LC-3 Assembler & Programming

CS 350: Computer Organization & Assembler Language Programming

10/16: Solved

A. Why?

- Assembler language is easier to read/write than machine language.
- Understanding low-level programs helps you understand what compilers do.
- (Close to) low-level programming is still sometimes done for some embedded hardware applications, or when extreme efficiency is needed.

B. Outcomes

After this lecture, you should

- Know the format of assembler programs, including instructions and declarations of initialized and uninitialized variables.
- Know the difference between assembler instructions and assembler directives ("pseudo-instructions").
- Know how to begin and end an assembler program.
- Understand how important pseudocode and commenting are for low-level programs.
- Know how to do some basic operations in LC-3 assembler.

C. Assembler Language

- **Machine language** is the language recognized by the hardware.
  - Programs written in 0's and 1's.
- **Assembler language** is a symbolic version of machine language.
- An **assembler** is similar to a compiler, but instead of translating high-level code into object code, it translates assembler language programs into object code.
- Assembler language provides symbolic opcodes, labels for memory locations, automatic conversion between hex and decimal.

 © James Sasaki, 2016

- Plus, **assembler directives** give the assembler non-instruction info: Where your program should go in memory, where it ends, when to reserve memory locations, and where to place constants.

- Sample `printstring.asm` program below simulates `PUTS` (`TRAP x22`):
  - Print string pointed to by `R0`, one character at a time.
  - Recall a string is a sequence of words each containing one character.
  - Right byte contains ASCII representation of character, left byte is zero
  - String terminated by a word containing `x0000`).

```
; printstring.asm
;
; Given: R0 points to first word of string.
; At end: We've printed the string.
; Temporary register: R2
;
       .ORIG    x3000       ; (Start program at x3000)
        LEA     R0, string  ; Pt R0 to string to print

        ADD     R2, R0, 0   ; R2 = &current char to print
Loop    LDR     R0, R2, 0   ; R0 = curr char to print
        BRZ     Done        ; (BRZ 3) Loop until we see '\0'
        OUT                 ; (TRAP x21) print char in R0
        ADD     R2, R2, 1   ; Pt R2 to next char
        BR      Loop        ; (BR -5) Continue loop
Done    HALT                ; (TRAP x25) Halt execution

string .STRINGZ "Hello, world!"
       .END               ; Tell assembler this ends the file
```

Notes

- Comments begin with semicolon and go to the end of the line.
- The `.ORIG x3000` is an **assembler directive** (a.k.a. **pseudo-instruction**)
- The `.ORIG` doesn't generate any instructions or data itself.
  - Note the dot in `.ORIG` — all assembler directives begin with a dot.
  - A `.ORIG` specifies where your program is supposed to begin in memory.
- For `.ORIG x3000`, the following instruction will be placed at `x3000` (the one after that at `x3001`, etc).

    - There isn't anything magic about `x3000`; code can start anywhere except for for very low memory (for the `TRAP` table) and very high memory (where the `TRAP`-handling code is).
- `LEA R0, string` (at `x3000`)
    - "`string`" is a **label** (it stands for a memory location; `x3008` as we'll see below). The assembler will automatically figure out the `PC` offset (`x3008 – x3001 = 7`) to use in the instruction. If we change the program so that string is declared at a different location, rerunning the assembler will cause the PC offset to be recalculated.
- `ADD R2, R0, 0` (at `x3001`)
    - We saw this in the simple assembler case: the `0` is an immediate field
- `Loop LDR R0, R2, 0` (at `x3002`)
    - `Loop` is a label because it isn't an opcode; it stands for location `x3002`. The `0` is the base register offset.
    - Labels are typically written in column 1 but don't have to be. (The assembler actually ignores white space, so instructions can be written in column 1 but typically aren't.)
- `BRZ Done` (at `x3003`)
    - Recall that the branch instruction mnemonics are `BR` (or `BRNZP`) for unconditional branch, `BRN`, `BRZ`, `BRP` (for <, =, > 0), `BRNZ`, `BRZP`, and `BRNP` (for ≤, ≥, ≠ 0), and `NOP` (for mask `000`, which never branches). Also recall that if more than one of `N`, `Z`, and `P` appear, they have to appear in that order.
    - We'll see below that label `Done` is at x3007, so the assembler will use a PC offset of `x3007 – x3004 = 3`.
- `OUT` (at `x3004`)
    - `OUT` is an assembler mnemonic for `TRAP x21`, which is what we wrote in the previous lecture. Similarly, you can use `GETC`, `PUTS`, `IN`, and `HALT` for traps `x20`, `x22`, `x23`, and `x25`.
- `ADD R2, R2, 1` (at x3005)

