Lab 4: ASCII Decimal to 2SC

Part A: Due Sunday, 18 November 2018, 11:59 PM
Part B: Due Tuesday, 27 November 2018, 11:59 PM

Minimum Submission Requirements

- Create a Lab4 folder (note the capitalization convention, include no extra characters in the directory name) that contains the following files:
  - Diagram.pdf
  - Lab4.asm
  - README.txt
- Commit and push your repository
- Tag the commit that you would like to be graded
  - You will submit two tags for this lab: Lab4a_submission_# for Part A and Lab4b_submission_# for Part B (note the capitalization convention)

Lab Objective

In this lab, you will develop a more detailed understanding of how data is represented, stored, and manipulated at the processor level. In addition to strengthening your understanding of MIPS coding, you will read a string input and learn how to convert numerical ASCII characters into a two's complement binary number. You will also learn how to print a two's complement integer stored in a register.

Lab Preparation

Read chapters 1 and 9 from Introduction To MIPS Assembly Language Programming.

Specification

Functionality

The functionality of your program will be as follows:

1. Read two program arguments: signed decimal numbers [-64, 63].
2. Print the user inputs.
3. Convert the ASCII strings into two sign-extended integer values.
   a. Convert the first program argument to a 32-bit two's complement number, stored in register $s1.
   b. Convert the second program argument to a 32-bit two's complement number, stored in register $s2.
4. Add the two integer values, store the sum in $s0.
5. Print the sum as a decimal to the console.
6. Print the sum as 32-bit two's complement binary number to the console.
7. (Extra credit) Print the sum as a decimal number expressed in Morse code.
   a. Use a period (ASCII code 0x2E) for "dots" and a hyphen (ASCII code 0x2D) for "dashes".
   b. Insert a space (ASCII code 0x20) between characters.
   c. Don't forget to print the Morse code for a minus sign if the number is negative!

Two Parts

Part A: Block Diagram & Pseudocode

Before coding, you will first create a top level block diagram or flowchart to show how the different portions of your program will work together. Use https://www.draw.io or a similar drafting program to create this document. This diagram will be contained in the file Diagram.pdf. This diagram must be computer generated to receive full credit.

Next, you will create pseudo code that outlines your program. Your pseudocode will appear underneath your header comment in Lab4.asm.

Part B: Assembly Code

Write a program in MIPS to implement the functionality.

Output

An example of the expected output is given below. Your code's output format should match this output format *exactly*. New line characters are printed after each number representation. If you skip the extra credit, the last line printed will be the new line character after the binary value.

```
You entered the decimal numbers:
45 -54

The sum in decimal is:
-9

The sum in two's complement binary is:
11111111111111111111111111110111

The sum in Morse code is:
-....- ----.

-- program is finished running --
```

Files

Diagram.pdf

This file will contain a block diagram or flowchart of how the different components of your code work together. To receive full credit, you must submit a computer generated diagram. Hand drawn diagrams will be worth half credit.

Lab4.asm

This file contains your pseudocode and assembly code.

Header Comment

Your code should include a header comment with your name, CruzID, date, lab number, course number, quarter, school, program description and notes. Every program you write should include information like this. This is a good opportunity to start developing effective code documentation skills. An example header comment is shown below.

```
##############################################################################
# Created by:  Last Name, First Name
#              CruzID
#              7 August 2018
#
# Assignment:  Lab 64: Hello World
#              CMPE 012, Computer Systems and Assembly Language
#              UC Santa Cruz, Fall 2018
#
# Description: This program prints 'Hello world.' to the screen.
#
# Notes:       This program is intended to be run from the MARS IDE.
##############################################################################
```

Notes can contain any messages that are important for the user to know in order to run the program properly. For this lab you should put a note about how to enter a program argument.

Every block or section of code should have a comment describing what that block of code is for. In-line comments should be lined up (using spaces) for ease of readability.

Register Usage

Registers $s1 and $s2 shall contain the two 32-bit two's complement integers entered by the user. Register $s0 shall be used to store the 32-bit two's complement sum.

You should try to use as few registers as possible. Try to only use $zero, $v0, $a0, $s0-$s2, and the temporary registers, $t0-$t9. If you run out of registers, you may use $s3-$s8.

