UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE012/L: COMPUTER SYSTEMS AND ASSEMBLY PROGRAMMING

# LAB 5: HEX TO DECIMAL CONVERSION

## MINIMUM SUBMISSION REQUIREMENTS:

- Create a lab5 folder
- Lab5.asm in the lab5 folder
- README.txt in the lab5 folder
- Commit and Push your repo
- A Google form submitted with the Commit ID taken from your GITLAB web interface verifying that the above files are correctly named in their correct folders
- All of the above must be completed by the lab due date

## LAB OBJECTIVE:

In this lab, you develop a more detailed understanding of how data is represented, stored, and manipulated at the processor level. You will need to traverse an array, convert between ASCII and binary numbers, convert a hexadecimal number to a native binary number, and convert the native binary number to an ASCII signed decimal number. You will also gain additional experience with the fundamentals of MIPS coding, such as looping and checking conditionals.

## LAB PREPARATION:

Read this document in its entirety, carefully. Read *Introduction to MIPS Assembly Language Programming* by Charles Kann, chapters 1 and 9.

## LAB SPECIFICATIONS:

In this lab, you will write a MIPS32 program that reads a program argument string from the user, repeats that string, translates the string into a binary integer (in two's complement format), stores that signed integer into a specific register ($s0), converts the signed integer into an ASCII decimal representation which is printed before exiting cleanly.

You will need to write your own pseudocode for this lab. Your pseudocode must be part of your lab submission (as a block comment). Your code must follow your pseudocode.

As always, your code should be clean and readable. Refer to Lab4 to see the standard of readability we expect.

In this lab, all input syscalls are **FORBIDDEN** (that is, any syscall whose description begins with "read"). Instead, the user input will be passed to the program using a "program argument" string. This string will be the ASCII sequence "0x" (zero followed by x) followed by a sequence of eight hexadecimal digits in ASCII ('0" – "9" and "A"-"F"). All hex letters are capitalized only! The input is 8 hex characters, so it is within the range of a 32 bit two's compliment representation.

Syscall 1 (print integer) and Syscall 36 (print integer as unsigned) are also **FORBIDDEN**. Note: you may use them for debugging your code, but you will get no points if you use it ANYWHERE(even in comments!) in the code you turn in. Instead, your code must convert the binary 2SC integer in register $s0 into an ASCII representation of the signed decimal number and output it using Syscall 11 (print character) or Syscall 4 (print string).

The number, as a single 32-bit binary integer, must be stored in $s0 at the time the program exits. You can use other registers as you wish, but $s0 is reserved!

Two examples of the expected output are given below. Your code's output format should match this output format exactly. Capitalization is required!:

```
Input a hex number:
0xDEADBEEF
The decimal value is:
-559038737
-- program is finished running --
```

```
Input a hex number:
0xFFFF0000
The decimal value is:
-65536
-- program is finished running --
```

```
Input a hex number:
0x000000F0
The decimal value is:
240
-- program is finished running --
```

Please note that part of our grading process is automated and any deviation from this format will cause our grading script to fail. You do not get partial credit when you do not follow the format requirements.

In general, we do not grade on efficiency, but we do expect your code to use loops where appropriate. For example, do not repeat nearly-identical blocks of code to check each bit or several bits. Instead, write one block of code and loop over it. Similarly, we do not expect the highest performance code. Make sure your code is reasonable and produces the correct output in a reasonable run-time.

**LAB STRATEGY:**

This lab can be divided into four main parts:

1. Read program argument string
2. Convert the program argument to a binary integer in $s0.
3. Convert the value in $s0 as a 2SC number to an ASCII decimal number.
4. Print the correct signed ASCII decimal number before exiting cleanly.

We strongly recommend that you attack all of these steps in parallel. If you get stuck on part 1 (you probably will), then work on part 2 and return to part 1 later. One possible strategy is to write four separate files, each one of which solves one part. Make sure to recombine them into Lab5.asm before turning it in.

**READING PROGRAM ARGUMENTS:**

In this lab, all input syscalls are forbidden. Instead, we will use "program arguments" to read user input. To introduce a program argument to your code, first use the MARS toolbar to turn on program arguments (Settings -> check "Program Arguments provided to MIPS program"). If you do this and then assemble your code, a white text field will appear at the top of your "Text Segment" window. This is where you enter a program argument.

