P09

Due No Due Date **Points** 100 **Submitting** an external tool **Available** after Oct 31 at 12am

P09

Programming Practice

These programming practice exercises are ***intentionally*** not auto-graded. **Use PennSim to test your code!**

Testing your own code and determining whether or not it functions correctly is a **critical** part of programming! **Nobody wants to hire a programmer that needs someone else to test their code! :-)**

Just writing your solution on paper is NOT an effective way to learn these skills! **Use PennSim!**

The only way to get good at programming (and make it easier) is to practice programming.

We are happy to help you in office hours if you are stuck!

Programming Practice: Control Flow

An expected instruction count is given for each to allow you to sanity-check your solution while creating it (if you have five times the required number of instructions, you're probably going about it wrong), and let you know whether or not you can improve the efficiency of code after you have verified it works correctly.

A common mistake is to focus too much on the instruction count at the start, and to think that writing the correct number of instructions is more important than writing ones that work. **That is not the case, so please do not fall into that trap.** Having the "correct" instruction count is **completely useless** if the code does not function correctly.

Note that the given instruction count does not include the BR START instruction given in the code provided below. In some cases the count is approximate because there are multiple reasonable ways to structure the code.

Always start your code from a template file containing the .ORIG directive, header comments, etc. You can paste the following into a blank .asm file as a starting point (and copy the subroutine header as needed if you need to create more than one subroutine):

```
; Filename:      <name of file>
; Author:         <your name>
; Description: <explanation of what the program does>

               .ORIG x0200
START

; add code here

DONE         BR START   ; repeat forever

; any test program data you need to create goes below here...
; (subroutine data should be immediately after the subroutine code that uses it!)


; Subroutine:   <subroutine name>
; Description: <explanation of what the subroutine does>
; Assumes:       <list/description of values passed to subroutine (and where they are)>
; Returns:       <list/description of values returned by subroutine (and where they are)>

; subroutine code and data goes here

               .END
```

Basic Skills

- Write a subroutine **ADD3** that takes as parameters the values in **R0**, **R1**, and **R2** and returns the sum in register **R3**. Write a test program that tests that your subroutine works for a few different values of these registers.
  Subroutine requires: 3 instructions.
- Write a subroutine **MAXIMUM** that takes as parameters the values in **R0** and **R1** and returns the maximum in register **R2**. Assume values are signed and in the range ±16383 to avoid overflow problems. Write a test program that checks that the subroutine works correctly for two negative numbers, two positive numbers, one negative and one positive number, both equal, etc. Be sure to test cases where **R0** is the maximum and where **R1** is the maximum!
  Subroutine requires: ~10 instructions and 1 .BLKW for register save/restore.

More Subroutines

- Write a subroutine named **MaxArray** that takes as parameters a starting address and a number of locations. It then should call the **MAXIMUM** subroutine above within a loop to find the maximum value in the array.

- Write a subroutine named **INITNEG1** that takes as parameters a starting memory address in **R1** and a number of locations in **R2**. It does not return a value in a register. This subroutine should use a loop to write the value **-1** to the indicated locations in memory. Write a test program that tests the subroutine for a few different starting addresses and numbers of locations. Be sure to test a location count of 0, 1, and some larger value. Do not worry about negative location counts.
  Hint: if you did not implement a program like this for the Week 10 programming practice, do that first before writing this subroutine...

String Processing Subroutines

- Write a subroutine **charcount** that takes a string address in **R1** and a character in **R0**. It returns the number of times the given character occurs in the string. This subroutine should be case-sensitive (i.e., 'a' should not match 'A', etc.)

The following subroutines are based on the standard C string library functions, so you can find information about what they should do by searching "<subroutine_name> C string library" on the Web.)

- Write a subroutine **strchr** that takes a string address in **R1** and a character in **R0**. It should return the address of the **first** occurrence of the character in the string, or if the character is not found in the string, it returns 0.
- Write a subroutine **strrchr** (note the extra **r**!) that takes a string address in **R1** and a character in **R0**. It should return the address of the **last** occurrence of the character in the string, or if the character is not found in the string, it returns 0.

Note: you can also turn any previous programming practice exercise into a subroutine!