CS061 – Lab 7
Emulating (a small part of) an Assembler

1   High Level Description

The purpose of this lab is to do some more advanced subroutine development, specifically nested subroutine calls.

You will construct and print a table of Assembly Language instructions with their Machine Language opcodes.

And you will build a search function for that table which emulates one basic element of an assembler or compiler.

2   Our Objectives for This Week

1.   Exercise 1 ~ Subroutine with helper: Prints out LC3 instructions & op-codes
2.   Exercise 2 ~ Subroutine with helper: Instruction locator & checker

REMEMBER: ALL your programs must now be written as subroutines!!

Call your subroutines with the JSRR instruction *(it works with subroutines located anywhere in memory)*

And remember to include all four steps of subroutine construction, being *especially* careful to back up & restore R7 *(or your program won't work and you won't be able to figure out why!)*

Exercise 1
Write the following subroutine:
```
;-----------------------------------------------------------------------------------
; Subroutine: SUB_PRINT_OPCODE_TABLE
; Parameters: None
; Postcondition: The subroutine has printed out a list of every LC3 instruction
;                   and corresponding opcode in the following format:
;                   ADD = 0001
;                   AND = 0101
;                   BR = 0000
;                   …
; Return Value: None
;-----------------------------------------------------------------------------------
```

Specifications:

- The data for the subroutine consists of two *remote* "parallel" arrays:
  ○ An array of numbers (not strings), each one representing an LC3 opcode (i.e. #1, #5, etc.)
  ○ An array of strings, each one representing the corresponding LC3 Assembly Language (AL) instruction, in the same order as the opcodes.
  (i.e. "ADD", "AND", etc.)
  ○ When invoked, the subroutine simply prints the tables as described.

Hints:

- The Op-code table from the text is provided at the end of this document.
- The two arrays will be stored remotely, with the remote addresses provided as local data *to the subroutine (this is so that a different subroutine can also access the same arrays)*
- Store the array of opcodes as a list of .FILL pseudo-ops
- Store the array of AL instructions as a list of .STRINGZ pseudo ops
  ○ Terminate this array of strings with a .FILL #-1
- To iterate through the two arrays in parallel, keep a pointer to each array (in separate registers)
  ○ Iterate through the opcode list one memory location at a time
  ○ You *could* print out each AL instruction using PUTS (Trap x22) - but we only have a pointer to the start of the entire array of strings, i.e. the address of the first instruction!! We don't know the start address of the rest of them! *So you are going to use the starting address of the whole array*, and iterate through it character by character, printing each with OUT (Trap x21), stopping at the #0 *(i.e. essentially make your own PUTS subroutine!)*
  At that point, you will
  - print the " = " *(which you will have stored with .STRINGZ)*;
  - print the opcode *(using your helper subroutine)*;
  - print a newline; and
  - finally, increment the two array pointers and start over.

○ Print the opcode with the helper subroutine described below, passing it in as a number in R2
○ Quit when the instruction array pointer points to the value xFFFF = #-1 *(use BRn)*
○ ***MAKE SURE YOU DO NOT PRINT THE TERMINATING NULL!!***

As always you must keep the simpl text window open to report runtime errors.

You will need a ***helper subroutine*** to print the op-codes to the console.
Create a version of your Assignment 3 as a subroutine *(just like you did in lab 6)* that takes a register parameter.
Skip the 12 MSBs, and print out just the 4 LSBs, as ascii 1s and 0s (<u>no terminating newline</u> - *leave that to the parent subroutine)*. So when passed e.g. the value #12 (x000C) in R2, the sub will print out "1100"

```
;----------------------------------------------------------------------------------------
; Subroutine: SUB_PRINT_OPCODE
; Parameters: R2 containing a 4-bit op-code in the 4 LSBs of the register
; Postcondition: The subroutine has printed out just the 4 bits as 4 ascii 1s and 0s
;                The output is NOT newline terminated.
; Return Value: None (but the input register R2 is unchanged)
;----------------------------------------------------------------------------------------
```

Test Harness:
Write a test harness that calls the SUB_PRINT_OPCODE_TABLE *(the SUB_PRINT_OPCODE will be called from <u>inside</u> that subroutine - this is the first time you will be using a nested subroutine call!*
Be very careful about backing up & restoring ONLY the necessary registers).

Fair Warning:
If you use .STRINGZ to simply store "ADD = 0001" (or any similar cheating hack-job) etc and print it out that way, you will not only get no credit for the lab, you will also hear a heavy sigh and will be walked away from in tired dismissal by the TA.

