The LC-3 Assembler

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Assembler language is easier to read/write than machine language.

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.

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.
    - 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
```

Discussion of `printstring` program

- Comments begin with semicolon and go to the end of the line.
- The `.ORIG x3000` is an **assembler directive** (a.k.a. **pseudoinstruction**)
  - 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 where pre-existing TRAP code is).
- `LEA R0, string` — `LEA` is the opcode, `R0` through `R7` can be used as registers
- "`string`" is a **label** (it stands for a memory location). The assembler will automatically figure out the `PC` offset to use in the instruction.
  - This instruction is at `x3000` and string is at `x3008`, so the offset is `7`.

  - `LEA R0, 7` would've been equivalent (but then your code would break if you added/subtracted code between here and `string`).
- `ADD R2, R0, 0` — The `0` indicates an immediate value of zero. You can use decimal or hexadecimal (precede hex constants by `x`). Book writes `#0`; the `#` is optional.
- `Loop LDR R0, R2, 0` — `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` — 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). Note if more than one of `N`, `Z`, and `P` appear, they have to appear in that order.
  - Note `BRNZ` means branch if negative or zero; to branch if `≠ 0`, use `BRNP`.
  - Similarly, `BRNP` means negative or positive; to branch if not `> 0`, use `BRNZ`.
- `OUT` — handy shortcut for `TRAP x21`. (Writing `TRAP x21` [or `TRAP 33`] works too.) Other abbreviations for `TRAP`s are as in previous lecture: `GETC`, `IN`, `PUTS`, `HALT`.
- `ADD R2, R2, 1` — The `1` indicates an immediate value of one.
- `BR Loop` — The assembler figures out that `Loop` is at `PC-5` and uses `-5` as the `PC` offset.
- `HALT` — abbreviation for `TRAP x25`; causes execution to stop. (At runtime, it actually jumps to OS code that includes an instruction that causes execution to stop.)
- `string .STRINGZ "Hello, world!"`. This allocates space for a string and gives it the name `string`, `.STRINGZ` is an assembler directive. 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 `x3008`, 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` — the directive that tells the assembler that this is the end of the program text. Note `.END` and `HALT` are different.

Other notes:

- Labels are case-sensitive but opcodes, assembler directives, and register names aren't.
- We don't put the declaration of data at the top of the program.
  - Otherwise we'd execute the data as instructions.
  - E.g., putting the `.STRINGZ` of Hello, world! would insert 14 words of information there (they behave like NOP instructions because the leading seven 0 bits get treated as opcode 0000 = Branch with mask 000 = never branch)

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

- The directive `.FILL` *number* is used to declare one numeric constant (decimal or hex). `.STRINGZ` is much nicer than equivalent sequence of fills
  - E.g., `.STRINGZ "Hi"` vs 3 lines `.FILL x48 .FILL x69 .FILL x00`.
  - By the way, you can't use `\n`, `\t`, etc as in C/Java, sigh.
- 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.mcgraw-hill.com/sites/0072467509/student_view0/` .
- 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.

(I used WineBottler to get WINE but I'm not sure that works anymore. There's a tutorial on installing WINE at http://www.davidbaumgold.com/tutorials/wine-mac/. I haven't tried it, but it looks reasonable.)

