Once you've logged into Codio via Coursera, follow the same instructions from the previous assignment (**LC4 Assembly Programming**) to open up PennSim in the XServer window.

Running user_echo.ASM in PennSim

1) From the Codio **File Tree**, click on the file: **os.asm**
   a. *This file contains the guts of our operating system for this HW*
   b. Look in the TRAP vector table, for the line that reads:
      `JMP TRAP_PUTC`

   Notice that it is the second instruction after the .ADDR x8000 directive.
   - When the "loader" loads this line into prog. mem, it will be placed in row x8001
   - We can call TRAP_PUTC, by using the TRAP instruction followed by x01.
     - *This will be done later in a file called: user_echo.asm*

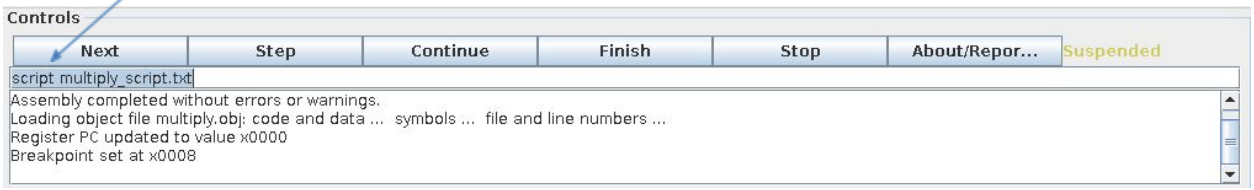
   c. Scroll down (approximately 115 lines or so) until you reach the lines that read:
      `.CODE`
      `TRAP_PUTC`
      This label marks the start of the PUTC Trap (aka OS – subroutine)
   d. When the "JMP TRAP_PUTC" instruction is run (from the vector table), the program counter will be advanced to the address labeled by "TRAP_PUTC"
   e. Further examine the code that follows; this is the operating system subroutine (aka – a TRAP) called TRAP_PUTC.
   f. Notice that this is the program we created in lecture to write 1 character to the ASCII display device on the LC4

2) From the **File Tree**, click on the file: **user_echo.asm**
   a. *This file is a program to test some of the operating systems TRAPs in os.asm*
   b. Scroll down to about the 24th line, look for the lines that read:
      `CONST R0, x54`
      `TRAP x01`
   c. The CONST inst. places hex 54 (which is actually the letter 'T' in ASCII code) in R0
      i. This serves as an argument to the TRAP_PUTC trap
   d. The TRAP x01 instruction forces the PC be set to x8001, w/ PSR[15]=1 (OS mode)
      i. TRAP x01 is TRAP_PUTC, so this TRAP call will have the effect of outputting a 'T' to the LC4's screen
   e. Examine the rest of this program, you'll see that its purpose is to output: "`Type Here>`" on the LC4 screen

3) From the "**File Tree**" click on the file: **user_echo_script.txt**
   a. *This file is the PennSim script that assembles and loads os.asm & user_echo.asm*
   b. Look carefully at the contents and compare how it differs from your last HW
      i. *Notice how it assembles and loads both os.asm and user_echo.asm*

4) Return back to the "PennSim" window you opened when you logged into Codio.
   a. In PennSim's "***Controls***" section, type in:

script user_echo_script.txt
Then press enter:



Note: this is an image from the last assignment (hence multiply_script.txt)

5) Press the "step" button and carefully go line by line until you see the letter 'T' outputted to the screen.
   a. Carefully watch how you start in user_echo.asm's code (in user program memory) and then with the call to TRAP, you enter into OS program memory.
   b. Understanding this process is crucial to understanding and eventually debugging this HW.

6) Finally, press the "Continue" button to see PennSim run the program until the END label is encountered.

7) Make certain you understand how these files work together before continuing on to the next section.

Problem 1) Working with existing TRAPS (GETC/PUTC) to "echo" keystrokes back to the user:

Once you are sure you understand how to run user_echo.asm and os.asm files, you will modify the file **user_echo.asm** in this problem. You will ***NOT*** need to modify **os.asm** in this problem.

Add to the end of user_echo.asm. Enable the program to read an ASCII character from the keyboard (using TRAP_GETC) and output it to the ASCII display (using TRAP_PUTC). Repeat this process in a loop until the user presses the <enter> key. Consider which type of loop is appropriate (for, while, do-while). As an aside, the process of taking in a key and displaying right back to a user is known as "echoing" their keystroke. *Make certain to test your code BEFORE starting problem #2.*

Problem 2) Writing a string to the ASCII Display:

Overview:

In this and the next problem, you will create a new traps: TRAP_PUTS and add it to the operating system file: **os.asm**. You'll also create a new file called: **user_string.asm** to test your new traps (similar to how user_echo.asm tested the GETC and PUTC traps in the last problem). This new traps will "put" **ASCII strings** to the screen.

What is an ASCII String?