After the header comment and pseudocode, but before your program, include a comment about register usage. Some registers might be reused in several parts of your code, this is ok. An example of this comment is shown below.

```
# REGISTER USAGE
# $t0: address of first character from user input,
#      holds value of 0xFFFFFFFF for sign extension
# $s0: stores 2SC sum of user inputs
```

White Space

Line up instructions, operands, and comments to increase readability. Code should be indented from labels.

Bad Example

```
LI $t1 2 #initialize $t1
LOOP:
ADDI $t0 $t0 1 #increment $t0
BLT $t0 $t1 LOOP #determine if code should re-enter loop
```

Good Example

```
      LI   $t1 2          # initialize $t1
LOOP:
      ADDI $t0 $t0 1      # increment $t0
      BLT  $t0 $t1 LOOP   # determine if code should re-enter loop
```

A Note About Tabs

It is preferable to line up comments using spaces as opposed to tabs. Text editors can have different standards for the width of one tab character. For this reason, it is preferable to line up comments using spaces, not tabs, so that the code appears the same regardless of text editor.

README.txt

This file must be a plain text (.txt) file. It should contain your first and last name (as it appears on Canvas) and your CruzID. If unsure of your CruzID, note that your email address is CruzID@ucsc.edu. Your answers to the questions should total at least 8 sentences with complete thoughts.

Your README should adhere to the following template:

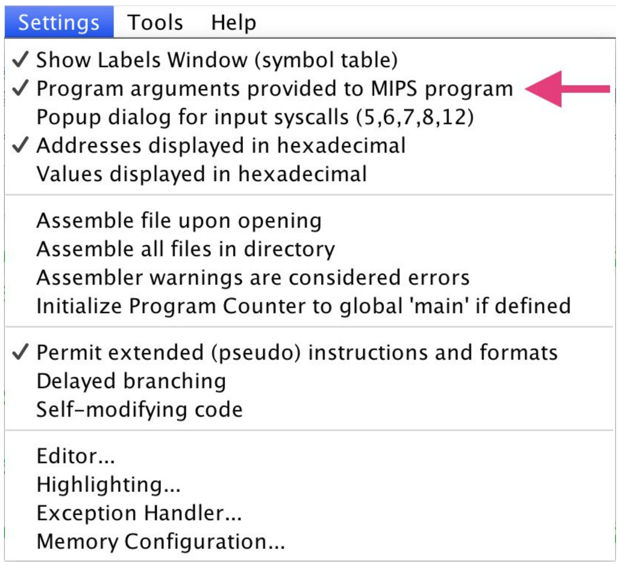
```
------------------------
Lab 4: ASCII Decimal to 2SC
CMPE 012 Fall 2018

Last Name, First Name
CruzID
-------------------------

What was your approach to converting each decimal number to two’s complement
form?
Write the answer here.

What did you learn in this lab?
Write the answer here.

Did you encounter any issues? Were there parts of this lab you found enjoyable?
Write the answer here.

How would you redesign this lab to make it better?
Write the answer here.
```

Syscalls

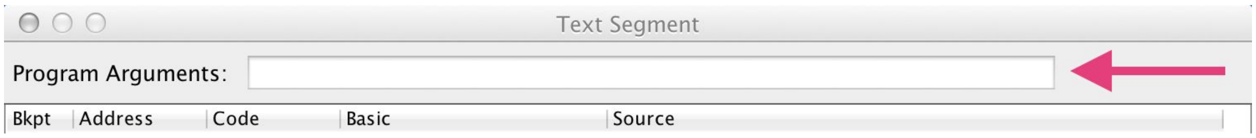
When printing the integer values, you may use syscall system services 4 (print string) and 11 (print character). You <u>may not</u> use syscall system service 1 (print integer) or syscall system service 35 (print integer as binary).

Input

In this lab you will obtain two user inputs, not using a syscall, but by using program arguments. The user will enter two integer values between -64 and 63. These numbers will be sign-extended to 32 bits and stored in $s1 and $s2. Turn on program arguments from the Settings menu as shown below.



✓ Show Labels Window (symbol table)
✓ Program arguments provided to MIPS program
Popup dialog for input syscalls (5,6,7,8,12)
✓ Addresses displayed in hexadecimal
Values displayed in hexadecimal
Assemble file upon opening
Assemble all files in directory
Assembler warnings are considered errors
Initialize Program Counter to global 'main' if defined
✓ Permit extended (pseudo) instructions and formats
Delayed branching
Self-modifying code
Editor...
Highlighting...
Exception Handler...
Memory Configuration...

