LC-3 Sample Programs

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Understanding low-level programs helps you understand what compilers do.
- Low-level (or close to low-level) programming is still done for some embedded hardware applications, or when extreme efficiency is needed.

B. Outcomes

After this lecture, you should

- Understand how important pseudocode and commenting are for low-level programs.
- Know how to multiply and read a string in LC-3 assembler.

C. 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's some pseudocode for it. Note the assumption `Y ≥ 0`.

```
Property: product = X*(Y-k) and 0 ≤ k ≤ Y.
(When k is 0, product = X*Y.)

product = 0       ; Initialize product and k
k = Y
until k = 0
   product += X
   k--
```

- And here's some code for it. The comments and program have been slightly modified relative to the auxiliary textbook.


```
; multiply.asm
; ...(Some comments omitted)...
;
; 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     BRZ     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   16
Y        .FILL   6
product  .BLKW   1              ; Holds X*Y at end
```

D. Sample Program: Reading a String

- The `readstring.asm` program prompts the user for input and reads in a sequence of characters one by one 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. Note we echo the characters as we read them (so that users can see what they type), and we store them into a buffer. When the user enters newline, we add a terminating `x00` character to the buffer so that we get a well-formed string. (The return won't be included.) Then we print out the string and halt.

```
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. The result makes up (most of) the block comments at the top of the `readstring.asm` file.. Note the comments use the C notations `&`*variable* and `*`*pointer,* where `&`*variable* means the address that the variable is stored at, and `*`*pointer* means the value stored at the pointed-to address.

```
; 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.
;
; Pseudocode:
; buffer_posn = &buffer  (the characters we read will
;     go into a buffer; buffer_posn points to our
;     position within the buffer (the location to store
;     the next character into).
;
; Print "Enter chars (return to halt): "
; Read char into R0
; Calculate R0 - return char
; until R0 - return char = 0
;    Print char in R0
;    *buffer_posn = R0
;    buffer_posn++
;    Read char into R0
;    Calculate R0 - return char
; end loop
; Print the return character
; *buffer_posn = null char to end the string
; Print the string
; Halt
```


- And here's the actual program:

```
; Register usage
;    R0 = GETC/OUT char, R1 = buffer_posn,
;    R2 = -(return char), R3 = temp
;
        .ORIG    x3000
         LEA     R1, buffer    ; buffer_posn = &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     ; *buffer_posn = char read in
         ADD     R1, R1, 1     ; buffer_posn++
         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     ; terminate string in buffer
         LEA     R0, buffer
         PUTS                  ; print the string we read in
         HALT

retChar .FILL    x0A           ; Return character (\n)
msg     .STRINGZ "Enter chars (return to halt): "
buffer  .BLKW    100           ; buffer space for string
        .END
```

E. 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` *register*, `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` *register*, `tablePtr` has the same effect as `LEA` *register*, `table` but the `LD` works even if `table` is inaccessible using `LEA`. (Note that `tablePtr` has to be declared close to the `LD` instruction, however.)

- Here's a short program to set `table[k]` to a value:

```
; table.asm
; Set table[k] = x where k and k are variables.
; We assume table entries take up one word each.

; Register usage: R0 = &table[0], R1 = &table[k],
; R2 is for temporary values

         .ORIG  x8000

; To make R0 = &table[0], we can always use the LD
; command below. If the table is close by, then the LEA
; will also work
         LD    R0, tablePtr    ; pt R0 to table[0]
; or     LEA   R0, table       ; pt R0 to table[0]

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

; Set table[k] = x
         LD    R2, x           ; R2 = value
         STR   R2, R1, 0       ; table[k] = value

         HALT

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

; tablePtr is a constant that contains the address of
; table[0]. If we are using the LEA of table, then
; tablePtr isn't necessary.
;
tablePtr .FILL  table          ; &table[0]
```



```
; We can make table be far away from the LEA by
; uncommenting the BLKW below
;        .BLKW  256

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