A string is simply a consecutive sequence of individual ASCII characters, an "array" of characters as it is properly known. To be considered a string an ASCII character array must end with a NULL character. A NULL character is a non-ASCII character; typically, the number 0 is used. It is never printed; it is just to mark the end of the string. Consider this hypothetical example of a string containing: "Tom" in data memory. It starts at x4000 and uses 4 spots in data memory to hold each character (including the NULL)

Example Address	Contents in HEX	*ASCII translation*
x4000	x0054	*'T'*
x4001	x004F	*'O'*
x4002	x004D	*'M'*
x4003	**x0000**	***NULL***

Requirements for this TRAP:

First, let's create TRAP_PUTS. Open up the file **os.asm**; look where TRAP_PUTS is located in the vector table. Then scroll down to the label: **TRAP_PUTS** to see the Trap's implementation, you'll notice it's empty, but this is where you will be working. Implement the trap to do the following:

- This purpose of this function is to output a NULL terminated string to the ASCII display.
- Because we can't pass the entire "string" to a trap – because we'd run out of registers quickly if we had strings with more than 8 characters…instead, we will pass the address of the first character of a string to the trap using R0
- When TRAP_PUTS is called register R0 should contain the **address** of the first character of the string where the caller has stored the string in DATA memory. R0 is considered the argument to the TRAP.
    - *Using the example string above, R0 would equal x4000 when the trap is called*
- The last character in the string should be zero (we call a string with a zero following it, a ***null terminated string***).
- This trap will not return anything to the caller

Pseudocode for this TRAP:


```
TRAP_PUTS (R0) {
      check the value of R0, is it a valid address in User Data memory?
        if it is, continue, if not, return to caller
      load the ASCII character from the address held in R0
      while (ASCII character != NULL) {
            check status register for ASCII Display
              if it's free, continue, if not, keep checking until its free
            write ASCII character to ASCII Display's data register
            load the next ASCII character from data memory
      }
      return to caller
}
```


Calling and Testing the TRAP:

Next, we must call our TRAP. Copy **user_echo.asm**, and call the copy: **user_string.asm**. Open up user_string.asm and perform the following tasks:

1) Using the .FILL directive, populate User Data Memory, starting at address: x4000 with the hex code for the string: "I love CIT 593" Look at the back cover of your book to find the HEX code for each character. Don't forget to also set the value of the last address after your string with a NULL. You may label address x4000 if you'd like, but its not required.
2) Populate R0 with the address of the first character in your string
3) Call TRAP_PUTS using the appropriate TRAP # from the TRAP Vector Table shown at the top of os.asm
4) Create a **user_string_script.txt** file (by copying user_echo_script.txt and modifying it).
5) Test our your work, if it's not working, debug by going "step" by "step"

Problem 3) Reading a string from the ASCII Display:

Do not attempt this problem until you have problem #1 working properly. Open up the file **os.asm**, scroll down to the label: **TRAP_GETS**, you'll notice under the label is where you will be working. Implement the trap to do the following:

Requirements for this TRAP:

- This purpose of this function is to read characters from the keyboard until the "enter" key is pressed, store them in user data memory as a string in a location requested by the caller, then return the length of the string to the caller.
- When the trap is called, the caller must pass as an argument R0. R0 must contain an address in User Data Memory where the string that will be read from the keyboard will be stored.
- The trap should check to ensure R0 contains a valid address in USER data memory.
- The TRAP must then read in characters one by one. As it reads in each character, it must store them in data memory consecutively starting from the address passed in by the caller. Once the "enter" key is pressed, which is HEX: x0D or x0A depending on your machine, the trap must "NULL" terminate the string in data memory and return the length of the string (without including the NULL or enter) to the caller.
- The TRAP should return the length of the string in R1.

Calling and Testing the TRAP:

As an example, let's say a caller called the TRAP as follows:

```
; caller sets R0=x2000
; caller sets R1=0
; caller "calls" TRAP:    TRAP_GETS(R0)
; R1 should contain the length of the string after the TRAP returns
```

Let's say that when the TRAP is called, the user types in "Hello" on the console, followed by an enter. User Data Memory would contain the following after the TRAP returns:

Example Address	Contents in HEX	*ASCII translation*
X2000	x0048	*H*
X2001	x0065	*e*
X2002	x006C	*l*
X2003	x006C	*l*
X2004	X006F	*o*
X2005	**X0000**	***NULL***

And R1 would contain the number: 5 when the trap returns. The above is just an example; any valid address in data memory can be passed in by the caller and any string could be entered by the caller on the keyboard.

Implement your trap in the same os.asm file as the last problem. However, create **user_string2.asm** to test your trap (as the caller). Also, create **user_string2_script.txt** to test your caller code and os code.

After you complete testing the trap, in **user_string2.asm**, do the following:

1) Call TRAP_GETS with address: x2020 in R0. This will allow a string to be entered by the user and it will be stored in address R0
2) Print to the ASCII Display (using TRAP_PUTS and TRAP_PUTC)
   "Length = " X (where X is the length of the read in string from step 1 - you can assume length will be < 10)
3) Call TRAP_PUTS with address: x2020 in R0. This should output the same string that was read in from step 1.