  - Again, the `1` indicates an immediate value of one.
- `BR Loop` (at `x3006`)
  - Since `Loop` is declared at `x3002`, the assembler will use `x3002 – x3007` = `–5` for the `PC` offset.
- `HALT` (at `x3007`)
  - This is an abbreviation for `TRAP x25`; it halts execution by resetting the CPU's running flag.
- `string .STRINGZ "Hello, world!"` (at `x3008`)
  - First, the assembler will note that the label `string` is declared `x3008`.
  - `.STRINGZ` (note the dot) is an assembler directive. It causes a sequence of words to be filled in with the ASCII character values of a string. In our case, `"Hello, world!"` takes 14 characters (13 plus the null character). So `'H'` = `x48` = `72` is stored at `x3008`, `'e'` = `x65` = `101` is stored at `x3009`, ..., `'!'` = `x21` = `33` at `x3014`, and `0` at `x3015`.
  - It causes 14 words to be given values for the 13 characters of `Hello, world!` plus `x0000` for the null character. The words start at, since that's the next location, and the name `string` gets bound to that location. The last word is at `x3015`. (If there were another `.STRINGZ` directive or something else that required us to allocate some space, it would be at `x3016`.)
- `.END` (would be at `x3016` if it stood for an executable instruction).
  - This directive that tells the assembler that this is the end of the program text. Note `.END` and `HALT` are different: `HALT` stands for executable code and you can have any number of them in your program; `.END` is a directive, doesn't stand for executable code, and must appear exactly once in your program file.

Other notes:

- Labels are case-sensitive but opcodes, assembler directives, and register names aren't.

- Unlike high-level languages, where you declare your identifiers (constants and variables) at the top of your program, in LC-3 assembler you declare them **after your code**, otherwise you'll execute your data as instructions.
  - E.g., putting the `.STRINGZ` of `"Hello, world!"` would insert 14 words of data there. Luckily (?) they would be have like `NOP` instructions because the leading seven 0 bits would be treated as opcode `0000` (i.e., `BR`) with mask `000` (i.e., never branch).

The Assembler Directives `.STRINGZ`, `.FILL`, and `.BLKW`

- The directive `.FILL` *value* is used to initialize a memory location to the specified value (which can be in decimal or hex). E.g., `.FILL x10` or `.FILL 16`
  - (We'll see labels (= names of memory location) later; they can also be used as a value.)
- `.STRINGZ` "*string*" stands for a sequence of fills for the individual characters of the string (plus a null character to terminate the string).
  - E.g., `.STRINGZ "Hi"` stands for `.FILL x48 .FILL x69 .FILL x00`.
  - The string can include `\n`, `\t`, etc as in C/Java.
  - The directive `.BLKW` *number* is the same as *number* occurrences of `.FILL 0`.
  - Typically used for (what we think of as) variables and arrays.
  - If a label is attached, it's associated with the first word allocated.

D. LC-3 Editor and Simulator

- The textbook-provided LC-3 editor and simulator runs under Windows.
- There's a link from the syllabus page of the course website. The direct link is: http://highered.mheducation.com/sites/0072467509/student_view0/index.html
- For Windows, the two programs you want are `LC3Edit.exe` and `Simulate.exe`. (The Unix version is different and buggy under Mac OS.)
- Me personally, I run the Windows version on my Mac using WINE, a collection of libraries that implement various Windows operations natively. (Me personally, I used WineBottler (see MacUpdate) to get WINE. There's a tutorial on installing WINE (I haven't tried it, but it looks reasonable.)

 © James Sasaki, 2016

E. The LC-3 Editor

- The `LC3Edit`.exe program is the editor for assembler programs. Programs should be saved as `*.asm` files. To assemble a program, click the `asm` button.
  - (We can also use the editor to write machine programs in binary or hex.)
  - Figure 1 shows an editor window after assembling `printstring.asm`.

