CSE 220: Systems Fundamentals I

Homework #1

Spring 2017

Assignment Due: Feb. 15, 2017 by 11:59 pm via Sparky

⚠ **PLEASE READ THE WHOLE DOCUMENT BEFORE STARTING!**

Introduction

The goal of this homework is to become familiar with basic MIPS instructions, syscalls, basic loops, conditional logic and memory representations. In this homework you will be creating a base conversion program. The program will convert values from base 2-15 to base 2-15.

⚠ You **MUST** download the SB version of MARS posted on the PIAZZA website. **DO NOT USE** the MARS available on the official webpage. The SB version has a reduced instruction set and additional system calls you will need to complete the homework assignments.

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

Your program will take three command-line arguments: `integer fromBase toBase`

- `integer`: The null-terminated ASCII string representing the value to convert.
- `fromBase`: The base of the value is given in.
- `toBase`: The base to convert the value to.

  Valid values for `fromBase` and `toBase` are 2-15. We will represent them as arguments using hexedecimal digits `[2-9, A-F]`.

ℹ Only capital letters A-F will be considered valid Hexadecimal digits for arguments

The program will convert the `integer` represented in base-`fromBase` to its corresponding value in base-`toBase` and print the new value out. If any of the arguments are invalid, the program will print "ERROR".

Examples:

Arguments	Program Prints	Example of
`123 5 F`	28	Valid Input
`1111111 3 7`	3121	Valid Input
`987654321 A 4`	322313212202301	Valid Input
`EE F B`	194	Valid Input
`89ABC F F`	89ABC	Valid Input
`123`	`ERROR`	Not enough arguments
`123 G 2`	`ERROR`	Invalid `fromBase`
`123 1 4`	`ERROR`	Invalid `toBase`
`123 22 A`	`ERROR`	Invalid base, too many characters
`123 3 5`	`ERROR`	Value is not representable in `fromBase`
`121 D a`	`ERROR`	Invalid `tobase`, invalid hex character
`ab4 F C`	`ERROR`	Invalid value, invalid hex characters

ℹ Use the algorithms taught in lecture to convert the integers. Base Conversion Tools are available online to check you work.

ℹ The numerical representation of the input value is guaranteed to fit in 32-bits. The maximum decimal value is $2^{31} - 1$

Part 1: Command-line arguments

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

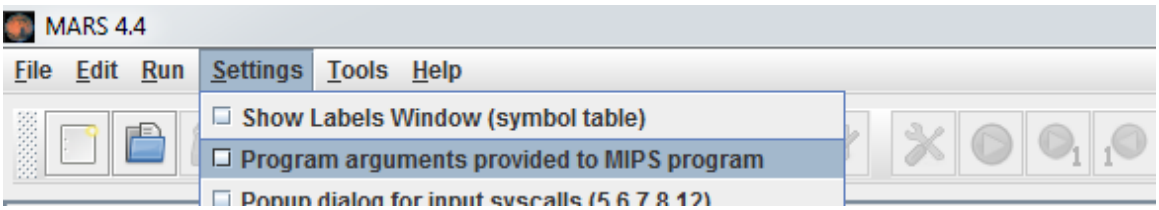
To begin writing your program:

- Create a new MIPS program file.
- At the top of your program in comments put your name and id.

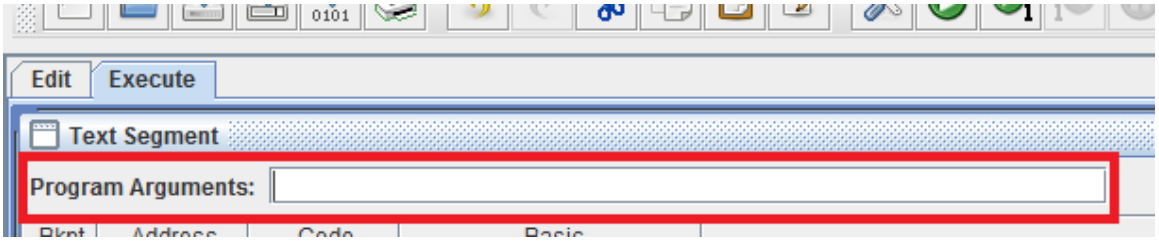
```
# Homework #1
# name: MY_FIRSTNAME_LASTNAME
# sbuid: MY_SBU_ID
...
```

Configuring Mars for command line arguments

Your program is going to accept command line arguments which will provide input to the program. To do this, we first need to tell Mars that we want to accept command line arguments. To do this you need to open Mars, navigate 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.



Each command line argument should be separated by a space.

! Your program must ALWAYS be run with at least 1 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.

In this assignment, we will be using three arguments. We have provided you boilerplate code for extracting each of the 3 arguments from the array.

Setting up the .data section

Add the following directives to the `.data` section to define memory space for the assignment:

```
.data
.align 2

    numargs: .word 0
    integer: .word 0
    fromBase: .word 0
    toBase: .word 0
    Err_string: .asciiz "ERROR\n"

    # buffer is 32 space characters
```

```
    buffer: .ascii "                                    "
    newline: .asciiz "\n"

# Helper macro for grabbing command line arguments
.macro load_args
    sw $a0, numargs
    lw $t0, 0($a1)
    sw $t0, integer
    lw $t0, 4($a1)
    sw $t0, fromBase
    lw $t0, 8($a1)
    sw $t0, toBase
.end_macro
```

`load_args` will store the total number of arguments provided to the program in the label `numargs`. In addition, the starting address of each argument string that is provided to your program can be accessed using labels `integer`, `fromBase`, and `toBase`.