Extra Credit:

Create a new trap called: TRAP_GETC_TIMER. You will need to modify **os.asm** and create new files : **user_string_ec.asm**, **user_string_ec_script.txt** for this problem. Your new TRAP_GETC_TIMER should do everything that TRAP_GETC used to do, except now it should "time out" if a user doesn't enter a key in 2 seconds. How to do this? Recall that TRAP_GETC checks the status register in a loop. Before entering that loop, you could set a timer for 2 seconds. Once you are inside the loop that checks the status register, you could also check the timer too. If the user doesn't press a key in 2 seconds, return back to the caller without checking the data register. I've included a TRAP called TRAP_TIMER to give you an example of how to work with the timer I/O device. Your user_string_ec.asm file should call TRAP_GETC_TIMER to get a character from the keyboard if it is entered within 2 seconds. If the user types a character within 2 seconds, print it to the ASCII display, otherwise your program should end gracefully with nothing printed.

Note: you cannot call one trap from another! Do you know why? Great exam question!!

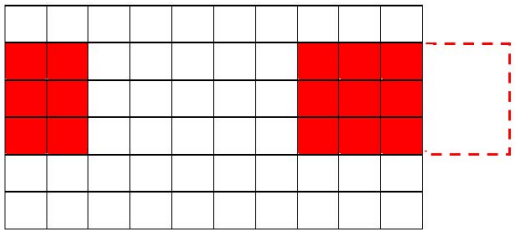
Problem 4) Drawing a box on the video display:

Overview:

For this problem you will create a new TRAP (and test code for it) that will draw a rectangle out to the video display I/O device. The size and color of the rectangle, will be set by the caller of the TRAP.

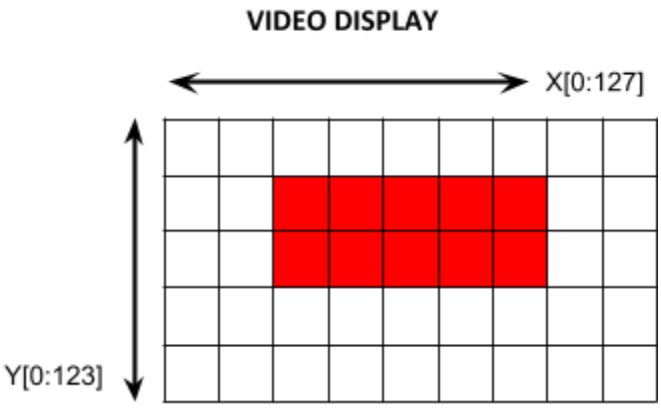
Requirements for this TRAP:

- Trap Name: TRAP_DRAW_RECT
- This trap will draw a rectangle whose location and dimensions will be set by the user.
- The color of the rectangle will also be an argument passed in by the caller
- When the trap is called the following registers should contain the following:
  - R0 – "x coordinate" of upper-left corner of the rectangle.
  - R1 – "y coordinate" of upper-left corner of the rectangle.
  - R2 – length of the rectangle (in number of pixels across the display).
  - R3 – width of the side of the rectangle (in number of pixels down the display).
  - R4 – the color of the rectangle
- Boundary Checking
  - The trap should check to see if the length/width are valid from the starting location of the box.
  - If invalid, return without drawing the box
- Comments:
  - Make certain to comment the TRAP's inputs and outputs as well as key components of your code so we can understand your work
  - Also make certain to comment the trap "test" program you write
- Extra Credit
  - Note: to implement the extra credit, you will be modifying the behavior of TRAP_DRAW_RECT. Once you get the TRAP working to the specifications above, it is encouraged that you make a backup of your code in case your E.C. implementation is not functional in time for submission.
  - The E.C. trap should still check to see if the length/width are valid from the starting location of the box, but if the starting coordinates are invalid, use (0,0) as the starting location instead. Keep the rectangle's width/length. (1 point)
  - If the box would go outside of video memory horizontally, correct the rectangle and make it "wrap around" the display (see diagram below). You do not need to implement this for rectangles that will go out of bounds vertically. (4 points)



Calling and Testing the TRAP:

As an example of using the TRAP, say the user calls the TRAP with the values of: R0=#2, R1=#1, R2=#5, R3=#2, R4=x7C00 (that's the color RED). The TRAP should then draw a RED box to the video display and it should look like this:



Implement the trap in os.asm. ***You can look to TRAP_DRAW_PIXEL (included in os.asm) for examples of working with video memory.***

To test the trap, create a new file: **user_draw.asm** which should contain a simple test of TRAP_DRAW_RECT that does the following:

- Call the TRAP for the following 3 rectangles, coordinates are given in (x, y) order:
  - A red box, upper left coordinates: (50, 5), length = 10 and width 5
  - A green box, upper left coordinates: (10, 10), length = 50, width 40
  - A yellow box, upper left coordinates: (120, 100), length = 27, width 10

Lastly, be certain to create a script file to assemble and load your program called: **user_draw_script.txt**.

Important Note on Plagiarism:

- We will scan your HW files for plagiarism using an automatic plagiarism detection tool.
- If you are unaware of the plagiarism policy, make certain to check the syllabus to see the possible repercussions of submitting plagiarized work (or letting someone submit yours)