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

CMPE012/L: COMPUTER SYSTEMS AND ASSEMBLY PROGRAMMING

LAB 5: DECIMAL TO BINARY CONVERTER

MINIMUM SUBMISSION REQUIREMENTS:

- 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, convert a decimal number to a native binary format, and use bitmasking techiques to analyze a binary 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 number, stores that number into a specific register ($s0), and then prints that number in binary 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 in the form of a decimal number, possibly preceded by a single minus sign. If the string begins with a minus sign (negative), your code must output the two's compliment representation. You may assume that the user input number is within the range of a 32 bit two's compliment representation.

Sycall 35 (print binary) is also **FORBIDDEN**. Note: you may use it for debugging your code, but you will get no points if you use it in the code you turn in. Instead, your code must use bitmasking to determine the status of each bit in the $s0 register.

The input number, once translated into a single 32-bit binary number, must be stored in $s0 at the time the program exits.

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

```
User input number:
48879
This number in binary is:
00000000000000001011111011101111
-- program is finished running --
```

```
User input number:
-8531
This number in binary is:
11111111111111111101111010101101
-- program is finished running --
```

Please note that part of our grading process is automated and any deviation from this format may cause our grading script to fail. This will waste our time and your points.

In general, we do not grade on efficiency, but we do expect your code to use loops where appropriate. For example, do not write 32 nearly-identical blocks of code to check each bit. Instead, write one block of code and loop over it 32 times.

LAB STRATEGY:

This lab can be divided into four main parts:

1. Read program argument string
2. Check if user input number is negative, and convert the rest of the program argument to a binary number in $s0.
3. Convert $s0 to 2SC, if appropriate
4. Print contents of $s0 in binary.

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, $a0 contains the address of a location in data memory, which in turn contains the address of your string, also stored in data memory.

Be sure you understand how this works!

CONVERTING AN ASCII STRING TO A BINARY NUMBER:

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 number. This number 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 decimal digit (or a minus sign). You will first need to convert each digit from ASCII representation into a binary number (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" is followed by a "3", what steps are required to combine 0b0101 with 0b0011 to yield 0b00110101?

CONVERTING TO TWO'S COMPLIMENT:

If the first character of the program argument string was a minus sign (-), you will need to convert the value in $s0 to its 2SC equivalent before you print it. There are several ways to accomplish this in MIPS.

PRINTING IN BINARY:

Though MARS provides a "print binary" syscall, you will receive zero points on this section if you use it. Instead, you must use bitmasking to check each bit of $s0 individually.

Bitmasking is a tool to examine a specific bit (or bits) of some binary value. It accomplishes this using a second, carefully prepared binary value that will hide (or "mask") the irrelevant information in the unknown value. For example, if the bitwise AND of 0bXXXX and 0b0010 is 0b0000, we can deduce that bit 1 of X is 0.

You will need to perform a bitmasking operation on every bit of $s0, and use the result to print a "0" or "1" as appropriate.

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 pseudocode will be included as part of your lab report, and it 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.

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. What happens if the user inputs a number that is too large to fit in a 32-bit 2SC number?
    2. What happens if the user inputs a number that is too small (large magnitude, but negative) to fit in a 32-bit 2SC number?
    3. What is the difference between the "mult" and the "multu" instructions? Which one did you use, and why?
    4. 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? How is the BDC like the DBC, and how is it different?

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.