ℹ `load_args` is an assembler macro. Macros are different from functions. We use it to simplify access to the command line arguments for this first assignment.
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.

ℹ You can declare more items in your `.data` section as you wish, but you must at minimum have the ones defined exactly as they appear in the above code section.

Writing the program

Create the `.text` section in your homework file and create a label `main:`. This is the main entry to your program. Next, we will extract the command line arguments using the `load_args` macro. This must be executed before any other code to avoid losing values in `$a0` and `$a1`.

In addition, NEVER call this macro again. Doing so may overwrite the command line arguments passed to your program.

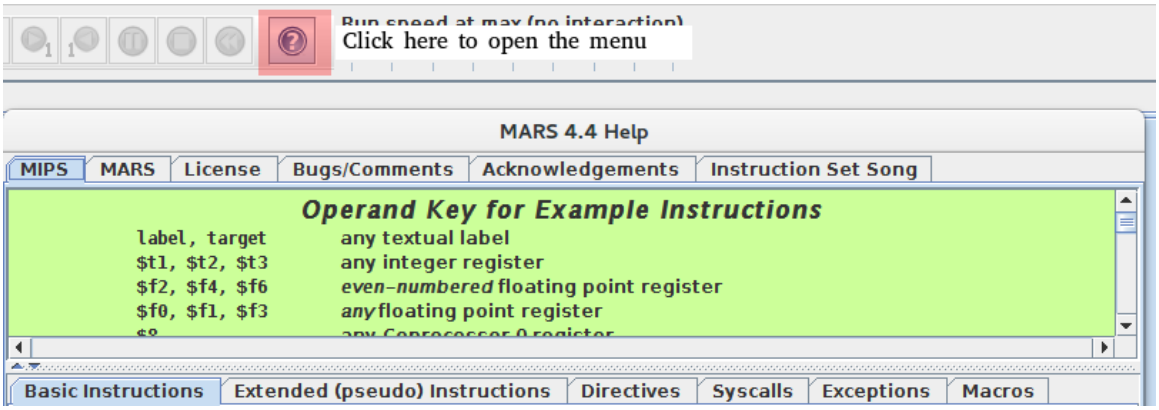
```
.text
.globl main
main:
    load_args()       # Only do this once
```

After the `load_args` macro is called, you will begin writing your program.

First add code to check the number of command line arguments provided. If the value string stored in memory at label `numargs` is not 3, the program should print out `Err_string` and exit the program. Note that the error string has already been defined in your `.data` section at label `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.



If the number of arguments is valid, next check the strings for the `fromBase` and `toBase` arguments.

Each of these arguments are valid if they each satisfy the following properties:

- is only 1 character in length, and
- is in the ASCII character set
  `'2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'`

ℹ To use the values placed in `integer`, `fromBase`, and `toBase`, you will need to use load instruction(s) to obtain the value(s) stored in memory. The labels `integer`, `fromBase`, and `toBase` each store the starting address of a null-terminated sequence of ASCII characters stored in memory.

Some other questions to ask yourself are:

? Consider the difference between the instructions `lw` and `lb`.

? How are strings stored in memory? How would you know if the argument string has a length of only 1 character? How do you check this?

? What is the relationship between the ASCII characters in their binary representation? Can you write simple checks, rather than a LARGE if/else or switch/case statement?

If either `fromBase` or `toBase` is not valid, print `Err_string` and exit the program.

If both `fromBase` and `toBase` are valid, next check each character in the string stored at the address in `integer` to ensure that each of the characters are valid in `fromBase`. If the integer is not representable in `fromBase`, print `Err_string` and exit the program.

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

Part 2: Converting `integer`

In this part of the assignment, convert `integer` from base-`fromBase` to base-`toBase`.

First convert `integer` from base-`fromBase` to decimal (base-10) using the multiplication method, then convert the decimal value to base-`toBase` using the division method. Both methods were discussed in lecture and are explained in the textbook.

❗ Remember that each ASCII character in the `integer` string must be converted to the digit value it represents. The same goes for the ASCII character representing `fromBase` and toBase.

❗ Remember that numbers are stored in computers using 2's complement notation. The decimal value will be stored in its 2's complement representation within the register!

❗ You may assume the decimal value is always representable in a single 32-bit register.

When using the algorithm for base conversion the value in the new base is created starting with the right-most digit (Least Significant digit) to the left-most digit (Most Significant digit). To store the new value as we create it we have allocated space for 32 characters in memory at the label `buffer`. Directly after this in memory is a newline character.

As you execute the algorithm, store the ASCII character for each digit of the new value into the buffer, starting from the right-most position. The memory address of the right most position is the address of label `buffer+31` or `newline-1`. With each additional digit to store, subtract 1 from the address.

When you print the result, print the string starting at the label `buffer`. The value printed will be right justified as shown in the examples at the beginning of the assignment. As `buffer` is not null-terminated (`.ascii` vs `.asciiz`) when printing starting at the address of `buffer`, the newline character will also be printed.

ℹ NOTE: There is a base case for conversion. If the `fromBase` and `toBase` are the same, the characters from `value` can be copied into the buffer directly. It is up to you if you want to implement this check or apply the regular conversion algorithm.

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 examples, one output line ONLY.

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 struggling, stop by office hours.

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.