# TCSS 372 PROBLEM 5

## EXPANDING MEMORY, WRITING A SIMPLE PROGRAM, AND RUNNING IT

### PURPOSE

The purpose of this problem is to begin approaching the capabilities of the full LC-3 (sans the interrupt capabilities). You will be testing your implementation of the LC-3 by running the same program on the book simulator and then running it on your simulator.

### DESCRIPTION

You will be implementing a few more instructions for your simulator along with the Run menu option in the debug monitor. The memory simulator will have to be expanded. Then, you will design and write an assembly program using the LC-3 editor, assemble it (and debug it as needed). The program will be run on the book simulator to determine final results. When you run the program on your simulator you can then check your results compared with the book simulator results.

### REQUIREMENTS

#### MEMORY

You should expand your memory array to several hundred words. You won't need all of it, but that size will allow you to show more memory and you will have plenty of room for the new assembly program. The actual number is not important as long as it is large enough to hold the program and data (see below).

#### NEW INSTRUCTIONS

Implement the following instructions:

- JSRR – jump to subroutine located at address in base register
- LEA – load effective address (needed to set up the base register)
- RET – return from subroutine
- TRAP x20 – getch (see code below if you are not using ncurses)
- TRAP x21 – out (same as putc or simple printf("%c");
- TRAP x22 – puts (simple printf without /n)

Don't forget to implement saving the return address in R7 when executing JSRR and restoring the PC from R7 when returning from a subroutine.

For TRAP x20, if you are not using ncurses, you will need the following code to implement the getch function:

```
#include <unistd.h>
#include <termios.h>

char getch() {
        char buf = 0;
        struct termios old = {0};
        if (tcgetattr(0, &old) < 0)
                perror("tcsetattr()");
        old.c_lflag &= ~ICANON;
        old.c_lflag &= ~ECHO;
        old.c_cc[VMIN] = 1;
        old.c_cc[VTIME] = 0;
        if (tcsetattr(0, TCSANOW, &old) < 0)
                perror("tcsetattr ICANON");
        if (read(0, &buf, 1) < 0)
                perror ("read()");
        old.c_lflag |= ICANON;
        old.c_lflag |= ECHO;
        if (tcsetattr(0, TCSADRAIN, &old) < 0)
                perror ("tcsetattr ~ICANON");
        return (buf);
}
```

This has worked in most of the C platforms I am familiar with. It will allow you to capture a single key (character) without echoing it to the console. It involves some deep systems routines which are beyond the scope of this course! Your trap x20 routine will then put the character into R0 before returning to your simulator controller.

**DEBUG MONITOR UPGRADE**

Note: After seeing how some of the other teams implemented their screens with ncurses, many of you might consider re-designing your debug monitor using that library. Those using ncurses will be free to layout the needed data in any way you wish as long as it conforms to the intent of the design given in problem 3.

Implement the Run option and include it in the menu for the debug monitor. The basic operation is that if someone is stepping through the program they should be able to select run at any point and have it finish the program running rather than stepping. Running will work just as it does in the book simulator. The debug monitor screen is not updated until after the HALT instruction terminates the program. Then it is printed out again with the final states of the CPU and memory.

Option: you might want to try to implement setting a breakpoint so that someone can run the program up to a specific address and then step through the rest of the program. This can be

done by setting up a small array of integers in which each slot can have a breakpoint address (in order). Each time through the loop the current value of the PC-1 is compared with items in the slots and if one matches, the program stops and the monitor switches to step mode. We will be doing this for the project so it wouldn't hurt to try it now. And it will make debugging a lot easier.

### ASSEMBLY PROGRAM

The assembly program will prompt a user to enter their first name. It will allow the user to type in their name. The string should be stored in a memory location (allow 20 or so words for storage, i.e. .BLKW #20). The program should then call a subroutine that "encrypts" the name by subtracting a small constant from each character and replacing the original string with the new one. The subroutine will need the starting address of the string location passed to it in R1. It will loop through the string subtracting the constant from each character and replacing it in the array. When the subroutine returns, with the starting address of the array returned in R0 to indicate success, the program prints a second prompt (Press any key to continue: ). When the user presses another key (without echo), the program proceeds to print out the array using the TRAP x22 facility.

Use the following procedure:

1. Design the flow chart for this program (example from pages 159-161 in the book). Be sure to indicate register usage in the processing blocks as shown in the book. This will be turned in with the other documentation and be graded.
2. Code the program using the LC-3 Editor. Assemble and debug if necessary.
3. Run the program on the LC-3 book simulator to verify that it works as described above.
4. Run the program on your LC3 simulator and preserve the results (of stepping through) in screen shots as before. You will only need shots of the first three calls to the subroutine to show it working. Get a screen shot of the final screen after the HALT.

Be sure to run the program multiple times with various names. You do not need to worry about error trapping and handling in this exercise.

### RECORDING AND COMPARING RESULTS

Using short names (say "Bob" or "Ted" or "Jane") step through the program and record the contents of the registers that are changed in the main portion and in the subroutine. Record the program counter (step) and whichever registers are changing. Also record the values that are being stored in memory (i.e. the name).

Running the program on the book simulator, do the same thing recording the registers and memory for each step.

Set these values up side by side for comparison. If there are any discrepancies between the two, and yet the program worked, see if you can figure out why. HINT: do not step into TRAP commands on the book simulator!

## SUBMISSION

Create a Word document as before with: 1) flow chart graphics, clearly labelled, 2) screen shots, and 3) comparison data from recoding.

Zip up: .h, .c, .docx into a file called problem5.zip and one member of the team submit it.

## GRADING

As in problem 3/4 this will be based on the clarity of your documentation, especially the flow graphs and the screen shots, but also the comparison data. I will look at the code only if there appears to be a problem.