CSE 220: Systems Fundamentals I

Homework #2

Spring 2017

Assignment Due: Febuary 24, 2017 by 11:59 pm

Assignment Overview

In this assignment you will be creating functions. The goal is to understand passing arguments, returning values, and the role of register conventions. The theme of the assignment is floating point numbers and will give you good practice manipulating register values.

You **MUST** implement all the functions in the assignment as defined. It is OK to implement additional helper functions of your own in `hw2.asm`.

⚠ **You MUST follow the MIPS calling and register conventions. If you do not, you WILL lose points.**

⚠ Do not submit a file with the functions/labels `main` or `_start` defined. You will obtain a ZERO for the assignment if you do this.

🛈 If you are having difficulties implementing these functions, write out the pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

🛈 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.

Getting started

Download `hw2.zip` from Piazza in the homework section of resources. This file contains `hw2.asm` and multiple `hw2_main` files, which you need for the assignment. At the top of your `hw2.asm` program in comments put your name and SBU ID number.

```
# Homework #2
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to test your functions

To test your functions, simply open one of the provided `hw2_main` files in MARS. Next, assemble the `main` file and run. Mars will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your `hw2.asm` file to the main file before assembling it. Once the contents have been substituted into the file, Mars will then assemble it as normal.

Each of the main files tests the functions you are to implement with one of the sample test cases. You should modify these files or create your own files, to test your functions with more test cases.

⚠ Your assignment will not be graded using these tests!

Any modifications to the `main` files will not be graded. You will only submit your `hw2.asm` file via Sparky. Make sure that all code require for implementing your functions (`.text` and `.data`) are included in the `hw2.asm` file! To make sure that your code is self-contained, try assembling your `hw2.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in a `main` file to `hw2.asm`.

⚠ It is highly advised to write your own main programs (new individual files) to test each of your functions thoroughly.

⚠ Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

Optional readings

If you are interested there are a few interesting readings about floating point arithmetic which contain information that you may not be aware of and can be used to assist you with the assignment.

- What every computer programmer should know about floating point
- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Floating-point arithmetic may give inaccurate results in Excel
- What Every JavaScript Developer Should Know About Floating Point Numbers

Part I: Helper Functions

In this part of the assignment you will implement basic leaf functions (a function which does not call any other functions). `hw2_main1.asm` contains a basic test for each of these functions.

All functions implemented in the assignment must be placed in `hw2.asm` and follow the standard MIPS register conventions for functions taught in lecture.

a. `int char2digit(char c)`

This function converts the ASCII character '0'-'9' to its integer value, 0-9. Any other ASCII character results in an error.

- `c` : the ASCII character
- *returns*: the integer value of the character or -1 for error

Examples:

Code	Return Value
`char2digit('9')`	9
`char2digit('3')`	3
`char2digit('@')`	-1
`char2digit('c')`	-1

b. `int memchar2digit(char *c)`

This function converts the ASCII character stored at the memory address given by c from '0'-'9' to its integer value, 0-9. Any other ASCII character results in an error.

ℹ The `*` is C notation for an address of a value stored in memory.

- `*c` : the address of an ASCII character in memory
- *returns*: the integer value of the character or -1 for error

Examples:

Note, the prefix `0x` indicates the following digits are shown in hexadecimal representation.

Code	Memory Address = Stored Value	Return Value
`memchar2digit(0x40000000)`	`Mem[0x40000000] = 0x39`	9
`memchar2digit(0x44404440)`	`Mem[0x44404440] = 0x33`	3
`memchar2digit(0x4FFFFFFF)`	`Mem[0x4FFFFFFF] = 0x4B`	-1
`memchar2digit(0x30303030)`	`Mem[0x30303030] = 0x20`	-1

c. `(int, int) fromExcessk(int value, int k)`

This function converts an excess-k value to its corresponding value in base-10.

- `value` : an excess-k value. If value is < 0, an error occurs.
- `k` : a integer value representing excess-k. If `k` ≤ 0, an error occurs.
- *returns*: (i) 0 for success, -1 for error, and (ii) the converted base 10 value upon success, the original `value` upon error. Note that the function returns two values: `$v0` holds the first return value and `$v1` holds the second return value.

⚠ Assume all input and resultant values are correctly representable in 32-bits.

Examples:

Code	Return Value
`fromExcessk(23, 127)`	0, -104
`fromExcessk(-23, 127)`	-1, -23
`fromExcessk(0, 127)`	0, -127
`fromExcessk(10, 35)`	0, -25
`fromExcessk(1000,-10)`	-1, 1000
`fromExcessk(2,0)`	-1, 2

d. `int printNbitBinary(int value, int m)`

This function prints out the `m` least-significant bits of the binary representation of `value` and returns success. If `m` is in the range $[1, 32]$ the function prints out the `m` least-significant bits of the binary representation of the value and returns success (0). Otherwise, the function prints nothing and returns error (-1).

- `value`: the value to print the binary digits of
- `m`: the number of bits to print of `value`.
- *returns*: 0 if the print was successful, -1 for error.

⚠ Remember, `value` is stored in 2's complement binary format in the register.

⚠ `value` is any 32-bit value in the register.

Basic Algorithm:

```
shift value left (32-m) bits
while (m > 0)
    if (value < 0) print '1', else print '0'
    shift value left 1 bit
    m = m-1
```

Examples:

Code	Return Value	Prints
`printNbitBinary(-23, 30)`	0	`11111111111111111111111101001`
`printNbitBinary(10, 3)`	0	`010`
`printNbitBinary(10, 9)`	0	`000001010`
`printNbitBinary(1000,-2)`	-1	
`printNbitBinary(2,100)`	-1	

Part II: Floating point

In this section, you will be writing functions to work with strings and floating point values and formats. You will implement the functions to complete `hw2_main2.asm`. These functions will call the functions you implemented in Part I.

All functions implemented in the assignment must be placed in `hw2.asm` and follow the standard MIPS register conventions for functions taught in lecture.

❗ Under no circumstances are you to use the floating point registers or floating point MIPS instructions.

e. `(int, float) btof(char[] input)`

ℹ The `[]` is C notation for the starting address of an array of the specified data type stored in memory.

`btof` will convert a string of binary digits or a string of characters representing a special value into its IEEE 754 single precision floating point binary representation. The function will return two values, an int and a 'float'. The int is a flag for success or error. The "float" is the IEEE-754 single floating point representation of the input value stored in the returned register.

- `input`: a string of binary digits or special value string representing an IEEE-754 number. Can be of any length. The IEEE representation should be truncated to fit in IEEE-754 single precision format.
- *returns*: (i) 0 conversion was successful, -1 for error and (ii) the IEEE-754 formatted value of the string, or unspecified value if an error occurred.

ℹ Your implementation may return any value you want for the unspecified value.

This function must parse the string until it reaches the end ('\0') or an invalid character. Valid binary digit characters are `0`, `1`, `.`, `+`, `-`, and the ASCII characters in `NaN` and `Inf`. If any non-valid character/substring is reached, an error occurs.

The SPECIAL float values which must be handled are `+0.0`, `-0.0`, `NaN`, `+Inf`, and `-Inf`. These strings are CASE-SENSITIVE.

❗ Your function MUST CALL `char2digit` or `memchar2digit`.

❗ All `input` argument strings, which are not SPECIAL float values, will always contain a single radix point (.) and have at least 1 binary digit on either side of the radix point.

Examples:

Note, the prefix `0b` indicates the following digits are the binary representation for the floating point representation. This notation is similar to the hexadecimal representation `0x`.

Code	Return Values
`btof("010.001")`	0, 0b01000000000010000000000000000000
`btof("+010.001")`	0, 0b01000000000010000000000000000000
`btof("-010.001")`	0, 0b11000000000010000000000000000000
`btof("-0.0001")`	0, 0b10111101100000000000000000000000
`btof("11110.00111000111000111000001")`	0, 0b01000001111100011100011100011100
`btof("-11110.00111000111000111000001")`	0, 0b11000001111100011100011100011100
`btof("+0.0")`	0, 0b00000000000000000000000000000000
`btof("-0.0")`	0, 0b10000000000000000000000000000000
`btof("NaN")`	0, 0b01111111111111111111111111111111
`btof("+Inf")`	0, 0b01111111100000000000000000000000
`btof("-Inf")`	0, 0b11111111100000000000000000000000
`btof("213.2")`	-1, ?
`btof("iNF")`	-1, ?
`btof("nan-12.3")`	-1, ?
`btof("-11.1A")`	-1, ?
`btof("@.2")`	-1, ?

ℹ ? denotes an unspecified return value. Your implementation may return any value you want.

f. `int print_parts(float value)`

This function takes the binary representation of a single precision IEEE-754 float value as a parameter and returns a 1 if the value is positive, 0 if the value is a special value (`+0` , `-0` , `NaN` , `+Inf` , and `-Inf`), and -1 if the value is negative. This function will print out the sign of the number, the binary and decimal value of the exponent, and the binary and decimal value of the mantissa.

❗ Your function MUST CALL `printNbitBinary` .

❗ Your function MUST print the EXACT format shown in the examples. All spaces are single spaces.

❗ DO NOT use system call 35 to print the binary. It always prints out 32 bits.

- `value` : the binary representation of a floating point value.
- *returns*: 1 if the value is positive, 0 if the value is a special value, and -1 if the value is negative

Examples - Note the argument is shown as a hexadecimal value to save space:

Code	Return Value	Prints

`print_parts(0x40080000)`
Value: 2.125
	1	`0 +`
		`10000000 128`
		`00010000000000000000000 524288`

`print_parts(0xC1F1C71C)`
Value: -30.222222
	-1	`1 -`
		`10000011 131`
		`11100011100011100011100 7456540`

`print_parts(0x7F800000)`
Value: +Inf
	0	`0 +`
		`11111111 255`
		`00000000000000000000000 0`

ℹ Use the IEEE 754 Converter to assist with checking your values.

g. `int print_binary_product(float value)`

This function takes the binary representation of a single precision IEEE-754 float value as a parameter prints the value in a binary product format (which is similar to scientific notation): $\pm 1.\text{mantissa} * 2^{+/-exponent}$ if it is a non-special value. The function will return 1 if the value is printable in binary product form or 0 if the value is a special value.

- `value`: the binary representation of a floating point value.
- *returns*: 1 if the value is printable in binary product form or 0 if the value is a special value

❗ Your function MUST CALL `printNbitBinary` and `fromExcessk`.

❗ Your function MUST print the EXACT format shown in the examples. All spaces are single spaces.

Examples - Note the argument is shown as a hexadecimal value to save space:

Code	Rtn Val	Prints
`print_binary_product(0x40080000)`	1	`+1.00010000000000000000000 x 2^+1`
`print_binary_product(0xC1F1C71C)`	1	`-1.11100011100011100011100 x 2^+4`
`print_binary_product(0xBDDC0000)`	1	`-1.10111000000000000000000 x 2^-4`
`print_binary_product(0x3DDC0000)`	1	`+1.10111000000000000000000 x 2^-4`
`print_binary_product(0x7F800000)`	0	

Hand-in Instructions

See Sparky Submission Instructions on piazza for hand-in instructions.

❗ There is no tolerance for homework submission via email. Work 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 for additional help.