



Lab 3: Decimal Converter
Worth 70 points (65 lab + 5 report)

Lab Objectives:

Now that we are moving into programming, it is important for you to focus on good programming practices. When programming in assembly this is especially true as assembly is not a pretty or easily readable language. You will need to rely on clear and useful comments to make your code easy to read and understand.

Having a clear plan for your program is essential in any language, but this is especially true in a low-level language like assembly. Thus, for all programs in this class, you will be required to create flowcharts to document your plan. *Write your flowchart BEFORE you write your code!* Once a good flowchart is created, mapping your solution to assembly code is much, much easier.

Part 1: Review

Review the Program Flow slides presented in class. The purpose of that lecture was to go over some of the basics of what a programmer does and the usage of flowcharting to solve a simple problem. Once the problem is solved it is easy to then implement it in whatever programming language you use, even if that language is LC3.

Part 2: Read and understand the program requirements

In this lab, you will design and write an LC3 program that implements the following:

- Prompt the user for input. The input will be either a signed integer in decimal, followed by a newline (ie, the 'Enter' key), or the character "X".
 - *You may assume the user enters correct input, and that the integer is between 32767 and -32768 (that is, you can fit the integer into a single LC3 word).*
- If the user enters "X", the program prints a Goodbye message and halts.
- If the user enters a positive integer, the program prints that integer in binary.
- If the user enters a negative integer, the program prints the 16-bit 2's complement representation of that integer in binary.
- The program returns to the prompt after printing.

Here is an example of what an interaction with the program might look like in the console (yours does not need to look exactly like this, but should have the same functionality):

```
Welcome to the conversion program!
Enter a decimal number or X to quit:
>12
Thanks, here it is in binary
00000000000001100
Enter a decimal number or X to quit:
>-3
Thanks, here it is in binary
1111111111111101
Enter a decimal number or X to quit:
>X
Bye. Have a great day.
----- Halting the processor -----
```

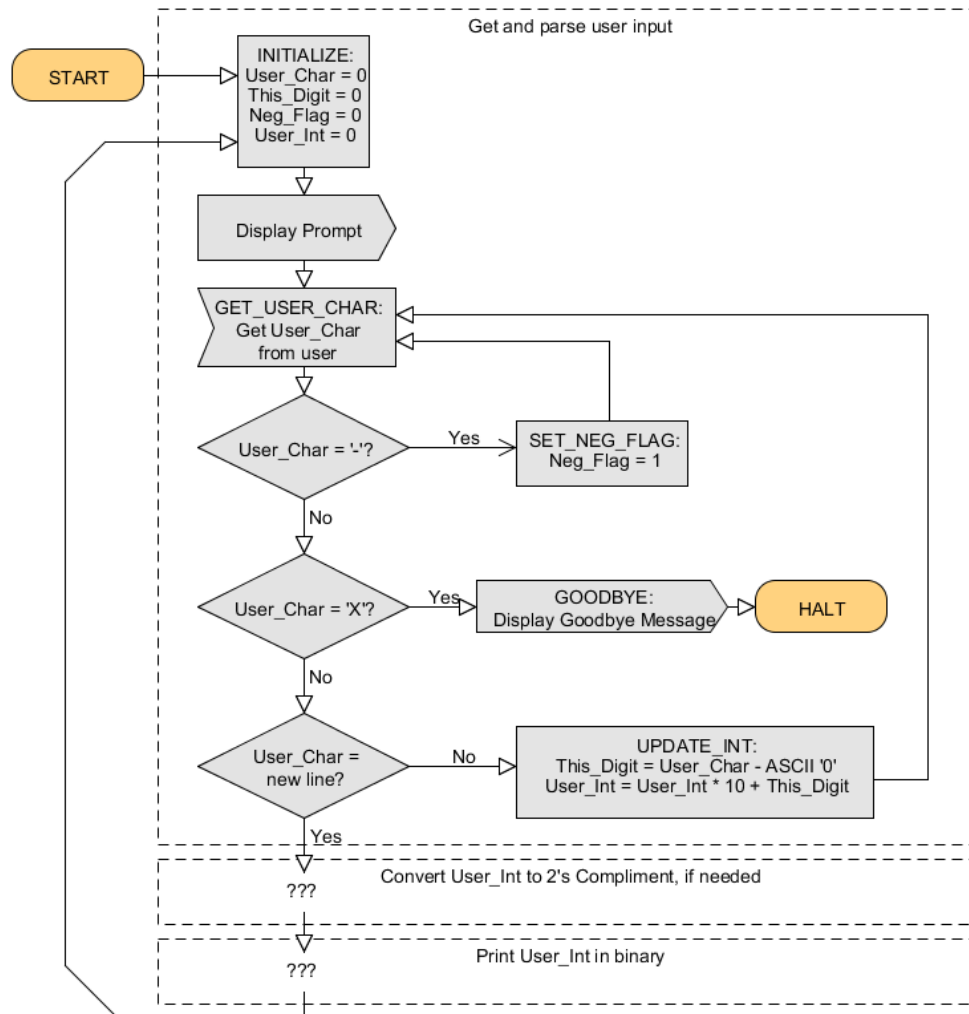
This code might seem simple, but some of the problems underlying this code are challenging. For example, LC3 can only read one single character at a time (using the trap directive called GETC). So your code has to first handle the “1,” because the user entered that first, then handle the “2” a moment later when the user enters it. How might a program accomplish this?