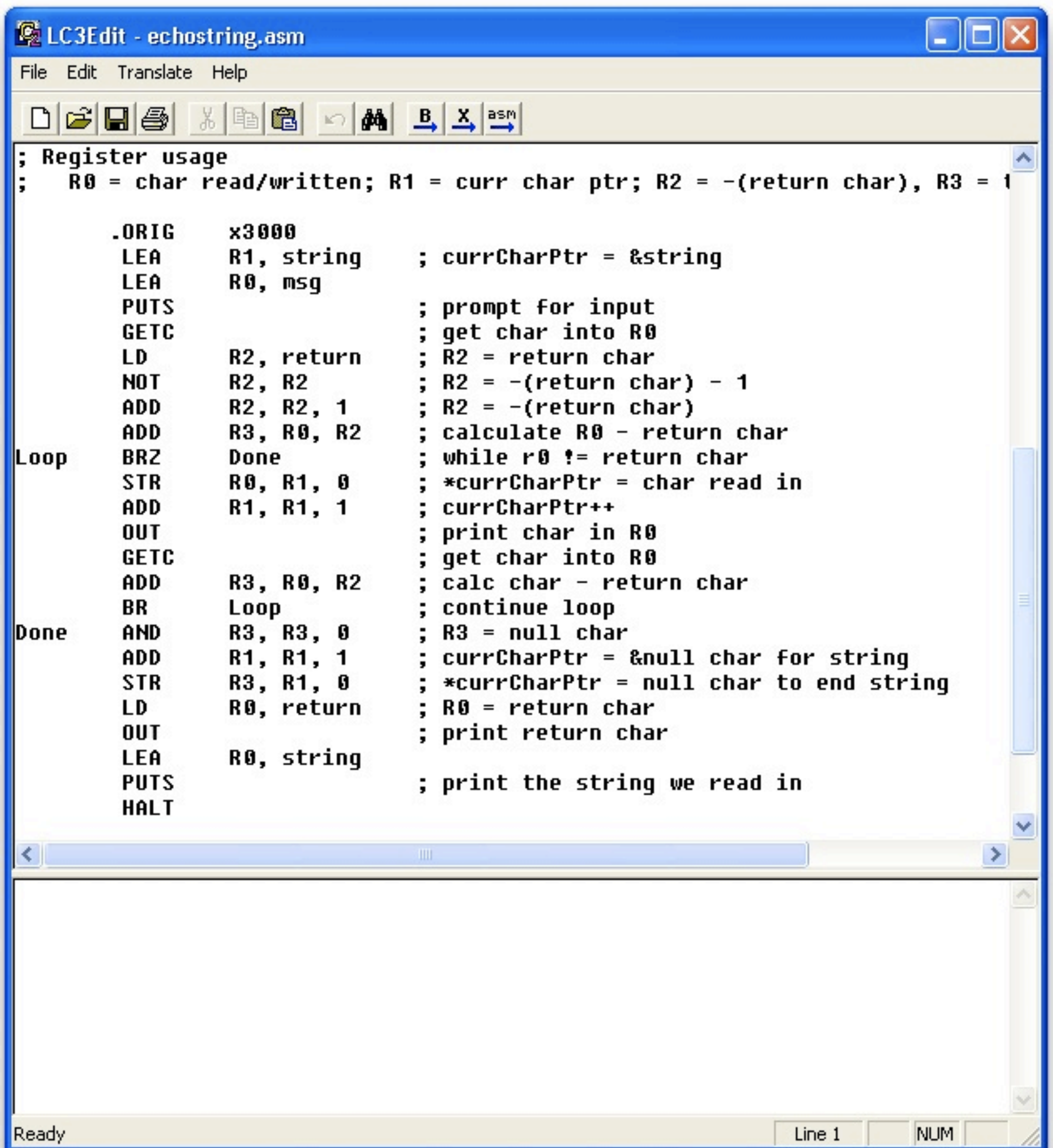


Figure 1. LC-3 Editor Window

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.)
- 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.

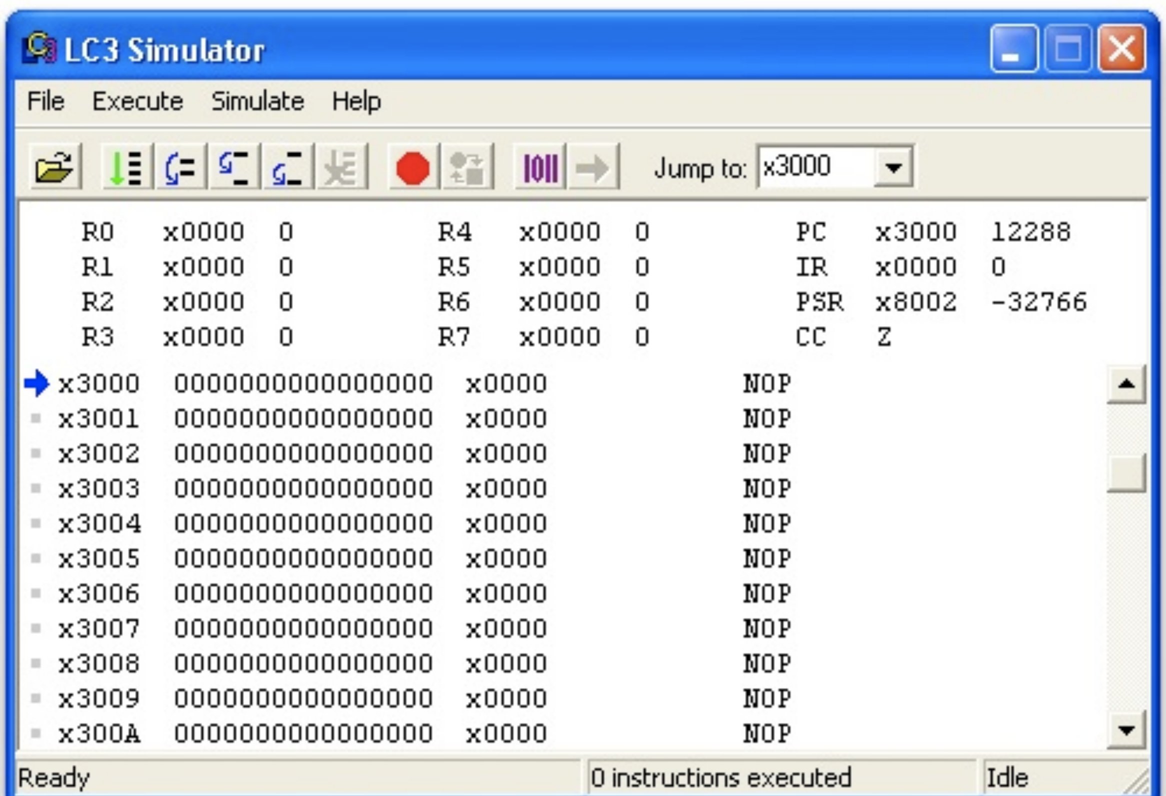


Figure 2: Freshly-Initialized LC-3 Simulator

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*).
- 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.

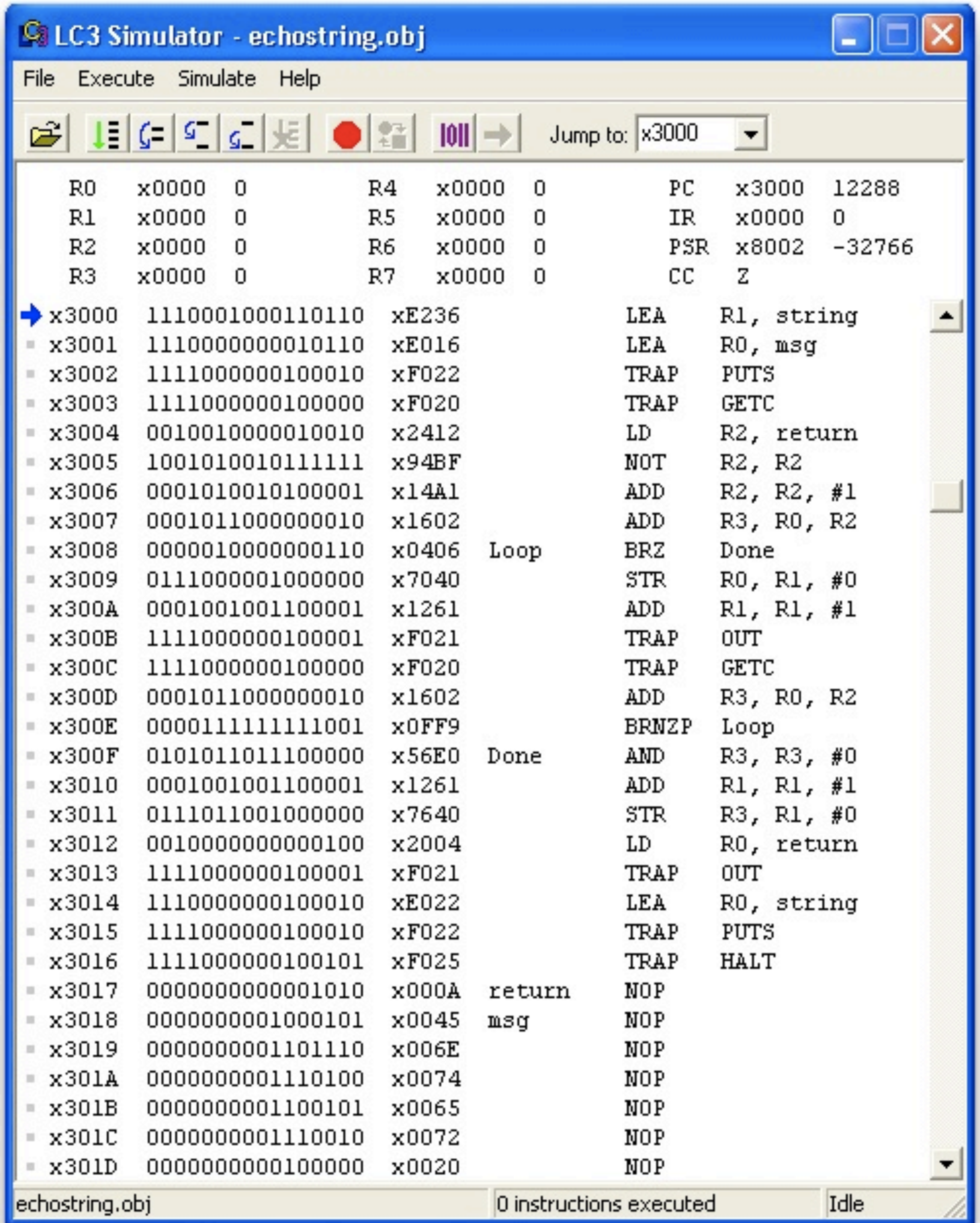
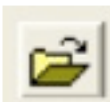


Figure 3: LC-3 Simulator: After Loading echostring.obj

E. LC Simulator Controls

- 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

 is for **Load Program** (same as *File* → *Load Program*).

