument should be separated by a space.

❗ Your program must ALWAYS be run with at least one command line argument! You can expect that command line arguments will always be given in the correct order for this assignment.

When your program is assembled and then run, the arguments to your program are placed in memory. 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.

ℹ All arguments are saved in memory as ASCII character strings.

Your program will take the following command-line arguments separated by space:
`NETID ID GRADE RECITATION FAVTOPICS PERCENTILE`

- `NETID`: Student's NetID.
- `ID`: Student's SBU ID.
- `GRADE`: Student's CSE 220 letter grade. This value can be one or two characters long. The first character will be in the inclusive range of uppercase `A` to uppercase `F`. The second optional character will either be `+` or `-`.
- `RECITATION`: Represents the recitation number.
- `FAVTOPICS`: This argument is a series of 4 (four) 1s and/or 0s. Each representing the topic the student liked.
- `PERCENTILE`: Represents the percentile ranking of student.

All the arguments, except `NETID`, need to be validated appropriately.

We have provided you boilerplate code in the `hw1.asm` file for extracting each of the arguments from the array and storing their values in accessible labels in your `.data` section

`load_args` is an assembler macro. Macros are a textual replacement of code at assemble time, while functions are changes in control flow during run time. We use it to simplify access to the command line arguments for this first assignment. `load_args` will store the total number of arguments provided to the program at an address in memory labeled `numargs`. In addition, the starting address of each argument string that is provided to your program can be accessed using their labels (eg. `AddressOfNetId`, `AddressOfId`, etc).

You may implement macros in your own programs. However, it is not a requirement. We caution their use, as they can introduce non-obvious coding bugs if not used carefully!

You can declare more items in your `.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.**

The `load_args` macro will crash if zero arguments are provided when you run the program. You must test with at least 1 argument specified. In addition, NEVER call this macro again within your program. Doing so could overwrite the command line arguments passed to your program.

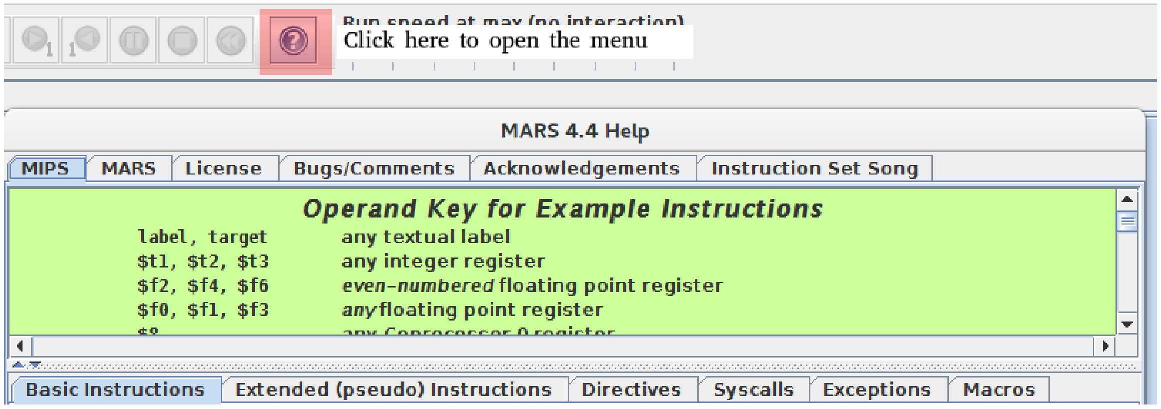
Writing the program

In `hw1.asm`, begin writing your program after the `load_args` macro.

First, add code to check the number of command line arguments provided. If the value stored in memory at label `numargs` is not 6, the program should print out `err_string` and exit the program (syscall 10). Note that the error string has already been defined in your `.data` section and is stored in memory at the address labeled `err_string`.

ℹ The number of arguments is stored in memory at the label `numargs` by the macro, but the value is also STILL in `$a0`, as the macro code does not modify the contents of `$a0`. Remember, values remain in registers until a new value is stored into the register, thereby overwriting it.

ℹ To print a string in MIPS, you need to use system call 4. You can find a listing of all the official MARS systems calls here. 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.



Operand Key for Example Instructions

label, target	any textual label
$t1, $t2, $t3	any integer register
$f2, $f4, $f6	*even-numbered* floating point register
$f0, $f1, $f3	*any* floating point register

If the number of arguments is valid, proceed to check each argument's validity.