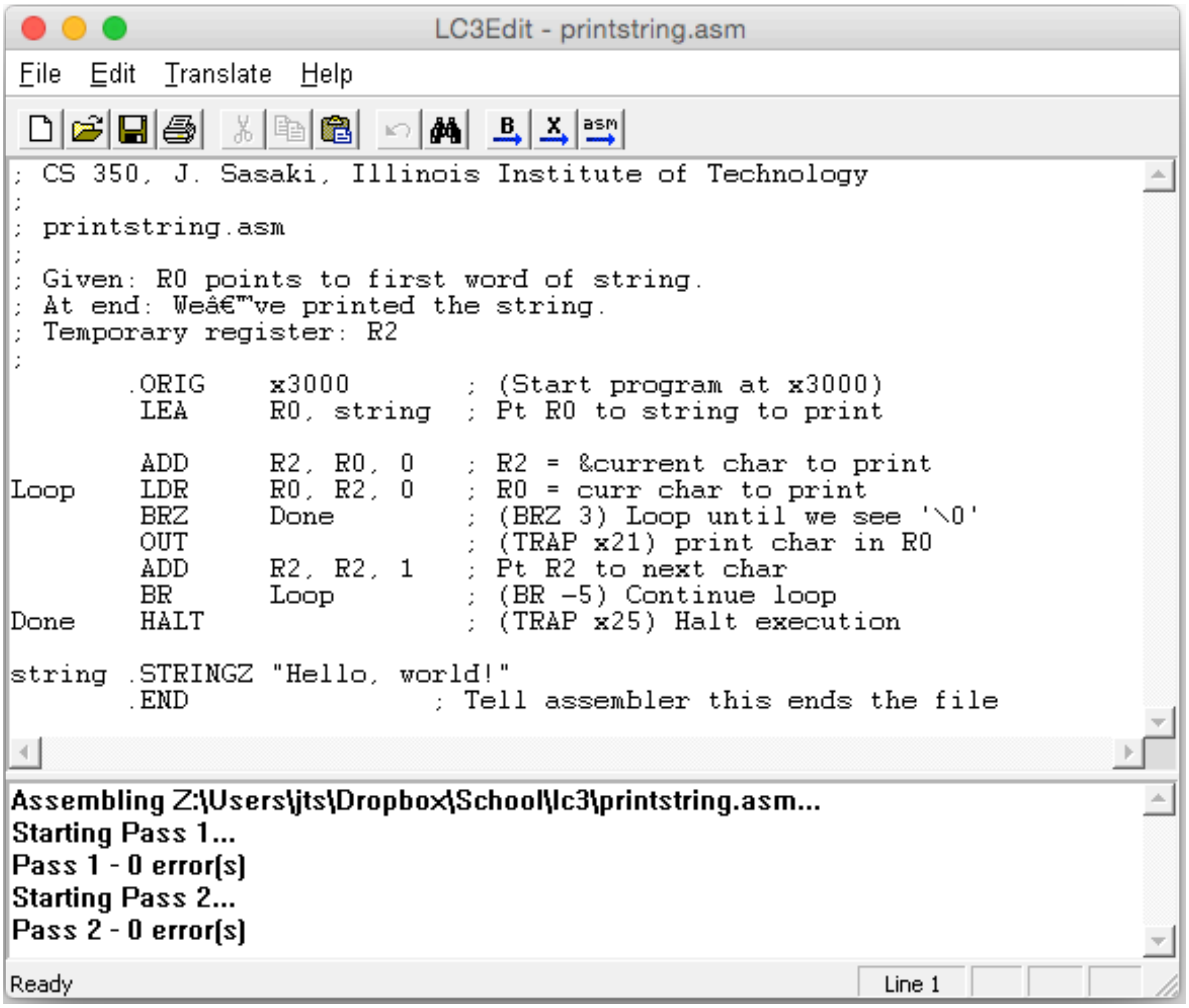

Figure 1. LC-3 Editor Window

- Assembly produces a `*.obj` ("object") file, which can be loaded into the simulator. It also produces some auxiliary files:

- `*.hex` and `*.bin` for compiled code: The first line is the `.ORIG` number; the remaining lines contain the contents with which to initialize memory (in 4-digit or 16-bit binary format).
- `*.sym` for the symbol table: This holds a list of labels defined in the program plus the locations the labels stand for.
- `*.lst` for a program listing.

The LC-3 Simulator

- The `Simulate.exe` program is the LC-3 simulator. It's a graphical simulator.
    - Figure 2 shows the initial display of the simulator. If you've been running a program and want to clean everything up, you can use *File → Reinitialize Machine* to get to Figure 2.
    - Figure 3 shows the simulator after loading in an object file (with *File → Load Program…* or *Ctrl+L*).