To see how it works, press "assemble," enter a string, and then advance your code by a single step. Notice that the $a0 and $a1 registers change. $a0 contains a small number, 1 (indicating that you passed one string into the program). $a1 contains an address. This address is a location in stack memory, it is the beginning of an array called the "program arguments pointer table." To view that array, use the drop-down box in the "Data Segment" window and select "current $sp." Find it. Currently, the table should contain exactly one element. That element is *another* address. It should be very nearby, and is the address of the program argument string.

To summarize: At time of startup, $a1 contains the address of a location in data memory, which in turn contains the address of your string, also stored in data memory.

Follow this method in your program to find the addresses and DO NOT hard code the address of your input. This information is passed "on the stack" and if the stack location changed, your program would break otherwise.

Be sure you understand how this works!

**CONVERTING AN ASCII STRING TO A BINARY INTEGER:**

Your program argument is stored as an ASCII string. You will need to handle each character of this string individually, and iterate over the string to combine all the information of the whole string into a 32-bit binary integer. This integer must be stored in $s0, and remain there for the rest of your code.

Each character of this string is an ASCII representation of a hexadecimal digit ("0" to "9" or "A" to "F" capitalized only). You will first need to convert each digit from ASCII representation into a binary integer (so, for example, you must turn "5" into 0b0101). Kann provides a useful description of the ASCII table that will help you find a procedure for this conversion.

You will need to use each successive digit of the input string to update the value stored in $s0. Think carefully about how to combine your current value and the next digit to find the next value. For example, if your "5" in hex is followed by a "3" (in hex), what steps are required to combine 0b0101 with 0b0011 to yield 0b1010011 (83 in decimal)?

**CONVERTING FROM TWO'S COMPLIMENT:**

If the most significant bit of your resultant number in $s0 is set, you have a negative number. You will need to take the two's complement of the value in $s0 before you print it to find the magnitude. There are several ways to accomplish this in MIPS.

**PRINTING IN DECIMAL:**

Though MARS provides "print integer" and "print integer as unsigned" syscalls, you will receive zero points on this section if you use it. Instead, you must convert $s0 to a decimal string first. One convenient way is to use Binary Coded Decimal (BCD) which represents each decimal value as a 4-bit value. Several combinations of the 4-bits are not valid. Once you have the BCD value, you can easily convert to ASCII using the opposite of the technique when you converted from ASCII.

## PSEUDOCODE:

The code that this lab requires is too complex to attempt without a plan. You will need pseudocode to guide you as you write (and to guide our staff as we help you debug your work). ***BEFORE*** you begin writing your complete program, write the pseudocode for your program.

As you code, you may find that your pseudocode was incorrect, or did not include enough detail to be useful. If you find that your code is diverging from your pseudocode, take the time to update your pseudocode! Pseudocode is a crucial tool for understanding and navigating your code, and it is rude to the staff to ask them to help you if you cannot provide that tool.

This must reflect your code accurately.

## LAB WRITEUP:

Insert your pseudocode that accurately reflects the structure of your code as a block comment at the top of your Lab5.asm file.

Please use our README.txt template on Canvas to make grading easier. Be sure your write-up (in the README.txt file) contains:

- Appropriate headers
- Describe what you learned, what was surprising, what worked well and what did not
- Answer the following questions:
    1. How many representations of 0 are there in your input number?
    2. What is the largest input value (in decimal) that your program supports?
    3. What is the smallest (most negative) input value (in decimal) that your program supports?
    4. What is the difference between signed and unsigned arithmetic in MIPS (add vs addu, mult vs multu)? Which type did you use? What were the advantages or disadvantages of this choice?
    5. Consider how you might write a binary-to-decimal converter, in which the user inputs a string of ASCII "0"s and "1"s and your code prints an equivalent decimal string. How would you write your code?

To alleviate file format issues we want lab reports in plain text. Feel free to use a word processor to type it up but please ensure that your README.txt is a plain text file and CHECK IT ON GITLAB to make sure it is readable.

## BUMPS AND ROAD HAZARDS:

This lab is not easy. Make sure to start it early.

This lab requires understanding how strings and values are stored at the bit level. It also requires an understanding of the relationship between addresses and memory. If the concepts in this lab are unclear to you, make sure you understand them before attempting to write long pieces of code. Do small experiments first.

As usual, you will find that the MARS debugging tools are essential. Use breakpoints and step-throughs, and make sure you understand what the "Registers" pane and the "Data Segment" window are telling you. Make sure that you read the error messages that MARS gives you. They are relevant.