We have added two special syscalls to MARS to assist with this assignment:

1. `atoi` syscall: convert an ASCII string of digit characters to a 32-bit integer. You should use this syscall to convert each argument that is supposed to be an integer from its ASCII string representation to an integer values, i.e `ID` and `RECITATION` should be converted to integers.

2. `atof` syscall: convert an ASCII string of digit characters to a 32-bit IEEE-754 Single Precision floating point number. You should use this syscall to convert each ASCII string representation of the float to its floating point value, i.e `PERCENTILE` should be converted to a float.

Service	**Code**	**Args**	**Result**
`atoi`	84	$a0 = Starting address of the string to convert	Converts an ASCII string to an integer and places the result in $v0. Success: $v1 = 0, Fail: $v1 = -1
`atof`	85	$a0 = Starting address of the string to convert	Converts an ASCII string to a float and places the resulting binary representation in $v0. Success: $v1 = 0, Fail: $v1 = -1

The `atoi` syscall returns failure if the provided string contains any ASCII character not in the following set:
{`‘-’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’`}.

The `atof` syscall returns failure if the provided string contains any ASCII character not in the following set:
{`‘.’, ‘-’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’`}.

The arguments are valid if `atoi/atof` returns success and the values returned are in the appropriate ranges.

NOTE: Real computers do not have these types of syscalls. These were created for your ease in this assignment.

Remember, the addresses of the arguments, which were null-terminated sequence of ASCII characters (strings), were stored at unique labels by the `load_args()` macro. You will need to use load instruction(s) to obtain the value(s) of these strings stored at the addresses.

The following is a summary of the arguments that need to be validated.

- `ID`: The value is in the inclusive range [0, 999999999]
- `GRADE`: This value can be one or two ASCII characters long. The first character will be in the inclusive range of uppercase `A` to uppercase `F`. The second optional character will either be `+` or `-`.

Example Input	Valid/Invalid
A-	valid
F	valid
a@	invalid
HELLOCSE220	invalid

For the single letter grades, the second character stored in the struct is a space character (ASCII 32).

- `RECITATION`: This value must be in the following set: {8, 9, 10, 12, 13, 14}.
- `FAVTOPICS`: This argument is a series of exactly 4 (four) 1s and 0s. Each representing the topic the student liked.

Example Input	Valid/Invalid
1011	valid
1	invalid
000asdf0	invalid

> ℹ Hint: Use bit wise instructions and remember that ASCII characters are not the same as bits. Convert from one form to the other to make sure that the comparison is valid.

- `PERCENTILE`: This value must be in the inclusive range [0.0, 100.0].

Print out `err_string` and exit the program (syscall 10) if any of the arguments is invalid.

At this point, your program should correctly handle and validate all of the command-line arguments passed to the program.

⚠ After checking `numargs`, you must parse the arguments in the following order: ID, NETID, PERCENTILE, GRADE, RECITATION, and FAVTOPICS. This matters because the print statements' ordering will vary if you deviate from the specified order. We will grade in this order.

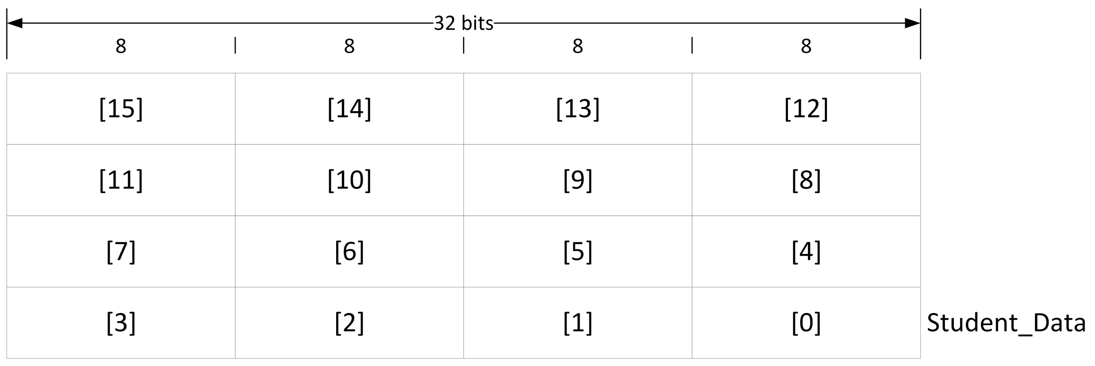
Part 2: Modifying the cse220_student struct in memory

