LC3-Your-Own-Adventure (part 1 of 2)

In the next two weeks, you will apply your knowledge of I/O and subroutines to develop an interactive program involving data manipulation, text output, and user input. The program simulates a popular type of book in which the reader encounters specific choices while reading through the story. The reader must decide on a particular choice, then flip to the page corresponding to that choice to continue the adventure. In our version of the game, the number of choices possible for any decision is limited to three. The point of this program is to give you hands-on experience with translating one type of data into another, to expose you to direct use of I/O device interfaces, and to help you make the connection between finite state machines and software.

You may want to refer back to first assignment for information about the weekly routine, the meaning of the challenge problems, instructions for checking out a copy, suggestions for tools, and instructions for handing in. Only details specific to this program are included in this document.

The Pieces

This week, you start with only a short piece of code that prints the low eight bits (one byte) of R3 as a two-digit hexadecimal number.

In programming studio, we will develop a set of subroutines that write a set of hexadecimal numbers to the monitor based on direct interaction with the LC-3’s DSR and DDR registers. A copy of the code provided for printing a one-byte hex number is attached at the end of this document. Our goals will be threefold: first, to develop the necessary PRINT_CHAR subroutine; second, to wrap the PRINT_HEX_DIGIT code into a properly working subroutine; and, third, to write a subroutine that uses it to print four space-separated values to the monitor.

On your own, you must write code to translate a database into a more useful format for use while playing the adventure game. (Precise details are given in the Details section below.) A given database consists of some number of records, which correspond to the “pages” of the virtual book. Each page (record) consists of a string and three page numbers corresponding to up to three possible choices allowed on that page.

After you have performed the translation, you must then validate your processing by using the PRINT_HEX_DIGIT subroutine to dump the contents of the array that you have built to the monitor.

Details

You will be given a text database starting at x5000. The database has a specific format, which is detailed below. You will need to write LC-3 assembly code to parse the text database to create an array that will help you to access the database for our adventure game.

The format of the database is as follows: the first string starts at x5000 and holds ASCII characters in the usual format (8 MSB are 0) in succession until a NUL value (ASCII x00) is found. The strings are of arbitrary non-zero length and NUL-terminated. The next three values immediately after the NUL are record numbers (stored as 16-bit two’s complement format) for choices 1, 2 and 3 respectively. The next text record in the database is located directly after the three choice memory locations. The end of the database is marked by a single NUL character, so if you find that the memory location after the three choice values for a record holds NUL, you should stop processing. (As a result, records are not allowed to use empty strings—keep that fact in mind when you create your own tests.)
The array that you build up from the database must store four pieces of information for each text record in the database in sequential order, namely the address of the first character of the string, and the choice 1, 2 and 3 values for that string. The array that you build must start at location x4000, and should terminate when data for the last string of the text database has been stored.

Finally, your program should print the values stored in the array out to the console in the format shown below and halt. The values that we print are the page number, choice 1, choice 2 and choice 3 respectively. Example output is given below.

Example Database

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x5000</td>
<td>'H'</td>
<td></td>
</tr>
<tr>
<td>x5001</td>
<td>'i'</td>
<td></td>
</tr>
<tr>
<td>x5002</td>
<td>NUL</td>
<td>Signifies end of string</td>
</tr>
<tr>
<td>x5003</td>
<td>1</td>
<td>Choice 1</td>
</tr>
<tr>
<td>x5004</td>
<td>4</td>
<td>Choice 2</td>
</tr>
<tr>
<td>x5005</td>
<td>3</td>
<td>Choice 3</td>
</tr>
<tr>
<td>x5006</td>
<td>'D'</td>
<td>Beginning of next text record</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x5031</td>
<td>'M'</td>
<td></td>
</tr>
<tr>
<td>x5032</td>
<td>'o'</td>
<td></td>
</tr>
<tr>
<td>x5033</td>
<td>'m'</td>
<td></td>
</tr>
<tr>
<td>x5034</td>
<td>NUL</td>
<td>Signifies end of string</td>
</tr>
<tr>
<td>x5035</td>
<td>2</td>
<td>Choice A</td>
</tr>
<tr>
<td>x5036</td>
<td>6</td>
<td>Choice B</td>
</tr>
<tr>
<td>x5037</td>
<td>-1</td>
<td>Choice C</td>
</tr>
<tr>
<td>x5038</td>
<td>NUL</td>
<td>End of database</td>
</tr>
</tbody>
</table>

Array Format

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4000</td>
<td>x5000</td>
<td>Address of text string for record 0</td>
</tr>
<tr>
<td>x4001</td>
<td>1</td>
<td>Choice 1 for record 0</td>
</tr>
<tr>
<td>x4002</td>
<td>4</td>
<td>Choice 2 for record 0</td>
</tr>
<tr>
<td>x4003</td>
<td>3</td>
<td>Choice 3 for record 0</td>
</tr>
<tr>
<td>x4004</td>
<td>x5006</td>
<td>Address of text string for record 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x4023</td>
<td>-1</td>
<td>Choice 3 for record 8</td>
</tr>
</tbody>
</table>

Example Output

00 01 04 03
01 05 02 08
... 08 02 06 FF

Challenges for Program 2

Here are a few challenges for this week.