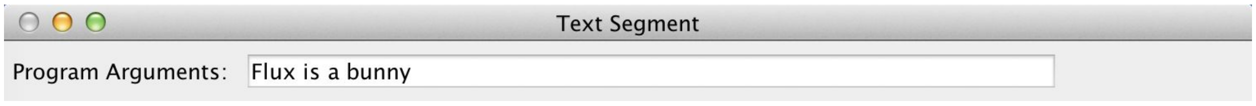


When program arguments are entered and the code is run, $a0 initially contains the number of program arguments entered (program arguments are separated by spaces). The register $a1 initially contains an address that we will call Address 1. If we go to Address 1 in memory, we will find another address, which we will call Address 2A. If we go to Address 2A in memory, we will find the ASCII encoding for the first byte of the first string entered in the program arguments.

Example Program Argument

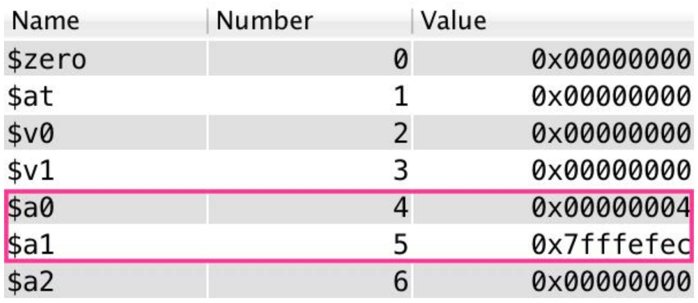
In this example, there are four program arguments. We will call these Program Arguments A, B, C, and D as follows:

Program Argument A: “Flux”
Program Argument B: “is”
Program Argument C: “a”
Program Argument D: “bunny”



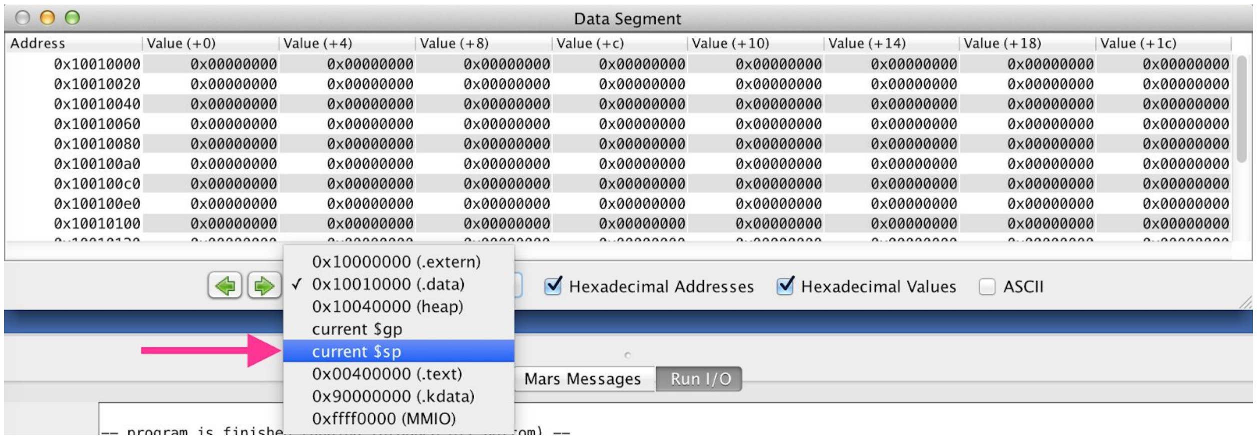
Program Arguments: Flux is a bunny

The value of $a0 has the value of 4 to match the 4 program arguments. We will call the value in $a1 “Address 1.”