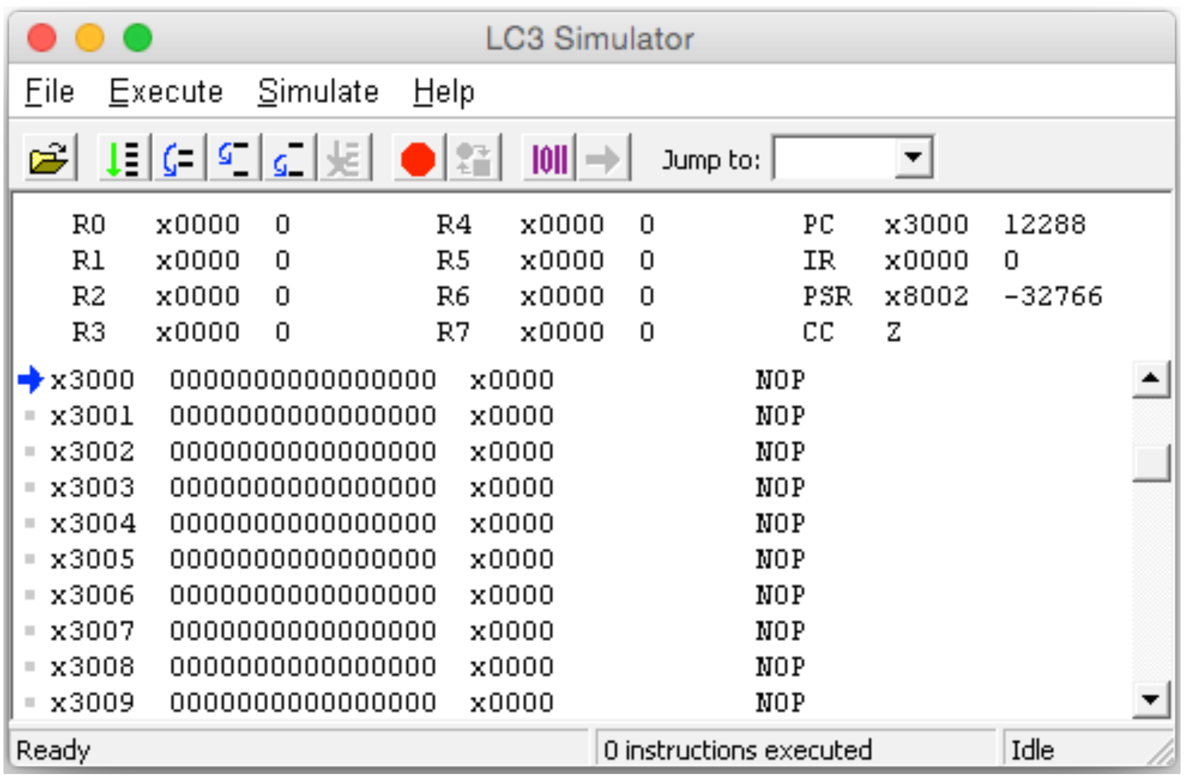

Figure 2: Freshly-Initialized LC-3 Simulator

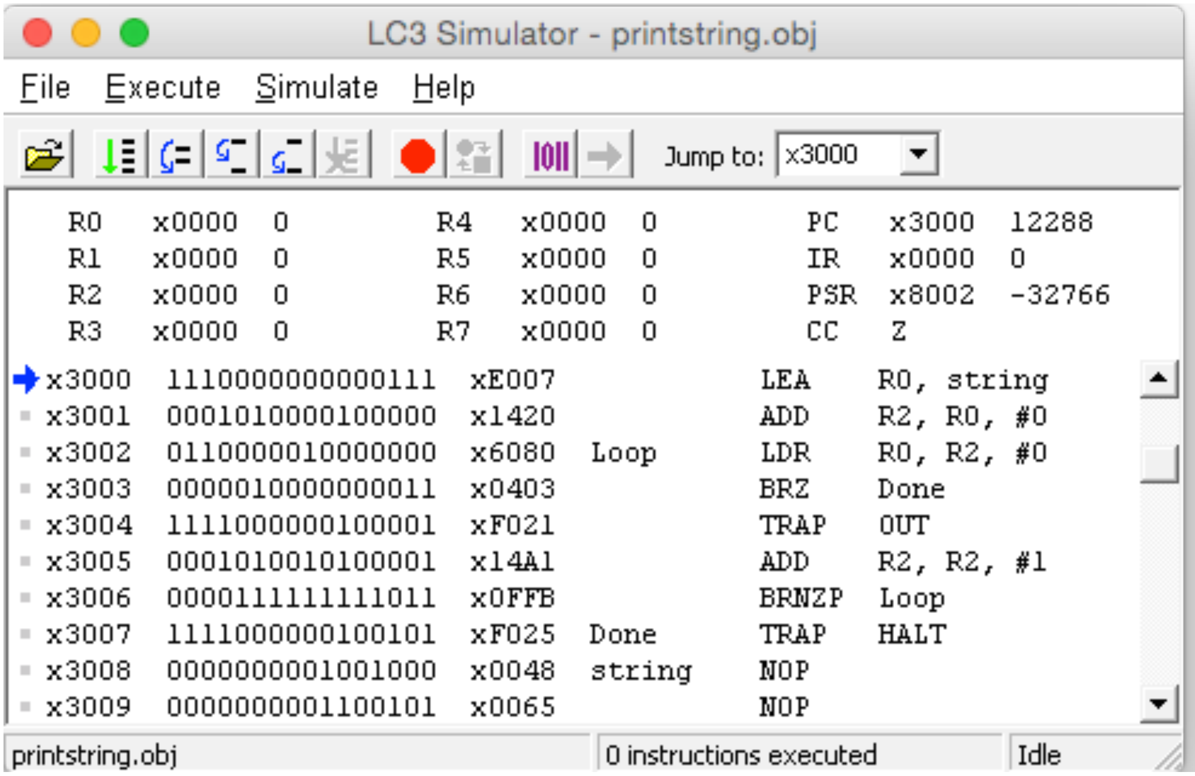

Figure 3: LC-3 Simulator: After Loading printstring.obj

Simulator Window Contents

- The top of the display contains the registers (in hex and decimal), the program counter (`PC`), instruction register (`IR`), condition code (`CC`, either `N`, `Z`, or `P`), and the program status register (`PSR`).
    - The `PSR` is used in I/O; also, `PSR[1]` is the CPU running bit: `TRAP x25` (a.k.a. `HALT`) sets this bit to `0` to stop the instruction cycle.
- Memory is displayed one row per address. The blue arrow points to the address in the `PC`.
    - The value of the address is shown in binary, hex, and as an instruction.
        - Uninitialized memory and characters (and more generally, words with value x00…) are shown as the `NOP` instruction because any word that begins with binary 0000 000 looks like a branch with 000 mask bits.
    - You can scroll the memory display; you can also move the display to a specific address by entering it in the `Jump to` area.

- The grey dot next to the address is red when a breakpoint is set at that location (see below).

- You can **change the value of a memory location** by double-clicking on it. This brings up a dialog box into which you can enter a new value for a memory location.

Simulator Control Area

- Figure 4 shows the 5 groups of controls at the top of the simulator window.
    - In general, a button is dimmed when its action is not available.