Also think about how your code might print an integer as a string of 1s and 0s. There’s no built-in function for this, so your code needs some way to check each individual bit in the integer. How might a program do that?

Part 3: Design a flowchart

Now that you understand the spec, it's time to design a flowchart that we can use to build our program.

We've started a flowchart for you. Note that many nodes have an all-caps LABEL. That makes it easy for you to figure out where a BRANCH instruction should branch to.



Spend some time thinking about the way the “get and parse user input” section works. Step through it a few times, imagining various things the user could input. What happens if the user enters “X”? What happens if the user enters “4”? What about “42”? Be sure it makes some sense to you before you move on!

You'll have to finish the flowchart by adding your own sub-flowcharts in place of the “???”.

CONVERTING TO 2's COMPLIMENT

Converting to 2SC is fairly straightforward. If Neg_Flag is clear (that is, 0), then convert User_Int to its 2SC. The procedure for doing this should be quite familiar to you by now! If Neg_Flag is set (anything besides zero), then leave User_Int unchanged. This should be a good warm up!

PRINTING USER_INT IN BINARY

This is much trickier! Displaying constant messages is fairly simple, because LC3 provides a trap directive called PRINT. However, printing variables in binary can be a real challenge.

You'll need to use a technique called bit-masking. You use one binary string, called a *mask*, to check a specific bit in another binary string. For example, if we wanted to check the third bit of User_Int, we could use a mask whose third bit is 1:

User_Int: 1101 1001	User_Int: 1101 0001
Mask: 0000 1000	Mask: 0000 1000
AND result: 0000 1000	AND result: 0000 0000

So if the third bit of User_Int is set, then AND result will not be zero. If the third bit of User_Int is clear, then the AND result will be zero.

Of course, you need to evaluate 16 bits, so you'll need some sort of loop structure, evaluating each bit in turn and printing the result. You'll need to implement a counter so that your loop exits after 16 iterations.

On each iteration, you'll need to check a new bit. One strategy is to store 16 masks, and load a different mask in each iteration of the loop (there's an example of this later in the lab). Another strategy is to use one constant mask, and left-shift User_Int on each iteration so that a new bit is revealed by the mask.

NOTES ON FLOWCHART DESIGN


Your flowchart is a map to your code. If your flowchart is good, then writing and navigating your code is easy.

It is very common to write a flowchart, feel great about it, then start coding and realize that your flowchart has serious flaws. It is very tempting in this situation to keep working on your code, and only update your flowchart later. This is usually *NOT* a good strategy! Instead, *fix your flowchart first*, and then fix your code.

Your final flowchart may be handwritten, but it must be neat, well-arranged, and completely legible. If you want to draw with software, that's fine too (we recommend draw.io or UMLET, but use whatever you like).

Part 3: Implement your flowchart in code:

Now that you have a nice flowchart, it's time to implement it in code! Use your flowchart to implement one node at a time. So start at START, and work your way out from there! For each node:

- Figure out how the instructions in the node can be implemented in assembly.
- Add the node to your code.
- Use the ASSEMBLE button () to check that your new code assembles.
- Test it in the simulator. Put a breakpoint at the beginning of your new node, and step through it a few times, watching the registers to be sure they behave as you expect.

Figuring out how to write something in Assembly can be tough. LC3 is Turing-complete, so it can compute anything that any other language can compute, but even doing straightforward things can involve several unintuitive steps.