(1 point) We implicitly assumed that page numbers fit in a byte. If a database contains more than 256 pages, print the following error message (exactly!) and do not print the list of page choices:

"Database invalid: contains more than 256 rooms!\n"

(2 points) Given a database containing $N$ pages, a choice $C$ is valid if (and only if) $−1 \leq C < N$. Verify that all choices in the database are valid, and print the following error message (exactly, other than for the hex values) for each invalid choice: "Page 1A contains invalid choice 8C.\n" Do not print the normal list of page choices if the database contains invalid choices.

(10 points) Even if the choices in a database are all valid, some of the pages may be unreachable. The game starts from page 0. Use the search ideas that we discussed in class to find all reachable pages (you’ll want another array to keep track of which pages are reachable). If some of the pages in the database are NOT reachable, print an error message (exactly, other than for the hex value) for each unreachable page: “Page 3C is not reachable.\n” Do not print the normal list of page choices if the database contains unreachable pages.

(7 points) (REQUIRES PREVIOUS CHALLENGE) If all choices are valid and all pages are reachable, print path information after the normal output. For each page, print a shortest path starting from page 00 and ending at that page, using the following format as an example for page 05: “00:13:2C:0F:05\n”
Specifics

- Your program must be written in LC-3 assembly language, and must be called `prog2.asm` — we will NOT grade files with any other name.
- Your code must begin at memory location x3000 (just don’t change the code you’re given in that sense).
- The last instruction executed by your program must be a HALT (TRAP x25).
- You must use the `PRINT_HEX_DIGIT` subroutine to print hex values to the monitor, and must not use any TRAPs in the `PRINT_CHAR` subroutine that `PRINT_HEX_DIGIT` must call. Note that the code given to you does not meet these constraints.
- You must use a subroutine to print four space-separated hex values from registers to the monitor by calling `PRINT_HEX_DIGIT`—the choice of subroutine name is yours; this code is to be developed during programming studio.
- All subroutine interfaces must be documented clearly, and the code must obey the documentation.
- You may assume that the database is valid for the purposes of the regular parts of the assignment. For the challenges, your code must handle those potential problems mentioned in that section.
- The text database’s records are implicitly indexed from 0 onwards. Since your array should be in the same sequential order as the text database, they have the same implicit indices. The values in the choice 1,2 and 3 memory locations refer to a record index.
- Certain choice values are -1. These do not point to any record, but rather indicate that the choice is invalid. You do not need to treat these values in any special way for Program 2.
- You must end your output text lines by outputting a linefeed (ASCII x0A) character to the monitor using the `PRINT_CHAR` subroutine.
- Your array must begin at x4000, and the text database begins at x5000.
- Your output must match the desired format exactly, as shown in this specification and in the test files provided.
- You may not make assumptions about the initial contents of any register.
- Your code must be well-commented. Comments begin with a semicolon. Follow the commenting style of the code examples provided in class and in the textbook.

Testing

You should test your program thoroughly before handing in your solution. Remember, when testing your program, you need to set the relevant memory contents appropriately in the simulator.

We have given you two sample test inputs, `test.asm` and `surreal.obj`. Note that you do not have the ASM version of surreal! (“Surreal” is a fairly extensive game in which you play a 198 student trying to finish your programming assignment, but to get to play the game, you have to finish your code...) You will need to assemble the `test.asm`. For each test, we have provided a script file to execute your program with the test input—`runtest` and `runreal`—and a correct version of the output: `testout` and `surrealout`.

Remember to debug your code before trying the tests! And remember that your final register values need not match those of the output that we provide, but all other outputs must match exactly.