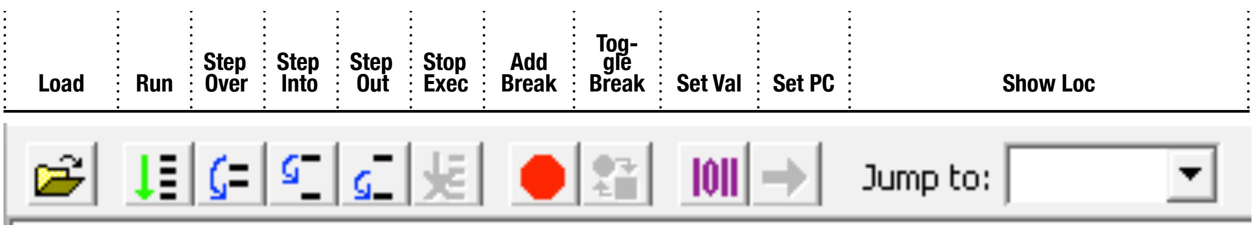

Figure 4: LC-3 Simulator: Simulator Buttons

Left-to-right, the controls are

- **Load Program** (same as *File* → *Load Program*)
- **Execute** buttons (also see the *Execute* menu).
    - **Run** (same as *Execute* → *Run*): Execute instructions until the HALT trap causes execution to stop, or until the Stop Execution button is pressed.
    - **Step Over** (same as *Execute* → *Step Over*): Execute one instruction and pause; if the instruction is a TRAP or subroutine call, execute the entire TRAP or call and then pause.
    - **Step Into** (same as *Execute* → *Step Into*): Execute instructions until you enter a TRAP or subroutine call, then pause.
    - **Step Out** (same as *Execute* → *Step Out*): Execute instructions until you return from a TRAP or subroutine call, then pause.
    - **Stop Execution** (same as *Execute* → *Stop*): Pause execution. (Definitely handy for stopping infinite loops.) You can tell if the program is running (not paused) if the count of the number instructions executed is increasing. This count is at the bottom-right of the simulator window (see Figure 5).

Figure 5: LC-3 Simulator: Count of Instructions Executed

- **Breakpoint Buttons** (also part of menu *Simulate*)
  - **Add Breakpoint**: Brings up a dialog box to add a breakpoint.
  - **Toggle Breakpoint** — if you click on a memory location to highlight it, then this button flips the location's breakpoint status. Clicking the grey or red circle to the left of a memory location also toggles its breakpoint status (see Figure 6).

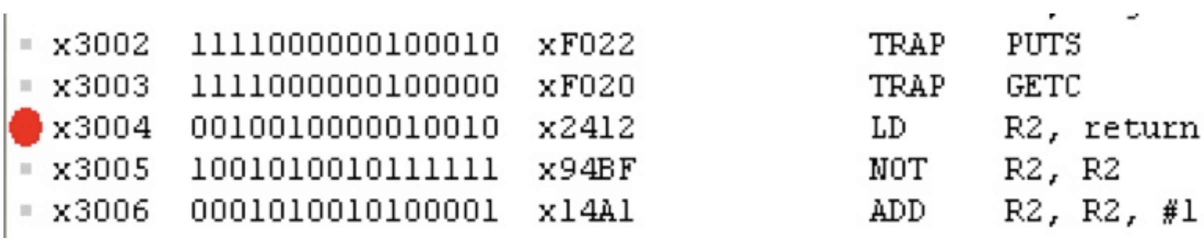

Figure 6: LC-3 Simulator: After Setting a Breakpoint

- **Set Buttons**
  - **Set the Value** of the highlighted location (the line highlighted in blue).
  - **Set the PC** to the highlighted location.
  - **Change the Displayed Location**: Scroll the window so that the specified location is shown. You can type in a location or pull-down a menu of recent locations.

F. Sample Program: Multiplying an Integer and a Natural Number

- `multiply.asm` is a sample program for multiplying two integers `X` and `Y` using repeated addition. Here is pseudocode for it. (Note the assumption `Y ≥ 0`.)

```
; multiply.asm
; ...
; Set product = X * Y (where Y >= 0), using repeated addition.
; If Y < 0, we halt.
;
; Pseudocode:
;
; Property: product = X*(Y-k) and 0 <= k <= Y
;    (So when k = 0, product = X*Y)
;
;    product = 0               ; Initialize product and k
;    k = Y
;    until k <= 0
;       product += X
;       k--
```

- And here is code for it. The comments and program have been slightly modified relative to the auxiliary textbook by Patt & Patel.

```
; Register usage: R1 = k, R2 = X, R3 = product
;
        .ORIG    x3050
         LD      R2, X          ; R2 = X
         AND     R3, R3, 0      ; R3 = X * (Y-k)
         LD      R1, Y          ; k = Y
Loop     BRNZ    Done           ; until k <= 0
         ADD     R3, R3, R2     ;    R3 = R3 + X
         ADD     R1, R1, -1     ;    k--
         BR      Loop
Done     ST      R3, product    ; product = X*Y
         HALT

X       .FILL    12
Y       .FILL    8
product .BLKW    1              ; Holds X*Y at end
        .END
```

G. Sample Program: Reading a String

- The `readstring.asm` program prompts the user for input and reads in a sequence of characters until the user enters return (the newline character, ASCII 10).