 **Execute** (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).

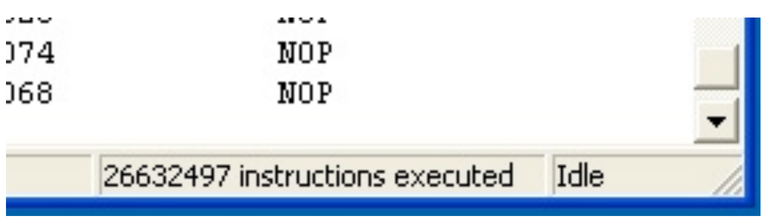


Figure 5: LC-3 Simulator: Count of Instructions Executed

 is for breakpoints (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).

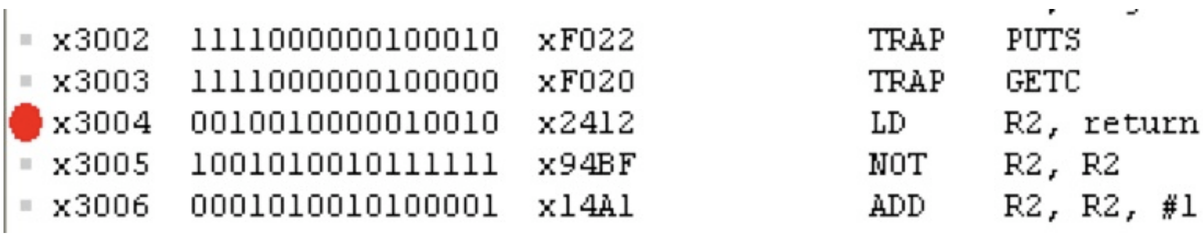
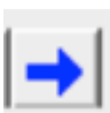


Figure 6: LC-3 Simulator: After Setting a Breakpoint

 **Sets the PC** to the highlighted location

Changes the **Displayed Location**

- You can type in a location or pull-down a menu of recent locations.

LC-3 Assembler

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Assembler language is easier to read/write than machine language.

B. Outcomes

After this activity, you should

- Be able to read assembler programs and differentiate the instructions from the directives.
- Be able to write assembler programs (declare where they go in memory, write instructions, declare constants and variables, and end programs).

C. Questions

For the following questions, 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. (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?

2. Do labels have to go in column 1? Which of the following are case-sensitive: labels, opcodes, pseudo-instructions, and register names?
3. What would happen if you replace each `.FILL` by a `.BLKW`?
4. How would the program behave if the five `.FILL` lines were replaced by `msg .STRINGZ "Hi !\n"` ? (Annoying, isn't it?)
5. 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`?