Program 3

For next week, you will extend this week’s program to make use of the array to play the adventure game. Just as a reader must know the current page to continue reading a story, your program must keep track of its state within the adventure game.
Grading Rubric

Functionality (50%)
- 25% - program builds the correct array
- 25% - program prints the array values correctly

Style (25%)
- 5% - subroutine PRINT_CHAR defined and written properly
- 5% - subroutine PRINT_HEX_DIGIT defined and written properly
- 5% - subroutine to print four hex values defined and written properly
- 5% - code uses PRINT_CHAR and PRINT_HEX_DIGIT subroutines rather than OUT traps
- 5% - clear code structure used for parsing of text database

Comments, clarity, and write-up (25%)
- 5% - introductory paragraph clearly explaining program’s purpose and approach used
- 10% - each subroutine specifies an interface definition and a table of registers
- 10% - code is clear and well-commented (every line)

Note that correct output (and the points awarded for it) also depends on building the array correctly.
; prog2.asm - starting code for ECE198KL Spring 2013 Program 2
; "Copyright (c) 2013 by Steven S. Lumetta.
; Permission to use, copy, modify, and distribute this software and its
documentation for any purpose, without fee, and without written agreement is
hereby granted, provided that the above copyright notice and the following
two paragraphs appear in all copies of this software.
; IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
; ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
; DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
; EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
; OF THE POSSIBILITY OF SUCH DAMAGE.
; THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
; WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
; MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
; PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
; THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
; SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."

; Author:    Steve Lumetta
; Version:   1.00
; Creation Date: 21 January 2013
; Filename:   prog2.asm
; History:
; SL 1.00 21 January 2013
; Culled from Program 1 old code, then adapted slightly.

; .ORG  x3000 ; starting address is x3000
LD  R3,SAMPLE
JSR  PRINT_HEX_DIGIT ; will not work as is!
HALT

SAMPLE .FILL xAB ; data for main program

; For Program 2, you must wrap the following code up as a subroutine.
; This code relies on another subroutine called PRINT_CHAR
; that prints a single ASCII character in the low 8 bits of R2
to the monitor. The PRINT_CHAR call must not change any registers
;(other than R7)?

SUBROUTINE PRINT_HEX_DIGIT
; input: R3 -- 8-bit value to be printed as hex (high bits ignored)
caller-saved registers: R7 (as always with LC-3)
callee-saved registers: all other registers

PRINT_HEX_DIGIT
; registers used in this subroutine
; R2 -- ASCII character to be printed
; R3 -- 8-bit value to be printed as hex (high bits ignored)
; R4 -- digit counter
; R6 -- temporary

; print low 8 bits of R3 as hexadecimal
; shift R3 up 8 bits
ADD  R2,R2,#0
; initialize shift count to 8
ADD  R2,R2,#8

SHIFT_LOOP
ADD  R3,R3,R3
; shift one bit
ADD  R2,R2,#-1
; count down
BNZ  SHIFT_LOOP
; keep going until we're done

ADD  R5,R5,#0
; initialize digit count to 0

DIG_LOOP
ADD  R4,R4,#0
; initialize bit count to 0
ADD  R2,R2,#0
; initialize current digit to 0

BIT_LOOP
ADD  R2,R2,R2
; double the current digit (shift left)
ADD  R3,R3,#0
; is the MSB set?
BNz  MBB CLEAR
; if so, add 1 to digit

MBB CLEAR
ADD  R3,R3,R3
; now get rid of that bit (shift left)
ADD  R4,R4,#1
; increment the bit count
ADD  R6,R6,#-4
; have four bits yet?
BNZ  BIT_LOOP
; if not, go get another

ADD  R6,R6,#-10
; is the digit > = 10?
BNz  HIGH DIGIT
; if so, we need to print a letter
LD  R6,ASC_ZERO
; add '0' to digits < 10
BNz  PRINT DIGIT

HIGH DIGIT
LD  R6,ASC_HIGH
; add 'A' - 10 to digits >= 10

PRINT DIGIT
ADD  R2,R2,R6
; calculate the digit
JSR  PRINT CHAR
; print the digit

ADD  R5,R5,#1
; increment the digit count
ADD  R6,R6,#-2
; printed both digits yet?
BNZ  DIG LOOP
; if not, go print another

; THIS CODE WILL NOT WORK AS IS!
RET

; data for PRINT HEX subroutine
ASC ZERO .FILL x0030 ; ASCII '0'
ASC HIGH .FILL x0037 ; ASCII 'A' - 10
; Subroutine PRINT_CHAR
; input: R2 -- 8-bit ASCII character to print to monitor
; caller-saved registers: R7 (as always with LC-3)
; callee-saved registers: all other registers
;
PRINT_CHAR
    ST   R5,PRINT_CHAR_R0       ; save R0 to memory
    ST   R7,PRINT_CHAR_R7       ; save R7 to memory...why?
    ADD  R8,R2,#0               ; copy from R2
    OUT  ; cheat! (you must fix...)
    LD   R0,PRINT_CHAR_R0       ; restore R0 from memory
    LD   R7,PRINT_CHAR_R7       ; restore R7 from memory...why?
    RET
;
; data for PRINT_CHAR subroutine
PRINT_CHAR_R0  .BLKW  #1
PRINT_CHAR_R7  .BLKW  #1

; the directive below tells the assembler that the program is done
; (so do not write any code below it!)
.END