- There is no built-in `TRAP` for reading strings, so this program is more useful than the `printstring.asm` program which simulated `PUTS` (`TRAP x22`) to print a string.
- Let's start with some high-level pseudocode for the program:

```
; readstring.asm
; ....
; Read and echo characters until we see a return. (Also echo
; the return.)  Store the characters (but not the return) as
; a string in a buffer.
;
; Pseudocode 1:
;    Point to the beginning of the buffer
;    Prompt user for the input
;    Read a character
;    until character = return
;       Echo the character
;       Copy the char to the pointed position in the buffer
;       Point to the next buffer position
;       Read the next character
;    Echo the return character
;    End the string in the buffer and print it
```

- If we break down operations and assign some variables and registers, we can get pseudocode that's closer to assembler code. Note the use of the C notations `&`*variable* and `*`*pointer*.

```
; Pseudocode 2:
;    char buffer[…], *bp; // bp = buffer position
;    bp = &buffer[0];
;    Print "Enter chars (return to halt): "
;    Read char into R0
;    Calculate R0 - return char
;    until R0 - return char == 0
;       Print char in R0
;       *bp = R0
;       ++bp
;       Read char into R0
;       Calculate R0 - return char
;    end loop
;    Print the return character
;    *bp = '\0' (end the string)
;    Print the string
;    Halt
```

- Here is the code. Note most of the pseudocode appears in the line comments.

```
; Register usage
;    R0 = GETC/OUT char, R1 = bp (buffer position)
;    R2 = -(return char), R3 = temp
;
        .ORIG    xC000
         LEA     R1, buffer    ; bp = &buffer
         LEA     R0, msg
         PUTS                  ; prompt for input
         GETC                  ; get char into R0
         LD      R2, retChar   ; R2 = return char
         NOT     R2, R2        ; R2 = -(return char) - 1
         ADD     R2, R2, 1     ; R2 = -(return char)
         ADD     R3, R0, R2    ; calculate R0 - return char

Loop     BRZ     Done          ; until r0 = return char
         OUT                   ;    print char in R0
         STR     R0, R1, 0     ;    *bp = char read in
         ADD     R1, R1, 1     ;    ++bp
         GETC                  ;    get char into R0
         ADD     R3, R0, R2    ;    calc char - return char
         BR      Loop          ; continue loop

Done     OUT                   ; print return char in R0
         AND     R3, R3, 0     ; R3 = null char ('\0')
         STR     R3, R1, 0     ; *bp = null char
         LEA     R0, buffer
         PUTS                  ; print the string we read in
         HALT

retChar .STRINGZ "\n"          ; Return character (\n)
msg     .STRINGZ "Enter chars (return to halt): "
buffer  .BLKW    80            ; buffer space for string
        .END
```

H. Sample Program: Accessing a Table Element

- In C we might write `table[k] = x` to set a table element to some value.
- To implement this in assembler, we need to get `&table[k]` into a register so that we can use `STR` (store using base register) to set `table[k]`.
- We can calculate `&table[k]` as `&table[0] + k *` *width of a table element*.
- If `table` is close enough to use the `LEA` instruction, then `LEA` *reg*, `table` sets the register to `&table[0]`. If `table` is too far away for an `LEA` to access it, then we need to store `&table[0]` somewhere and access that.

- The declaration

  ```
  tablePtr .FILL table    ; &table[0]
  ```

  uses `.FILL` to initialize a memory location with the address of `table`: The assembler will substitute the memory address associated with the label `table`.

- So `LD` *reg*, `tablePtr` has the same effect as `LEA` *reg*, `table` but the `LD` works even if `table` is inaccessible using `LEA`.
  - (But `tablePtr` has to be declared close to the `LD` instruction.)
- There are a couple of versions of the program to set `table`[k] to a value
  - `table_near.asm` (uses `LEA` on the nearby table)
  - `table_far.asm` (uses `LD` on `tablePtr` because `table` is too far away to be accessed with `LEA`).
- Here is `table_near.asm`:

```
; table_near.asm
; ....
; Set table[k] = x where k and k are variables and
; the table is with PC-offset distance.
;
; We assume table entries take up one word each
;
; Register usage: R0 = &table[0], R1 = &table[k]. R2: temp
;

; Since the table is within a PC-offset distance, we
; can use LEA to get &table[0].
         .ORIG  x8000
          LEA   R0, table       ; R0 = &table[0]

; Make R2 = &table[k]
          LD    R2, k           ; R2 = k
          ADD   R1, R0, R2      ; R1 = &table[k]

; Set table[k] = x
          LD    R2, x           ; R2 = value
          STR   R2, R1, 0       ; *R1 (= table[k]) = value
          HALT

k        .FILL  4               ; index into table
x        .FILL  -1              ; value to copy into table

table    .BLKW  100             ; space for table[0..99]
         .END
```

- And here is `table_far.asm` (some comments abbreviated)

```
; table_far.asm
; ....

; To make R0 = &table[0], we load the address of the table,
; as stored in tablePtr.
         .ORIG  x8000
          LD    R0, tablePtr    ; R0 = tablePtr = &table[0]

; Make R2 = &table[k]
          LD    R2, k           ; R2 = k
          ADD   R1, R0, R2      ; R1 = &table[k]

; Set table[k] = x
          LD    R2, x           ; R2 = value
          STR   R2, R1, 0       ; *R1 (= table[k]) = value
          HALT

k        .FILL  4               ; index into table
x        .FILL  -1              ; value to copy into table

; Define tablePtr = &table[0], then declare some space to
; ensure that the table is not PC-offset-accessible.  Note
; table starts at x8109.
;
tablePtr .FILL  table           ; &table[0]
         .BLKW  256
table    .BLKW  100             ; space for table[0..99]
         .END
```

LC-3 Assembler & Programming

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Assembler language is easier to read/write than machine language.
- Understanding low-level programs helps you understand what compilers do.
- (Close to) low-level programming is still sometimes done for some embedded hardware applications, or when extreme efficiency is needed.