6. In the `ADD R2, R2, 1` instruction, what would happen if we replace the `1` by `R1`?
7. 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?
8. [**A Surprisingly Important Question**] What would happen if we moved the `.FILL` lines to be directly after the `.ORIG`?
9. Here a the machine-level program for summing the contents of `x3100` – `x310B`. Rewrite it as a complete program: Translate the instructions into assembler, add labels for the branch instructions, and add `.ORIG`, `HALT`, and `.END` lines. In addition, assume the values are stored starting at the label `VALUES` (instead of address `x3100`). Declare the values as twelve `.FILL` lines after the `HALT` instruction. Make the twelve values 2, 4, 6, 8, ..., 24.

Addr	*Op*	*Instruction*	*Comments*
`x3000`	`LEA`	`1110 001 011111111`	`R1 ← x3100 (PC + 0xFF)`
`x3001`	`AND`	`0101 011 011 1 00000`	`R3 ← 0`
`x3002`	`AND`	`0101 010 010 1 00000`	`R2 ← 0`
`x3003`	`ADD`	`0001 010 010 1 01100`	`R2 ← 12`
`x3004`	`BR`	`0000 010 000000101`	`Loop: If Z, quit loop`
`x3005`	`LDR`	`0110 100 001 000000`	`R4 = *R1`
`x3006`	`ADD`	`0001 011 011 0 00 100`	`R3 ← R3 + R4`
`x3007`	`ADD`	`0001 001 001 1 00001`	`++R1 (pointer)`
`x3008`	`ADD`	`0001 010 010 1 11111`	`--R2 (counter)`
`x3009`	`BR`	`0000 111 111111010`	`End loop (BR top of loop)`
`x300A`	`...`	*(... 246 words ...)*	
`x3100`	`...`	*(... the data ...)*	

10. 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) Would it make a difference to declare `NBR` before the values or after the values? (**Hint**: What if `NBR` is, say, 256?)

Solution

1. (a) Directives begin with a period: The lines with `.ORIG`, `.FILL`, and `.END` contain directives; the others contain assembler instructions. (b) 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`.
2. 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.
3. A `.BLKW` *c* (where *c* is a constant) stands for c occurrences of `.FILL 0`, so `msg .BLKW x48` would declare $48_{16} = 72_{10}$ words of zeros and so on. (It turns out that `.BLKW 0` is not treated as an error by LC3 assembler, so a following instruction/directive will be at the same location as the `.BLKW 0`. So `Label1 .BLKW 0 Label2 .FILL 7` would declare `Label1` and `Label2` to be at the same location.)
4. Unlike C/C++/Java, the LC3 assembler doesn't recognize `\n` in strings as an escape sequence: It treats it as two characters, backslash and `n`.
5. The "`, 0`" in `LDR R0, R2, 0` instruction is necessary. Replacing `0` by `R0`, `R0` by `0`, or `R2` by `2` causes errors
6. Replacing the `1` in `ADD R2, R2, 1` with R1 causes an error.
7. `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.
8. 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.
9. (Note the `LEA` now loads the address of `VALUES`, which is at `x300B`, not `x3100`.)

```
        .ORIG  x3000
         LEA   R1, VALUES
         AND   R3, R3, 0
         AND   R2, R2, 0
         ADD   R2, R2, 12
Loop     BRZ   Done
         LDR   R4, R1, 0
         ADD   R3, R3, R4
         ADD   R1, R1, 1
         ADD   R2, R2, -1
         BR    Loop
Done     HALT
VALUES  .FILL  2
        .FILL  4
        .FILL  6
        .FILL  8
        .FILL  10
        .FILL  12
        .FILL  14
        .FILL  16
        .FILL  18
        .FILL  20
        .FILL  22
        .FILL  24
        .END
```

10. (a) We need to add the `NBR .FILL 12` declaration; instead of setting `R2` ← `12` (using the `AND` and `ADD` at `x3002` and `x3003`) we can use `LD R2, NBR` .
(b) The furthest location a `LD` at `x3002` can access is `x3102`; since `VALUES` will be at `x300A`, we can only have `x3102`−`x300A` = `xF8` = `248` values. So if `NBR` > `248`, then it must be declared before `VALUES`; for `NBR` ≤ `248`, we can declare it before or after `VALUES`.