Exercise 2
Build a second pair of subroutines (same master/helper structure) that allow a user to repeatedly type in instruction names (example: "ADD", "JSR", "BR") and be told whether the instruction is valid (i.e. whether the requested instruction exists).
```
;----------------------------------------------------------------------------------------
; Subroutine: SUB_FIND_OPCODE
; Parameters: None
; Postcondition: The subroutine has invoked the SUB_GET_STRING subroutine and stored a string
;                as local data; it has searched the AL instruction list for that string, and reported
;                either the instruction/opcode pair, OR "Invalid instruction"
; Return Value: None
;----------------------------------------------------------------------------------------
;----------------------------------------------------------------------------------------
; Subroutine: SUB_GET_STRING
; Parameters: R2 - the address to which the null-terminated string will be stored.
; Postcondition: The subroutine has prompted the user to enter a short string, terminated
;                by [ENTER]. That string has been stored as a null-terminated character array
;                at the address in R2
; Return Value: None (the address in R2 does not need to be preserved)
```

;---

Specifications:

- The FIND subroutine invokes the GET_STRING sub, which prompts the user to type an [ENTER]-terminated string, which is stored as local data in the FIND sub (make sure to allocate enough memory locally for this string)
- The input string is compared with the array of LC3 instructions.
  As in the previous subroutine, the two arrays are accessed via locally-stored addresses.
- If the input string matches one of the instructions, then that line from the opcode table is printed out. Otherwise "Invalid instruction" is printed.

Examples:

- The user types "JSRR[ENTER]"
  - The subroutine prints "JSRR = 0100"
- The user types "AMD[ENTER]"
  - The subroutine prints "Invalid instruction"

Test Harness:
Just add a call to SUB_FIND_OPCODE to your harness for exercise 1.

Submission
Demo your lab exercises to your TA **_before you leave lab_**.
If you are unable to complete all exercises in lab, show your TA how far you got, and request permission to complete it after lab.
Your TA will usually give you partial credit for what you have done, and allow you to complete & demo the rest later for full credit, so long as you have worked at it seriously.
When you're done, demo it to any of the TAs or instructors in office hours **_<u>before</u> your next lab_**.
Office hours are posted on Piazza, under the "Staff" tab.

	15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD⁺	0001	DR	SR1	0	00	SR2

ADD^+

	15 14 13 12	11 10 9	8 7 6	5	4 3 2 1 0
ADD^+	0001	DR	SR1	1	imm5
AND^+	0101	DR	SR1	0	00 SR2
AND^+	0101	DR	SR1	1	imm5

	15 14 13 12	11	10	9	8 7 6 5 4 3 2 1 0
BR	0000	n	z	p	PCoffset9

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
JMP	1100	000	BaseR	000000

	15 14 13 12	11	10 9 8 7 6 5 4 3 2 1 0
JSR	0100	1	PCoffset11

	15 14 13 12	11	10 9	8 7 6	5 4 3 2 1 0
JSRR	0100	0	00	BaseR	000000

	15 14 13 12	11 10 9	8 7 6 5 4 3 2 1 0
LD^+	0010	DR	PCoffset9
LDI^+	1010	DR	PCoffset9

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
LDR^+	0110	DR	BaseR	offset6

	15 14 13 12	11 10 9	8 7 6 5 4 3 2 1 0
LEA^+	1110	DR	PCoffset9

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
NOT^+	1001	DR	SR	111111

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
RET	1100	000	111	000000

	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
RTI	1000	000000000000

	15 14 13 12	11 10 9	8 7 6 5 4 3 2 1 0
ST	0011	SR	PCoffset9
STI	1011	SR	PCoffset9

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
STR	0111	SR	BaseR	offset6

	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
TRAP	1111	0000	trapvect8

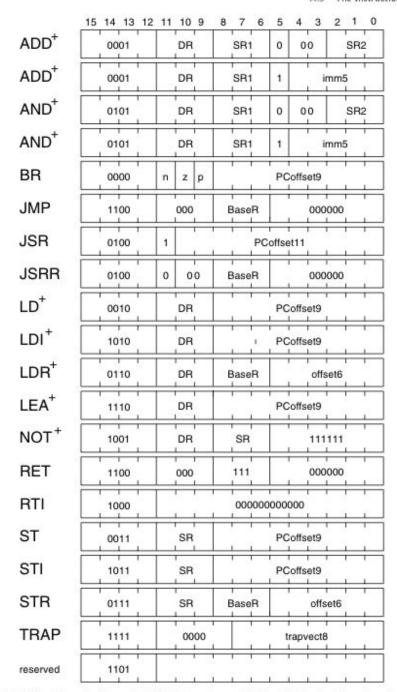|	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
reserved	1101	

Figure A.2   Format of the entire LC-3 instruction set. **Note:** + indicates instructions that modify condition codes