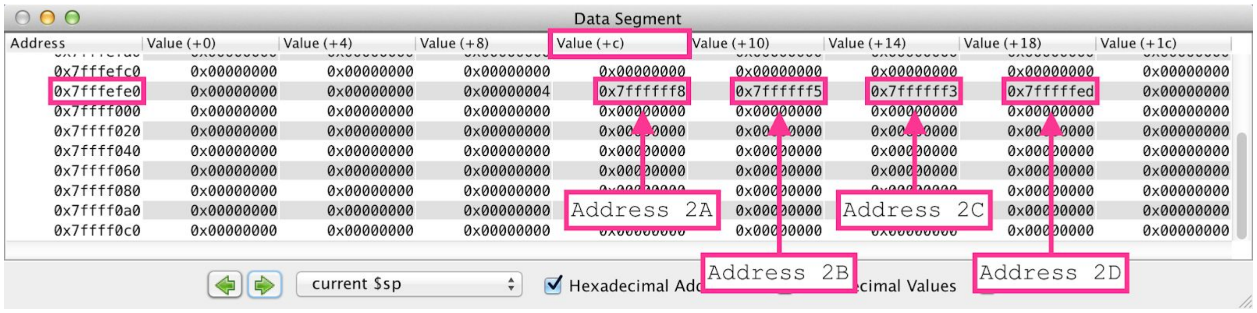
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000004
$a1	5	0x7fffefec
$a2	6	0x00000000

Address 1 should be found in the stack. You can jump to the stack area of the program by selecting "current $sp" from the drop down menu on the bottom of the memory window.



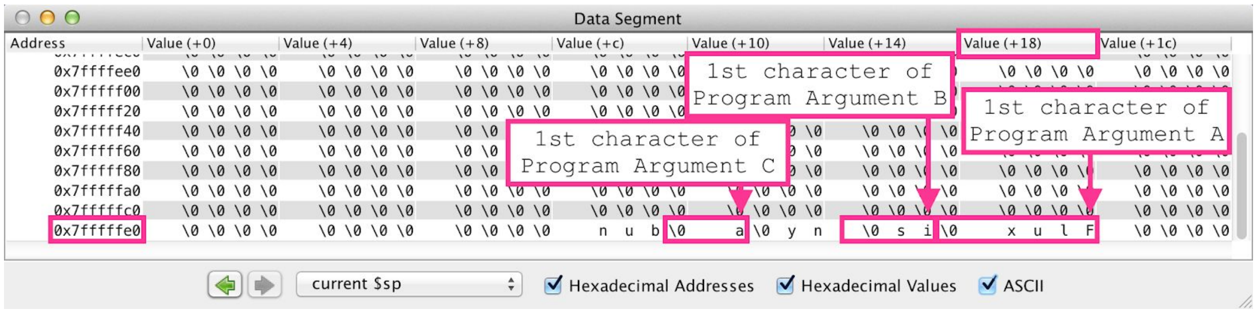
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

If we go to the Address 1 in memory (0x7fffefec in this example), we will find the first of 4 more addresses. We will call these addresses "Address 2A, 2B, 2C, and 2D." Address 2A is the location of the first character in Program Argument A, Address 2B is the location of the first character in Program Argument B, and so on.



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefc0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffefe0	0x00000000	0x00000000	0x00000004	0x7ffffff8	0x7ffffff5	0x7ffffff3	0x7fffffed	0x00000000
0x7ffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7ffff080	0x00000000	0x00000000	0x00000000	[illegible]	0x00000000	[illegible]	0x00000000	0x00000000
0x7ffff0a0	0x00000000	0x00000000	0x00000000	[illegible]	0x00000000	[illegible]	0x00000000	0x00000000
0x7ffff0c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

If we go to Address 2A (0x7ffffff8 in this example), we will find the ASCII encoding for first character of Program Argument A. If we go to Address 2B (0x7ffffff5 in this example), we will find the ASCII encoding for first character of Program Argument B, and so on. Check the ASCII box to display the data as ASCII characters. Note that each program argument ends with a null character.