B. Outcomes

After this activity, you should

- Be able to read assembler programs and differentiate instructions from directives.
- Be able to write assembler programs (declare where they go in memory, write instructions, declare constants and variables, and end programs).
- Connect pseudocode and assembler code for some small examples.
- Assemble and repeatedly run a program, changing its data values between runs.

C. Problems

For Problem 1, use the following program

```
; Print a message string (Note: it happens to end in a line feed)
;
        .ORIG    x3000          ; Start the program at x3000
         LEA     R2, msg        ; Pt R2 -> start of message string
Loop     LDR     R0, R2, 0      ; R0 = next char of string
         BRZ     Done           ; Loop until end of string
         TRAP    x21            ; Print current char of string
         ADD     R2, R2, 1      ; Pt R2 to next char of string
         BR      Loop           ; Continue loop
Done     HALT                   ; Stop program
msg     .FILL    x48            ; "H"
        .FILL    x69            ; "i"
        .FILL    x20            ; " "
        .FILL    x21            ; "!"
        .FILL    x0A            ; Line feed
        .FILL    0              ; end of string
        .END                    ; End of program
```

1. For the program above
   a. Which lines of the program contain assembler instructions? Assembler directives (pseudo-instructions)?
   b. Where will each instruction and fill be stored in memory, and what addresses do the labels indicate?
   c. Do labels have to go in column 1? Which of the following are case-sensitive: labels, opcodes, pseudo-instructions, and register names?
   d. What would happen if you replace each `.FILL` by a `.BLKW`?
   e. In the `LDR R0, R2, 0` instruction (a) Is the "`, 0`" necessary? (b) What would happen if we replace the `0` by `R0`? (c) If we replace `R0` and `R2` by `0` and `2`?
   f. In the `ADD R2, R2, 1` instruction, what would happen if we replace the `1` by `R1`?

2. What is the difference between a `HALT` instruction (`TRAP x25`) and the `.END` assembler directive? How many halt instructions can you have in a program? How many `.END` directives?
3. With assembler programs, we declare our data and variables after the program instead of before — what would happen if we moved the `.FILL` lines to be directly after the `.ORIG`?

4. Complete the following assembler program that sums the contents of the array `Data` and leaves the result in `R3`. (Note the assembler doesn't know about arrays: To it, `Data` is a label that stands for `x8100`, so what we're actually doing is summing the values at `x8100` – `x810B`.) Declare the values of the array to be 2, 4, 6, ..., 24 by using twelve lines of `.FILL` after the `HALT` instruction. (The 254 words of 0 declared before `Data` simulate other code and data that the program might have.)

Addr	***Instruction***		***Asm***	***Comments***
			???	(origin x8000)
x8000	1110 001 011111111		LEA R1, ???	R1 ← addr of Data
x8001	0101 011 011 1 00000		AND R3,R3,???	R3 ← 0
x8002	0101 010 010 1 00000		ADD R2,R3,???	R2 ← R3 = 0
x8003	0001 010 010 1 01100		ADD R2,R2,???	R2 ← 12
x8004	0000 010 000000101	Loop	BRZ ???	Loop: If Z, quit loop
x8005	0110 100 001 000000		LDR R4, ???	. R4 = val pt'd by R1
x8006	0001 011 011 000 100		ADD ???	. R3 ← R3 + R4
x8007	0001 001 001 1 00001		???	. ++R1 (pointer)
x8008	0001 010 010 1 11111		???	. --R2 (counter)
x8009	0000 111 111111010		???	End loop (go to top)
x800A	1111 0000 0010 0101	Done	???	HALT
x800B …	(245 words of 0)		???	(space to get to x8100)
x8100	0000000000000010	Data	.FILL 2	(The values to sum)
x8101	0000000000000100		???	
x8102	0000000000000110		???	
x8103	0000000000001000		???	
x8104	0000000000001010		???	
x8105	0000000000001100		???	
x8106	0000000000001110		???	
x8106	0000000000010000		???	
x8107	0000000000010010		???	
x8108	0000000000010100		???	
x8109	0000000000010110		???	
x810A	0000000000011000		???	
x810B	0000000000011010		.FILL 26	(Last value to sum)
			???	(end of program)

5. Continuing with the program from the previous problem, say we want to parameterize the number of values to add by declaring it using `Nbr .FILL 12`.
   a. How would we have to modify the program?
   b. Why would declaring `Nbr` just before `Data` cause a problem that could be solved by decreasing the 245 words of 0 to a smaller value?

These next problems involve experimenting with the `multiply`, `readstring`, and `table_near` / `table_far` programs. (There are no questions to answer.)