A few LC3 examples might help you along:

Loading a stored value into a register:

If the value you want to load is between -16 and +15, you'll need two steps. First, set the register to 0 with AND, and then add an immediate value:

```
;Set R3 = -16
AND R3, R3, #0
ADD R3, R3, #-16
```

But you often want to load a value that can't fit in a five-bit immediate. For example, to load the ASCII value of "X", you'll need to store it in memory and load the value with the LD instruction:

```
;Set R3 = Ascii value of X
LD R3, ASCII_X

.
.  ;some other code here uses ASCII_X
.

HALT

ASCII_X .fill 88    ;ascii value of X
```

Note that the stored data (ASCII_X) is stored *AFTER* the HALT instruction. This is important, because you don't ever want to execute data! (Question: Why is executing data bad?)

Decision node:

Here is an example of a decision node:

```
.
. ;code that stores User_Char in R0
.

CHECK_FOR_X:
    ;Find    User_Char - ascii_x:
    LD R4, ASCII_X
    ADD R4, R4, R0

    ;if User Char is X, result is zero, so exit:
    BRZ GOODBYE
    ;else, move on:
    BR CHECK_FOR_NEWLINE

.
. ;more code
.

HALT

ASCII_X .fill -88 ; inverse of ascii value of X
```

Loading a mask from an array:

Here is an example of code that pre-defines an array of masks, and uses R2 to pick one mask out of the array. It works by loading the address of the array into R1, increases that address by R2, then loads the value at the new address into R0:

```
;R0 = MASK_ARRAY[ R2 ]
LEA R1, MASK_ARRAY
ADD R1, R1, R2
LDR R0, R1, #0

.
. ;use the mask
.

HALT

MASK_ARRAY:
.fill b10000000000000000
.fill b01000000000000000
.fill b00100000000000000
. . .
.fill b00000000000000001
```

Tips for Readable Assembly code:

- Use comments to break down the elements of your code. Many beginners comment every line – that’s understandable, since you’re just learning and it’s easy to forget how instructions work. But it’s more important to have comments that describe what blocks of code are for.
- Comments, labels, and variables should correspond to elements of your flowchart. It should be easy for a reader to find which node maps to which lines of code, and vice versa. (This makes debugging much, much easier!)
- Use whitespace (tabs and enters) to show your code’s structure. Assembly doesn’t have if-thens or loops, so you have to write your own, but you can still use indents to show where your hand-made loops are.
- Use labels that accurately describe the purpose of that label or variable.

Here’s a fairly readable block of assembly to illustrate the last few points:

```
; Left-shift R6 by N places:
LD R1, N
FOR_N:
    ;Left-shift R6 by multiplying by 2:
    ADD R6, R6, R6

    ;Check if for loop is finished:
```

```

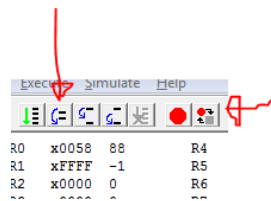
        ADD R1, R1, #-1
        BRZ  BREAK_FOR_N
        BR  FOR_N

BREAK_FOR_N:
.
.
.

```

Debugging Tips:

Learn to use the LC3 simulator's debugging tools. In particular, spend some time playing with breakpoints and step-throughs:



When you debug, try to isolate bugs. If something's not working, first try to figure out which node of your flowchart has the problem. Put a breakpoint at the entry to each node, and when that breakpoint triggers, check to see that the previous node did what it was supposed to do.

Lab Requirements

You must submit the following files to Canvas for this lab:

- Lab3.asm
- Lab3_writeup.txt

Furthermore, you must have the following during checkoff:

- A flowchart of your program

The code and the flowchart should agree—that is, the flowchart should accurately describe the program.

Check lab1 if you do not recall our expectations for lab write-ups. In addition:

- Discuss the algorithm(s) you designed. Were there any issues in implementing them?
- Discuss any assembly language techniques you developed or discovered.
- Describe how you used your flowchart when coding your program.

To alleviate file format issues we want lab reports in plain text. Feel free to use a word processor if you like to type it up but please submit a plain text file.

Collaboration: *You are allowed to discuss this lab with other students on this lab only from a high level, such as discussing your flowcharts, NOT by working on code together.*

Point Breakdown

5 pts: Print greeting message and quit when prompted

20 pts: Converts character decimal string into a 2SC number

20 pts: Print as a binary string

10 pts: Work with negative input

5 pts: Good flow chart

5 pts: Have good comments in the code

5 pts: Writeup