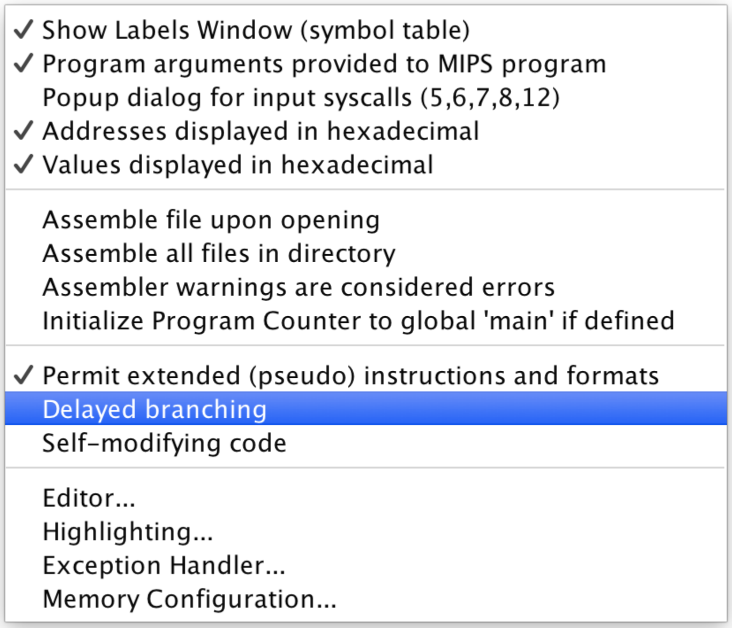

Data Segment
1st character of Program Argument B
1st character of Program Argument A
1st character of Program Argument C
current $sp
✓ Hexadecimal Addresses
✓ Hexadecimal Values
✓ ASCII

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffee0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	[illegible]	[illegible]	\0 \0 \0 \0	\0 \0 \0 \0
0x7fffff00	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	[illegible]	[illegible]	[illegible]	[illegible]
0x7fffff20	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	[illegible]	[illegible]	[illegible]	[illegible]
0x7fffff40	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 [illegible]	[illegible]	[illegible] \0	\0 \0 \0 \0	[illegible]	[illegible]
0x7fffff60	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 [illegible]	[illegible]	[illegible] \0	\0 \0 \0 \0	[illegible]	[illegible]
0x7fffff80	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 [illegible]	[illegible]	[illegible] \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x7fffffa0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x7fffffc0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x7fffffe0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	n u b \0	a \0 y n	\0 s i \0	x u l F	\0 \0 \0 \0

It is important that you do not hard-code the values for any of the addresses in your program.

Turn Off Delayed Branching

From the settings menu, make sure Delayed branching is unchecked.



✓ Show Labels Window (symbol table)
✓ Program arguments provided to MIPS program
Popup dialog for input syscalls (5,6,7,8,12)
✓ Addresses displayed in hexadecimal
✓ Values displayed in hexadecimal

Assemble file upon opening
Assemble all files in directory
Assembler warnings are considered errors
Initialize Program Counter to global 'main' if defined

✓ Permit extended (pseudo) instructions and formats
Delayed branching
Self-modifying code

Editor...
Highlighting...
Exception Handler...
Memory Configuration...

Checking this option will insert a “delay slot” which makes the next instruction after a branch execute, no matter the outcome of the branch. To avoid having your program behave in unpredictable ways, make sure Delayed branching is turned off. In addition, add a nop instruction after each branch instruction. For example:

```
      LI   $t1 2
LOOP:
      ADDI $t0 $t0 1
      BLT  $t0 $t1 LOOP
      NOP                      # nop added after the branch instruction
      ADD  $t3 $t5 $t6
```

Submission Instructions

This assignment will be submitted in two parts. You will use the tags Lab4a_submission_# and Lab4b_submission_#.

Late hours will not be used for Part A of the assignment. If you do not submit a diagram or pseudocode by the Part A deadline, you can submit it with your Part B submission for less points.

Grading Rubric

Part A

```
 2 pt pseudo code
 1 pt pseudo code submitted by Part A deadline
 2 pt block diagram (-1 for hand drawn diagram)
 1 pt block diagram submitted by Part A deadline
```

Part B

```
 3 pt prompts match specification exactly (1 per prompt)
 6 pt register values
      2 pt  correct value stored in $s1
      2 pt  correct value stored in $s2
      2 pt  correct value stored in $s0
      Note: -1 per register if not sign extended properly
 1 pt program argument print
16 pt decimal sum output
      Note:  0 pt  if used print integer syscall
            -1 pt  if no negative sign
12 pt binary sum output
      Note:  0 pt  if used print integer as binary syscall
            -2 pt  if not sign extended properly
 6 pt style and documentation
      1 pt  comment on register usage
      1 pt  useful and sufficient comments
      1 pt  labels, instructions, operands, comments lined up in columns
      2 pt  README contains at least 8 sentences total with complete thoughts
      1 pt  complete headers for code and README
            Note: program header must include name, CruzID, date, lab name,
                  course name, quarter, school, program description, note on
                  program argument usage
                  README must include name, CruzID, lab name, course name
 8 pt (extra credit) Morse code sum output
      Note: -1 pt no minus sign
```