6. Use an LC-3 editor to assemble `multiply.asm` and load the object file into the simulator. Put a breakpoint at the `HALT` instruction and run the program (single-step or continuously; your choice). When it reaches the `HALT`, verify for yourself that the result product is correct. Using the blue arrow button, reset the PC to the start of the program; double-click on the `X` and `Y` locations, give them different values, and rerun the program (without reloading it). Repeat this a few times until you feel comfortable with the program and the simulator. (One suggestion: Try positive, negative, and zero values for `X` and for `Y`.)
7. Repeat this process of assembling and re-running on `readstring.asm`. Try re-running the program without reloading it and enter a shorter string; notice how old characters remain in the buffer. Try changing the `STR` instruction that terminates the string with `'\0'` with a `NOP` (double-click on the instruction and change it to 0). Re-run with a shorter string and verify that extra leftover characters get printed.
8. With `table_near.asm` and `table_far.asm`, try playing with the distance between the code and the table by changing or adding `.BLKW` between the code and the table and re-assembling. Verify that `table_far.asm` still works even if the table is within PC-offset distance. On the other hand, `table_near.asm` should fail to assemble if `table` is too far away from its `LEA`. (You should get an error message saying that the offset needed is too large.)

Activity 11 Solution

1. (Assembler program)

   1a. Directives begin with a period: The lines with `.ORIG`, `.FILL`, and `.END` contain directives; the others contain assembler instructions.

   1b. From the `LEA` through the final `.FILL`, we have addresses `x3000`, `x3002`, …, `x300C`. So `Loop` is at `x3001`, `Done` at `x3006`, and `msg` at `x3007`.

   1c. Labels don't have to go in column 1 (the assembler is actually whitespace-insensitive). Labels are case-sensitive but opcodes, pseudo-instructions, and register names are not.

   1d. Since `.BLKW` *c* (where *c* is a constant) stands for *c* occurrences of `.FILL 0`, we have that `msg .BLKW x48` would declare $48_{16} = 72_{10}$ words of zeros. (Useless fact: `.BLKW 0` isn't an error.)

   1e. The "`, 0`" in `LDR R0, R2, 0` instruction is necessary. Replacing `0` by `R0`, `R0` by `0`, or `R2` by `2` causes errors

   1f. Replacing the `1` in `ADD R2, R2, 1` with `R1` changes the instruction from an incrementation (`R2` ← `R2` + `1`) to a register + register addition (`R2` ← `R2` + `R1`).

2. `HALT` (`TRAP x25`) is an executable instruction; `.END` is an assembler directive. You can have any number of HALT instructions but only one `.END` directive, at the end of the file.

3. Moving the the `.FILL` lines to be directly after the `.ORIG` would make the `.FILL` values be executed as instructions. A `.FILL` assigns a value to a memory location, and if control reaches that location, then the value will be treated as an instruction even if we intended the value to be data.

4. (Complete program)

Addr	*Instruction*	*Asm*	*Comments*
		.ORIG x8000	
x8000	1110 001 011111111	LEA R1,Data	R1 ← & Data
x8001	0101 011 011 1 00000	AND R3,R3,0	R3 ← 0
x8002	0101 010 010 1 00000	AND R2,R2,0	R2 ← 0
x8003	0001 010 010 1 01100	ADD R2,R2,12	R2 ← 12
x8004	0000 010 000000101	Loop BRZ Done	Loop: If Z, quit loop
x8005	0110 100 001 000000	LDR R4,R1,0	. R4 = val pt'd by R1
x8006	0001 011 011 000 100	ADD R3,R3,R4	. R3 ← R3 + R4
x8007	0001 001 001 1 00001	ADD R1,R1,1	. ++R1 (pointer)
x8008	0001 010 010 1 11111	ADD R2,R2,-1	. --R2 (counter)
x8009	0000 111 111111010	BR Loop	End loop (go to top)
x800A	1111 0000 0010 0101	Done TRAP x25	HALT
x800B ... x80FF	(245 words of 0)	.BLKW 245	(space to get to x8100)
x8100	0000000000000010	Data .FILL 2	(The values to sum)
x8101	0000000000000100	.FILL 4	
x8102	0000000000000110	.FILL 6	
x8103	0000000000001000	.FILL 8	
x8104	0000000000001010	.FILL 10	
x8105	0000000000001100	.FILL 12	
x8106	0000000000001110	.FILL 14	
x8106	0000000000010000	.FILL 16	
x8107	0000000000010010	.FILL 18	
x8108	0000000000010100	.FILL 20	
x8109	0000000000010110	.FILL 22	
x810A	0000000000011000	.FILL 24	
x810B	0000000000011010	.FILL 26	(Last value to sum)
		.END	(end of program)

5. (Declare `Nbr .FILL 12` and parameterize previous program by using `Nbr`.)

   a. At `x8003`, to initialize `R2`, use `LD R2, Nbr`

   b. With `Data` at `x8100`, the PC offset used by `LEA R1, Data` is at its maximum. Adding a a declaration of `Nbr` would push the label `Data` to be at `x8101`, and the `LEA R1, Data` would produce an assembler error message. The easiest solution to this problem is to decrease the 254 words of 0 to 253, then declare `Nbr`, and then declare `Data`.

(Actually, now that I think about it, if you replace the two instructions at `x8002` and `x8003` with the `LD R2, Nbr`, that would free up an extra word of space, in which case declaring `Nbr` just before `Data` would work out.)

6. (No question)
7. (No question)
8. (No question)

 © James Sasaki, 2016