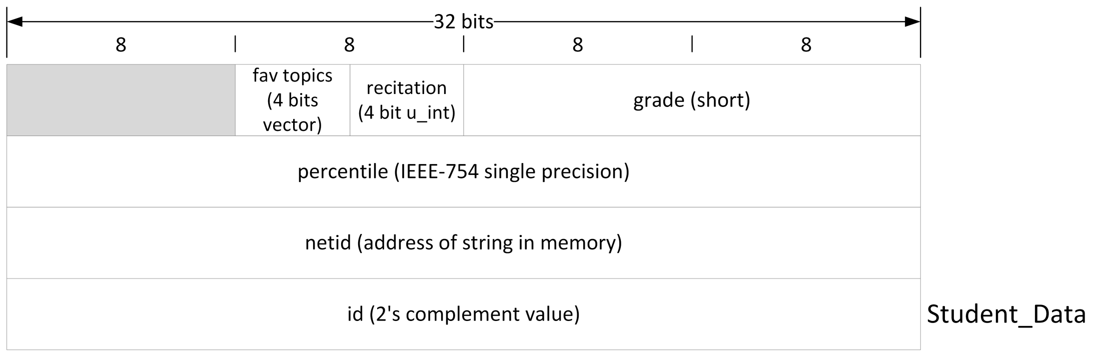
Make sure you completed ALL of part 1 before doing this part.

To refresh, the struct is shown below.

```
struct cse220_student {
  int      id;            // size: 4b (b == bytes)
  String   netid;         // size: 4b; starting address of the string
  float    percentile;    // size: 4b; single precision IEEE-754
  short    grade;         // size: 2b; byte[0] is letter and byte[1] is sign
  nibble   recitation; // size: 4 bits; NOTE: nibble isn't a true C datatype
  nibble   favtopics;  // size: 4 bits; bit vector
};
```

We have provided you with a set of structs in different files with names like `Struct1.asm`. In each file is a label `Student_Data:`, which is the address of the first byte of the struct stored in memory.




32 bits
8
8
8
8
fav topics (4 bits vector)
recitation (4 bit u_int)
grade (short)
percentile (IEEE-754 single precision)
netid (address of string in memory)
id (2's complement value)
Student_Data

The image displayed above will help you visualize the fields of the struct in memory.

Compare each field passed through the command line with the corresponding field in the struct stored at the label `Student_Data`. If they are the same, then print the appropriate message corresponding to the field. For example, if the `RECITATION` argument **is the same as** the `recitation` field of the struct at the `Student_Data` label, then print the message at `unchanged_Recitation`.

If the data that is passed to your program through the command line arguments is **not** the same as the data that is available at `Student_Data`, then save the argument value to the corresponding field in the struct in memory. For example, if the `RECITATION` argument **is the different from** the `recitation` field of the struct at the `Student_Data` label, then print the message at `updated_Recitation` and update the value of the argument in memory.

Remember, to compare the strings representing to the `netid`, you will have to compare the strings character by character. If they are different you will have to update the ADDRESS, i.e. the `netid` field should reference the string passed to the program as a command-line argument.

⚠ You are guaranteed that the space at the `Student_Data` label is large enough to fit the full struct but may not necessarily contain all the correct data. You are also guaranteed that the addresses referenced by the `netid` field will be valid.

Once you are done modifying all the fields, print out the hex value of each byte in the struct at `Student_Data`, using the hex system call. Start at byte 0 and end at byte 14. You are shown sample output in the provided the sample files. You **must** use a loop.

Sample files are provided (with expected outputs in the comments of the files) on PIAZZA. These files must be placed in the same directory as your Mars executable! If they are in a different location, you need to change the `.include “Struct1.asm”` to contain the full path of the file.

⚠ Look at the sample result files provided to you in the hw1 zip you downloaded from Piazza for examples of what should be printed.

❗ It is critical that you follow the output format exactly as shown. If you don’t the grading system will not be able to detect it and you will lose points. Beware of printing additional "invisible" characters such as `\0` (NULL byte).

ℹ We HIGHLY encourage you to create your own samples to test and verify your programs on different packet values.

The provided sample files MAY BE used in grading of the assignment. Additional tests WILL BE used for grading.

Hand-in instructions

Do not add any miscellaneous printouts, as this will probably make the grading script give you a zero. Please print out the text **EXACTLY** as it is displayed in the provided examples.

When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

See Sparky Submission Instructions on Piazza for hand-in instructions. There is no tolerance for homework submission via email. They must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are having trouble submitting, stop by office hours prior to the deadline for assistance.